Hyperparameter Tuning Cookbook

A guide for scikit-learn, PyTorch, river, and spotpython

Thomas Bartz-Beielstein

Nov 1, 2025

Table of contents

Preface **1**
Book Structure . . . 1
Software Used in this Book . . . 2
Citation . . . 2

I. Optimization **5**

1. Aircraft Wing Weight Example **7**
1.1. AWWE Equation . . . 7
1.2. AWWE Parameters and Equations (Part 1) . . . 7
1.3. Goals: Understanding and Optimization . . . 8
1.4. Properties of the Python "Solver" . . . 10
1.5. Plot 1: Load Factor (N_z) and Aspect Ratio (A) . . . 11
1.6. Plot 2: Taper Ratio and Fuel Weight . . . 13
1.7. The Big Picture: Combining all Variables . . . 14
1.8. AWWE Landscape . . . 17
1.9. Summary of the First Experiments . . . 18
1.10. Exercise . . . 18
1.10.1. Adding Paint Weight . . . 18
1.11. Jupyter Notebook . . . 19

2. Introduction to `scipy.optimize` **21**
2.1. Derivative-free Optimization Algorithms . . . 22
2.1.1. Nelder-Mead Simplex Algorithm . . . 22
2.1.2. Powell's Method . . . 23
2.2. Gradient-based Optimization Algorithms . . . 25
2.2.1. An Introductory Example: Broyden-Fletcher-Goldfarb-Shanno Algorithm (BFGS) . . . 25
2.2.2. Background and Basics for Gradient-based Optimization . . . 26
2.2.3. Gradient . . . 26
2.2.4. Jacobian Matrix . . . 26
2.2.5. Hessian Matrix . . . 27
2.2.6. Gradient Descent . . . 28
2.2.7. Newton Method . . . 31
2.2.8. BFGS-Algorithm . . . 33
2.2.9. Procedure: . . . 33

2.2.10. Visualization BFGS for Rosenbrock . . . 35
2.3. Global Optimization . . . 35
2.3.1. Local vs Global Optimization . . . 35
2.3.2. Dual Annealing Optimization . . . 40
2.3.3. Differential Evolution . . . 44
2.3.4. Procedure . . . 45
2.3.5. Other global optimization algorithms . . . 48
2.3.6. DIRECT . . . 48
2.3.7. SHGO . . . 48
2.3.8. Basin-hopping . . . 48
2.4. Project: One-Mass Oscillator Optimization . . . 48
2.4.1. Introduction . . . 48
2.4.2. One-Mass Oscillator Model . . . 49
2.4.3. The Real-World Data . . . 49
2.4.4. Objective Function . . . 51
2.4.5. Results . . . 52
2.5. Exercises . . . 54
2.6. Jupyter Notebook . . . 55

II. Numerical Methods 57

3. Simulation and Surrogate Modeling 59
3.1. Surrogates . . . 59
3.1.1. Costs of Simulation . . . 60
3.1.2. Mathematical Models and Meta-Models . . . 60
3.1.3. Surrogates = Trained Meta-models . . . 60
3.1.4. Computer Experiments . . . 60
3.1.5. Limits of Mathematical Modeling . . . 61
3.1.6. Why Computer Simulations are Necessary . . . 61
3.1.7. Simulation Requirements . . . 61
3.2. Applications of Surrogate Models . . . 61
3.3. DACE and RSM . . . 62
3.3.1. Noise Handling in RSM and DACE . . . 63
3.4. Updating a Surrogate Model . . . 63

4. Sampling Plans 65
4.1. Ideas and Concepts . . . 65
4.1.1. The ‘Curse of Dimensionality’ and How to Avoid It . . . 67
4.1.2. Physical versus Computational Experiments . . . 67
4.1.3. Designing Preliminary Experiments (Screening) . . . 68
4.1.4. Special Considerations When Deploying Screening Algorithms . . . 77
4.2. Analyzing Variable Importance in Aircraft Wing Weight . . . 77
4.3. Designing a Sampling Plan . . . 80
4.3.1. Stratification . . . 80

4.3.2. Latin Squares and Random Latin Hypercubes 85
4.3.3. Space-filling Designs: Maximin Plans 88
4.3.4. Memory Management 93
4.3.5. Optimizing the Morris-Mitchell Criterion Φ_q 101
4.3.6. Evolutionary Operation 103
4.3.7. Putting it all Together 104
4.4. Experimental Analysis of the Morris-Mitchell Criterion 111
4.4.1. Evaluation of Sampling Designs 111
4.4.2. Demonstrate the Impact of mmphi Parameters 115
4.4.3. Morris-Mitchell Criterion: Impact of Adding Points 115
4.5. A Sample-Size Invariant Version of the Morris-Mitchell Criterion 120
4.5.1. Comparison of `mmphi()` and `mmphi_intensive()` 120
4.5.2. Plotting the Two Morris-Mitchell Criteria for Different Sample Sizes 121
4.6. Jupyter Notebook 124

5. Constructing a Surrogate 125
5.1. Stage One: Preparing the Data and Choosing a Modelling Approach . . 125
5.2. Stage Two: Parameter Estimation and Training 127
5.3. Stage Three: Model Testing 129
5.4. Jupyter Notebook 130

6. Response Surface Methods 131
6.1. What is RSM? 131
6.1.1. Visualization: Problems in Practice 134
6.1.2. RSM: Strategies 134
6.1.3. RSM: Noise in the Empirical Model 135
6.1.4. RSM: Natural and Coded Variables 135
6.1.5. RSM Low-order Polynomials 136
6.2. First-Order Models (Main Effects Model) 136
6.2.1. First-Order Model Properties 137
6.2.2. First-order Model with Interactions in python 137
6.2.3. Observations: First-Order Model with Interactions 139
6.3. Second-Order Models 139
6.3.1. Second-Order Models: Properties 140
6.3.2. Example: Stationary Ridge 140
6.3.3. Observations: Second-Order Model (Ridge) 141
6.3.4. Example: Rising Ridge 142
6.3.5. Summary: Rising Ridge 143
6.3.6. Falling Ridge 143
6.3.7. Saddle Point 143
6.3.8. Interpretation: Saddle Points 145
6.3.9. Summary: Ridge Analysis 145
6.4. General RSM Models 145
6.4.1. Ordinary Least Squares 145

6.5. General Linear Regression . . . 146
6.6. Designs . . . 147
6.6.1. Different Designs . . . 148
6.7. RSM Experimentation . . . 148
6.7.1. First Step . . . 148
6.7.2. Second Step . . . 148
6.7.3. Third Step . . . 148
6.8. RSM: Review and General Considerations . . . 149
6.8.1. Historical Considerations about RSM . . . 150
6.8.2. Status Quo . . . 150
6.8.3. The Role of Statistics . . . 150
6.8.4. New RSM is needed: DACE . . . 150
6.9. Exercises . . . 151
6.10. Jupyter Notebook . . . 151

7. Polynomial Models 153
7.1. Fitting a Polynomial . . . 153
7.2. Polynomial Fitting in Python . . . 154
7.2.1. Fitting the Polynomial . . . 154
7.2.2. Explaining the k-fold Cross-Validation . . . 155
7.2.3. Making Predictions . . . 156
7.2.4. Plotting the Results . . . 157
7.3. Example One: Aerofoil Drag . . . 158
7.4. Example Two: A Multimodal Test Case . . . 160
7.5. Extending to Multivariable Polynomial Models . . . 161
7.6. Jupyter Notebook . . . 163

8. Radial Basis Function Models 165
8.1. Radial Basis Function Models . . . 165
8.1.1. Fitting Noise-Free Data . . . 166
8.1.2. Numerical Stability Through Positive Definite Matrices . . . 170
8.1.3. Ill-Conditioning . . . 171
8.1.4. Parameter Optimization: A Two-Level Approach . . . 172
8.2. Python Implementation of the RBF Model . . . 175
8.2.1. The Rbf Class . . . 176
8.3. RBF Example: The One-Dimensional `sin` Function . . . 182
8.4. RBF Example: The Two-Diemnsional `dome` Function . . . 184
8.4.1. The Connection Between RBF Models and Neural Networks . . . 187
8.5. Radial Basis Function Models for Noisy Data . . . 188
8.5.1. Ridge Regularization Approach . . . 188
8.5.2. Reduced Basis Approach . . . 188
8.6. Jupyter Notebook . . . 189

9. Kriging (Gaussian Process Regression) 191
9.1. From Gaussian RBF to Kriging Basis Functions . . . 191

9.2. Building the Kriging Model . . . 192
9.3. MLE to estimate θ and p . . . 199
9.3.1. The Log-Likelihood . . . 199
9.3.2. Differentiation with Respect to μ . . . 201
9.3.3. Differentiation with Respect to σ . . . 202
9.3.4. Results of the Optimizations . . . 203
9.3.5. The Concentrated Log-Likelihood Function . . . 203
9.3.6. Optimizing the Parameters $\vec{\theta}$ and $\vec{p}$ . . . 204
9.3.7. Correlation and Covariance Matrices Revisited . . . 204
9.4. Implementing an MLE of the Model Parameters . . . 205
9.5. Kriging Prediction . . . 205
9.6. Kriging Example: Sinusoid Function . . . 207
9.6.1. Calculating the Correlation Matrix Ψ . . . 207
9.6.2. Computing the Ψ Matrix . . . 209
9.6.3. Selecting the New Locations . . . 210
9.6.4. Computing the ψ Vector . . . 210
9.6.5. Predicting at New Locations . . . 211
9.6.6. Visualization . . . 211
9.6.7. The Complete Python Code for the Example . . . 212
9.7. Jupyter Notebook . . . 218

10. Matrices **219**
10.1. Derivatives of Quadratic Forms . . . 219
10.2. The Condition Number . . . 220
10.3. The Moore-Penrose Pseudoinverse . . . 221
10.3.1. Definitions . . . 221
10.3.2. Implementation in Python . . . 221
10.4. Strictly Positive Definite Kernels . . . 222
10.4.1. Definition . . . 222
10.4.2. Connection to Positive Definite Matrices . . . 222
10.4.3. Connection to RBF Models . . . 223
10.5. Cholesky Decomposition and Positive Definite Matrices . . . 223
10.5.1. Example of Cholesky Decomposition . . . 226
10.5.2. Inverse Matrix Using Cholesky Decomposition . . . 227
10.6. Nyström Approximation . . . 228
10.6.1. What's the Big Idea? . . . 228
10.6.2. How Does It Work? . . . 228
10.6.3. Example . . . 229
10.6.4. Why Is This Useful? . . . 229
10.6.5. Applying the Nyström Approximation: How Nyström Approximation Helps Kriging . . . 230
10.6.6. Example: Predicting Temperature with Nyström-Kriging . . . 230
10.6.7. Details: Woodbury Matrix Identity for Avoiding the Big Inversion 231
10.6.8. The Example: Step-by-Step . . . 232

10.7. Extending spotpython's Kriging Surrogate with Nyström Approximation for Enhanced Scalability . . . 234
10.7.1. Introduction: Overcoming the Scalability Challenge in Kriging for Sequential Optimization . . . 234
10.7.2. Report Objectives and Structure . . . 235
10.8. Theoretical Foundations: The Nyström–Kriging Framework . . . 235
10.8.1. A Primer on Kriging (Gaussian Process Regression) . . . 235
10.8.2. The Nyström Method for Low-Rank Kernel Approximation . . . 236
10.8.3. Justification for Landmark Selection . . . 236
10.9. Implementation: A Scalable Kriging Class for spotpython . . . 236
10.9.1. Updated kriging.py with Nyström Approximation (excerpt) . . . 236
10.10 Implementation Details . . . 239
10.10.1 Architectural Enhancements to `init` . . . 239
10.10.2 The `fit()` Method: A Dual-Pathway Approach . . . 239
10.10.3 The `predict()` Method: Conditional Prediction Logic . . . 239
10.10.4 Critical Detail: Preserving Mixed Variable Type Functionality . 240
10.10.5 Seamless Integration into the Nyström Workflow . . . 240
10.11 Jupyter Notebook . . . 240

11. Infill Criteria 241
11.1. Balancing Exploitation and Exploration . . . 242
11.2. Expected Improvement (EI) . . . 242
11.3. Expected Improvement in the Hyperparameter Tuning Cookbook (Python Implementation) . . . 243
11.4. Jupyter Notebook . . . 244

III. Sequential Parameter Optimization Toolbox (SPOT) 245

12. Introduction to Sequential Parameter Optimization 247
12.1. Surrogate Model Based Optimization . . . 247
12.2. Advantages of the `spotpython` Approach . . . 248
12.3. Disadvantages of the spotpython Approach . . . 249
12.4. An Initial Example: Optimization of the One-dimensional Sphere Function 250
12.4.1. The Objective Function: Sphere . . . 250
12.4.2. The `Spot` Method as an Optimization Algorithm Using a Surrogate Model . . . 251
12.5. A Second Example: Optimization of a User-Specified Multi-dimensional Function . . . 253
12.6. A Third Example: Hyperparameter Tuning of a Neural Network . . . 258
12.7. Organization . . . 272
12.8. Summary: Basic Attributes and Methods of the `Spot` Object . . . 273
12.9. Run . . . 275
12.10 Print the Results . . . 275
12.11 Show the Progress . . . 276

12.12 Visualize the Surrogate . . . 276
12.13 Init: Build Initial Design . . . 277
12.14 Replicability . . . 278
12.15 Surrogates . . . 279
12.15.1 A Simple Predictor . . . 279
12.16 Tensorboard Setup . . . 279
12.16.1 Tensorboard Configuration . . . 279
12.16.2 Starting TensorBoard . . . 280
12.17 Demo/Test: Objective Function Fails . . . 280
12.18 Handling Results: Printing, Saving, and Loading . . . 281
12.19 `spotpython` as a Hyperparameter Tuner . . . 282
12.19.1 Modifying Hyperparameter Levels . . . 282
12.20 Sampling in spotpython . . . 286
12.21 Example: `Spot` and the Sphere Function . . . 287
12.21.1 The Objective Function: Sphere . . . 287
12.21.2 The `Spot` Method as an Optimization Algorithm Using a Surrogate Model . . . 288
12.22 `Spot` Parameters: `fun_evals`, `init_size` and `show_models` . . . 290
12.23 Print the Results . . . 292
12.24 Show the Progress . . . 292
12.25 Visualizing the Optimization and Hyperparameter Tuning Process with TensorBoard . . . 293
12.26 Jupyter Notebook . . . 298

13. Multi-dimensional Functions 299
13.1. The Objective Function: 3-dim Sphere . . . 299
13.1.1. Results . . . 301
13.1.2. TensorBoard . . . 305
13.1.3. Conclusion . . . 306
13.2. Exercises . . . 306
13.3. Selected Solutions . . . 308
13.4. Jupyter Notebook . . . 318

14. Isotropic and Anisotropic Kriging 319
14.1. Example: Isotropic `Spot` Surrogate and the 2-dim Sphere Function . . . 319
14.1.1. The Objective Function: 2-dim Sphere . . . 319
14.1.2. Results . . . 320
14.2. Example With Anisotropic Kriging . . . 321
14.2.1. Taking a Look at the `theta` Values . . . 323
14.3. Exercises . . . 324
14.3.1. 1. The Branin Function `fun_branin` . . . 324
14.3.2. 2. The Two-dimensional Sin-Cos Function `fun_sin_cos` . . . 325
14.3.3. 3. The Two-dimensional Runge Function `fun_runge` . . . 325
14.3.4. 4. The Ten-dimensional Wing-Weight Function `fun_wingwt` . . . 325
14.3.5. 5. The Two-dimensional Rosenbrock Function `fun_rosen` . . . 326

14.4. Selected Solutions . . . 326
14.4.1. Solution to Exercise Section 14.3.5: The Two-dimensional Rosenbrock Function `fun_rosen` . . . 326
14.5. Jupyter Notebook . . . 333

15. Sequential Parameter Optimization: Using `scipy` Optimizers 335
15.1. The Objective Function Branin . . . 335
15.2. The Optimizer . . . 336
15.2.1. TensorBoard . . . 337
15.3. Print the Results . . . 338
15.4. Show the Progress . . . 338
15.5. Exercises . . . 340
15.5.1. `dual_annealing` . . . 340
15.5.2. `direct` . . . 342
15.5.3. `shgo` . . . 344
15.5.4. `basinhopping` . . . 346
15.5.5. Performance Comparison . . . 348
15.6. Jupyter Notebook . . . 349

16. Using `sklearn` Surrogates in `spotpython` 351
16.1. Example: Branin Function with `spotpython`'s Internal Kriging Surrogate 351
16.1.1. The Objective Function Branin . . . 351
16.1.2. Running the surrogate model based optimizer `Spot`: . . . 352
16.1.3. TensorBoard . . . 353
16.1.4. Print the Results . . . 354
16.1.5. Show the Progress and the Surrogate . . . 354
16.2. Example: Using Surrogates From scikit-learn . . . 355
16.2.1. GaussianProcessRegressor as a Surrogate . . . 356
16.3. Example: One-dimensional Sphere Function With `spotpython`'s Kriging 358
16.3.1. Results . . . 366
16.4. Example: `Sklearn` Model GaussianProcess . . . 367
16.5. Exercises . . . 376
16.5.1. 1. A decision tree regressor: `DecisionTreeRegressor` . . . 376
16.5.2. 2. A random forest regressor: `RandomForestRegressor` . . . 377
16.5.3. 3. Ordinary least squares Linear Regression: `LinearRegression` 377
16.5.4. 4. Linear least squares with l2 regularization: `Ridge` . . . 377
16.5.5. 5. Gradient Boosting: `HistGradientBoostingRegressor` . . . 377
16.5.6. 6. Comparison of Surrogates . . . 377
16.6. Selected Solutions . . . 378
16.6.1. Solution to Exercise Section 16.5.5: Gradient Boosting . . . 378
16.7. Jupyter Notebook . . . 401

17. Sequential Parameter Optimization: Gaussian Process Models 403
17.1. Gaussian Processes Regression: Basic Introductory `scikit-learn` Example . . . 403
17.1.1. Train and Test Data . . . 404
17.1.2. Building the Surrogate With `Sklearn` . . . 404
17.1.3. Plotting the `Sklearn`Model . . . 404
17.1.4. The `spotpython` Version . . . 405
17.1.5. Visualizing the Differences Between the `spotpython` and the `sklearn` Model Fits . . . 406
17.2. Exercises . . . 407
17.2.1. `Schonlau Example Function` . . . 407
17.2.2. `Forrester Example Function` . . . 407
17.2.3. `fun_runge Function (1-dim)` . . . 408
17.2.4. `fun_cubed (1-dim)` . . . 409
17.2.5. The Effect of Noise . . . 409

18. Infill Criteria 411
18.1. Expected Improvement . . . 411
18.1.1. The Philosophy Behind Expected Improvement . . . 411
18.1.2. Mathematical Definition . . . 412
18.1.3. Understanding the Components . . . 413
18.2. EI: Implementation in spotpython . . . 413
18.2.1. Key Implementation Details . . . 413
18.2.2. Code Example from the Kriging Class . . . 414
18.3. Practical Advantages of Expected Improvement . . . 414
18.4. Connection to the Hyperparameter Tuning Cookbook . . . 414
18.5. Example: `Spot` and the 1-dim Sphere Function . . . 415
18.5.1. The Objective Function: 1-dim Sphere . . . 415
18.5.2. Results . . . 416
18.6. Same, but with EI as infill_criterion . . . 417
18.7. Non-isotropic Kriging . . . 419
18.8. Using `sklearn` Surrogates . . . 422
18.8.1. The spot Loop . . . 422
18.8.2. spot: The Initial Model . . . 423
18.8.3. Init: Build Initial Design . . . 424
18.8.4. Evaluate . . . 427
18.8.5. Build Surrogate . . . 427
18.8.6. A Simple Predictor . . . 427
18.9. Gaussian Processes regression: basic introductory example . . . 428
18.10. The Surrogate: Using scikit-learn models . . . 430
18.11. Additional Examples . . . 432
18.11.1. Optimize on Surrogate . . . 436
18.11.2. Evaluate on Real Objective . . . 436
18.11.3. Impute / Infill new Points . . . 436
18.12. Tests . . . 436

18.13EI: The Famous Schonlau Example . . . 437
18.14EI: The Forrester Example . . . 440
18.15Noise . . . 442
18.16Cubic Function . . . 448
18.17Modifying Lambda Search Space . . . 453

19. Handling Noise 455
19.1. Example: `Spot` and the Noisy Sphere Function . . . 455
19.1.1. The Objective Function: Noisy Sphere . . . 455
19.1.2. Reproducibility: Noise Generation and Seed Handling . . . 457
19.2. spotpython's Noise Handling Approaches . . . 459
19.3. Print the Results . . . 465
19.4. Noise and Surrogates: The Nugget Effect . . . 466
19.4.1. The Noisy Sphere . . . 466
19.5. Exercises . . . 469
19.5.1. Noisy `fun_cubed` . . . 469
19.5.2. `fun_runge` . . . 469
19.5.3. `fun_forrester` . . . 469
19.5.4. `fun_xsin` . . . 469

20. Optimal Computational Budget Allocation in spotpython 471
Citation . . . 471
20.1. Example: `spotpython`, OCBA, and the Noisy Sphere Function . . . 472
20.1.1. The Objective Function: Noisy Sphere . . . 472
20.2. Using Optimal Computational Budget Allocation (OCBA) . . . 473
20.2.1. The Noisy Sphere . . . 474
20.2.2. Print the Results . . . 484
20.3. Noise and Surrogates: The Nugget Effect . . . 485
20.3.1. The Noisy Sphere . . . 485
20.4. Exercises . . . 488
20.4.1. Noisy `fun_cubed` . . . 488
20.4.2. `fun_runge` . . . 489
20.4.3. `fun_forrester` . . . 489
20.4.4. `fun_xsin` . . . 489
20.5. Jupyter Notebook . . . 490

21. Kriging with Varying Correlation-p 491
21.1. Example: `Spot` Surrogate and the 2-dim Sphere Function . . . 491
21.1.1. The Objective Function: 2-dim Sphere . . . 491
21.1.2. Results . . . 492
21.2. Example With Modified p . . . 494
21.2.1. Taking a Look at the `p_val` Values . . . 495
21.3. Optimization of the `p_val` Values . . . 496
21.4. Optimization of Multiple `p_val` Values . . . 498

21.5. Exercises . . . 499
21.5.1. `fun_branin` . . . 499
21.5.2. `fun_sin_cos` . . . 500
21.5.3. `fun_runge` . . . 500
21.5.4. `fun_wingwt` . . . 500
21.6. Jupyter Notebook . . . 501

22. Factorial Variables **503**
22.1. Jupyter Notebook . . . 507

23. User-Specified Functions: Extending the Analytical Class **509**
23.1. Software Requirements . . . 509
23.2. The Single-Objective Function: User Specified . . . 510
23.3. The Objective Function: Extending the Analytical Class . . . 512
23.4. Results . . . 514
23.5. A Contour Plot . . . 515
23.6. Multi-Objective Functions . . . 517
23.6.1. Response Surface Experiment . . . 517
23.7. Jupyter Notebook . . . 523

24. Kriging with spotPython based on the Forrester et al. textbook **525**
24.1. High-level model and notation . . . 525
24.2. Correlation matrix build: build_Psi() . . . 525
24.3. Likelihood and hyperparameters: likelihood() . . . 526
24.4. Hyperparameter optimization: max_likelihood() . . . 527
24.5. Model fitting workflow: fit() . . . 528
24.6. Correlation vector for a new point: build_psi_vec() . . . 529
24.7. Prediction at a new x: _pred() and predict() . . . 530
24.8. Batch prediction: predict() . . . 531
24.9. Method variants and options . . . 532
24.10 Relation to Forrester (Ch. 2/3/6) . . . 532
24.11 Practical notes . . . 532
24.12 Kriging 1D demo: 03bc, 03c3^2, s(x), and EI (Forrester et al., 2008, Ch. 2/3/6) . . . 533

IV. Data-Driven Modeling and Optimization **537**

25. Basic Statistics and Data Analysis **539**
25.1. Exploratory Data Analysis . . . 539
25.1.1. Histograms . . . 539
25.1.2. Boxplots . . . 542
25.2. Probability Distributions . . . 542
25.2.1. Sampling from a Distribution . . . 543

25.3. Discrete Distributions . . . 543
25.3.1. Bernoulli Distribution . . . 543
25.3.2. Binomial Distribution . . . 544
25.4. Continuous Distributions . . . 545
25.4.1. Distribution functions: PDFs and CDFs . . . 545
25.5. Background: Expectation, Mean, Standard Deviation . . . 546
25.6. Distributions and Random Numbers in Python . . . 548
25.6.1. Calculation of the Standard Deviation with Python . . . 548
25.6.2. The Argument "axis" . . . 549
25.7. Expectation (Continuous) . . . 550
25.8. Variance and Standard Deviation (Continuous) . . . 550
25.8.1. The Uniform Distribution . . . 551
25.9. The Uniform Distribution . . . 552
25.9.1. The Normal Distribution . . . 553
25.9.2. Visualization of the Standard Deviation . . . 554
25.9.3. Realizations of a Normal Distribution . . . 555
25.10. The Normal Distribution . . . 556
25.11. The Exponential Distribution . . . 561
25.11.1. Standardization of Random Variables . . . 561
25.12. Covariance and Correlation . . . 561
25.12.1. The Multivariate Normal Distribution . . . 561
25.13. Covariance . . . 564
25.14. Correlation . . . 566
25.14.1. Definitions . . . 566
25.14.2. Computations . . . 568
25.14.3. The Outer-product and the `np.outer` Function . . . 569
25.14.4. Correlation and Independence . . . 572
25.14.5. Pearson's Correlation . . . 574
25.14.6. Interpreting the Correlation: Correlation Squared . . . 574
25.14.7. Partial Correlation . . . 575
25.15. Hypothesis Testing and the Null-Hypothesis . . . 578
25.15.1. Alternative Hypotheses, Main Ideas . . . 578
25.15.2. p-values: What they are and how to interpret them . . . 578
25.16. Statistical Power . . . 578
25.17. The Central Limit Theorem . . . 578
25.18. Maximum Likelihood . . . 578
25.19. Maximum Likelihood Estimation: Multivariate Normal Distribution . . . 580
25.19.1. The Joint Probability Density Function of the Multivariate Normal Distribution . . . 580
25.19.2. The Log-Likelihood Function . . . 581
25.20. Cross-Validation . . . 581
25.21. Mutual Information . . . 582
25.22. t-SNE . . . 583

26. Hypothesis Testing **585**
26.1. Paired-Sample Analysis . . . 585
26.1.1. Directional Hypotheses for a Paired Experiment . . . 586
26.1.2. The Paired Samples t-Test: A Technical Examination . . . 588
26.1.3. A Priori Sample Size Determination for Desired Power . . . 590
26.1.4. Effect Size (Δ) . . . 591
26.2. Non-inferiority Test . . . 596

27. Addressing Multicollinearity: Principle Component Analysis (PCA) and Factor Analysis (FA) **599**
27.1. Introduction . . . 599
27.2. The Car-Sales Data Set . . . 600
27.2.1. The Target Variable . . . 601
27.2.2. The Features . . . 601
27.2.3. Combining Non-categorical and Categorical (encoded) Data . . . 603
27.3. Fit the Linear Regression Model . . . 603
27.3.1. Model Summary and Interpretation . . . 604
27.4. Collinearity Diagnostics . . . 605
27.4.1. The Coefficient Table . . . 605
27.4.2. Eigenvalues and Condition Indices . . . 606
27.4.3. Kayser-Meyer-Olkin (KMO) Measure . . . 607
27.5. Addressing Multicollinearity with Principal Component Analysis (PCA) 608
27.5.1. Introduction to PCA . . . 608
27.5.2. Application of PCA in Regression Problems: . . . 608
27.5.3. Scree Plot . . . 609
27.5.4. Loading Scores (for PCA) . . . 609
27.5.5. PCA for Car Sales Example . . . 610
27.5.6. Creating the Regression Model with Principal Components . . . 614
27.5.7. Collinearity Diagnostics for PCA Regression Model . . . 616
27.5.8. PCA: Creating the Regression Model with three Principle Components only . . . 617
27.6. Addressing Multicollinearity and Latent Structure with Factor Analysis (FA) . . . 618
27.6.1. Introduction to Factor Analysis . . . 618
27.6.2. Determining the Number of Factors for Factor Analysis . . . 619
27.6.3. Scree Plot for Factor Analysis . . . 619
27.6.4. Factor Loadings . . . 620
27.6.5. Factor Scores . . . 622
27.6.6. Creating the Regression Model with Extracted Factors (from FA) 623
27.6.7. Factor Analysis: Creating the Regression Model with three Extracted Factors only . . . 626
27.7. Summary: Comparing OLS, PCA, and Factor Analysis Models . . . 628
27.7.1. Interpretation of the Regression Models . . . 628
27.7.2. Differences Compared to the Standard OLS Model . . . 629

27.7.3. Key Differences Between Loading Scores (PCA) and Factor Loadings (FA) . . . 630
27.7.4. Advantages of Using PCA and FA . . . 630
27.7.5. Disadvantages of Using PCA and FA . . . 631
27.7.6. When to Use Which Method . . . 631
27.8. Using Principal Components / Factors in Other Models . . . 632
27.8.1. Random Forest Regressor with the Full Dataset . . . 632
27.8.2. Random Forest Regressor with PCA Components . . . 633
27.8.3. Random Forest Regressor with Extracted Factors (from FA) . . . 634
27.8.4. Comparison of the Random Forest Models . . . 634
27.9. Videos: Principal Component Analysis (PCA) . . . 636
27.10 Jupyter Notebook . . . 636

28. Regression **637**
28.1. Supervised and Unsupervised Learning . . . 637
28.1.1. Starting point . . . 637
28.1.2. Philosophy . . . 637
28.1.3. Supervised Learning . . . 637
28.1.4. Unsupervised Learning . . . 638
28.2. Linear Regression . . . 639
28.2.1. The main ideas of fitting a line to data (The main ideas of least squares and linear regression.) . . . 639
28.2.2. Optimal Predictor . . . 640
28.2.3. Curse of Dimensionality and Parametric Models . . . 642
28.2.4. Assessing Model Accuracy and Bias-Variance Trade-off . . . 647
28.3. Multiple Regression . . . 651
28.4. R-squared . . . 651
28.4.1. R-Squared in Simple Linear Regression . . . 651
28.5. Assessing Confounding Effects in Multiple Regression . . . 652
28.6. Cross-Validation . . . 655

29. Classification **657**
29.1. Classification: Some Details . . . 658
29.2. k-Nearest Neighbor Classification . . . 658
29.2.1. Minkowski Distance . . . 659
29.3. Decision and Classification Trees . . . 661
29.3.1. Decision Trees . . . 661
29.3.2. Regression Trees . . . 661
29.4. The Confusion Matrix . . . 661
29.4.1. Sensitivity and Specificity . . . 661
29.5. Naive Bayes . . . 662
29.6. Gaussian Naive Bayes . . . 662

30. Clustering **663**
30.1. DBSCAN . . . 663

30.2. k-Means Clustering 663
30.3. DDMO-Additional Videos 663
30.4. DDMO-Exercises 668

V. Machine Learning and AI 675

31. Machine Learning and Artificial Intelligence 677
31.1. Jupyter Notebooks 677
31.2. Videos 677
31.2.1. June, 11th 2024 677
31.2.2. June, 18th 2024 677
31.2.3. June, 25th 2024 678
31.2.4. CNNs 678
31.2.5. RNN 679
31.2.6. LSTM 679
31.2.7. Pytorch/Lightning 679
31.2.8. July, 2nd 2024 680
31.2.9. Additional Lecture (July, 9th 2024)? 680
31.2.10. Additional Videos 681
31.2.11. All Videos in a Playlist 681
31.3. The StatQuest Introduction to PyTorch 681
31.3.1. Build a Simple Neural Network in PyTorch 682
31.3.2. Use the Neural Network and Graph the Output 684
31.3.3. Optimize (Train) a Parameter in the Neural Network and Graph the Output 685
31.4. Build a Long Short-Term Memory unit by hand using PyTorch + Lightning 691
31.4.1. Train the LSTM unit and use Lightning and TensorBoard to evaluate: Part 1 - Getting Started 696
31.4.2. Optimizing (Training) the Weights and Biases in the LSTM that we made by hand: Part 2 - Adding More Epochs without Starting Over 698
31.5. Using and optimzing the PyTorch LSTM, nn.LSTM() 701

VI. Introduction to Hyperparameter Tuning 705

32. Hyperparameter Tuning 707
32.1. Structure of the Hyperparameter Tuning Chapters 707
32.2. Goals of Hyperparameter Tuning 707

VII. Hyperparameter Tuning with Sklearn 709

33. HPT: sklearn 711
33.1. Introduction to sklearn . . . 711

34. HPT: sklearn SVC on Moons Data 713
34.1. Packages used in this Chapter . . . 713
34.2. Step 1: Setup . . . 714
34.3. Step 2: Initialization of the Empty `fun_control` Dictionary . . . 714
34.4. Step 3: SKlearn Load Data (Classification) . . . 715
34.5. Step 4: Specification of the Preprocessing Model . . . 717
34.6. Step 5: Select Model (`algorithm`) and `core_model_hyper_dict` . . . 717
34.7. Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` . . . 720
34.7.1. Modify hyperparameter of type numeric and integer (boolean) . 720
34.7.2. Modify hyperparameter of type factor . . . 721
34.7.3. Optimizers . . . 721
34.8. Step 7: Selection of the Objective (Loss) Function . . . 721
34.8.1. Predict Classes or Class Probabilities . . . 722
34.9. Step 8: Calling the SPOT Function . . . 722
34.9.1. The Objective Function . . . 722
34.9.2. Run the `Spot` Optimizer . . . 722
34.9.3. TensorBoard . . . 724
34.10 Step 9: Results . . . 725
34.11 Get Default Hyperparameters . . . 727
34.12 Get SPOT Results . . . 728
34.12.1 Plot: Compare Predictions . . . 728
34.12.2 Detailed Hyperparameter Plots . . . 731
34.12.3 Parallel Coordinates Plot . . . 734
34.12.4 Plot all Combinations of Hyperparameters . . . 734

35. HPT: sklearn SVR on Regression Data 737
35.1. Step 1: Setup . . . 737
35.2. Step 2: Initialization of the Empty `fun_control` Dictionary . . . 737
35.3. Step 3: SKlearn Load Data (Classification) . . . 738
35.4. Step 4: Specification of the Preprocessing Model . . . 739
35.5. Step 5: Select Model (`algorithm`) and `core_model_hyper_dict` . . . 740
35.6. Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` . . . 742
35.6.1. Modify hyperparameter of type numeric and integer (boolean) . 742
35.6.2. Modify hyperparameter of type factor . . . 743
35.6.3. Optimizers . . . 743
35.7. Step 7: Selection of the Objective (Loss) Function . . . 744
35.7.1. Predict Classes or Class Probabilities . . . 744

35.8. Step 8: Calling the SPOT Function 744
35.8.1. The Objective Function 744
35.8.2. Run the `Spot` Optimizer 745
35.8.3. TensorBoard 748
35.9. Get Default Hyperparameters 749
35.10. Get SPOT Results 750
35.10.1. Plot: Compare Predictions 750
35.10.2. Detailed Hyperparameter Plots 751
35.10.3. Parallel Coordinates Plot 766

VIII Hyperparameter Tuning with River 767

36. HPT: River 769
36.1. Introduction to River 769

37. Simplifying Hyperparameter Tuning in Online Machine Learning—The spotRiverGUI 771
37.1. Introduction 771
37.2. Installation and Starting 772
37.2.1. Installation 772
37.2.2. Starting the GUI 773
37.3. Binary Classification 773
37.3.1. Binary Classification Options 773
37.3.2. Experiment Options 776
37.3.3. Evaluation Options 777
37.3.4. Online Machine Learning Model Options 779
37.4. Regression 781
37.5. Showing the Data 781
37.6. Saving and Loading 785
37.6.1. Saving the Experiment 785
37.6.2. Loading an Experiment 786
37.7. Running a New Experiment 786
37.7.1. Starting and Stopping Tensorboard 786
37.8. Performing the Analysis 786
37.9. Summary and Outlook 790
37.10. Appendix 791
37.10.1. Adding new Tasks 791

38. `river` Hyperparameter Tuning: Hoeffding Tree Regressor with Friedman Drift Data 793
38.1. The Friedman Drift Data Set 793
38.2. Setup 794
38.2.1. General Experiment Setup 794
38.2.2. Data Setup 795

38.2.3. Evaluation Setup . . . 796
38.2.4. River-Specific Setup . . . 796
38.2.5. Model Setup . . . 797
38.2.6. Objective Function Setup . . . 798
38.2.7. Surrogate Model Setup . . . 798
38.2.8. Summary: Setting up the Experiment . . . 798
38.2.9. Run the `Spot` Optimizer . . . 799
38.3. Using the `spotgui` . . . 802
38.4. Results . . . 803
38.5. Performance of the Model with Default Hyperparameters . . . 804
38.5.1. Get Default Hyperparameters and Fit the Model . . . 804
38.5.2. Evaluate the Model with Default Hyperparameters . . . 805
38.5.3. Show Predictions of the Model with Default Hyperparameters . . . 806
38.6. Get SPOT Results . . . 807
38.7. Visualize Regression Trees . . . 809
38.7.1. Spot Model . . . 810
38.8. Detailed Hyperparameter Plots . . . 811
38.9. Parallel Coordinates Plots . . . 814

39. The Friedman Drift Data Set 815
39.1. Setup . . . 816
39.1.1. Select a User Hyperdictionary . . . 816
39.2. Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` . . . 817
39.2.1. Run the `Spot` Optimizer . . . 818
39.3. Results . . . 819
39.4. Performance of the Model with Default Hyperparameters . . . 821
39.4.1. Get Default Hyperparameters and Fit the Model . . . 821
39.4.2. Evaluate the Model with Default Hyperparameters . . . 821
39.4.3. Show Predictions of the Model with Default Hyperparameters . . . 822
39.5. Get SPOT Results . . . 823
39.6. Detailed Hyperparameter Plots . . . 825
39.7. Parallel Coordinates Plots . . . 828

IX. Hyperparameter Tuning with PyTorch Lightning 829

40. Basic Lightning Module 831
40.1. Introduction . . . 831
40.2. Starter Example: Transformer . . . 832
40.3. Lightning Core Methods . . . 833
40.3.1. Training Step . . . 834
40.3.2. Validation Step . . . 835
40.3.3. Test Step . . . 836
40.3.4. Predict Step . . . 837

40.4. Lightning Extras . . . 838
40.4.1. Lightning: Save Hyperparameters . . . 838
40.4.2. Lightning: Model Loading . . . 839
40.5. Starter Example: Linear Neural Network . . . 839
40.5.1. Hidden Layers . . . 839
40.5.2. Hyperparameters . . . 839
40.5.3. The LightningBasic Class . . . 840
40.5.4. The Data Set: Diabetes . . . 844
40.5.5. The DataLoaders . . . 845
40.5.6. The Trainer . . . 845
40.5.7. Using a DataModule . . . 847
40.6. Using spotpython with Pytorch Lightning . . . 848

41. Details of the Lightning Module Integration in spotpython 877
41.1. Introduction . . . 877
41.2. 1. spotpython.fun.hyperlight.HyperLight.fun() . . . 877
41.3. 2. spotpython.light.trainmodel.train_model() . . . 878
41.4. 3. Trainer: fit and validate . . . 880
41.4.1. Commented: Using the fun_control dictionary . . . 881
41.4.2. Using the source code: . . . 881
41.4.3. DataModule . . . 888

42. User Specified Basic Lightning Module With spotpython 893
42.1. Introduction . . . 893
42.1.1. Dataset . . . 893
42.1.2. DataModule . . . 895
42.2. The Neural Network: MyRegressor . . . 900
42.3. Calling the Neural Network With spotpython . . . 906
42.4. Looking at the Results . . . 908
42.4.1. Tuning Progress . . . 908
42.4.2. Tuned Hyperparameters and Their Importance . . . 908
42.4.3. Get the Tuned Architecture . . . 908

43. HPT PyTorch Lightning: Data 909
43.1. Setup . . . 909
43.2. Initialization of the `fun_control` Dictionary . . . 910
43.3. Loading the Diabetes Data Set . . . 910
43.3.1. Data Set and Data Loader . . . 910
43.3.2. Preparing Training, Validation, and Test Data . . . 911
43.3.3. Dataset for spotpython . . . 913
43.4. The LightDataModule . . . 914
43.4.1. The `prepare_data()` Method . . . 915
43.4.2. The `setup()` Method . . . 915
43.4.3. The `train_dataloader()` Method . . . 918
43.4.4. The `val_dataloader()` Method . . . 919

43.4.5. The `test_dataloader()` Method . . . 921
43.4.6. The `predict_dataloader()` Method . . . 922
43.5. Using the `LightDataModule` in the `train_model()` Method . . . 923
43.6. Further Information . . . 924
43.6.1. Preprocessing . . . 924

44. Hyperparameter Tuning with `spotpython` and `PyTorch` Lightning for the Diabetes Data Set 925
44.1. The Basic Setting . . . 925
44.2. Looking at the Results . . . 953
44.2.1. Tuning Progress . . . 953
44.2.2. Tuned Hyperparameters and Their Importance . . . 954
44.2.3. Get the Tuned Architecture . . . 958
44.2.4. Test on the full data set . . . 959
44.3. Cross Validation With Lightning . . . 960
44.4. Extending the Basic Setup . . . 961
44.4.1. General Experiment Setup . . . 962
44.4.2. Data Setup . . . 962
44.4.3. Objective Function `fun` . . . 962
44.4.4. Core-Model Setup . . . 962
44.4.5. Hyperdict Setup . . . 963
44.4.6. Other Settings . . . 963
44.5. Tensorboard . . . 963
44.6. Loading the Saved Experiment and Getting the Hyperparameters of the Tuned Model . . . 963
44.7. Using the `spotgui` . . . 964
44.8. Summary . . . 965

45. Early Stopping Explained: HPT with `spotpython` and `PyTorch` Lightning for the Diabetes Data Set 967
45.1. The Basic Setting . . . 967
45.2. Looking at the Results . . . 985
45.2.1. Tuning Progress . . . 985
45.2.2. Tuned Hyperparameters and Their Importance . . . 986
45.2.3. Get the Tuned Architecture . . . 990
45.2.4. Test on the full data set . . . 991
45.3. Cross Validation With Lightning . . . 992
45.4. Extending the Basic Setup . . . 993
45.4.1. General Experiment Setup . . . 994
45.4.2. Data Setup . . . 994
45.4.3. Objective Function `fun` . . . 994
45.4.4. Core-Model Setup . . . 994
45.4.5. Hyperdict Setup . . . 995
45.4.6. Other Settings . . . 995
45.5. Tensorboard . . . 995

45.6. Loading the Saved Experiment and Getting the Hyperparameters of the Tuned Model . . . 995
45.7. Using the `spotgui` . . . 996
45.8. Summary . . . 997

46. Hyperparameter Tuning with PyTorch Lightning and User Data Sets 999
46.1. Loading a User Specified Data Set . . . 999
46.2. Summary . . . 1012

47. Hyperparameter Tuning with PyTorch Lightning and User Models 1013
47.1. Using a User Specified Model . . . 1013
47.2. Details . . . 1040
47.2.1. Model Setup . . . 1040
47.2.2. The `my_hyper_dict.py` File . . . 1040
47.2.3. The `my_hyper_dict.json` File . . . 1041
47.2.4. The `my_regressor.py` File . . . 1043
47.3. Connection with the LightDataModule . . . 1048
47.3.1. The `prepare_data()` Method . . . 1048
47.3.2. The `setup()` Method . . . 1048
47.3.3. The `train_dataloader()` Method . . . 1053
47.3.4. The `val_dataloader()` Method . . . 1055
47.3.5. The `test_dataloader()` Method . . . 1056
47.3.6. The `predict_dataloader()` Method . . . 1057
47.4. Using the `LightDataModule` in the `train_model()` Method . . . 1059
47.5. The Last Connection: The `HyperLight` Class . . . 1060
47.6. Further Information . . . 1061
47.6.1. Preprocessing . . . 1061

48. Hyperparameter Tuning with PyTorch Lightning: ResNets 1063
48.1. Residual Neural Networks . . . 1063
48.1.1. Residual Connections . . . 1063
48.1.2. Implementation of the Original ResNet Block . . . 1064
48.1.3. Implementation of the Pre-Activation ResNet Block . . . 1068
48.1.4. The Overall ResNet Architecture . . . 1071

49. Neural ODEs 1077
49.1. Neural Ordinary Differential Equations . . . 1077
49.2. Regression Example . . . 1083
49.2.1. Specifying the Dynamics Layer . . . 1083
49.3. Further Reading . . . 1086

50. Neural ODE Example 1089
50.1. Implementation of a Neural ODE . . . 1089

51. Physics Informed Neural Networks 1103
51.1. PINNs . . . 1103

51.2. Generation and Visualization of the Training Data and the Ground Truth (Function) . . . 1103
51.3. Gradient With Autograd . . . 1106
51.4. Network . . . 1106
51.5. Basic Neutral Network . . . 1109
51.6. PINNs . . . 1112
51.6.1. PINNs: Parameter Estimation . . . 1115
51.7. Summary . . . 1120

52. Hyperparameter Tuning with PyTorch Lightning: Physics Informed Neural Networks 1121
52.1. PINNs . . . 1121
52.1.1. The Ground Truth Model . . . 1121
52.1.2. Required Files . . . 1123
52.1.3. The New `pinn_hyperdict.py` File . . . 1124
52.1.4. The New `pinn_regressor.py` File . . . 1125
52.1.5. The New `pinn_hyperdict.json` File . . . 1132

53. Explainable AI with SpotPython and Pytorch 1133
53.1. Running the Hyperparameter Tuning or Loading the Existing Model . . 1134
53.2. Results from the Hyperparameter Tuning Experiment . . . 1156
53.2.1. Getting the Best Model, i.e, the Tuned Architecture . . . 1157
53.3. Training the Tuned Architecture on the Test Data . . . 1158
53.4. Visualizing the Neural Network Architecture . . . 1161
53.5. XAI Methods . . . 1162
53.5.1. Weights . . . 1162
53.5.2. Activations . . . 1162
53.5.3. Gradients . . . 1162
53.5.4. Getting the Weights . . . 1163
53.5.5. Getting the Activations . . . 1171
53.5.6. Getting the Gradients . . . 1174
53.6. Feature Attributions . . . 1183
53.6.1. Integrated Gradients . . . 1183
53.6.2. Deep Lift . . . 1184
53.6.3. Feature Ablation . . . 1186
53.7. Conductance . . . 1188

54. HPT PyTorch Lightning Transformer: Introduction 1193
54.1. Transformer Basics . . . 1193
54.1.1. Embedding . . . 1193
54.1.2. Attention . . . 1194
54.1.3. Self-Attention . . . 1196
54.1.4. Masked Self-Attention . . . 1196
54.1.5. Generation of Outputs . . . 1196
54.1.6. End-Of-Sequence-Token . . . 1197

54.2. Details of the Implementation . . . 1197
54.2.1. Dot Product Attention . . . 1199
54.2.2. Scaled Dot Product Attention . . . 1200
54.3. Example: Transformer in Lightning . . . 1201
54.3.1. Downloading the Pretrained Models . . . 1202
54.3.2. The Transformer Architecture . . . 1203
54.3.3. Attention Mechanism . . . 1203
54.3.4. Multi-Head Attention . . . 1204
54.3.5. Permutation Equivariance . . . 1207
54.3.6. Transformer Encoder . . . 1207
54.3.7. Layer Normalization and Feed-Forward Network . . . 1209
54.3.8. Positional Encoding . . . 1211
54.3.9. Learning Rate Warm-up . . . 1214
54.3.10. PyTorch Lightning Module . . . 1217
54.4. Experiment: Sequence to Sequence . . . 1219
54.4.1. Dataset and Data Loaders . . . 1219
54.4.2. The Reverse Predictor Class . . . 1221
54.4.3. Gradient Clipping . . . 1221
54.4.4. Implementation of the Lightning Trainer . . . 1222
54.4.5. Training the Model . . . 1223
54.5. Visualizing Attention Maps . . . 1223
54.6. Conclusion . . . 1226
54.7. Additional Considerations . . . 1226
54.7.1. Complexity and Path Length . . . 1226
54.8. Further Reading . . . 1227

55. Hyperparameter Tuning of a Transformer Network with PyTorch Lightning 1229
55.1. Basic Setup . . . 1229
55.2. Looking at the Results . . . 1237
55.2.1. Tuning Progress . . . 1237
55.2.2. Tuned Hyperparameters and Their Importance . . . 1238
55.3. Hyperparameter Considerations . . . 1238
55.3.1. Important: Constraints and Interconnections: . . . 1239
55.3.2. Practical Considerations: . . . 1240
55.4. Summary . . . 1241

56. Saving and Loading 1243
56.1. Required Python Packages . . . 1243
56.2. Quick Overview of `spotpython` Save and Load Functions . . . 1244
56.3. spotpython: Saving and Loading Optimization Experiments . . . 1303
56.3.1. Getting the Tuned Hyperparameters . . . 1307
56.4. spotpython as a Hyperparameter Tuner . . . 1307
56.4.1. The Diabetes Data Set . . . 1307
56.5. Saving and Loading PyTorch Lightning Models . . . 1359
56.5.1. Get the Tuned Architecture . . . 1359

56.5.2. Load a Model from Checkpoint 1360
56.6. Converting a Lightning Model to a Plain Torch Model 1367
56.6.1. The Function `get_removed_attributes_and_base_net` 1367
56.6.2. An Example how to use the Plain Torch Net 1370

57. Hyperparameter Tuning with `spotpython` and `PyTorch` Lightning for the Diabetes Data Set Using a ResNet Model 1373
57.1. Looking at the Results 1384
57.1.1. Tuning Progress 1384
57.1.2. Tuned Hyperparameters and Their Importance 1384
57.1.3. Get the Tuned Architecture 1388
57.1.4. Test on the full data set 1389
57.1.5. Cross Validation With Lightning 1390
57.2. Summary 1391

58. Hyperparameter Tuning with `spotpython` and `PyTorch` Lightning for the Diabetes Data Set Using a User Specified ResNet Model 1393
58.1. Looking at the Results 1434
58.1.1. Tuning Progress 1434
58.1.2. Tuned Hyperparameters and Their Importance 1435
58.1.3. Get the Tuned Architecture 1438
58.2. Details of the User-Specified ResNet Model 1439
58.2.1. `my_resnet.py` 1439
58.2.2. `my_hyperdict.py` 1443
58.2.3. `my_hyperdict.json` 1444
58.3. Summary 1447

59. Hyperparameter Tuning with `spotpython` and `PyTorch` Lightning Using a CondNet Model 1449
59.1. Looking at the Results 1480
59.1.1. Tuning Progress 1480
59.1.2. Tuned Hyperparameters and Their Importance 1480
59.1.3. Get the Tuned Architecture 1484

X. Multi Objective Optimization 1487

60. Introduction to Desirability Functions 1489
60.1. The `Python` Packages Used in This Article 1491
60.2. Desirability 1492
60.2.1. Basic Desirability Functions 1492
60.2.2. Overall Desirability 1496
60.2.3. Non-Standard Features 1496
60.3. Related Work 1501

60.4. An Example With Two Objectives: Chemical Reaction 1502
60.4.1. The Two Objective Functions: Conversion and Activity 1503
60.4.2. Contour Plot Generation 1504
60.5. Multi-Objective Optimization and Maximizing Desirability 1513
60.6. Surrogate-Model Based Optimization Using Desirability 1519
60.7. Surrogate Model Hyperparameter Tuning 1530
60.7.1. The Single-Objective Approach 1531
60.7.2. Weighted Multi-Objective Function 1535
60.7.3. Multi-Objective Hyperparameter Tuning With Desirability . . . 1536
60.8. Conclusion 1542
60.9. Appendix 1543
60.9.1. Alternative Optimization Approach Using a Circular Design Region1543

XI. Lernmodule 1547

61. Lernmodul: Aircraft Wing Weight Example (AWWE) 1549
61.1. Einleitung 1549
61.2. Die AWWE-Gleichung 1549
61.3. Eingabegrößen (Parameter) 1550
61.4. Ausgabegröße 1551
61.5. Analyse von Effekten, Interaktionen und Wichtigkeit 1551
61.5.1. Mathematische Eigenschaften und Ziele 1551
61.5.2. Wichtigkeit der Variablen und deren Effekte 1551
61.5.3. Expertenwissen und Screening-Studien 1553
61.6. Zusatzmaterialien 1554

62. Lernmodul: Versuchspläne (Sampling-Pläne) für Computerexperimente 1555
62.1. Einführung 1555
62.2. Der “Fluch der Dimensionen” 1556
62.2.1. Das Volumen in hochdimensionalen Räume 1556
62.2.2. Volumen einer Kugel in hohen Dimensionen 1556
62.2.3. Der mathematische Grund: Ein Wettlauf zweier Funktionen . . . 1556
62.2.4. Der geometrische Grund: Die Dominanz der Ecken 1557
62.3. Entwurf von Vorab-Experimenten (Screening) 1558
62.4. Entwurf eines umfassenden Sampling-Plans 1559
62.4.1. Stratifikation 1559
62.4.2. Maximin-Pläne 1561
62.4.3. Das Morris-Mitchell-Kriterium (Phi_q) 1561
62.5. Alternative Sampling-Pläne: Sobol-Sequenzen 1563
62.6. Zusammenfassung 1563
62.7. Zusatzmaterialien 1564

63. Lernmodul: Eine Einführung in Kriging 1565
63.1. Konzeptionelle Grundlagen des Kriging . . . 1565
63.1.1. Von einfachen Modellen zur intelligenten Interpolation . . . 1565
63.1.2. Die Kernphilosophie des Kriging: Eine stochastische Prozessperspektive . . . 1567
63.2. Die mathematische Architektur eines Kriging-Modells . . . 1568
63.2.1. Glossar der Kriging-Notation . . . 1568
63.2.2. Der Korrelationskernel: Quantifizierung von Beziehungen . . . 1569
63.2.3. Aufbau des Systems: Die Korrelationsmatrizen Ψ und $\vec{\psi}$ . . . 1571
63.2.4. Modellkalibrierung durch Maximum-Likelihood-Schätzung (MLE) 1573
63.3. Implementierung und Vorhersage . . . 1575
63.3.1. Der Kriging-Prädiktor: Generierung neuer Werte . . . 1575
63.3.2. Numerische Best Practices und der Nugget-Effekt . . . 1576
63.4. Eine vollständige exemplarische Vorgehensweise: Kriging der Sinusfunktion 1580
63.4.1. Schritt-für-Schritt-Codeausführung und Interpretation . . . 1586
63.4.2. Fazit und Ausblick . . . 1589
63.5. Zusatzmaterialien . . . 1590

64. Lernmodul: Die Cholesky-Zerlegung 1591
64.1. Einführung . . . 1591
64.2. Definition und Eigenschaften . . . 1591
64.2.1. Symmetrische, positiv definite Matrizen . . . 1591
64.2.2. Die Zerlegung . . . 1592
64.2.3. Vorteile der Cholesky-Zerlegung . . . 1592
64.3. Berechnung der Cholesky-Zerlegung . . . 1592
64.3.1. Beispiel mit Ψ . . . 1593
64.4. Anwendungen der Cholesky-Zerlegung . . . 1593
64.4.1. Lösung linearer Gleichungssysteme . . . 1594
64.4.2. Berechnung von Determinanten . . . 1594
64.4.3. Kriging und Gauß-Prozess-Regression (GPR) . . . 1594
64.4.4. Generierung von Stichproben aus multivariaten Normalverteilungen 1595
64.4.5. Anwendungen in Optimierungsalgorithmen . . . 1595
64.5. Implementierung in Python . . . 1595
64.6. Fazit . . . 1598
64.7. Zusatzmaterialien . . . 1598

65. Lernmodul: Erweiterung des Kriging-Modells: Numerische Optimierung der Hyperparameter 1601
65.1. Einleitung . . . 1601
65.2. Kriging-Hyperparameter: Theta ($\vec{\theta}$) und p ($\vec{p}$) . . . 1601
65.3. Die Notwendigkeit der Optimierung: Die konzentrierte Log-Likelihood . 1602
65.4. Numerische Optimierungsalgorithmen . . . 1602
65.5. Erweiterung des Sinusfunktions-Beispiels mit Hyperparameter-Optimierung . . . 1603
65.5.1. Kriging-Basisfunktionen (Definition der Korrelation) . . . 1604

65.5.2. Zielfunktion für die Hyperparameter-Optimierung (Negative Log-Likelihood) . . . 1605
65.5.3. Datenpunkte für das Sinusfunktions-Beispiel . . . 1606
65.5.4. Optimierung von Theta . . . 1607
65.5.5. Vorhersage mit optimiertem Theta . . . 1608
65.5.6. Visualisierung der Ergebnisse . . . 1608
65.5.7. 6. Ergebnisse und Diskussion . . . 1609
65.5.8. 7. Fazit und Ausblick . . . 1609
65.6. Zusatzmaterialien . . . 1610

66. Lernmodul: Erweiterung des Kriging-Modells zu einer Klasse (Python Code) 1611
66.1. Ein erste Beispielverwendung: . . . 1615
66.2. Ein zweites Beispiel mit train_test_split: . . . 1616

67. Lernmodul: Kriging Projekt 1619

68. 2. Die Black-Box-Funktion 1625

69. Hauptskript für die sequentielle Optimierung 1627

70. — Schritt 4: Surrogat-basierte Optimierung zur Suche des nächsten Infill-Punktes — 1631

71. Suchgrenzen für die Optimierung auf dem Surrogatmodell 1633

72. Fitte das finale Modell mit allen verfügbaren N_max_evaluations Punkten 1635

73. Lernmodul: Kriging Projekt mit Expected Improvement 1637
73.1. Einleitung . . . 1637
73.2. 1. Die KrigingRegressor-Klasse (Erweiterung) . . . 1637
73.3. 2. Die Black-Box-Funktion `f(x)` (Wiederholung) . . . 1638
73.4. 3. Initialer Stichprobenplan `X` (Wiederholung) . . . 1638
73.5. 4. Sequenzieller Optimierungsablauf mit Expected Improvement . . . 1638
73.6. Der entsprechende Python-Code: . . . 1639

74. 2. Die Black-Box-Funktion f(x) 1645
74.1. Hauptskript für die sequentielle Optimierung . . . 1646
74.2. Ergebnisse und Diskussion . . . 1659

Appendices 1661

A. Introduction to Jupyter Notebook 1661
A.1. Different Notebook cells . . . 1661
A.1.1. Code cells . . . 1661
A.1.2. Markdown cells . . . 1661

A.1.3. Raw cells . . . 1662
A.2. Install Packages . . . 1662
A.3. Load Packages . . . 1663
A.4. Functions in Python . . . 1663
A.5. List of Useful Jupyter Notebook Shortcuts . . . 1664

B. Git Introduction **1667**
B.1. Learning Objectives . . . 1667
B.2. Basics of Git . . . 1667
B.2.1. Initializing a Repository: `git init` . . . 1667
B.2.2. Ignoring Files: `.gitignore` . . . 1668
B.2.3. Adding Changes to the Staging Area: `git add` . . . 1668
B.2.4. Transferring Changes to Memory: `git commit` . . . 1669
B.2.5. Check the Status of Your Repository: `git status` . . . 1670
B.2.6. Review Your Repository's History: `git log` . . . 1671
B.3. Branches (Timelines) . . . 1671
B.3.1. Creating an Alternative Timeline: `git branch` . . . 1671
B.3.2. The Pointer to the Current Branch: `HEAD` . . . 1672
B.3.3. Switching to an Alternative Timeline: `git switch` . . . 1672
B.3.4. Switching to an Alternative Timeline and Making Changes: `git checkout` . . . 1672
B.3.5. The Difference Between `checkout` and `switch` . . . 1673
B.4. Merging Branches and Resolving Conflicts . . . 1674
B.4.1. git `merge`: Merging Two Timelines . . . 1674
B.4.2. Resolving Conflicts When Merging . . . 1675
B.4.3. git `revert`: Undoing Something . . . 1677
B.5. Downloading from GitLab . . . 1679
B.6. Advanced . . . 1679
B.6.1. git `rebase`: Moving the Base of a Branch . . . 1679
B.7. Exercises . . . 1681
B.7.1. Create project folder . . . 1681
B.8. Initialize repo . . . 1681
B.8.1. Do not upload / ignore certain file types . . . 1682
B.8.2. Create file and stage it . . . 1682
B.8.3. Create another file and check status . . . 1682
B.8.4. Commit changes . . . 1682
B.8.5. Create a new branch and switch to it . . . 1682
B.8.6. Commit changes in the new branch . . . 1683
B.8.7. Merge branch into main . . . 1683
B.8.8. Resolve merge conflict . . . 1683

C. Python **1685**
C.1. Data Types and Precision in Python . . . 1685
C.2. Recommendations . . . 1686

D. Gaussian Processes—Some Background Information **1687**
D.1. Gaussian Process Prior . . . 1687
D.2. Covariance Function . . . 1687
D.2.1. Positive Definiteness . . . 1689
D.3. Construction of the Covariance Matrix . . . 1689
D.4. Generation of Random Samples and Plotting the Realizations of the Random Function . . . 1692
D.5. Properties of the 1d Example . . . 1694
D.5.1. Several Bumps: . . . 1694
D.5.2. Smoothness: . . . 1695
D.5.3. Scale of Two: . . . 1695
D.6. Jupyter Notebook . . . 1697

E. Datasets **1699**
E.1. The Diabetes Data Set . . . 1699
E.1.1. Data Exploration of the sklearn Diabetes Data Set . . . 1699
E.1.2. Generating the PyTorch Data Set . . . 1700
E.2. The Friedman Drift Dataset . . . 1701
E.2.1. The Friedman Drift Dataset as Implemented in `river` . . . 1701
E.2.2. The Friedman Drift Data Set from `spotpython` . . . 1704

F. Using Slurm **1707**
F.1. Introduction . . . 1707
F.2. Packages important for this Chapter . . . 1707
F.3. Prepare the Slurm Scripts for Runs on the Remote Machine . . . 1707
F.4. Generate a `spotpython` Configuration . . . 1709
F.5. Copy the Configuration to the Remote Machine . . . 1711
F.6. Run the `spotpython` Code on the Remote Machine . . . 1711
F.7. Copy the Results to the Local Machine . . . 1711
F.8. Analyze the Results on the Local Machine . . . 1711
F.8.1. Visualizing the Tuning Progress . . . 1712
F.8.2. Design Table with Default and Tuned Hyperparameters . . . 1712
F.8.3. Plotting Important Hyperparameters . . . 1712
F.8.4. The Tuned Hyperparameters . . . 1712
F.9. Slurm Command Reference . . . 1712
F.10. Jupyter Notebook . . . 1713

G. Python Package Building **1715**
Introduction . . . 1715
G.1. Create a Conda Environment . . . 1715
G.2. Download the User Package . . . 1715
G.3. Build the User Package . . . 1715
G.4. Open the Documentation of the User Package . . . 1716

H. Parallelism in Initial Design **1717**
H.1. Setup . . . 1717
H.2. Experiments . . . 1718
H.2.1. Sequential Execution . . . 1718
H.2.2. Parallel Execution . . . 1723
H.2.3. Results . . . 1740

I. Solutions to Selected Exercises **1743**
I.1. Data-Driven Modeling and Optimization . . . 1743
I.1.1. Histograms . . . 1743
I.1.2. The Normal Distribution . . . 1743
I.1.3. The mean, the media, and the mode . . . 1744
I.1.4. The exponential distribution . . . 1744
I.1.5. Population and Estimated Parameters . . . 1744
I.1.6. Calculating the Mean, Variance and Standard Deviation . . . 1744
I.1.7. Hypothesis Testing and the Null-Hypothesis . . . 1744
I.1.8. Alternative Hypotheses, Main Ideas . . . 1745
I.1.9. p-values: What they are and how to interpret them . . . 1745
I.1.10. How to calculate p-values . . . 1745
I.1.11. p-hacking: What it is and how to avoid it . . . 1746
I.1.12. Covariance . . . 1746
I.1.13. Pearson's Correlation . . . 1746
I.1.14. Boxplots . . . 1747
I.1.15. Power Analysis . . . 1747
I.1.16. The Central Limit Theorem . . . 1747
I.1.17. Boxplots . . . 1747
I.1.18. R-squared . . . 1748
I.1.19. Linear Regression . . . 1748
I.1.20. Multiple Regression . . . 1748
I.1.21. A Gentle Introduction to Machine Learning . . . 1748
I.1.22. Maximum Likelihood . . . 1748
I.1.23. Probability is not Likelihood . . . 1748
I.1.24. Cross Validation . . . 1749
I.1.25. The Confusion Matrix . . . 1749
I.1.26. Sensitivity and Specificity . . . 1749
I.1.27. Bias and Variance . . . 1749
I.1.28. Mutual Information . . . 1749
I.1.29. Principal Component Analysis (PCA) . . . 1750
I.1.30. t-SNE . . . 1750
I.1.31. K-means clustering . . . 1751
I.1.32. DBSCAN . . . 1751
I.1.33. K-nearest neighbors . . . 1752
I.1.34. Naive Bayes . . . 1752
I.1.35. Gaussian Naive Bayes . . . 1752
I.1.36. Trees . . . 1752

I.2. Machine Learning and Artificial Intelligence 1753
I.2.1. Backpropagation 1753
I.2.2. Gradient Descent 1753
I.2.3. ReLU 1753
I.2.4. CNNs 1753
I.2.5. RNN 1754
I.2.6. LSTM 1754
I.2.7. Pytorch/Lightning 1754
I.2.8. Embeddings 1754
I.2.9. Sequence to Sequence Models 1755
I.2.10. Transformers 1755

References **1757**

Preface

This document provides a comprehensive guide to hyperparameter tuning using spotpython for scikit-learn, scipy-optimize, River, and PyTorch. The first part introduces fundamental ideas from optimization. The second part discusses numerical issues and introduces spotpython's surrogate model-based optimization process. The thirs part focuses on hyperparameter tuning. Several case studies are presented, including hyperparameter tuning for sklearn models such as Support Vector Classification, Random Forests, Gradient Boosting (XGB), and K-nearest neighbors (KNN), as well as a Hoeffding Adaptive Tree Regressor from river. The integration of spotpython into the PyTorch and PyTorch Lightning training workflow is also discussed. With a hands-on approach and step-by-step explanations, this cookbook serves as a practical starting point for anyone interested in hyperparameter tuning with Python. Highlights include the interplay between Tensorboard, PyTorch Lightning, spotpython, spotriver, and River. This publication is under development, with updates available on the corresponding webpage.

! Important: This book is still under development.

The most recent version of this book is available at https://sequential-parameter-optimization.github.io/Hyperparameter-Tuning-Cookbook/

Book Structure

This document is structured in three parts. The first part presents an introduction to optimization. The second part describes numerical methods, and the third part presents hyperparameter tuning.

Hyperparameter Tuning Reference

- The open access book Bartz et al. (2022) provides a comprehensive overview of hyperparameter tuning. It can be downloaded from https://link.springer.com/book/10.1007/978-981-19-5170-1.

> **Note**
>
> The `.ipynb` notebook (Bartz-Beielstein 2023a) is updated regularly and reflects updates and changes in the `spotpython` package. It can be downloaded from https://github.com/sequential-parameter-optimization/spotpython/blob/main/notebooks/14_spot_ray_hpt_torch_cifar10.ipynb.

Software Used in this Book

scikit-learn is a Python module for machine learning built on top of SciPy and is distributed under the 3-Clause BSD license. The project was started in 2007 by David Cournapeau as a Google Summer of Code project, and since then many volunteers have contributed.

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs. Lightning is a lightweight PyTorch wrapper for high-performance AI research. It allows you to decouple the research from the engineering.

River is a Python library for online machine learning. It is designed to be used in real-world environments, where not all data is available at once, but streaming in.

spotpython ("Sequential Parameter Optimization Toolbox in Python") is the Python version of the well-known hyperparameter tuner SPOT, which has been developed in the R programming environment for statistical analysis for over a decade. The related open-access book is available here: Hyperparameter Tuning for Machine and Deep Learning with R—A Practical Guide.

spotriver provides an interface between spotpython and River.

Citation

If this document has been useful to you and you wish to cite it in a scientific publication, please refer to the following paper, which can be found on arXiv: https://arxiv.org/abs/2307.10262.

```
@ARTICLE{bart23iArXiv,
      author = {{Bartz-Beielstein}, Thomas},
      title = "{Hyperparameter Tuning Cookbook:
          A guide for scikit-learn, PyTorch, river, and spotpython}",
     journal = {arXiv e-prints},
    keywords = {Computer Science - Machine Learning,
      Computer Science - Artificial Intelligence, 90C26, I.2.6, G.1.6},
```

```
          year = 2023,
         month = jul,
           eid = {arXiv:2307.10262},
         pages = {arXiv:2307.10262},
           doi = {10.48550/arXiv.2307.10262},
archivePrefix = {arXiv},
        eprint = {2307.10262},
 primaryClass = {cs.LG},
        adsurl = {https://ui.adsabs.harvard.edu/abs/2023arXiv230710262B},
       adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

Part I.

Optimization

1. Aircraft Wing Weight Example

> **Note**
>
> - This section is based on chapter 1.3 "A ten-variable weight function" in Forrester, Sóbester, and Keane (2008).
> - The following Python packages are imported:

```
import math
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.axes_grid1 import ImageGrid
```

1.1. AWWE Equation

- Example from Forrester, Sóbester, and Keane (2008)
- Understand the **weight** of an unpainted light aircraft wing as a function of nine design and operational parameters:

$$W = 0.036 S_W^{0.758} \times W_{fw}^{0.0035} \left(\frac{A}{\cos^2 \Lambda}\right)^{0.6} \times q^{0.006} \times \lambda^{0.04}$$
$$\times \left(\frac{100 R_{tc}}{\cos \Lambda}\right)^{-0.3} \times (N_z W_{dg})^{0.49}$$

1.2. AWWE Parameters and Equations (Part 1)

Table 1.1.: Aircraft Wing Weight Parameters

Symbol	Parameter	Baseline	Minimum	Maximum
S_W	Wing area (ft^2)	174	150	200
W_{fw}	Weight of fuel in wing (lb)	252	220	300

Symbol	Parameter	Baseline	Minimum	Maximum
A	Aspect ratio	7.52	6	10
Λ	Quarter-chord sweep (deg)	0	-10	10
q	Dynamic pressure at cruise (lb/ft^2)	34	16	45
λ	Taper ratio	0.672	0.5	1
R_{tc}	Aerofoil thickness to chord ratio	0.12	0.08	0.18
N_z	Ultimate load factor	3.8	2.5	6
W_{dg}	Flight design gross weight (lb)	2000	1700	2500
W_p	paint weight (lb/ft^2)	0.064	0.025	0.08

The study begins with a baseline Cessna C172 Skyhawk Aircraft as its reference point. It aims to investigate the impact of wing area and fuel weight on the overall weight of the aircraft. Two crucial parameters in this analysis are the aspect ratio (A), defined as the ratio of the wing's length to the average chord (thickness of the airfoil), and the taper ratio (λ), which represents the ratio of the maximum to the minimum thickness of the airfoil or the maximum to minimum chord.

It's important to note that the equation used in this context is not a computer simulation but will be treated as one for the purpose of illustration. This approach involves employing a true mathematical equation, even if it's considered unknown, as a useful tool for generating realistic settings to test the methodology. The functional form of this equation was derived by "calibrating" known physical relationships to curves obtained from existing aircraft data, as referenced in Raymer (2006). Essentially, it acts as a surrogate for actual measurements of aircraft weight.

Examining the mathematical properties of the AWWE (Aircraft Weight With Wing Area and Fuel Weight Equation), it is evident that the response is highly nonlinear concerning its inputs. While it's common to apply the logarithm to simplify equations with complex exponents, even when modeling the logarithm, which transforms powers into slope coefficients and products into sums, the response remains nonlinear due to the presence of trigonometric terms. Given the combination of nonlinearity and high input dimension, simple linear and quadratic response surface approximations are likely to be inadequate for this analysis.

1.3. Goals: Understanding and Optimization

The primary goals of this study revolve around understanding and optimization:

1. **Understanding**: One of the straightforward objectives is to gain a deep understanding of the input-output relationships in this context. Given the global perspective implied by this setting, it becomes evident that a more sophisticated

model is almost necessary. At this stage, let's focus on this specific scenario to establish a clear understanding.

2. **Optimization**: Another application of this analysis could be optimization. There may be an interest in minimizing the weight of the aircraft, but it's likely that there will be constraints in place. For example, the presence of wings with a nonzero area is essential for the aircraft to be capable of flying. In situations involving (constrained) optimization, a global perspective and, consequently, the use of flexible modeling are vital.

The provided Python code serves as a genuine computer implementation that "solves" a mathematical model. It accepts arguments encoded in the unit cube, with defaults used to represent baseline settings, as detailed in the table labeled as Table 1.1. To map values from the interval $[a, b]$ to the interval $[0, 1]$, the following formula can be employed:

$$y = f(x) = \frac{x - a}{b - a}. \tag{1.1}$$

To reverse this mapping and obtain the original values, the formula

$$g(y) = a + (b - a)y \tag{1.2}$$

can be used. The function `wingwt()` expects inputs from the unit cube, which are then transformed back to their original scales using Equation 1.2. The function is defined as follows:

```
def wingwt(Sw=0.48, Wfw=0.4, A=0.38, L=0.5, q=0.62, l=0.344,  Rtc=0.4, Nz=0.37, Wdg=0.38):
    # put coded inputs back on natural scale
    Sw = Sw * (200 - 150) + 150
    Wfw = Wfw * (300 - 220) + 220
    A = A * (10 - 6) + 6
    L = (L * (10 - (-10)) - 10) * np.pi/180
    q = q * (45 - 16) + 16
    l = l * (1 - 0.5) + 0.5
    Rtc = Rtc * (0.18 - 0.08) + 0.08
    Nz = Nz * (6 - 2.5) + 2.5
    Wdg = Wdg*(2500 - 1700) + 1700
    # calculation on natural scale
    W = 0.036 * Sw**0.758 * Wfw**0.0035 * (A/np.cos(L)**2)**0.6 * q**0.006
    W = W * l**0.04 * (100*Rtc/np.cos(L))**(-0.3) * (Nz*Wdg)**(0.49)
    return(W)
```

1.4. Properties of the Python "Solver"

The compute time required by the "wingwt" solver is extremely short and can be considered trivial in terms of computational resources. The approximation error is exceptionally small, effectively approaching machine precision, which indicates the high accuracy of the solver's results.

To simulate time-consuming evaluations, a deliberate delay is introduced by incorporating a `sleep(3600)` command, which effectively synthesizes a one-hour execution time for a particular evaluation.

Moving on to the AWWE visualization, plotting in two dimensions is considerably simpler than dealing with nine dimensions. To aid in creating visual representations, the code provided below establishes a grid within the unit square to facilitate the generation of sliced visuals. This involves generating a "meshgrid" as outlined in the code.

```
x = np.linspace(0, 1, 3)
y = np.linspace(0, 1, 3)
X, Y = np.meshgrid(x, y)
zp = zip(np.ravel(X), np.ravel(Y))
list(zp)
```

```
[(np.float64(0.0), np.float64(0.0)),
 (np.float64(0.5), np.float64(0.0)),
 (np.float64(1.0), np.float64(0.0)),
 (np.float64(0.0), np.float64(0.5)),
 (np.float64(0.5), np.float64(0.5)),
 (np.float64(1.0), np.float64(0.5)),
 (np.float64(0.0), np.float64(1.0)),
 (np.float64(0.5), np.float64(1.0)),
 (np.float64(1.0), np.float64(1.0))]
```

The coding used to transform inputs from natural units is largely a matter of taste, so long as it's easy to undo for reporting back on original scales

```
%matplotlib inline
# plt.style.use('seaborn-white')
x = np.linspace(0, 1, 100)
y = np.linspace(0, 1, 100)
X, Y = np.meshgrid(x, y)
```

1.5. Plot 1: Load Factor (N_z) and Aspect Ratio (A)

We will vary N_z and A, with other inputs fixed at their baseline values.

```
z = wingwt(A = X, Nz = Y)
fig = plt.figure(figsize=(7., 5.))
plt.contourf(X, Y, z, 20, cmap='jet')
plt.xlabel("A")
plt.ylabel("Nz")
plt.title("Load factor (Nz) vs. Aspect Ratio (A)")
plt.colorbar()
plt.show()
```

Contour plots can be refined, e.g., by adding explicit contour lines as shown in the following figure.

```
contours = plt.contour(X, Y, z, 4, colors='black')
plt.clabel(contours, inline=True, fontsize=8)
```

```
plt.xlabel("A")
plt.ylabel("Nz")

plt.imshow(z, extent=[0, 1, 0, 1], origin='lower',
           cmap='jet', alpha=0.9)
plt.colorbar()
```

The interpretation of the AWWE plot can be summarized as follows:

- The figure displays the weight response as a function of two variables, N_z and A, using an image-contour plot.
- The slight curvature observed in the contours suggests an interaction between these two variables.
- Notably, the range of outputs depicted in the figure, spanning from approximately 160 to 320, nearly encompasses the entire range of outputs observed from various input settings within the full 9-dimensional input space.
- The plot indicates that aircraft wings tend to be heavier when the aspect ratios (A) are high.
- This observation aligns with the idea that wings are designed to withstand and accommodate high gravitational forces (g-forces, large N_z), and there may be a compounding effect where larger values of N_z contribute to increased wing weight.

- It's plausible that this phenomenon is related to the design considerations of fighter jets, which cannot have the efficient and lightweight glider-like wings typically found in other types of aircraft.

1.6. Plot 2: Taper Ratio and Fuel Weight

- The same experiment for two other inputs, e.g., taper ratio λ and fuel weight W_{fw}

```
z = wingwt(Wfw = X,  Nz = Y)
contours = plt.contour(X, Y, z, 4, colors='black')
plt.clabel(contours, inline=True, fontsize=8)
plt.xlabel("WfW")
plt.ylabel("l")

plt.imshow(z, extent=[0, 1, 0, 1], origin='lower',
           cmap='jet', alpha=0.9)
plt.colorbar();
```

- Interpretation of Taper Ratio (l) and Fuel Weight (W_{fw})
  - Apparently, neither input has much effect on wing weight:

    * with λ having a marginally greater effect, covering less than 4 percent of the span of weights observed in the $A \times N_z$ plane
  - There's no interaction evident in $\lambda \times W_{fw}$

1.7. The Big Picture: Combining all Variables

```
pl = ["Sw", "Wfw", "A", "L", "q", "l",  "Rtc", "Nz", "Wdg"]
```

```
Z = []
Zlab = []
l = len(pl)
# lc = math.comb(l,2)
for i in range(l):
    for j in range(i+1, l):
    # for j in range(l):
        # print(pl[i], pl[j])
        d = {pl[i]: X, pl[j]: Y}
        Z.append(wingwt(**d))
        Zlab.append([pl[i],pl[j]])
```

Now we can generate all 36 combinations, e.g., our first example is combination `p = 19`.

```
p = 19
Zlab[p]
```

```
['A', 'Nz']
```

To help interpret outputs from experiments such as this one—to level the playing field when comparing outputs from other pairs of inputs—code below sets up a color palette that can be re-used from one experiment to the next. We use the arguments `vmin=180` and `vmax =360` to implement comparibility

```
plt.contourf(X, Y, Z[p], 20, cmap='jet', vmin=180, vmax=360)
plt.xlabel(Zlab[p][0])
plt.ylabel(Zlab[p][1])
plt.colorbar()
```

- Let's plot the second example, taper ratio λ and fuel weight W_{fw}
- This is combination `11`:

```
p = 11
Zlab[p]
```

```
['Wfw', 'l']
```

```
plt.contourf(X, Y, Z[p], 20, cmap='jet', vmin=180, vmax=360)
plt.xlabel(Zlab[p][0])
plt.ylabel(Zlab[p][1])
plt.colorbar()
```

- Using a global colormap indicates that these variables have minor effects on the wing weight.
- Important factors can be detected by visual inspection
- Plotting the Big Picture: we can plot all 36 combinations in one figure.

```
fig = plt.figure(figsize=(20., 20.))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
                 nrows_ncols=(6,6),  # creates 2x2 grid of axes
                 axes_pad=0.5,  # pad between axes in inch.
                 share_all=True,
                 label_mode="all",
                 )
i = 0
for ax, im in zip(grid, Z):
    # Iterating over the grid returns the Axes.
    ax.set_xlabel(Zlab[i][0])
    ax.set_ylabel(Zlab[i][1])
    # ax.set_title(Zlab[i][1] + " vs. " + Zlab[i][0])
    ax.contourf(X, Y, im, 30, cmap = "jet",  vmin = 180, vmax = 360)
    i = i + 1

plt.show()
```

1.8. AWWE Landscape

- Our Observations

  1. The load factor N_z, which determines the magnitude of the maximum aerodynamic load on the wing, is very active and involved in interactions with other variables.

  – Classic example: the interaction of N_z with the aspect ratio A indicates a heavy wing for high aspect ratios and large g-forces
  – This is the reaon why highly manoeuvrable fighter jets cannot have very efficient, glider wings)

2. Aspect ratio A and airfoil thickness to chord ratio R_{tc} have nonlinear interactions.
3. Most important variables:
   - Ultimate load factor N_z, wing area S_w, and flight design gross weightW_{dg}.
4. Little impact: dynamic pressure q, taper ratio l, and quarter-chord sweep L.

- Expert Knowledge
  - Aircraft designers know that the overall weight of the aircraft and the wing area must be kept to a minimum
  - the latter usually dictated by constraints such as required stall speed, landing distance, turn rate, etc.

1.9. Summary of the First Experiments

- First, we considered two pairs of inputs, out of 36 total pairs
- Then, the "Big Picture":
  - For each pair we evaluated `wingwt` 10,000 times
- Doing the same for all pairs would require 360K evaluations:
  - not a reasonable number with a real computer simulation that takes any non-trivial amount of time to evaluate
  - Only 1s per evaluation: > 100 hours
- Many solvers take minutes/hours/days to execute a single run
- And: three-way interactions?
- Consequence: a different strategy is needed

1.10. Exercise

1.10.1. Adding Paint Weight

- Paint weight is not considered.
- Add Paint Weight W_p to formula (the updated formula is shown below) and update the functions and plots in the notebook.

$$W = 0.036 S_W^{0.758} \times W_{fw}^{0.0035} \times \left(\frac{A}{\cos^2 \Lambda}\right)^{0.6} \times q^{0.006} \times \lambda^{0.04}$$
$$\times \left(\frac{100 R_{tc}}{\cos \Lambda}\right)^{-0.3} \times (N_z W_{dg})^{0.49} + S_w W_p$$

1.11. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

2. Introduction to `scipy.optimize`

SciPy provides algorithms for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, statistics and many other classes of problems. SciPy is a collection of mathematical algorithms and convenience functions built on NumPy. It adds significant power to Python by providing the user with high-level commands and classes for manipulating and visualizing data.

SciPy optimize provides functions for minimizing (or maximizing) objective functions, possibly subject to constraints. It includes solvers for nonlinear problems (with support for both local and global optimization algorithms), linear programing, constrained and nonlinear least-squares, root finding, and curve fitting.

In this notebook, we will learn how to use the `scipy.optimize` module to solve optimization problems. See: https://docs.scipy.org/doc/scipy/tutorial/optimize.html

> **Note**
>
> - This content is based on information from the scipy.optimize package.
> - The `scipy.optimize` package provides several commonly used optimization algorithms. A detailed listing is available in `scipy.optimize` (can also be found by `help(scipy.optimize)`).

Common functions and objects, shared across different SciPy optimize solvers, are shown in Table 2.1.

Table 2.1.: Common functions and objects, shared across different SciPy optimize solvers

Function or Object	Description
show_options([solver, method, disp])	Show documentation for additional options of optimization solvers.
OptimizeResult	Represents the optimization result.
OptimizeWarning	Warning issued by solvers.

We will introduce unconstrained minimization of multivariate scalar functions in this chapter. The minimize function provides a common interface to unconstrained and constrained minimization algorithms for multivariate scalar functions in `scipy.optimize`.

To demonstrate the minimization function, consider the problem of minimizing the Rosenbrock function of N variables:

$$f(J) = \sum_{i=1}^{N-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$$

The minimum value of this function is 0, which is achieved when (x_i = 1).

Note that the Rosenbrock function and its derivatives are included in `scipy.optimize`. The implementations shown in the following sections provide examples of how to define an objective function as well as its Jacobian and Hessian functions. Objective functions in `scipy.optimize` expect a numpy array as their first parameter, which is to be optimized and must return a float value. The exact calling signature must be `f(x, *args)`, where `x` represents a numpy array, and `args` is a tuple of additional arguments supplied to the objective function.

2.1. Derivative-free Optimization Algorithms

Section 2.1.1 and Section 2.1.2 present two approaches that do not need gradient information to find the minimum. They use function evaluations to find the minimum.

2.1.1. Nelder-Mead Simplex Algorithm

The Nelder Mead is a simple local optimization algorithm. It requires only function evaluations and is a good choice for simple minimization problems. However, because it does not use any gradient evaluations, it may take longer to find the minimum. It can be devided into the following steps:

1. Initialize the simplex
2. Evaluate the function at each vertex of the simplex
3. Order the vertices by function value
4. **Reflect** the worst point through the centroid of the remaining points
5. If the reflected point is better than the second worst, replace the worst point with the reflected point
6. If the reflected point is worse than the worst point, try **contracting** the simplex
7. If the reflected point is better than the best point, try **expanding** the simplex
8. If none of the above steps improve the simplex, **shrink** the simplex towards the best point
9. Check for convergence

`method='Nelder-Mead'`: In the example below, the `minimize` routine is used with the *Nelder-Mead* simplex algorithm (selected through the `method` parameter):

```python
import numpy as np
from scipy.optimize import minimize

def rosen(x):
    """The Rosenbrock function"""
    return sum(100.0 * (x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0)

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen, x0, method='nelder-mead',
               options={'xatol': 1e-8, 'disp': True})

print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 339
         Function evaluations: 571
[1. 1. 1. 1. 1.]
```

The simplex algorithm is probably the simplest way to minimize a well-behaved function. It requires only function evaluations and is a good choice for simple minimization problems. However, because it does not use any gradient evaluations, it may take longer to find the minimum.

2.1.2. Powell's Method

Another optimization algorithm that needs only function calls to find the minimum is *Powell*'s method, which can be selected by setting the `method` parameter to `'powell'` in the `minimize` function. This algorithm consists of a conjugate direction method. It performs sequential one-dimensional minimizations along each vector of the directions set, which is updated at each iteration of the main minimization loop. It can be described by the following steps:

1. Initialization
2. Minimization along each direction
3. Create conjugate direction
4. Line search along the conjugate direction
5. Check for convergence

Example 2.1. To demonstrate how to supply additional arguments to an objective function, let's consider minimizing the Rosenbrock function with an additional scaling factor a and an offset b:

$$f(J, a, b) = \sum_{i=1}^{N-1} a(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 + b$$

You can achieve this using the `minimize` routine with the example parameters $a = 0.5$ and $b = 1$:

```
def rosen_with_args(x, a, b):
    """The Rosenbrock function with additional arguments"""
    return sum(a * (x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0) + b

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen_with_args, x0, method='nelder-mead',
               args=(0.5, 1.), options={'xatol': 1e-8, 'disp': True})

print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 1.000000
         Iterations: 319
         Function evaluations: 525
[1.         1.         1.         1.         0.99999999]
```

As an alternative to using the `args` parameter of `minimize`, you can wrap the objective function in a new function that accepts only `x`. This approach is also useful when it is necessary to pass additional parameters to the objective function as keyword arguments.

```
def rosen_with_args(x, a, *, b):  # b is a keyword-only argument
    return sum(a * (x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0) + b

def wrapped_rosen_without_args(x):
    return rosen_with_args(x, 0.5, b=1.)  # pass in `a` and `b`

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(wrapped_rosen_without_args, x0, method='nelder-mead',
               options={'xatol': 1e-8,})

print(res.x)
```

```
[1.         1.         1.         1.         0.99999999]
```

Another alternative is to use `functools.partial`.

```
from functools import partial

partial_rosen = partial(rosen_with_args, a=0.5, b=1.)
res = minimize(partial_rosen, x0, method='nelder-mead',
               options={'xatol': 1e-8,})

print(res.x)
```

```
[1.         1.         1.         1.         0.99999999]
```

2.2. Gradient-based Optimization Algorithms

2.2.1. An Introductory Example: Broyden-Fletcher-Goldfarb-Shanno Algorithm (BFGS)

This section introduces an optimization algorithm that uses gradient information to find the minimum. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm (selected by setting `method='BFGS'`) is an optimization algorithm that aims to converge quickly to the solution. This algorithm uses the gradient of the objective function. If the gradient is not provided by the user, it is estimated using first-differences. The BFGS method typically requires fewer function calls compared to the simplex algorithm, even when the gradient needs to be estimated.

Example 2.2 (BFGS)**.** To demonstrate the BFGS algorithm, let's use the Rosenbrock function again. The gradient of the Rosenbrock function is a vector described by the following mathematical expression:

$$\frac{\partial f}{\partial x_j} = \sum_{i=1}^{N} 200(x_i - x_{i-1}^2)(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}) - 2(1 - x_{i-1})\delta_{i-1,j} \tag{2.1}$$

$$= 200(x_j - x_{j-1}^2) - 400x_j(x_{j+1} - x_j^2) - 2(1 - x_j) \tag{2.2}$$

This expression is valid for interior derivatives, but special cases are:

$$\frac{\partial f}{\partial x_0} = -400x_0(x_1 - x_0^2) - 2(1 - x_0)$$

$$\frac{\partial f}{\partial x_{N-1}} = 200(x_{N-1} - x_{N-2}^2)$$

Here's a Python function that computes this gradient:

```
def rosen_der(x):
    xm = x[1:-1]
    xm_m1 = x[:-2]
    xm_p1 = x[2:]
    der = np.zeros_like(x)
    der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
    der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
    der[-1] = 200*(x[-1]-x[-2]**2)
    return der
```

You can specify this gradient information in the minimize function using the jac parameter as illustrated below:

```
res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
               options={'disp': True})

print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 25
         Function evaluations: 30
         Gradient evaluations: 30
[1.00000004 1.0000001  1.00000021 1.00000044 1.00000092]
```

2.2.2. Background and Basics for Gradient-based Optimization

2.2.3. Gradient

The gradient $\nabla f(J)$ for a scalar function $f(J)$ with n different variables is defined by its partial derivatives:

$$\nabla f(J) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n}\right]$$

2.2.4. Jacobian Matrix

The Jacobian matrix $J(J)$ for a vector-valued function $F(J) = [f_1(J), f_2(J), \dots, f_m(J)]$ is defined as:

$$J(J) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

It consists of the first order partial derivatives and gives therefore an overview about the gradients of a vector valued function.

Example 2.3 (acobian matrix)**.** Consider a vector-valued function $f : \mathbb{R}^2 \to \mathbb{R}^3$ defined as follows:

$$f(J) = \begin{bmatrix} x_1^2 + 2x_2 \\ 3x_1 - \sin(x_2) \\ e^{x_1+x_2} \end{bmatrix}$$

Let's compute the partial derivatives and construct the Jacobian matrix:

$\frac{\partial f_1}{\partial x_1} = 2x_1, \quad \frac{\partial f_1}{\partial x_2} = 2$

$\frac{\partial f_2}{\partial x_1} = 3, \quad \frac{\partial f_2}{\partial x_2} = -\cos(x_2)$

$\frac{\partial f_3}{\partial x_1} = e^{x_1+x_2}, \quad \frac{\partial f_3}{\partial x_2} = e^{x_1+x_2}$

So, the Jacobian matrix is:

$$J(J) = \begin{bmatrix} 2x_1 & 2 \\ 3 & -\cos(x_2) \\ e^{x_1+x_2} & e^{x_1+x_2} \end{bmatrix}$$

This Jacobian matrix provides information about how small changes in the input variables x_1 and x_2 affect the corresponding changes in each component of the output vector.

2.2.5. Hessian Matrix

The Hessian matrix $H(J)$ for a scalar function $f(J)$ is defined as:

$$H(J) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

The Hessian matrix consists of the second order derivatives of the function. It provides information about the local curvature of the function with respect to changes in the input variables.

Example 2.4 (Hessian matrix)**.** Consider a scalar-valued function:

$$f(J) = x_1^2 + 2x_2^2 + \sin(x_1 x_2)$$

The Hessian matrix of this scalar-valued function is the matrix of its second-order partial derivatives with respect to the input variables:

$$H(J) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix}$$

Let's compute the second-order partial derivatives and construct the Hessian matrix:

$$\frac{\partial^2 f}{\partial x_1^2} = 2 + \cos(x_1 x_2) x_2^2 \tag{2.3}$$

$$\frac{\partial^2 f}{\partial x_1 \partial x_2} = 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) \tag{2.4}$$

$$\frac{\partial^2 f}{\partial x_2 \partial x_1} = 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) \tag{2.5}$$

$$\frac{\partial^2 f}{\partial x_2^2} = 4x_2^2 + \cos(x_1 x_2) x_1^2 \tag{2.6}$$

So, the Hessian matrix is:

$$H(J) = \begin{bmatrix} 2 + \cos(x_1 x_2) x_2^2 & 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) \\ 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) & 4x_2^2 + \cos(x_1 x_2) x_1^2 \end{bmatrix}$$

2.2.6. Gradient Descent

In optimization, the goal is to find the minimum or maximum of a function. Gradient-based optimization methods utilize information about the gradient (or derivative) of the function to guide the search for the optimal solution. This is particularly useful when dealing with complex, high-dimensional functions where an exhaustive search is impractical.

The gradient descent method can be divided in the following steps:

- **Initialize:** start with an initial guess for the parameters of the function to be optimized.
- **Compute Gradient:** Calculate the gradient (partial derivatives) of the function with respect to each parameter at the current point. The gradient indicates the direction of the steepest increase in the function.

- **Update Parameters:** Adjust the parameters in the opposite direction of the gradient, scaled by a learning rate. This step aims to move towards the minimum of the function:
  - $x_{k+1} = x_k - \alpha \times \nabla f(x_k)$
  - x_x is current parameter vector or point in the parameter space.
  - α is the learning rate, a positive scalar that determines the step size in each iteration.
  - $\nabla f(x)$ is the gradient of the objective function.
- **Iterate:** Repeat the above steps until convergence or a predefined number of iterations. Convergence is typically determined when the change in the function value or parameters becomes negligible.

Example 2.5 (Gradient Descent)**.** We consider a simple quadratic function as an example:

$$f(x) = x^2 + 4x + y^2 + 2y + 4.$$

We'll use gradient descent to find the minimum of this function.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define the quadratic function
def quadratic_function(x, y):
    return x**2 + 4*x + y**2 + 2*y + 4

# Define the gradient of the quadratic function
def gradient_quadratic_function(x, y):
    grad_x = 2*x + 4
    grad_y = 2*y + 2
    return np.array([grad_x, grad_y])

# Gradient Descent for optimization in 2D
def gradient_descent(initial_point, learning_rate, num_iterations):
    points = [np.array(initial_point)]
    for _ in range(num_iterations):
        current_point = points[-1]
        gradient = gradient_quadratic_function(*current_point)
        new_point = current_point - learning_rate * gradient
        points.append(new_point)
    return points

# Visualization of optimization process with 3D surface and consistent arrow sizes
```

```
def plot_optimization_process_3d_consistent_arrows(points):
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    x_vals = np.linspace(-10, 2, 100)
    y_vals = np.linspace(-10, 2, 100)
    X, Y = np.meshgrid(x_vals, y_vals)
    Z = quadratic_function(X, Y)

    ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.6)
    ax.scatter(*zip(*points), [quadratic_function(*p) for p in points], c='red', label

    for i in range(len(points) - 1):
        x, y = points[i]
        dx, dy = points[i + 1] - points[i]
        dz = quadratic_function(*(points[i + 1])) - quadratic_function(*points[i])
        gradient_length = 0.5

        ax.quiver(x, y, quadratic_function(*points[i]), dx, dy, dz, color='blue', leng

    ax.set_title('Gradient-Based Optimization with 2D Quadratic Function')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('f(x, y)')
    ax.legend()
    plt.show()

# Initial guess and parameters
initial_guess = [-9.0, -9.0]
learning_rate = 0.2
num_iterations = 10

# Run gradient descent in 2D and visualize the optimization process with 3D surface ar
trajectory = gradient_descent(initial_guess, learning_rate, num_iterations)
plot_optimization_process_3d_consistent_arrows(trajectory)
```

2.2.7. Newton Method

Initialization: Start with an initial guess for the optimal solution: x_0.

Iteration: Repeat the following three steps until convergence or a predefined stopping criterion is met:

1. Calculate the gradient (∇) and the Hessian matrix (∇^2) of the objective function at the current point:

$$\nabla f(x_k) \quad \text{and} \quad \nabla^2 f(x_k)$$

2. Update the current solution using the Newton-Raphson update formula

$$x_{k+1} = x_k - [\nabla^2 f(x_k)]^{-1} \nabla f(x_k),$$

where

```
* $\nabla f(x_k)$ is the gradient (first derivative) of the objective function wi
```

- $\nabla^2 f(x_k)$: The Hessian matrix (second derivative) of the objective function with respect to x, evaluated at the current solution x_k.
- x_k: The current solution or point in the optimization process.
- $\nabla^2 f(x_k)]^{-1}$: The inverse of the Hessian matrix at the current point, representing the approximation of the curvature of the objective function.
- x_{k+1}: The updated solution or point after applying the Newton-Raphson update.

3. Check for convergence.

Example 2.6 (Newton Method)**.** We want to optimize the Rosenbrock function and use the Hessian and the Jacobian (which is equal to the gradient vector for scalar objective function) to the `minimize` function.

```
def rosenbrock(x):
    return 100 * (x[1] - x[0]**2)**2 + (1 - x[0])**2

def rosenbrock_gradient(x):
    dfdx0 = -400 * x[0] * (x[1] - x[0]**2) - 2 * (1 - x[0])
    dfdx1 = 200 * (x[1] - x[0]**2)
    return np.array([dfdx0, dfdx1])

def rosenbrock_hessian(x):
    d2fdx0 = 1200 * x[0]**2 - 400 * x[1] + 2
    d2fdx1 = -400 * x[0]
    return np.array([[d2fdx0, d2fdx1], [d2fdx1, 200]])

def classical_newton_optimization_2d(initial_guess, tol=1e-6, max_iter=100):
    x = initial_guess.copy()

    for i in range(max_iter):
        gradient = rosenbrock_gradient(x)
        hessian = rosenbrock_hessian(x)

        # Solve the linear system H * d = -g for d
        d = np.linalg.solve(hessian, -gradient)

        # Update x
```

```
        x += d

        # Check for convergence
        if np.linalg.norm(gradient, ord=np.inf) < tol:
            break

    return x

# Initial guess
initial_guess_2d = np.array([0.0, 0.0])

# Run classical Newton optimization for the 2D Rosenbrock function
result_2d = classical_newton_optimization_2d(initial_guess_2d)

# Print the result
print("Optimal solution:", result_2d)
print("Objective value:", rosenbrock(result_2d))
```

```
Optimal solution: [1. 1.]
Objective value: 0.0
```

2.2.8. BFGS-Algorithm

BFGS is an optimization algorithm designed for unconstrained optimization problems. It belongs to the class of quasi-Newton methods and is known for its efficiency in finding the minimum of a smooth, unconstrained objective function.

2.2.9. Procedure:

1. **Initialization:**
   - Start with an initial guess for the parameters of the objective function.
   - Initialize an approximation of the Hessian matrix (inverse) denoted by H.

2. **Iterative Update:**
   - At each iteration, compute the gradient vector at the current point.
   - Update the parameters using the BFGS update formula, which involves the inverse Hessian matrix approximation, the gradient, and the difference in parameter vectors between successive iterations:

$$x_{k+1} = x_k - H_k^{-1} \nabla f(x_k).$$

- Update the inverse Hessian approximation using the BFGS update formula for the inverse Hessian.

$$H_{k+1} = H_k + \frac{\Delta x_k \Delta x_k^T}{\Delta x_k^T \Delta g_k} - \frac{H_k g_k g_k^T H_k}{g_k^T H_k g_k},$$

  where:
- x_k and x_{k+1} are the parameter vectors at the current and updated iterations, respectively.
- $\nabla f(x_k)$ is the gradient vector at the current iteration.
- $\Delta x_k = x_{k+1} - x_k$ is the change in parameter vectors.
- $\Delta g_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ is the change in gradient vectors.

3. **Convergence:**

   - Repeat the iterative update until the optimization converges. Convergence is typically determined by reaching a sufficiently low gradient or parameter change.

Example 2.7 (BFGS for Rosenbrock)**.**

```
import numpy as np
from scipy.optimize import minimize

# Define the 2D Rosenbrock function
def rosenbrock(x):
    return (1 - x[0])**2 + 100 * (x[1] - x[0]**2)**2

# Initial guess
initial_guess = np.array([0.0, 0.0])

# Minimize the Rosenbrock function using BFGS
minimize(rosenbrock, initial_guess, method='BFGS')
```

```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 2.8440052847381483e-11
        x: [ 1.000e+00  1.000e+00]
      nit: 19
      jac: [ 3.987e-06 -2.844e-06]
 hess_inv: [[ 4.948e-01  9.896e-01]
            [ 9.896e-01  1.984e+00]]
     nfev: 72
     njev: 24
```

2.2.10. Visualization BFGS for Rosenbrock

A visualization of the BFGS search process on Rosenbrock's function can be found here: https://upload.wikimedia.org/wikipedia/de/f/ff/Rosenbrock-bfgs-animation.gif

2.3. Global Optimization

Global optimization aims to find the global minimum of a function within given bounds, in the presence of potentially many local minima. Typically, global minimizers efficiently search the parameter space, while using a local minimizer (e.g., minimize) under the hood.

2.3.1. Local vs Global Optimization

2.3.1.1. Local Optimizater:

- Seeks the optimum in a **specific region** of the search space
- Tends to **exploit** the local environment, to find solutions in the immediate area
- Highly **sensitive to initial conditions**; may converge to different local optima based on the starting point
- Often **computationally efficient for low-dimensional problems** but may struggle with high-dimensional or complex search spaces
- Commonly used in situations where the objective is to refine and improve existing solutions

2.3.1.2. Global Optimizer:

- Explores the **entire search space** to find the global optimum
- Emphasize **exploration over exploitation**, aiming to search broadly and avoid premature convergence to local optima
- Aim to **mitigate the risk of premature convergence** to local optima by employing strategies for global exploration
- **Less sensitive to initial conditions**, designed to navigate diverse regions of the search space
- Equipped to handle **high-dimensional** and **complex** problems, though computational demands may vary depending on the specific algorithm
- Preferred for applications where a comprehensive search of the solution space is crucial, such as in parameter tuning, machine learning, and complex engineering design

Example 2.8 (Global Optimizers in SciPy)**.** SciPy contains a number of good global optimizers. Here, we'll use those on the same objective function, namely the (aptly named) eggholder function:

```
def eggholder(x):
    return (-(x[1] + 47) * np.sin(np.sqrt(abs(x[0]/2 + (x[1]  + 47))))
            -x[0] * np.sin(np.sqrt(abs(x[0] - (x[1]  + 47)))))

bounds = [(-512, 512), (-512, 512)]
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

x = np.arange(-512, 513)
y = np.arange(-512, 513)
xgrid, ygrid = np.meshgrid(x, y)
xy = np.stack([xgrid, ygrid])

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.view_init(45, -45)
ax.plot_surface(xgrid, ygrid, eggholder(xy), cmap='terrain')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('eggholder(x, y)')
plt.show()
```

We now use the global optimizers to obtain the minimum and the function value at the minimum. We'll store the results in a dictionary so we can compare different optimization results later.

```
from scipy import optimize
results = dict()
results['shgo'] = optimize.shgo(eggholder, bounds)
results['shgo']
```

```
 message: Optimization terminated successfully.
 success: True
     fun: -935.3379515605778
    funl: [-9.353e+02]
       x: [ 4.395e+02  4.540e+02]
      xl: [[ 4.395e+02  4.540e+02]]
     nit: 1
    nfev: 47
   nlfev: 42
   nljev: 12
   nlhev: 0
```

```
results['DA'] = optimize.dual_annealing(eggholder, bounds)
results['DA']
```

```
 message: ['Maximum number of iteration reached']
```

```
 success: True
  status: 0
     fun: -935.3379515600866
       x: [ 4.395e+02  4.540e+02]
     nit: 1000
    nfev: 4055
    njev: 18
    nhev: 0
```

All optimizers return an `OptimizeResult`, which in addition to the solution contains information on the number of function evaluations, whether the optimization was successful, and more. For brevity, we won't show the full output of the other optimizers:

```
results['DE'] = optimize.differential_evolution(eggholder, bounds)
results['DE']
```

```
             message: Optimization terminated successfully.
             success: True
                 fun: -956.9182316158974
                   x: [ 4.824e+02  4.329e+02]
                 nit: 34
                nfev: 1086
          population: [[ 4.822e+02  4.327e+02]
                       [ 4.863e+02  4.373e+02]
                       ...
                       [ 4.788e+02  4.293e+02]
                       [ 4.824e+02  4.329e+02]]
 population_energies: [-9.569e+02 -9.468e+02 ... -9.545e+02 -9.569e+02]
                 jac: [ 2.956e-04 -1.933e-04]
```

`shgo` has a second method, which returns all local minima rather than only what it thinks is the global minimum:

```
results['shgo_sobol'] = optimize.shgo(eggholder, bounds, n=200, iters=5,
                                      sampling_method='sobol')
results['shgo_sobol']
```

```
 message: Optimization terminated successfully.
 success: True
     fun: -959.640662720848
    funl: [-9.596e+02 -9.353e+02 ... -6.591e+01 -6.387e+01]
       x: [ 5.120e+02  4.042e+02]
      xl: [[ 5.120e+02  4.042e+02]
```

```
           [ 4.395e+02  4.540e+02]
           ...
           [ 3.165e+01 -8.523e+01]
           [ 5.865e+01 -5.441e+01]]
     nit: 5
    nfev: 3482
   nlfev: 2280
   nljev: 626
   nlhev: 0
```

We'll now plot all found minima on a heatmap of the function:

```
fig = plt.figure()
ax = fig.add_subplot(111)
im = ax.imshow(eggholder(xy), interpolation='bilinear', origin='lower',
               cmap='gray')
ax.set_xlabel('x')
ax.set_ylabel('y')

def plot_point(res, marker='o', color=None):
    ax.plot(512+res.x[0], 512+res.x[1], marker=marker, color=color, ms=10)

plot_point(results['DE'], color='c')  # differential_evolution - cyan
plot_point(results['DA'], color='w')  # dual_annealing.        - white

# SHGO produces multiple minima, plot them all (with a smaller marker size)
plot_point(results['shgo'], color='r', marker='+')
plot_point(results['shgo_sobol'], color='r', marker='x')
for i in range(results['shgo_sobol'].xl.shape[0]):
    ax.plot(512 + results['shgo_sobol'].xl[i, 0],
            512 + results['shgo_sobol'].xl[i, 1],
            'ro', ms=2)

ax.set_xlim([-4, 514*2])
ax.set_ylim([-4, 514*2])
plt.show()
```

2.3.2. Dual Annealing Optimization

This function implements the Dual-Annealing optimization, which is a variant of the famous simulated annealing optimization.

Simulated Annealing is a **probabilistic** optimization algorithm inspired by the annealing process in metallurgy. The algorithm is designed to find a good or optimal **global** solution to a problem by exploring the solution space in a controlled and adaptive manner.

> **ℹ Annealing in Metallurgy**
>
> Simulated Annealing draws inspiration from the physical process of annealing in metallurgy. Just as metals are gradually cooled to achieve a more stable state, Simulated Annealing uses a similar approach to explore solution spaces in the digital world.

Heating Phase: In metallurgy, a metal is initially heated to a high temperature. At this elevated temperature, the atoms or molecules in the material become more energetic and chaotic, allowing the material to overcome energy barriers and defects.

Analogy Simulated Annealing (Exploration Phase): In Simulated Annealing, the algorithm starts with a high "temperature," which encourages exploration of the

solution space. At this stage, the algorithm is more likely to accept solutions that are worse than the current one, allowing it to escape local optima and explore a broader region of the solution space.

Cooling Phase: The material is then gradually cooled at a controlled rate. As the temperature decreases, the atoms or molecules start to settle into more ordered and stable arrangements. The slow cooling rate is crucial to avoid the formation of defects and to ensure the material reaches a well-organized state.

Analogy Simulated Annealing (Exploitation Phase): As the algorithm progresses, the temperature is gradually reduced over time according to a cooling schedule. This reduction simulates the cooling process in metallurgy. With lower temperatures, the algorithm becomes more selective and tends to accept only better solutions, focusing on refining and exploiting the promising regions discovered during the exploration phase.

2.3.2.1. Key Concepts

Temperature: The temperature is a parameter that controls the likelihood of accepting worse solutions. We start with a high temperature, allowing the algorithm to explore the solution space braodly. The temperature decreases with the iterations of the algorithm.

Cooling Schedule: The temperature parameter is reduced according to this schedule. The analogy to the annealing of metals: a slower cooling rate allows the material to reach a more stable state.

Neighborhood Exploration: At each iteration, the algorithm explores the neighborhood of the current solution. The neighborhood is defined by small perturbations or changes to the current solution.

Acceptance Probability: The algorithm evaluates the objective function for the new solution in the neighborhood. If the new solution is better, it is accepted. If the new solution is worse, it may still be accepted with a certain probability. This probability is determined by both the difference in objective function values and the current temperature.

For minimization: If:

$$f(x_t) > f(x_{t+1})$$

Then:

$$P(accept_new_point) = 1$$

If:

$$f(x_t) < f(x_{t+1})$$

Then:

$$P(accept_new_point) = e^{-(\frac{f(x_{t+1})-f(x_t)}{Tt})}$$

Termination Criterion: The algorithm continues iterations until a termination condition is met. This could be a fixed number of iterations, reaching a specific temperature threshold, or achieving a satisfactory solution.

2.3.2.2. Steps

1. Initialization: Set an initial temperature (T_0) and an initial solution ($f(x_0)$). The temperature is typically set high initially to encourage exploration.

2. Generate a Neighbor: Perturb the current solution to generate a neighboring solution. The perturbation can be random or follow a specific strategy.

3. Evaluate the Neighbor: Evaluate the objective function for the new solution in the neighborhood.

4. Accept or Reject the Neighbor: + If the new solution is better (lower cost for minimization problems or higher for maximization problems), accept it as the new current solution. + If the new solution is worse, accept it with a probability determined by an acceptance probability function as mentioned above. The probability is influenced by the difference in objective function values and the current temperature.

5. Cooling: Reduce the temperature according to a cooling schedule. The cooling schedule defines how fast the temperature decreases over time. Common cooling schedules include exponential or linear decay.

6. Termination Criterion: Repeat the iterations (2-5) until a termination condition is met. This could be a fixed number of iterations, reaching a specific temperature threshold, or achieving a satisfactory solution.

2.3.2.3. Scipy Implementation of the Dual Annealing Algorithm

In Scipy, we utilize the Dual Annealing optimizer, an extension of the simulated annealing algorithm that is versatile for both discrete and continuous problems.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import dual_annealing

def rastrigin_function(x):
    return 20 + x[0]**2 - 10 * np.cos(2 * np.pi * x[0]) + x[1]**2 - 10 * np.cos(2 * np

# Define the Rastrigin function for visualization
def rastrigin_visualization(x, y):
    return 20 + x**2 - 10 * np.cos(2 * np.pi * x) + y**2 - 10 * np.cos(2 * np.pi * y)
```

```
# Create a meshgrid for visualization
x_vals = np.linspace(-10, 10, 100)
y_vals = np.linspace(-10, 10, 100)
x_mesh, y_mesh = np.meshgrid(x_vals, y_vals)
z_mesh = rastrigin_visualization(x_mesh, y_mesh)

# Visualize the Rastrigin function
plt.figure(figsize=(10, 8))
contour = plt.contour(x_mesh, y_mesh, z_mesh, levels=50, cmap='viridis')
plt.colorbar(contour, label='Rastrigin Function Value')
plt.title('Visualization of the 2D Rastrigin Function')

# Optimize the Rastrigin function using dual annealing
result = dual_annealing(func = rastrigin_function,
                        x0=[5.0,3.0],                       #Initial Guess
                        bounds= [(-10, 10), (-10, 10)],
                        initial_temp = 5230,                #Intial Value for temperature
                        restart_temp_ratio = 2e-05,         #Temperature schedule
                        seed=42)

# Plot the optimized point
optimal_x, optimal_y = result.x
plt.plot(optimal_x, optimal_y, 'ro', label='Optimal Point')

# Set labels and legend
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()

# Show the plot
plt.show()

# Display the optimization result
print("Optimal parameters:", result.x)
print("Minimum value of the Rastrigin function:", result.fun)
```

```
Optimal parameters: [-4.60133247e-09 -4.31928660e-09]
Minimum value of the Rastrigin function: 7.105427357601002e-15
```

2.3.3. Differential Evolution

Differential Evolution is an algorithm used for finding the global minimum of multivariate functions. It is stochastic in nature (does not use gradient methods), and can search large areas of candidate space, but often requires larger numbers of function evaluations than conventional gradient based techniques.

Differential Evolution (DE) is a versatile and global optimization algorithm inspired by natural selection and evolutionary processes. Introduced by Storn and Price in 1997, DE mimics the survival-of-the-fittest principle by evolving a population of candidate solutions through iterative mutation, crossover, and selection operations. This nature-inspired approach enables DE to efficiently explore complex and non-linear solution spaces, making it a widely adopted optimization technique in diverse fields such as engineering, finance, and machine learning.

2.3.4. Procedure

The procedure boils down to the following steps:

1. **Initialization:**
   - Create a population of candidate solutions randomly within the specified search space.
2. **Mutation:**
   - For each individual in the population, select three distinct individuals (vectors) randomly.
   - Generate a mutant vector `V` by combining these three vectors with a scaling factor.
3. **Crossover:**
   - Perform the crossover operation between the target vector `U` and the mutant vector `V`. Information from both vectors is used to create a trial vector `U´`

> **Cross-Over Strategies in DE**
>
> - There are several crossover strategies in the literature. Two examples are:
>
> **Binominal Crossover:**
> In this strategy, each component of the trial vector is selected from the mutant vector with a probability equal to the crossover rate (CR). This means that each element of the trial vector has an independent probability of being replaced by the corresponding element of the mutant vector.
>
> $$U_i' = \begin{cases} V_i, & \text{if a random number } \sim U(0,1) \leq CR \text{ (Crossover Rate)} \\ U_i, & \text{otherwise} \end{cases}$$
>
> **Exponential Crossover:**
> In exponential crossover, the trial vector is constructed by selecting a random starting point and copying elements from the mutant vector with a certain probability. The probability decreases exponentially with the distance from the starting point. This strategy introduces a correlation between neighboring elements in the trial vector.

4. **Selection:**
   - Evaluate the fitness of the trial vector obtained from the crossover.
   - Replace the target vector with the trial vector if its fitness is better.
5. **Termination:**
   - Repeat the mutation, crossover, and selection steps for a predefined number of generations or until convergence criteria are met.

6. **Result:**
   - The algorithm returns the best-found solution after the specified number of iterations.

The key parameters in DE include the population size, crossover probability, and the scaling factor. Tweak these parameters based on the characteristics of the optimization problem for optimal performance.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

# Define the Rastrigin function
def rastrigin(x):
    A = 10
    return A * len(x) + sum([(xi**2 - A * np.cos(2 * np.pi * xi)) for xi in x])

# Create a grid for visualization
x_vals = np.linspace(-5.12, 5.12, 100)
y_vals = np.linspace(-5.12, 5.12, 100)
X, Y = np.meshgrid(x_vals, y_vals)
Z = rastrigin(np.vstack([X.ravel(), Y.ravel()]))

# Reshape Z to match the shape of X and Y
Z = Z.reshape(X.shape)

# Plot the Rastrigin function
plt.contour(X, Y, Z, levels=50, cmap='viridis', label='Rastrigin Function')

# Initial guess (starting point for the optimization)
initial_guess = (4,3,4,2)

# Define the bounds for each variable in the Rastrigin function
bounds = [(-5.12, 5.12)] * 4  # 4D problem, each variable has bounds (-5.12, 5.12)

# Run the minimize function
result = minimize(rastrigin, initial_guess, bounds=bounds, method='L-BFGS-B')

# Extract the optimal solution
optimal_solution = result.x

# Plot the optimal solution
plt.scatter(optimal_solution[0], optimal_solution[1], color='red', marker='x', label=

# Add labels and legend
```

```
plt.title('Optimization of Rastrigin Function with Minimize')
plt.xlabel('Variable 1')
plt.ylabel('Variable 2')
plt.legend()

# Show the plot
plt.show()

# Print the optimization result
print("Optimal Solution:", optimal_solution)
print("Optimal Objective Value:", result.fun)
```

```
Optimal Solution: [-2.52869119e-08 -2.07795060e-08 -2.52869119e-08 -
1.62721002e-08]
Optimal Objective Value: 3.907985046680551e-13
```

2.3.5. Other global optimization algorithms

2.3.6. DIRECT

DIviding RECTangles (DIRECT) is a deterministic global optimization algorithm capable of minimizing a black box function with its variables subject to lower and upper bound constraints by sampling potential solutions in the search space

2.3.7. SHGO

SHGO stands for "simplicial homology global optimization". It is considered appropriate for solving general purpose NLP and blackbox optimization problems to global optimality (low-dimensional problems).

2.3.8. Basin-hopping

Basin-hopping is a two-phase method that combines a global stepping algorithm with local minimization at each step. Designed to mimic the natural process of energy minimization of clusters of atoms, it works well for similar problems with "funnel-like, but rugged" energy landscapes

2.4. Project: One-Mass Oscillator Optimization

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import minimize
```

2.4.1. Introduction

In this project, you will apply various optimization algorithms to fit a one-mass oscillator model to real-world data. The objective is to minimize the sum of the squared residuals between the model predictions and the observed amplitudes of a one-mass oscillator system across different frequencies.

2.4.2. One-Mass Oscillator Model

The one-mass oscillator is characterized by the following equation, representing the amplitudes of the system:

$$V(\omega) = \frac{F}{\sqrt{(1-\nu^2)^2 + 4D^2\nu^2}}$$

Here, ω represents the angular frequency of the system, ν is the ratio of the excitation frequency to the natural frequency, i.e.,

$$\nu = \frac{\omega_{\text{err}}}{\omega_{\text{eig}}},$$

D is the damping ratio, and F is the force applied to the system.

The goal of the project is to determine the optimal values for the parameters ω_{eig}, D, and F that result in the best fit of the one-mass oscillator model to the observed amplitudes.

2.4.3. The Real-World Data

There are two different measurements. `J` represents the measured frequencies, and `N` represents the measured amplitudes.

```
df1 = pd.read_pickle("./data/Hcf.d/df1.pkl")
df2 = pd.read_pickle("./data/Hcf.d/df2.pkl")
df1.describe()
```

	J	N
count	33.000000	33.000000
mean	8148.750252	10.430887
std	6.870023	2.846469
min	8137.649210	4.698761
25%	8143.799766	8.319253
50%	8146.942295	10.152119
75%	8153.934051	13.407260
max	8162.504002	14.382749

```
df1.head()
```

	J	N
14999	8162.504002	5.527511
15011	8156.384831	7.359789
15016	8159.199238	6.532958
15020	8159.200889	5.895933
15025	8153.934051	9.326749

```
# plot the data, i.e., the measured amplitudes as a function of the measured frequenci
plt.scatter(df1["J"], df1["N"], color="black", label="Spektralpunkte", zorder=5, s=10)
plt.xlabel("Frequency [Hz]")
plt.ylabel("Amplitude")
plt.show()
```

Note: Low amplitudes distort the fit and are negligible therefore we define a lower threshold for `N`.

```
threshold = 0.4
df1.sort_values("N")
max_N = max(df1["N"])
df1 = df1[df1["N"]>=threshold*max_N]
```

We extract the frequency value for maximum value of the amplitude. This serves as the initial value for one decision variable.

```
df_max=df1[df1["N"]==max(df1["N"])]
initial_Oeig = df_max["J"].values[0]
max_N = df_max["N"].values[0]
```

We also have to define the other two initial guesses for the damping ratio and the force, e.g.,

```
initial_D = 0.006
initial_F = 0.120
initial_values = [initial_Oeig, initial_D, initial_F]
```

Additionally, we define the bounds for the decision variables:

```
min_Oerr = min(df1["J"])
max_Oerr = max(df1["J"])
```

```
bounds = [(min_Oerr, max_Oerr), (0, 0.03), (0, 1)]
```

2.4.4. Objective Function

Then we define the objective function:

```
def one_mass_oscillator(params, Oerr) -> np.ndarray:
    # returns amplitudes of the system
    # Defines the model of a one mass oscilator
    Oeig, D, F = params
    nue = Oerr / Oeig
    V = F / (np.sqrt((1 - nue**2) ** 2 + (4 * D**2 * nue**2)))
    return V
```

```
def objective_function(params, Oerr, amplitudes) -> np.ndarray:
    # objective function to compare calculated and real amplitudes
    return np.sum((amplitudes - one_mass_oscillator(params, Oerr)) ** 2)
```

We define the options for the optimzer and start the optimization process:

```
options = {
    "maxfun": 100000,
    "ftol": 1e-9,
    "xtol": 1e-9,
    "stepmx": 10,
    "eta": 0.25,
    "gtol": 1e-5}
```

```
J = np.array(df1["J"]) # measured frequency
N = np.array(df1["N"]) # measured amplitude
```

```
result = minimize(
    objective_function,
    initial_values,
    args=(J, N),
    method='Nelder-Mead',
    bounds=bounds,
    options=options)
```

2.4.5. Results

We can observe the results:

```
# map optimized values to variables
resonant_frequency = result.x[0]
D = result.x[1]
F = result.x[2]
# predict the resonant amplitude with the fitted one mass oscillator.
X_pred = np.linspace(min_Oerr, max_Oerr, 1000)
ypred_one_mass_oscillator = one_mass_oscillator(result.x, X_pred)
resonant_amplitude = max(ypred_one_mass_oscillator)
print(f"result: {result}")
```

```
result:        message: Optimization terminated successfully.
       success: True
        status: 0
           fun: 53.54144061205875
             x: [ 8.148e+03  7.435e-04  2.153e-02]
           nit: 93
          nfev: 169
 final_simplex: (array([[ 8.148e+03,  7.435e-04,  2.153e-02],
                       [ 8.148e+03,  7.435e-04,  2.153e-02],
                       [ 8.148e+03,  7.435e-04,  2.153e-02],
                       [ 8.148e+03,  7.435e-04,  2.153e-02]]), array([ 5.354e+01,  5.3
```

Finally, we can plot the optimized fit and the real values:

```
plt.scatter(
    df1["J"],
    df1["N"],
    color="black",
    label="Spektralpunkte filtered",
    zorder=5,
    s=10,
)
# color the max amplitude point red
plt.scatter(
    initial_Oeig,
    max_N,
    color="red",
    label="Max Amplitude",
    zorder=5,
    s=10,
)

plt.plot(
        X_pred,
        ypred_one_mass_oscillator,
        label="Alpha",
        color="blue",
        linewidth=1,
    )
plt.scatter(
    resonant_frequency,
    resonant_amplitude,
    color="blue",
    label="Max Curve Fit",
    zorder=10,
    s=20,
)
```

2.5. Exercises

Exercise 2.1 (Nelder-Mead)**.**

1. What are the steps of the Nelder-Mead algorithm?
2. What are the advantages and disadvantages of the Nelder-Mead algorithm?

Exercise 2.2 (Powell's Method)**.**

1. What are the steps of Powell's method?
2. What are the advantages and disadvantages of Powell's method?
3. What are similarities between the Nelder-Mead and Powell's methods?

Exercise 2.3 (Gradient Descent)**.**

1. What are the steps of the gradient descent algorithm?
2. What is the learning rate in the gradient descent algorithm?

Exercise 2.4 (Newton Method)**.**

1. What is the difference between the gradient descent and Newton method?
2. Which of the two methods converges faster?

Exercise 2.5 (BFGS)**.**

- In which situations is it possible to use algorithms like BFGS, but not the classical Newton method?
- Would you choose Gradient Descent or BFGS for a large-scale optimization problem?

Exercise 2.6 (Dual Annealing)**.**

1. When should you use Simulated Annealing or Dual Annealing over a local optimization algorithm?
2. Describe the Temperature parameter in Simulated Annealing.

Exercise 2.7 (Differential Evolution)**.**

1. What are the key steps in the Differential Evolution algorithm?
2. Explain the crossover operation in Differential Evolution.

2.6. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

Part II.

Numerical Methods

3. Simulation and Surrogate Modeling

- We will consider the interplay between
  - mathematical models,
  - numerical approximation,
  - simulation,
  - computer experiments, and
  - field data
- Experimental design will play a key role in our developments, but not in the classical regression and response surface methodology sense
- Challenging real-data/real-simulation examples benefiting from modern surrogate modeling methodology
- We will consider the classical, response surface methodology (RSM) approach, and then move on to more modern approaches
- All approaches are based on surrogates

3.1. Surrogates

- Gathering data is **expensive**, and sometimes getting exactly the data you want is impossible or unethical
- **Surrogate**: substitute for the real thing
- In statistics, draws from predictive equations derived from a fitted model can act as a surrogate for the data-generating mechanism
- Benefits of the surrogate approach:
  - Surrogate could represent a cheaper way to explore relationships, and entertain "what ifs?"
  - Surrogates favor faithful yet pragmatic reproduction of dynamics:
    - interpretation,
    - establishing causality, or
    - identification
  - Many numerical simulators are **deterministic**, whereas field observations are noisy or have measurement error

3.1.1. Costs of Simulation

- Computer simulations are generally cheaper (but not always!) than physical observation
- Some computer simulations can be just as expensive as field experimentation, but computer modeling is regarded as easier because:
  - the experimental apparatus is better understood
  - more aspects may be controlled.

3.1.2. Mathematical Models and Meta-Models

- Use of mathematical models leveraging numerical solvers has been commonplace for some time
- Mathematical models became more complex, requiring more resources to simulate/solve numerically
- Practitioners increasingly relied on **meta-models** built off of limited simulation campaigns

3.1.3. Surrogates = Trained Meta-models

- Data collected via expensive computer evaluations tuned flexible functional forms that could be used in lieu of further simulation to
  - save money or computational resources;
  - cope with an inability to perform future runs (expired licenses, off-line or over-impacted supercomputers)
- Trained meta-models became known as **surrogates**

3.1.4. Computer Experiments

- **Computer experiment**: design, running, and fitting meta-models.
  - Like an ordinary statistical experiment, except the data are generated by computer codes rather than physical or field observations, or surveys
- **Surrogate modeling** is statistical modeling of computer experiments

3.1.5. Limits of Mathematical Modeling

- Mathematical biologists, economists and others had reached the limit of equilibrium-based mathematical modeling with cute closed-form solutions
- **Stochastic simulations replace deterministic solvers** based on FEM, Navier–Stokes or Euler methods
- Agent-based simulation models are used to explore predator-prey (Lotka–Voltera) dynamics, spread of disease, management of inventory or patients in health insurance markets
- Consequence: the distinction between surrogate and statistical model is all but gone

3.1.6. Why Computer Simulations are Necessary

- You can't seed a real community with Ebola and watch what happens
- If there's (real) field data, say on a historical epidemic, further experimentation may be almost entirely limited to the mathematical and computer modeling side
- Classical statistical methods offer little guidance

3.1.7. Simulation Requirements

- Simulation should
  - enable rich **diagnostics** to help criticize that models
  - **understanding** its sensitivity to inputs and other configurations
  - providing the ability to **optimize** and
  - refine both **automatically** and with expert intervention
- And it has to do all that while remaining **computationally tractable**
- One perspective is so-called **response surface methods** (RSMs):
- a poster child from industrial statistics' heyday, well before information technology became a dominant industry

3.2. Applications of Surrogate Models

The four most common usages of surrogate models are:

- Augmenting Expensive Simulations: Surrogate models act as a 'curve fit' to approximate the results of expensive simulation codes, enabling predictions without rerunning the primary source. This provides significant speed improvements while maintaining useful accuracy.

- Calibration of Predictive Codes: Surrogates bridge the gap between simpler, faster but less accurate models and more accurate, slower models. This multi-fidelity approach allows for improved accuracy without the full computational expense.
- Handling Noisy or Missing Data: Surrogates smooth out random or systematic errors in experimental or computational data, filling gaps and revealing overall trends while filtering out extraneous details.
- Data Mining and Insight Generation: Surrogates help identify functional relationships between variables and their impact on results. They enable engineers to focus on critical variables and visualize data trends effectively.

3.3. DACE and RSM

Mathematical models implemented in computer codes are used to circumvent the need for expensive field data collection. These models are particularly useful when dealing with highly nonlinear response surfaces, high signal-to-noise ratios (which often involve deterministic evaluations), and a global scope. As a result, a new approach is required in comparison to Response Surface Methodology (RSM), which is discussed in Section 6.1.

With the improvement in computing power and simulation fidelity, researchers gain higher confidence and a better understanding of the dynamics in physical, biological, and social systems. However, the expansion of configuration spaces and increasing input dimensions necessitates more extensive designs. High-performance computing (HPC) allows for thousands of runs, whereas previously only tens were possible. This shift towards larger models and training data presents new computational challenges.

Research questions for DACE (Design and Analysis of Computer Experiments) include how to design computer experiments that make efficient use of computation and how to meta-model computer codes to save on simulation effort. The choice of surrogate model for computer codes significantly impacts the optimal experiment design, and the preferred model-design pairs can vary depending on the specific goal.

The combination of computer simulation, design, and modeling with field data from similar real-world experiments introduces a new category of computer model tuning problems. The ultimate goal is to automate these processes to the greatest extent possible, allowing for the deployment of HPC with minimal human intervention.

One of the remaining differences between RSM and DACE lies in how they handle noise. DACE employs replication, a technique that would not be used in a deterministic setting, to separate signal from noise. Traditional RSM is best suited for situations where a substantial proportion of the variability in the data is due to noise, and where the acquisition of data values can be severely limited. Consequently, RSM is better suited for a different class of problems, aligning with its intended purposes.

Two very good texts on computer experiments and surrogate modeling are Santner, Williams, and Notz (2003) and Forrester, Sóbester, and Keane (2008). The former is the canonical reference in the statistics literature and the latter is perhaps more popular in engineering.

Example 3.1 (Example: DACE and RSM)**.** Imagine you are a chemical engineer tasked with optimizing a chemical process to maximize yield. You can control temperature and pressure, but repeated experiments show variability in yield due to inconsistencies in raw materials.

- Using RSM: You would use RSM to design a series of experiments varying temperature and pressure. You would then fit a response surface (a mathematical model) to the data, helping you understand how changes in temperature and pressure affect yield. Using this model, you can identify optimal conditions for maximizing yield despite the noise.
- Using DACE: If instead you use a computational model to simulate the chemical process and want to account for numerical noise or uncertainty in model parameters, you might use DACE. You would run simulations at different conditions, possibly repeating them to assess variability and build a surrogate model that accurately predicts yields, which can be optimized to find the best conditions.

3.3.1. Noise Handling in RSM and DACE

Noise in RSM: In experimental settings, noise often arises due to variability in experimental conditions, measurement errors, or other uncontrollable factors. This noise can significantly affect the response variable, Y. Replication is a standard procedure for handling noise in RSM. In the context of computer experiments, noise might not be present in the traditional sense since simulations can be deterministic. However, variability can arise from uncertainty in input parameters or model inaccuracies. DACE predominantly utilizes advanced interpolation to construct accurate models of deterministic data, sometimes considering statistical noise modeling if needed.

3.4. Updating a Surrogate Model

A surrogate model is updated by incorporating new data points, known as infill points, into the model to improve its accuracy and predictive capabilities. This process is iterative and involves the following steps:

- Identify Regions of Interest: The surrogate model is analyzed to determine areas where it is inaccurate or where further exploration is needed. This could be regions with high uncertainty or areas where the model predicts promising results (e.g., potential optima).

- Select Infill Points: Infill points are new data points chosen based on specific criteria, such as:
- Exploitation: Sampling near predicted optima to refine the solution. Exploration: Sampling in regions of high uncertainty to improve the model globally. Balanced Approach: Combining exploitation and exploration to ensure both local and global improvements.
- Evaluate the True Function: The true function (e.g., a simulation or experiment) is evaluated at the selected infill points to obtain their corresponding outputs.
- Update the Surrogate Model: The surrogate model is retrained or updated using the new data, including the infill points, to improve its accuracy.
- Repeat: The process is repeated until the model meets predefined accuracy criteria or the computational budget is exhausted.

Definition 3.1 (Infill Points)**.** Infill points are strategically chosen new data points added to the surrogate model. They are selected to:

- Reduce uncertainty in the model.
- Improve predictions in regions of interest.
- Enhance the model's ability to identify optima or trends.

The selection of infill points is often guided by infill criteria, such as:

- Expected Improvement (EI): Maximizing the expected improvement over the current best solution.
- Uncertainty Reduction: Sampling where the model's predictions have high variance.
- Probability of Improvement (PI): Sampling where the probability of improving the current best solution is highest.

The iterative infill-points updating process ensures that the surrogate model becomes increasingly accurate and useful for optimization or decision-making tasks.

4. Sampling Plans

Note

- This section is based on chapter 1 in Forrester, Sóbester, and Keane (2008).
- The following Python packages are imported:

```
import numpy as np
import pandas as pd
import numpy as np
from typing import Tuple, Optional
import matplotlib.pyplot as plt
from spotpython.utils.sampling import rlh
from spotpython.utils.effects import screening_plot, screeningplan
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.sampling import (fullfactorial, bestlh,
     jd, mm, mmphi, mmsort, perturb, mmlhs, phisort, mmphi_intensive)
from spotpython.design.poor import Poor
from spotpython.design.clustered import Clustered
from spotpython.design.sobol import Sobol
from spotpython.design.spacefilling import SpaceFilling
from spotpython.design.random import Random
from spotpython.design.grid import Grid
```

4.1. Ideas and Concepts

Definition 4.1 (Sampling Plan)**.** In the context of computer experiments, the term sampling plan refers to the set of input values, say X,at which the computer code is evaluated.

The goal of a sampling plan is to efficiently explore the input space to understand the behavior of the computer code and build a surrogate model that accurately represents the code's behavior. Traditionally, Response Surface Methodology (RSM) has been used to design sampling plans for computer experiments. These sampling plans are

based on procedures that generate points by means of a rectangular grid or a factorial design.

However, more recently, Design and Analysis of Computer Experiments (DACE) has emerged as a more flexible and powerful approach for designing sampling plans.

Engineering design often requires the construction of a surrogate model $\hat{f}$ to approximate the expensive response of a black-box function f. The function $f(x)$ represents a continuous metric (e.g., quality, cost, or performance) defined over a design space $D \subset \mathbb{R}^k$, where x is a k-dimensional vector of design variables. Since evaluating f is costly, only a sparse set of samples is used to construct $\hat{f}$, which can then provide inexpensive predictions for any $x \in D$.

The process involves:

- Sampling discrete observations:
- Using these samples to construct an approximation $\hat{f}$.
- Ensuring the surrogate model is well-posed, meaning it is mathematically valid and can generalize predictions effectively.

A sampling plan

$$X = \left\{x^{(i)} \in D | i = 1, \dots, n\right\}$$

determines the spatial arrangement of observations. While some models require a minimum number of data points n, once this threshold is met, a surrogate model can be constructed to approximate f efficiently.

A well-posed model does not always perform well because its ability to generalize depends heavily on the sampling plan used to collect data. If the sampling plan is poorly designed, the model may fail to capture critical behaviors in the design space. For example:

- Extreme Sampling: Measuring performance only at the extreme values of parameters may miss important behaviors in the center of the design space, leading to incomplete understanding.
- Uneven Sampling: Concentrating samples in certain regions while neglecting others forces the model to extrapolate over unsampled areas, potentially resulting in inaccurate or misleading predictions. Additionally, in some cases, the data may come from external sources or be limited in scope, leaving little control over the sampling plan. This can further restrict the model's ability to generalize effectively.

4.1.1. The 'Curse of Dimensionality' and How to Avoid It

The "curse of dimensionality" refers to the exponential increase in computational complexity and data requirements as the number of dimensions (variables) in a problem grows. For a one-dimensional space, sampling n locations may suffice for accurate predictions. In high-dimensional spaces, the amount of data needed to maintain the same level of accuracy or coverage increases dramatically. For example, if a one-dimensional space requires n samples for a certain accuracy, a k-dimensional space would require n^k samples. This makes tasks like optimization, sampling, and modeling computationally expensive and often impractical in high-dimensional settings.

Example 4.1 (Example: Curse of Dimensionality)**.** Consider a simple example where we want to model the cost of a car tire based on its wheel diameter. If we have one variable (wheel diameter), we might need 10 simulations to get a good estimate of the cost. Now, if we add 8 more variables (e.g., tread pattern, rubber type, etc.), the number of simulations required increases to 10^8 (10 million). This is because the number of combinations of design variables grows exponentially with the number of dimensions. This means that the computational budget required to evaluate all combinations of design variables becomes infeasible. In this case, it would take 11,416 years to complete the simulations, making it impractical to explore the design space fully.

4.1.2. Physical versus Computational Experiments

Physical experiments are prone to experimental errors from three main sources:

- Human error: Mistakes made by the experimenter.
- Random error: Measurement inaccuracies that vary unpredictably.
- Systematic error: Consistent bias due to flaws in the experimental setup.

The key distinction is repeatability: systematic errors remain constant across repetitions, while random errors vary.

Computational experiments, on the other hand, are deterministic and free from random errors. However, they are still affected by:

- Human error: Bugs in code or incorrect boundary conditions.
- Systematic error: Biases from model simplifications (e.g., inviscid flow approximations) or finite resolution (e.g., insufficient mesh resolution).

The term "noise" is used differently in physical and computational contexts. In physical experiments, it refers to random errors, while in computational experiments, it often refers to systematic errors.

Understanding these differences is crucial for designing experiments and applying techniques like Gaussian process-based approximations. For physical experiments, replication mitigates random errors, but this is unnecessary for deterministic computational experiments.

4.1.3. Designing Preliminary Experiments (Screening)

Minimizing the number of design variables $x_1, x_2, \dots, x_k$ is crucial before modeling the objective function f. This process, called screening, aims to reduce dimensionality without compromising the analysis. If f is at least once differentiable over the design domain D, the partial derivative $\frac{\partial f}{\partial x_i}$ can be used to classify variables:

- Negligible Variables: If $\frac{\partial f}{\partial x_i} = 0$, $\forall x \in D$, the variable x_i can be safely neglected.
- Linear Additive Variables: If $\frac{\partial f}{\partial x_i} = \text{constant} \neq 0$, $\forall x \in D$, the effect of x_i is linear and additive.
- Nonlinear Variables: If $\frac{\partial f}{\partial x_i} = g(x_i)$, $\forall x \in D$, where $g(x_i)$ is a non-constant function, f is nonlinear in x_i.
- Interactive Nonlinear Variables: If $\frac{\partial f}{\partial x_i} = g(x_i, x_j, \dots), /, \forall x \in D$, where $g(x_i, x_j, \dots)$ is a function involving interactions with other variables, f is nonlinear in x_i and interacts with x_j.

Measuring $\frac{\partial f}{\partial x_i}$ across the entire design space is often infeasible due to limited budgets. The percentage of time allocated to screening depends on the problem: If many variables are expected to be inactive, thorough screening can significantly improve model accuracy by reducing dimensionality. If most variables are believed to impact the objective, focus should shift to modeling instead. Screening is a trade-off between computational cost and model accuracy, and its effectiveness depends on the specific problem context.

4.1.3.1. Estimating the Distribution of Elementary Effects

In order to simplify the presentation of what follows, we make, without loss of generality, the assumption that the design space $D = [0, 1]^k$; that is, we normalize all variables into the unit cube. We shall adhere to this convention for the rest of the book and strongly urge the reader to do likewise when implementing any algorithms described here, as this step not only yields clearer mathematics in some cases but also safeguards against scaling issues.

Before proceeding with the description of the Morris algorithm, we need to define an important statistical concept. Let us restrict our design space D to a k-dimensional, p-level full factorial grid, that is,

$$x_i \in \{0, \frac{1}{p-1}, \frac{2}{p-1}, \dots, 1\}, \quad \text{for } i = 1, \dots, k.$$

Definition 4.2 (Elementary Effect)**.** For a given baseline value $x \in D$, let $d_i(x)$ denote the elementary effect of x_i, where:

$$d_i(x) = \frac{f(x_1, \dots, x_i + \Delta, \dots, x_k) - f(x_1, \dots, x_i - \Delta, \dots, x_k)}{2\Delta}, \quad i = 1, \dots, k, \tag{4.1}$$

where Δ is the step size, which is defined as the distance between two adjacent levels in the grid. In other words, we have:

with

$$\Delta = \frac{\xi}{p-1}, \quad \xi \in \mathbb{N}^*, \quad \text{and} \quad x \in D, \text{ such that its components } x_i \leq 1 - \Delta.$$

Δ is the step size. The elementary effect $d_i(x)$ measures the sensitivity of the function f to changes in the variable x_i at the point x.

Morris's method aims to estimate the parameters of the distribution of elementary effects associated with each variable. A large measure of central tendency indicates that a variable has a significant influence on the objective function across the design space, while a large measure of spread suggests that the variable is involved in interactions or contributes to the nonlinearity of f. In practice, the sample mean and standard deviation of a set of $d_i(x)$ values, calculated in different parts of the design space, are used for this estimation.

To ensure efficiency, the preliminary sampling plan X should be designed so that each evaluation of the objective function f contributes to the calculation of two elementary effects, rather than just one (as would occur with a naive random spread of baseline x values and adding Δ to one variable). Additionally, the sampling plan should provide a specified number (e.g., r) of elementary effects for each variable, independently drawn with replacement. For a detailed discussion on constructing such a sampling plan, readers are encouraged to consult Morris's original paper (Morris, 1991). Here, we focus on describing the process itself.

The random orientation of the sampling plan B can be constructed as follows:

- Let B be a $(k+1) \times k$ matrix of 0s and 1s, where for each column i, two rows differ only in their i-th entries.
- Compute a random orientation of B, denoted B^*:

$$B^* = \left(1_{k+1,k} x^* + (\Delta/2)\left[(2B - 1_{k+1,k})D^* + 1_{k+1,k}\right]\right) P^*,$$

where:

- D^* is a k-dimensional diagonal matrix with diagonal elements ± 1 (equal probability),

- $\mathbf{1}$ is a matrix of 1s,
- x^* is a randomly chosen point in the p-level design space (limited by Δ),
- P^* is a $k \times k$ random permutation matrix with one 1 per column and row.

`spotpython` provides a `Python` implementation to compute B^*, see https://github.com/sequential-parameter-optimization/spotPython/blob/main/src/spotpython/utils/effects.py.

Here is the corresponding code:

```
def randorient(k, p, xi, seed=None):
    # Initialize random number generator with the provided seed
    if seed is not None:
        rng = np.random.default_rng(seed)
    else:
        rng = np.random.default_rng()

    # Step length
    Delta = xi / (p - 1)

    m = k + 1

    # A truncated p-level grid in one dimension
    xs = np.arange(0, 1 - Delta, 1 / (p - 1))
    xsl = len(xs)
    if xsl < 1:
        print(f"xi = {xi}.")
        print(f"p = {p}.")
        print(f"Delta = {Delta}.")
        print(f"p - 1 = {p - 1}.")
        raise ValueError(f"The number of levels xsl is {xsl}, but it must be greater t

    # Basic sampling matrix
    B = np.vstack((np.zeros((1, k)), np.tril(np.ones((k, k)))))

    # Randomization

    # Matrix with +1s and -1s on the diagonal with equal probability
    Dstar = np.diag(2 * rng.integers(0, 2, size=k) - 1)

    # Random base value
    xstar = xs[rng.integers(0, xsl, size=k)]

    # Permutation matrix
    Pstar = np.zeros((k, k))
```

```
    rp = rng.permutation(k)
    for i in range(k):
        Pstar[i, rp[i]] = 1

    # A random orientation of the sampling matrix
    Bstar = (np.ones((m, 1)) @ xstar.reshape(1, -1) +
        (Delta / 2) * ((2 * B - np.ones((m, k))) @ Dstar +
        np.ones((m, k)))) @ Pstar

    return Bstar
```

The code following snippet generates a random orientation of a sampling matrix `Bstar` using the `randorient()` function. The input parameters are:

- k = 3: The number of design variables (dimensions).
- p = 3: The number of levels in the grid for each variable.
- xi = 1: A parameter used to calculate the step size Delta.

Step-size calculation is performed as follows: `Delta = xi / (p - 1) = 1 / (3 - 1) = 0.5`, which determines the spacing between levels in the grid.

Next, random sampling matrix construction is computed:

- A truncated grid is created with levels `[0, 0.5]` (based on Delta).
- A basic sampling matrix B is constructed, which is a lower triangular matrix with 0s and 1s.

Then, randomization is applied:

- `Dstar`: A diagonal matrix with random entries of +1 or -1.
- `xstar`: A random starting point from the grid.
- `Pstar`: A random permutation matrix.

Random orientation is applied to the basic sampling matrix `B` to create `Bstar`. This involves scaling, shifting, and permuting the rows and columns of B.

The final output is the matrix `Bstar`, which represents a random orientation of the sampling plan. Each row corresponds to a sampled point in the design space, and each column corresponds to a design variable.

Example 4.2 (Random Orientation of the Sampling Matrix in 2-D)**.**

```
k = 2
p = 3
xi = 1
Bstar = randorient(k, p, xi, seed=123)
print(f"Random orientation of the sampling matrix:\n{Bstar}")
```

```
Random orientation of the sampling matrix:
[[0.5 0. ]
 [0.  0. ]
 [0.  0.5]]
```

We can visualize the random orientation of the sampling matrix in 2-D as shown in Figure 4.1.

```
plt.figure(figsize=(6, 6))
plt.scatter(Bstar[:, 0], Bstar[:, 1], color='blue', s=50, label='Hypercube Points')
for i in range(Bstar.shape[0]):
    plt.text(Bstar[i, 0] + 0.01, Bstar[i, 1] + 0.01, str(i), fontsize=9)
plt.xlim(-0.1, 1.1)
plt.ylim(-0.1, 1.1)
plt.xlabel('x1')
plt.ylabel('x2')
plt.grid()
```

Figure 4.1.: Random orientation of the sampling matrix in 2-D. The labels indicate the row index of the points.

Example 4.3 (Random Orientation of the Sampling Matrix)**.**

```
k = 3
p = 3
xi = 1
Bstar = randorient(k, p, xi)
print(f"Random orientation of the sampling matrix:\n{Bstar}")
```

```
Random orientation of the sampling matrix:
```

```
[[0.5 0.5 0. ]
 [0.5 0.  0. ]
 [0.  0.  0. ]
 [0.  0.  0.5]]
```

To obtain r elementary effects for each variable, the screening plan is built from r random orientations:

$$X = \begin{pmatrix} B_1^* \\ B_2^* \\ \vdots \\ B_r^* \end{pmatrix}$$

The function `screeningplan()` generates a screening plan by calling the `randorient()` function `r` times. It creates a list of random orientations and then concatenates them into a single array, which represents the screening plan. The screening plan implementation in `Python` is as follows (see https://github.com/sequential-parameter-optimization/spotPython/blob/main/src/spotpython/utils/effects.py):

```python
def screeningplan(k, p, xi, r):
    # Empty list to accumulate screening plan rows
    X = []
    for i in range(r):
        X.append(randorient(k, p, xi))
    # Concatenate list of arrays into a single array
    X = np.vstack(X)
    return X
```

It works like follows:

- The value of the objective function f is computed for each row of the screening plan matrix X. These values are stored in a column vector t of size $(r*(k+1))\times 1$, where:
  - `r` is the number of random orientations.
  - `k` is the number of design variables.

The elementary effects are calculated using the following formula:

- For each random orientation, adjacent rows of the screening plan matrix X and their corresponding function values from t are used.
- These values are inserted into Equation 4.1 to compute elementary effects for each variable. An elementary effect measures the sensitivity of the objective function to changes in a specific variable.

Results can be used for a statistical analysis. After collecting a sample of r elementary effects for each variable:

- The sample mean (central tendency) is computed to indicate the overall influence of the variable.
- The sample standard deviation (spread) is computed to capture variability, which may indicate interactions or nonlinearity.

The results (sample means and standard deviations) are plotted on a chart for comparison. This helps identify which variables have the most significant impact on the objective function and whether their effects are linear or involve interactions. This is implemented in the function `screening_plot()` in `Python`, which uses the helper function `_screening()` to calculate the elementary effects and their statistics.

```python
def _screening(X, fun, xi, p, labels, bounds=None) -> tuple:
    """Helper function to calculate elementary effects for a screening design.

    Args:
        X (np.ndarray): The screening plan matrix, typically structured
            within a [0,1]^k box.
        fun (object): The objective function to evaluate at each
            design point in the screening plan.
        xi (float): The elementary effect step length factor.
        p (int): Number of discrete levels along each dimension.
        labels (list of str): A list of variable names corresponding to
            the design variables.
        bounds (np.ndarray): A 2xk matrix where the first row contains
            lower bounds and the second row contains upper bounds for
            each variable.

    Returns:
        tuple: A tuple containing two arrays:
            - sm: The mean of the elementary effects for each variable.
            - ssd: The standard deviation of the elementary effects for
            each variable.
    """
    k = X.shape[1]
    r = X.shape[0] // (k + 1)

    # Scale each design point
    t = np.zeros(X.shape[0])
    for i in range(X.shape[0]):
        if bounds is not None:
            X[i, :] = bounds[0, :] + X[i, :] * (bounds[1, :] - bounds[0, :])
        t[i] = fun(X[i, :])
```

```
    # Elementary effects
    F = np.zeros((k, r))
    for i in range(r):
        for j in range(i * (k + 1), i * (k + 1) + k):
            idx = np.where(X[j, :] - X[j + 1, :] != 0)[0][0]
            F[idx, i] = (t[j + 1] - t[j]) / (xi / (p - 1))

    # Statistical measures (divide by n)
    ssd = np.std(F, axis=1, ddof=0)
    sm = np.mean(F, axis=1)
    return sm, ssd


def screening_plot(X, fun, xi, p, labels, bounds=None, show=True) -> None:
    """Generates a plot with elementary effect screening metrics.

    This function calculates the mean and standard deviation of the
    elementary effects for a given set of design variables and plots
    the results.

    Args:
        X (np.ndarray):
            The screening plan matrix, typically structured within a [0,1]^k box.
        fun (object):
            The objective function to evaluate at each design point in the screening p
        xi (float):
            The elementary effect step length factor.
        p (int):
            Number of discrete levels along each dimension.
        labels (list of str):
            A list of variable names corresponding to the design variables.
        bounds (np.ndarray):
            A 2xk matrix where the first row contains lower bounds and
            the second row contains upper bounds for each variable.
        show (bool):
            If True, the plot is displayed. Defaults to True.

    Returns:
        None: The function generates a plot of the results.
    """
    k = X.shape[1]
    sm, ssd = _screening(X=X, fun=fun, xi=xi, p=p, labels=labels, bounds=bounds)
    plt.figure()
    for i in range(k):
```

```
        plt.text(sm[i], ssd[i], labels[i], fontsize=10)
    plt.axis([min(sm), 1.1 * max(sm), min(ssd), 1.1 * max(ssd)])
    plt.xlabel("Sample means")
    plt.ylabel("Sample standard deviations")
    plt.gca().tick_params(labelsize=10)
    plt.grid(True)
    if show:
        plt.show()
```

4.1.4. Special Considerations When Deploying Screening Algorithms

When implementing the screening algorithm described above, two specific scenarios require special attention:

- Duplicate Design Points: If the dimensionality k of the space is relatively low and you can afford a large number of elementary effects r, we should be be aware of the increased probability of duplicate design points appearing in the sampling plan X. *Since the responses at sample points are deterministic, there's no value in evaluating the same point multiple times. Fortunately, this issue is relatively uncommon in practice, as screening high-dimensional spaces typically requires large numbers of elementary effects, which naturally reduces the likelihood of duplicates.
- Failed Simulations: Numerical simulation codes occasionally fail to return valid results due to meshing errors, non-convergence of partial differential equation solvers, numerical instabilities, or parameter combinations outside the stable operating range.

From a screening perspective, this is particularly problematic because an entire random orientation B^* becomes compromised if even a single point within it fails to evaluate properly. Implementing error handling strategies or fallback methods to manage such cases should be considered.

For robust screening studies, monitoring simulation success rates and having contingency plans for failed evaluations are important aspects of the experimental design process.

4.2. Analyzing Variable Importance in Aircraft Wing Weight

Let us consider the following analytical expression used as a conceptual level estimate of the weight of a light aircraft wing as discussed in Chapter 1.

```
fun = Analytical()
k = 10
p = 10
xi = 1
r = 25
X = screeningplan(k=k, p=p, xi=xi, r=r)  # shape (r x (k+1), k)
value_range = np.array([
    [150, 220,   6, -10, 16, 0.5, 0.08, 2.5, 1700, 0.025],
    [200, 300,  10,  10, 45, 1.0, 0.18, 6.0, 2500, 0.08 ],
])
labels = [
    "S_W", "W_fw", "A", "Lambda",
    "q",   "lambda", "tc", "N_z",
    "W_dg", "W_p"
]
screening_plot(
    X=X,
    fun=fun.fun_wingwt,
    bounds=value_range,
    xi=xi,
    p=p,
    labels=labels,
)
```

Figure 4.2.: Estimated means and standard deviations of the elementary effects for the 10 design variables of the wing weight function. Example based on Forrester, Sóbester, and Keane (2008).

> **Nondeterministic Results**
>
> The code will generate a slightly different screening plan each time, as it uses random orientations of the sampling matrix B.

Figure 4.2 provides valuable insights into variable activity without requiring domain expertise. The screening study with $r = 25$ elementary effects reveals distinct patterns in how variables affect wing weight:

- Variables with Minimal Impact: A clearly defined group of variables clusters around the origin - indicating their minimal impact on wing weight:
  - Paint weight (W_p) - as expected, contributes little to overall wing weight
  - Dynamic pressure (q) - within our chosen range, this has limited effect (essentially representing different cruise altitudes at the same speed)
  - Taper ratio (λ) and quarter-chord sweep (Λ) - these geometric parameters have minor influence within the narrow range (-10° to 10°) typical of light aircraft
- Variables with Linear Effects:

  - While still close to the origin, fuel weight (W_{fw}) shows a slightly larger central tendency with very low standard deviation. This indicates moderate importance but minimal involvement in interactions with other variables.

- Variables with Nonlinear/Interactive Effects:
  - Aspect ratio (A) and airfoil thickness ratio (R_{tc}) show similar importance levels, but their high standard deviations suggest significant nonlinear behavior and interactions with other variables.

- Dominant Variables: The most significant impacts come from:
  - Flight design gross weight (W_{dg})
  - Wing area (S_W)
  - Ultimate load factor (N_z)

These variables show both large central tendency values and high standard deviations, indicating strong direct effects and complex interactions. The interaction between aspect ratio and load factor is particularly important - high values of both create extremely heavy wings, explaining why highly maneuverable fighter jets cannot use glider-like wing designs.

What makes this screening approach valuable is its ability to identify critical variables without requiring engineering knowledge or expensive modeling. In real-world applications, we rarely have the luxury of creating comprehensive parameter space visualizations, which is precisely why surrogate modeling is needed. After identifying the active variables through screening, we can design a focused sampling plan for these key variables. This forms the foundation for building an accurate surrogate model of the objective function.

When the objective function is particularly expensive to evaluate, we might recycle the runs performed during screening for the actual model fitting step. This is most effective when some variables prove to have no impact at all. However, since completely inactive variables are rare in practice, engineers must carefully balance the trade-off between reusing expensive simulation runs and introducing potential noise into the model.

4.3. Designing a Sampling Plan

4.3.1. Stratification

A feature shared by all of the approximation models discussed in Forrester, Sóbester, and Keane (2008) is that they are more accurate in the vicinity of the points where we have evaluated the objective function. In later chapters we will delve into the laws that quantify our decaying trust in the model as we move away from a known, sampled point, but for the purposes of the present discussion we shall merely draw the intuitive conclusion that a uniform level of model accuracy throughout the design

space requires a uniform spread of points. A sampling plan possessing this feature is said to be space-filling.

The most straightforward way of sampling a design space in a uniform fashion is by means of a rectangular grid of points. This is the full factorial sampling technique.

Here is the simplified version of a `Python` function that will sample the unit hypercube at all levels in all dimensions, with the k-vector q containing the number of points required along each dimension, see https://github.com/sequential-parameter-optimization/spotPython/blob/main/src/spotpython/utils/sampling.py.

The variable `Edges` specifies whether we want the points to be equally spaced from edge to edge (`Edges=1`) or we want them to be in the centres of $n = q_1 \times q_2 \times ... \times q_k$ bins filling the unit hypercube (for any other value of `Edges`).

```
def fullfactorial(q_param, Edges=1) -> np.ndarray:
    """Generates a full factorial sampling plan in the unit cube.

    Args:
        q (list or np.ndarray):
            A list or array containing the number of points along each dimension (k-vector).
        Edges (int, optional):
            Determines spacing of points. If `Edges=1`, points are equally spaced from edge to edge
            Otherwise, points will be in the centers of n = q[0]*q[1]*...*q[k-1] bins filling the un

    Returns:
        (np.ndarray): Full factorial sampling plan as an array of shape (n, k), where n is the total

    Raises:
        ValueError: If any dimension in `q` is less than 2.
    """
    q_levels = np.array(q_param) # Use a distinct variable for original levels
    if np.min(q_levels) < 2:
        raise ValueError("You must have at least two points per dimension.")

    n = np.prod(q_levels)
    k = len(q_levels)
    X = np.zeros((n, k))

    # q_for_prod_calc is used for calculating repetitions, includes the phantom element.
    # This matches the logic of the user-provided snippet where 'q' was modified.
    q_for_prod_calc = np.append(q_levels, 1)

    for j in range(k): # k is the original number of dimensions
        # current_dim_levels is the number of levels for the current dimension j
        # In the user's snippet, q[j] correctly refers to the original level count
```

```
        # as j ranges from 0 to k-1, and q_for_prod_calc[j] = q_levels[j] for this ran
        current_dim_levels = q_for_prod_calc[j]

        if Edges == 1:
            one_d_slice = np.linspace(0, 1, int(current_dim_levels))
        else:
            # Corrected calculation for bin centers
            if current_dim_levels == 1: # Should not be hit if np.min(q_levels) >= 2
                one_d_slice = np.array([0.5])
            else:
                one_d_slice = np.linspace(1 / (2 * current_dim_levels),
                                          1 - 1 / (2 * current_dim_levels),
                                          int(current_dim_levels))

        column = np.array([])
        # The product q_for_prod_calc[j + 1 : k] correctly calculates
        # the product of remaining original dimensions' levels.
        num_consecutive_repeats = np.prod(q_for_prod_calc[j + 1 : k])

        # This loop structure replicates the logic from the user's snippet
        while len(column) < n:
            for ll_idx in range(int(current_dim_levels)): # Iterate through levels of
                val_to_repeat = one_d_slice[ll_idx]
                column = np.append(column, np.ones(int(num_consecutive_repeats)) * val
        X[:, j] = column
    return X
```

```
q = [3, 2]
X = fullfactorial(q, Edges=0)
print(X)
```

```
[[0.16666667 0.25      ]
 [0.16666667 0.75      ]
 [0.5        0.25      ]
 [0.5        0.75      ]
 [0.83333333 0.25      ]
 [0.83333333 0.75      ]]
```

Figure 4.3 shows the points in the unit hypercube for the case of 3x2 points.

```
X = fullfactorial(q, Edges=1)
print(X)
```

Figure 4.3.: 2D Full Factorial Sampling (3x2 Points). Edges = 0

```
[[0.  0. ]
 [0.  1. ]
 [0.5 0. ]
 [0.5 1. ]
 [1.  0. ]
 [1.  1. ]]
```

Figure 4.4 shows the points in the unit hypercube for the case of 3x2 points with edges.

Figure 4.4.: 2D Full Factorial Sampling (3x2 Points). Edges = 1

The full factorial sampling plan method generates a uniform sampling design by creating a grid of points across all dimensions. For example, calling fullfactorial([3, 4, 5], 1) produces a three-dimensional sampling plan with 3, 4, and 5 levels along each dimension, respectively. While this approach satisfies the uniformity criterion, it has two significant limitations:

- Restricted Design Sizes: The method only works for designs where the total number of points n can be expressed as the product of the number of levels in each dimension, i.e., $n = q_1 \times q_2 \times \cdots \times q_k$.
- Overlapping Projections: When the sampling points are projected onto individual axes, sets of points may overlap, reducing the effectiveness of the sampling plan. This can lead to non-uniform coverage in the projections, which may not fully represent the design space.

4.3.2. Latin Squares and Random Latin Hypercubes

To improve the uniformity of projections for any individual variable, the range of that variable can be divided into a large number of equal-sized bins, and random subsamples of equal size can be generated within these bins. This method is called stratified random sampling. Extending this idea to all dimensions results in a stratified sampling plan, commonly implemented using Latin hypercube sampling.

Definition 4.3 (Latin Squares and Hypercubes)**.** In the context of statistical sampling, a square grid containing sample positions is a Latin square if (and only if) there is only one sample in each row and each column. A Latin hypercube is the generalisation of this concept to an arbitrary number of dimensions, whereby each sample is the only one in each axis-aligned hyperplane containing it

For two-dimensional discrete variables, a Latin square ensures uniform projections. An $(n \times n)$ Latin square is constructed by filling each row and column with a permutation of $\{1, 2, \ldots, n\}$, ensuring each number appears only once per row and column.

Example 4.4 (Latin Square)**.** For $n = 4$, a Latin square might look like this:

```
2   1   3   4
3   2   4   1
1   4   2   3
4   3   1   2
```

Latin Hypercubes are the multidimensional extension of Latin squares. The design space is divided into equal-sized hypercubes (bins), and one point is placed in each bin. The placement ensures that moving along any axis from an occupied bin does not encounter another occupied bin. This guarantees uniform projections across all dimensions. To construct a Latin hypercube, the following steps are taken:

- Represent the sampling plan as an $n \times k$ matrix X, where n is the number of points and k is the number of dimensions.
- Fill each column of X with random permutations of $\{1, 2, \ldots, n\}$.
- Normalize the plan into the unit hypercube $[0, 1]^k$.

This approach ensures multidimensional stratification and uniformity in projections. Here is the code:

```
def rlh(n: int, k: int, edges: int = 0) -> np.ndarray:
    # Initialize array
    X = np.zeros((n, k), dtype=float)

    # Fill with random permutations
    for i in range(k):
        X[:, i] = np.random.permutation(n)

    # Adjust normalization based on the edges flag
    if edges == 1:
        # [X=0..n-1] -> [0..1]
        X = X / (n - 1)
    else:
        # Points at true midpoints
        # [X=0..n-1] -> [0.5/n..(n-0.5)/n]
        X = (X + 0.5) / n

    return X
```

Example 4.5 (Random Latin Hypercube)**.** The following code can be used to generate a 2D Latin hypercube with 5 points and edges=0:

```
X = rlh(n=5, k=2, edges=0)
print(X)
```

```
[[0.9 0.5]
 [0.1 0.1]
 [0.5 0.3]
 [0.7 0.7]
 [0.3 0.9]]
```

Figure 4.5 shows the points in the unit hypercube for the case of 5 points with `edges=0`.

Example 4.6 (Random Latin Hypercube with Edges)**.** The following code can be used to generate a 2D Latin hypercube with 5 points and edges=1:

Figure 4.5.: 2D Latin Hypercube Sampling (5 Points, Edges=0)

```
X = rlh(n=5, k=2, edges=1)
print(X)
```

```
[[1.   0.5 ]
 [0.5  0.25]
 [0.   0.  ]
 [0.75 1.  ]
 [0.25 0.75]]
```

Figure 4.6 shows the points in the unit hypercube for the case of 5 points with `edges=1`.

4.3.3. Space-filling Designs: Maximin Plans

A widely adopted measure for assessing the uniformity, or 'space-fillingness', of a sampling plan is the maximin metric, initially proposed by Johnson, Moore, and Ylvisaker (1990). This criterion can be formally defined as follows.

Consider a sampling plan X. Let $d_1, d_2, \dots, d_m$ represent the unique distances between all possible pairs of points within X, arranged in ascending order. Furthermore, let $J_1, J_2, \dots, J_m$ be defined such that J_j denotes the count of point pairs in X separated by the distance d_j.

Definition 4.4 (Maximin plan)**.** A sampling plan X is considered a maximin plan if, among all candidate plans, it maximizes the smallest inter-point distance d_1. Among plans that satisfy this condition, it further minimizes J_1, the number of pairs separated by this minimum distance.

While this definition is broadly applicable to any collection of sampling plans, our focus is narrowed to Latin hypercube designs to preserve their desirable stratification properties. However, even within this restricted class, Definition 4.4 may identify multiple equivalent maximin designs. To address this, a more comprehensive 'tie-breaker' definition, as proposed by Morris and Mitchell (1995), is employed:

Definition 4.5 (Maximin plan with tie-breaker)**.** A sampling plan X is designated as the maximin plan if it sequentially optimizes the following conditions: it maximizes d_1; among those, it minimizes J_1; among those, it maximizes d_2; among those, it minimizes J_2; and so forth, concluding with minimizing J_m.

Johnson, Moore, and Ylvisaker (1990) established that the maximin criterion (Definition 4.4) is equivalent to the D-optimality criterion used in linear regression. However, the extended maximin criterion incorporating a tie-breaker (Definition 4.5) is often preferred due to its intuitive nature and practical utility. Given that the sampling plans

Figure 4.6.: 2D Latin Hypercube Sampling (5 Points, Edges=1)

under consideration make no assumptions about model structure, the latter criterion (Definition 4.5) will be employed.

To proceed, a precise definition of 'distance' within these contexts is necessary. The p-norm is the most widely adopted metric for this purpose:

Definition 4.6 (p-norm)**.** The p-norm of a vector $\vec{x} = (x_1, x_2, \dots, x_k)$ is defined as:

$$d_p(\vec{x}^{(i_1)}, \vec{x}^{(i_2)}) = \left(\sum_{j=1}^{k} |x_j^{(i_1)} - x_j^{(i_2)}|^p \right)^{1/p}. \tag{4.2}$$

When $p = 1$, Equation 4.2 defines the rectangular distance, occasionally referred to as the Manhattan norm (an allusion to a grid-like city layout). Setting $p = 2$ yields the Euclidean norm. The existing literature offers limited evidence to suggest the superiority of one norm over the other for evaluating sampling plans when no model structure assumptions are made. It is important to note, however, that the rectangular distance is considerably less computationally demanding. This advantage can be quite significant, particularly when evaluating large sampling plans.

For the computational implementation of Definition 4.5, the initial step involves constructing the vectors $d_1, d_2, \dots, d_m$ and $J_1, J_2, \dots, J_m$. The `jd` function facilitates this task.

4.3.3.1. The Function `jd`

The function `jd` computes the distinct p-norm distances between all pairs of points in a given set and counts their occurrences. It returns two arrays: one for the distinct distances and another for their multiplicities.

```
def jd(X: np.ndarray, p: float = 1.0) -> Tuple[np.ndarray, np.ndarray]:
    """
    Args:
        X (np.ndarray):
            A 2D array of shape (n, d) representing n points
            in d-dimensional space.
        p (float, optional):
            The distance norm to use.
            p=1 uses the Manhattan (L1) norm, while p=2 uses the
            Euclidean (L2) norm. Defaults to 1.0 (Manhattan norm).

    Returns:
        (np.ndarray, np.ndarray):
            A tuple (J, distinct_d), where:
```

```
            - distinct_d is a 1D float array of unique,
            sorted distances between points.
            - J is a 1D integer array that provides
            the multiplicity (occurrence count)
            of each distance in distinct_d.
    """
    n = X.shape[0]

    # Allocate enough space for all pairwise distances
    # (n*(n-1))/2 pairs for an n-point set
    pair_count = n * (n - 1) // 2
    d = np.zeros(pair_count, dtype=float)

    # Fill the distance array
    idx = 0
    for i in range(n - 1):
        for j in range(i + 1, n):
            # Compute the p-norm distance
            d[idx] = np.linalg.norm(X[i] - X[j], ord=p)
            idx += 1

    # Find unique distances and their multiplicities
    distinct_d = np.unique(d)
    J = np.zeros_like(distinct_d, dtype=int)
    for i, val in enumerate(distinct_d):
        J[i] = np.sum(d == val)
    return J, distinct_d
```

Example 4.7 (The Function `jd`)**.** Consider a small 3-point set in 2D space, with points located at (0,0), (1,1), and (2,2) as shown in Figure 4.7. The distinct distances and their occurrences can be computed using the `jd` function, as shown in the following code:

```
J, distinct_d = jd(X, p=2.0)
print("Distinct distances (d_i):", distinct_d)
print("Occurrences (J_i):", J)
```

```
Distinct distances (d_i): [1.41421356 2.82842712]
Occurrences (J_i): [2 1]
```

Figure 4.7.: 3-Point Set in 2D Space

4.3.4. Memory Management

A computationally intensive part of the calculation performed with the `jd`-function is the creation of the vector $\vec{d}$ containing all pairwise distances. This is particularly true for large sampling plans; for instance, a 1000-point plan requires nearly half a million distance calculations.

Definition 4.7 (Pre-allocation of Memory)**.** Pre-allocation of memory is a programming technique where a fixed amount of memory is reserved for a data structure (like an array or vector) before it is actually filled with data. This is done to avoid the computational overhead associated with dynamic memory allocation, which involves repeatedly requesting and resizing memory as new elements are added.

Consequently, pre-allocating memory for the distance vector $\vec{d}$ is essential. This necessitates a slightly less direct method for computing the indices of $\vec{d}$, rather than appending each new element, which would involve costly dynamic memory allocation.

The implementation of Definition 4.5 is now required. Finding the most space-filling design involves pairwise comparisons. This problem can be approached using a 'divide and conquer' strategy, simplifying it to the task of selecting the better of two sampling plans. The function `mm(X1,X2,p)` is designed for this purpose. It returns an index indicating which of the two designs is more space-filling, or `0` if they are equally space-filling, based on the p-norm for distance computation.

4.3.4.1. The Function `mm`

The function `mm` compares two sampling plans based on the Morris-Mitchell criterion. It uses the `jd` function to compute the distances and multiplicities, constructs vectors for comparison, and determines which plan is more space-filling.

```
def mm(X1: np.ndarray, X2: np.ndarray, p: Optional[float] = 1.0) -> int:
    """
    Args:
        X1 (np.ndarray): A 2D array representing the first sampling plan.
        X2 (np.ndarray): A 2D array representing the second sampling plan.
        p (float, optional): The distance metric. p=1 uses Manhattan (L1) distance,
            while p=2 uses Euclidean (L2). Defaults to 1.0.

    Returns:
        int:
            - 0 if both plans are identical or equally space-filling
            - 1 if X1 is more space-filling
            - 2 if X2 is more space-filling
    """
```

```
    X1_sorted = X1[np.lexsort(np.rot90(X1))]
    X2_sorted = X2[np.lexsort(np.rot90(X2))]
    if np.array_equal(X1_sorted, X2_sorted):
        return 0  # Identical sampling plans

    # Compute distance multiplicities for each plan
    J1, d1 = jd(X1, p)
    J2, d2 = jd(X2, p)
    m1, m2 = len(d1), len(d2)

    # Construct V1 and V2: alternate distance and negative multiplicity
    V1 = np.zeros(2 * m1)
    V1[0::2] = d1
    V1[1::2] = -J1

    V2 = np.zeros(2 * m2)
    V2[0::2] = d2
    V2[1::2] = -J2

    # Trim the longer vector to match the size of the shorter
    m = min(m1, m2)
    V1 = V1[:m]
    V2 = V2[:m]

    # Compare element-by-element:
    # c[i] = 1 if V1[i] > V2[i], 2 if V1[i] < V2[i], 0 otherwise.
    c = (V1 > V2).astype(int) + 2 * (V1 < V2).astype(int)

    if np.sum(c) == 0:
        # Equally space-filling
        return 0
    else:
        # The first non-zero entry indicates which plan is better
        idx = np.argmax(c != 0)
        return c[idx]
```

Example 4.8 (The Function `mm`)**.** We can use the `mm` function to compare two sampling plans. The following code creates two 3-point sampling plans in 2D (shown in Figure 4.8) and compares them using the Morris-Mitchell criterion:

```
X1 = np.array([[0.0, 0.0],[0.5, 0.5],[0.0, 1.0], [1.0, 1.0]])
X2 = np.array([[0.1, 0.1],[0.4, 0.6],[0.1, 0.9], [0.9, 0.9]])
```

We can compare which plan has better space-filling (Morris-Mitchell). The output is

Figure 4.8.: Comparison of Two Sampling Plans

either 0, 1, or 2 depending on which plan is more space-filling.

```
better = mm(X1, X2, p=2.0)
print(f"Plan {better} is more space-filling.")
```

```
Plan 1 is more space-filling.
```

4.3.4.2. The Function mmphi

Searching across a space of potential sampling plans can be accomplished by pairwise comparisons. An optimization algorithm could, in theory, be written with mm as the comparative objective. However, experimental evidence (Morris and Mitchell 1995) suggests that the resulting optimization landscape can be quite deceptive, making it difficult to search reliably. This difficulty arises because the comparison process terminates upon finding the first non-zero element in the comparison array c. Consequently, the remaining values in the distance $(d_1, d_2, ..., d_m)$ and multiplicity $(J_1, J_2, ..., J_m)$ arrays are disregarded. These disregarded values, however, might contain potentially useful 'slope' information about the global landscape for the optimization process.

To address this, Morris and Mitchell (1995) defined the following scalar-valued criterion function, which is used to rank competing sampling plans. This function, while based on the logic of Definition 4.5, incorporates the complete vectors $d_1, d_2, ..., d_m$ and $J_1, J_2, ..., J_m$.

Definition 4.8 (Morris-Mitchell Criterion)**.** The Morris-Mitchell criterion is defined as:

$$\Phi_q(X) = \left(\sum_{j=1}^{m} J_j d_j^{-q} \right)^{1/q}, \tag{4.3}$$

where X is the sampling plan, d_j is the distance between points, J_j is the multiplicity of that distance, and q is a user-defined exponent. The parameter q can be adjusted to control the influence of smaller distances on the overall metric.

The smaller the value of Φ_q, the better the space-filling properties of X will be.

The function mmphi computes the Morris-Mitchell sampling plan quality criterion for a given sampling plan. It takes a 2D array of points and calculates the space-fillingness metric based on the distances between points. This can be implemented in Python as follows:

```python
def mmphi(X: np.ndarray,
          q: Optional[float] = 2.0,
          p: Optional[float] = 1.0) -> float:
    """
    Args:
        X (np.ndarray):
            A 2D array representing the sampling plan,
            where each row is a point in
            d-dimensional space (shape: (n, d)).
        q (float, optional):
            Exponent used in the computation of the metric.
            Defaults to 2.0.
        p (float, optional):
            The distance norm to use.
            For example, p=1 is Manhattan (L1),
            p=2 is Euclidean (L2). Defaults to 1.0.

    Returns:
        float:
            The space-fillingness metric Phiq. Larger values typically indicate a more
            space-filling plan according to the Morris-Mitchell criterion.
    """
    # Compute the distance multiplicities: J, and unique distances: d
    J, d = jd(X, p)
    # Summation of J[i] * d[i]^(-q), then raised to 1/q
    # This follows the Morris-Mitchell definition.
    Phiq = np.sum(J * (d ** (-q))) ** (1.0 / q)
    return Phiq
```

Example 4.9 (The Function `mmphi`)**.** We can use the `mmphi` function to evaluate the space-filling quality of the two sampling plans from Example 4.8. The following code uses these two 3-point sampling plans in 2D and computes their quality using the Morris-Mitchell criterion:

```python
# Two simple sampling plans from above
quality1 = mmphi(X1, q=2, p=2)
quality2 = mmphi(X2, q=2, p=2)
print(f"Quality of sampling plan X1:  {quality1}")
print(f"Quality of sampling plan X2:  {quality2}")
```

```
Quality of sampling plan X1:  2.91547594742265
Quality of sampling plan X2:  3.917162046269215
```

This equation provides a more compact representation of the maximin criterion, but the selection of the q value is an important consideration. Larger values of q ensure that terms in the sum corresponding to smaller inter-point distances (the d_j values, which are sorted in ascending order) have a dominant influence. As a result, Φ_q will rank sampling plans in a way that closely emulates the original maximin definition (Definition 4.5). This implies that the optimization landscape might retain the challenging characteristics that the Φ_q metric, especially with smaller q values, is intended to alleviate. Conversely, smaller q values tend to produce a Φ_q landscape that, while not perfectly aligning with the original definition, is generally more conducive to optimization.

To illustrate the relationship between Equation 4.3 and the maximin criterion of Definition 4.5, sets of 50 random Latin hypercubes of varying sizes and dimensionalities were considered by Forrester, Sóbester, and Keane (2008). The correlation plots from this analysis suggest that as the sampling plan size increases, a smaller q value is needed for the Φ_q-based ranking to closely match the ranking derived from Definition 4.5.

Rankings based on both the direct maximin comparison (`mm`) and the Φ_q metric (`mmphi`), determined using a simple bubble sort algorithm, are implemented in the Python function `mmsort`.

4.3.4.3. The Function `mmsort`

The function `mmsort` is designed to rank multiple sampling plans based on their space-filling properties using the Morris-Mitchell criterion. It takes a 3D array of sampling plans and returns the indices of the plans sorted in ascending order of their space-filling quality.

```
def mmsort(X3D: np.ndarray, p: Optional[float] = 1.0) -> np.ndarray:
    """
    Args:
        X3D (np.ndarray):
            A 3D NumPy array of shape (n, d, m), where m is the number of
            sampling plans, and each plan is an (n, d) matrix of points.
        p (float, optional):
            The distance metric to use. p=1 for Manhattan (L1), p=2 for
            Euclidean (L2). Defaults to 1.0.

    Returns:
        np.ndarray:
            A 1D integer array of length m that holds the plan indices in
            ascending order of space-filling quality. The first index in the
            returned array corresponds to the most space-filling plan.
    """
    # Number of plans (m)
    m = X3D.shape[2]
```

```
    # Create index array (1-based to match original MATLAB convention)
    Index = np.arange(1, m + 1)

    swap_flag = True
    while swap_flag:
        swap_flag = False
        i = 0
        while i < m - 1:
            # Compare plan at Index[i] vs. Index[i+1] using mm()
            # Note: subtract 1 from each index to convert to 0-based array indexing
            if mm(X3D[:, :, Index[i] - 1], X3D[:, :, Index[i + 1] - 1], p) == 2:
                # Swap indices if the second plan is more space-filling
                Index[i], Index[i + 1] = Index[i + 1], Index[i]
                swap_flag = True
            i += 1

    return Index
```

Example 4.10 (The Function `mmsort`)**.** The `mmsort` function can be used to rank multiple sampling plans based on their space-filling properties. The following code demonstrates how to use `mmsort` to compare two 3-point sampling plans in 3D space:

Suppose we have two 3-point sampling plans X1 and X1 from above. They are sorted using the Morris-Mitchell criterion with $p = 2.0$. For example, the output `[1, 2]` indicates that X1 is more space-filling than X2:

```
X3D = np.stack([X1, X2], axis=2)
ranking = mmsort(X3D, p=2.0)
print(ranking)
```

```
[1 2]
```

To determine the optimal Latin hypercube for a specific application, a recommended approach by Morris and Mitchell (1995) involves minimizing Φ_q for a set of q values (1, 2, 5, 10, 20, 50, and 100). Subsequently, the best plan from these results is selected based on the actual maximin definition. The mmsort function can be utilized for this purpose: a 3D matrix, X3D, can be constructed where each 2D slice represents the best sampling plan found for each Φ_q. Applying `mmsort(X3D,1)` then ranks these plans according to Definition 4.5, using the rectangular distance metric. The subsequent discussion will address the methods for finding these optimized Φ_q designs.

4.3.4.4. The Function `phisort`

`phisort` only differs from `mmsort` in having q as an additional argument, as well as the comparison line being:

```
if mmphi(X3D[:, :, Index[i] - 1], q=q, p=p) >
    mmphi(X3D[:, :, Index[i + 1] - 1], q=q, p=p):
```

```
def phisort(X3D: np.ndarray,
            q: Optional[float] = 2.0,
            p: Optional[float] = 1.0) -> np.ndarray:
    """
    Args:
        X3D (np.ndarray):
            A 3D array of shape (n, d, m),
            where m is the number of sampling plans.
        q (float, optional):
            Exponent for the mmphi metric. Defaults to 2.0.
        p (float, optional):
            Distance norm for mmphi.
            p=1 is Manhattan; p=2 is Euclidean.
            Defaults to 1.0.

    Returns:
        np.ndarray:
            A 1D integer array of length m, giving the plan indices in ascending
            order of mmphi. The first index in the returned array corresponds
            to the numerically lowest mmphi value.
    """
    # Number of 2D sampling plans
    m = X3D.shape[2]
    # Create a 1-based index array
    Index = np.arange(1, m + 1)
    # Bubble-sort: plan with lower mmphi() climbs toward the front
    swap_flag = True
    while swap_flag:
        swap_flag = False
        for i in range(m - 1):
            # Retrieve mmphi values for consecutive plans
            val_i = mmphi(X3D[:, :, Index[i] - 1], q=q, p=p)
            val_j = mmphi(X3D[:, :, Index[i + 1] - 1], q=q, p=p)

            # Swap if the left plan's mmphi is larger (i.e. 'worse')
            if val_i > val_j:
```

```
                Index[i], Index[i + 1] = Index[i + 1], Index[i]
                swap_flag = True
    return Index
```

Example 4.11 (The Function `phisort`)**.** The `phisort` function can be used to rank multiple sampling plans based on the Morris-Mitchell criterion. The following code demonstrates how to use `phisort` to compare two 3-point sampling plans in 3D space:

```
X1 = bestlh(n=5, k=2, population=5, iterations=10)
X2 = bestlh(n=5, k=2, population=15, iterations=20)
X3 = bestlh(n=5, k=2, population=25, iterations=30)
# Map X1 and X2 so that X3D has the two sampling plans
# in X3D[:, :, 0] and X3D[:, :, 1]
X3D = np.array([X1, X2])
print(phisort(X3D))
X3D = np.array([X3, X2])
print(phisort(X3D))
```

```
[1 2]
[1 2]
```

4.3.5. Optimizing the Morris-Mitchell Criterion Φ_q

Once a criterion for assessing the quality of a Latin hypercube sampling plan has been established, a systematic method for optimizing this metric across the space of Latin hypercubes is required. This task is non-trivial; as the reader may recall from the earlier discussion on Latin squares, this search space is vast. In fact, its vastness means that for many practical applications, locating the globally optimal solution is often infeasible. Therefore, the objective becomes finding the best possible sampling plan achievable within a specific computational time budget.

This budget is influenced by the computational cost associated with obtaining each objective function value. Determining the optimal allocation of total computational effort—between generating the sampling plan and actually evaluating the objective function at the selected points—remains an open research question. However, it is typical for no more than approximately 5% of the total available time to be allocated to the task of generating the sampling plan itself.

Forrester, Sóbester, and Keane (2008) draw an analogy to the process of devising a revision timetable before an exam. While a well-structured timetable enhances the effectiveness of revision, an excessive amount of the revision time itself should not be consumed by the planning phase.

A significant challenge in devising a sampling plan optimizer is ensuring that the search process remains confined to the space of valid Latin hypercubes. As previously discussed, the defining characteristic of a Latin hypercube X is that each of its columns represents a permutation of the possible levels for the corresponding variable. Consequently, the smallest modification that can be applied to a Latin hypercube—without compromising its crucial multidimensional stratification property—involves swapping two elements within any single column of X. A Python implementation for 'mutating' a Latin hypercube through such an operation, generalized to accommodate random changes applied to multiple sites, is provided below:

4.3.5.1. The Function `perturb()`

The function `perturb` randomly swaps elements in a Latin hypercube sampling plan. It takes a 2D array representing the sampling plan and performs a specified number of random element swaps, ensuring that the result remains a valid Latin hypercube.

```
def perturb(X: np.ndarray,
            PertNum: Optional[int] = 1) -> np.ndarray:
    """
    Args:
        X (np.ndarray):
            A 2D array (sampling plan) of shape (n, k),
            where each row is a point
            and each column is a dimension.
        PertNum (int, optional):
            The number of element swaps (perturbations)
            to perform. Defaults to 1.

    Returns:
        np.ndarray:
            The perturbed sampling plan,
            identical in shape to the input, with
            one or more random column swaps executed.
    """
    # Get dimensions of the plan
    n, k = X.shape
    if n < 2 or k < 2:
        raise ValueError("Latin hypercubes require at least 2 points and 2 dimensions"
    for _ in range(PertNum):
        # Pick a random column
        col = int(np.floor(np.random.rand() * k))
        # Pick two distinct row indices
        el1, el2 = 0, 0
        while el1 == el2:
```

```
            el1 = int(np.floor(np.random.rand() * n))
            el2 = int(np.floor(np.random.rand() * n))
        # Swap the two selected elements in the chosen column
        X[el1, col], X[el2, col] = X[el2, col], X[el1, col]
    return X
```

Example 4.12 (The Function `perturb()`). The `perturb` function can be used to randomly swap elements in a Latin hypercube sampling plan. The following code demonstrates how to use `perturb` to create a perturbed version of a 4x2 sampling plan:

```
X_original = np.array([[1, 3],[2, 4],[3, 1],[4, 2]])
print("Original Sampling Plan:")
print(X_original)
print("Perturbed Sampling Plan:")
X_perturbed = perturb(X_original, PertNum=1)
print(X_perturbed)
```

```
Original Sampling Plan:
[[1 3]
 [2 4]
 [3 1]
 [4 2]]
Perturbed Sampling Plan:
[[2 3]
 [1 4]
 [3 1]
 [4 2]]
```

Forrester, Sóbester, and Keane (2008) uses the term 'mutation', because this problem lends itself to nature-inspired computation. Morris and Mitchell (1995) use a simulated annealing algorithm, the detailed pseudocode of which can be found in their paper. As an alternative, a method based on evolutionary operation (EVOP) is offered by Forrester, Sóbester, and Keane (2008).

4.3.6. Evolutionary Operation

As introduced by G. E. P. Box (1957), evolutionary operation was designed to optimize chemical processes. The current parameters of the reaction would be recorded in a box at the centre of a board, with a series of 'offspring' boxes along the edges containing values of the parameters slightly altered with respect to the central, 'parent' values. Once the reaction was completed for all of these sets of variable values and the corresponding yields recorded, the contents of the central box would be replaced with

that of the setup with the highest yield and this would then become the parent of a new set of peripheral boxes.

This is generally viewed as a local search procedure, though this depends on the mutation step sizes, that is on the differences between the parent box and its offspring. The longer these steps, the more global is the scope of the search.

For the purposes of the Latin hypercube search, a variable scope strategy is applied. The process starts with a long step length (that is a relatively large number of swaps within the columns) and, as the search progresses, the current best basin of attraction is gradually approached by reducing the step length to a single change.

In each generation the parent is mutated (randomly, using the `perturb` function) a pertnum number of times. The sampling plan that yields the smallest Φ_q value (as per the Morris-Mitchell criterion, calculated using`mmphi`) among all offspring and the parent is then selected; in evolutionary computation parlance this selection philosophy is referred to as elitism.

The EVOP based search for space-filling Latin hypercubes is thus a truly evolutionary process: the optimized sampling plan results from the nonrandom survival of random variations.

4.3.7. Putting it all Together

All the pieces of the optimum Latin hypercube sampling process puzzle are now in place: the random hypercube generator as a starting point for the optimization process, the 'spacefillingness' metric that needs to be optimized, the optimization engine that performs this task and the comparison function that selects the best of the optima found for the various q's. These pieces just need to be put into a sequence. Here is the Python embodiment of the completed puzzle. It results in a function `bestlh` that uses the function `mmlhs` to find the best Latin hypercube sampling plan for a given set of parameters.

4.3.7.1. The Function `mmlhs`

Performs an evolutionary search (using perturbations) to find a Morris-Mitchell optimal Latin hypercube, starting from an initial plan X_start.

This function does the following:

1. Initializes a "best" Latin hypercube (`X_best`) from the provided `X_start`.
2. Iteratively perturbs `X_best` to create offspring.
3. Evaluates the space-fillingness of each offspring via the Morris-Mitchell metric (using `mmphi`).
4. Updates the best plan whenever a better offspring is found.

```python
def mmlhs(X_start: np.ndarray,
          population: int,
          iterations: int,
          q: Optional[float] = 2.0,
          plot=False) -> np.ndarray:
    """
    Args:
        X_start (np.ndarray):
            A 2D array of shape (n, k) providing the initial Latin hypercube
            (n points in k dimensions).
        population (int):
            Number of offspring to create in each generation.
        iterations (int):
            Total number of generations to run the evolutionary search.
        q (float, optional):
            The exponent used by the Morris-Mitchell space-filling criterion.
            Defaults to 2.0.
        plot (bool, optional):
            If True, a simple scatter plot of the first two dimensions will be
            displayed at each iteration. Only if k >= 2. Defaults to False.

    Returns:
        np.ndarray:
            A 2D array representing the most space-filling Latin hypercube found
            after all iterations, of the same shape as X_start.
    """
    n = X_start.shape[0]
    if n < 2:
        raise ValueError("Latin hypercubes require at least 2 points")
    k = X_start.shape[1]
    if k < 2:
        raise ValueError("Latin hypercubes are not defined for dim k < 2")
    # Initialize best plan and its metric
    X_best = X_start.copy()
    Phi_best = mmphi(X_best, q=q)
    # After 85% of iterations, reduce the mutation rate to 1
    leveloff = int(np.floor(0.85 * iterations))
    for it in range(1, iterations + 1):
        # Decrease number of mutations over time
        if it < leveloff:
            mutations = int(round(1 + (0.5 * n - 1) * (leveloff - it) / (leveloff - 1)))
        else:
            mutations = 1
        X_improved = X_best.copy()
```

```
        Phi_improved = Phi_best
        # Create offspring, evaluate, and keep the best
        for _ in range(population):
            X_try = perturb(X_best.copy(), mutations)
            Phi_try = mmphi(X_try, q=q)

            if Phi_try < Phi_improved:
                X_improved = X_try
                Phi_improved = Phi_try
        # Update the global best if we found a better plan
        if Phi_improved < Phi_best:
            X_best = X_improved
            Phi_best = Phi_improved
        # Simple visualization of the first two dimensions
        if plot and (X_best.shape[1] >= 2):
            plt.clf()
            plt.scatter(X_best[:, 0], X_best[:, 1], marker="o")
            plt.grid(True)
            plt.title(f"Iteration {it} - Current Best Plan")
            plt.pause(0.01)
    return X_best
```

Example 4.13 (The Function `mmlhs`)**.** The `mmlhs` function can be used to optimize a Latin hypercube sampling plan. The following code demonstrates how to use `mmlhs` to optimize a 4x2 Latin hypercube starting from an initial plan:

```
# Suppose we have an initial 4x2 plan
X_start = np.array([[0.1, 0.3],[.1, .4],[.2, .9],[.9, .2]])
print("Initial plan:")
print(X_start)
# Search for a more space-filling plan
X_opt = mmlhs(X_start, population=10, iterations=100, q=2)
print("Optimized plan:")
print(X_opt)
```

```
Initial plan:
[[0.1 0.3]
 [0.1 0.4]
 [0.2 0.9]
 [0.9 0.2]]
Optimized plan:
[[0.1 0.2]
 [0.1 0.9]
```

```
[0.2 0.4]
[0.9 0.3]]
```

Figure 4.9 shows the initial and optimized plans in 2D. The blue points represent the initial plan, while the red points represent the optimized plan.

Figure 4.9.: Comparison of the initial and optimized plans in 2D.

4.3.7.2. The Function `bestlh`

Generates an optimized Latin hypercube by evolving the Morris-Mitchell criterion across multiple exponents (q values) and selecting the best plan.

```
def bestlh(n: int,
           k: int,
           population: int,
           iterations: int,
           p=1,
```

```
        plot=False,
        verbosity=0,
        edges=0,
        q_list=[1, 2, 5, 10, 20, 50, 100]) -> np.ndarray:
"""
Args:
    n (int):
        Number of points required in the Latin hypercube.
    k (int):
        Number of design variables (dimensions).
    population (int):
        Number of offspring in each generation of the evolutionary search.
    iterations (int):
        Number of generations for the evolutionary search.
    p (int, optional):
        The distance norm to use. p=1 for Manhattan (L1), p=2 for Euclidean (L2).
        Defaults to 1 (faster than 2).
    plot (bool, optional):
        If True, a scatter plot of the optimized plan in the first two dimensions
        will be displayed. Only if k>=2.  Defaults to False.
    verbosity (int, optional):
        Verbosity level. 0 is silent, 1 prints the best q value found. Defaults to
    edges (int, optional):
        If 1, places centers of the extreme bins at the domain edges ([0,1]).
        Otherwise, bins are fully contained within the domain, i.e. midpoints.
        Defaults to 0.
    q_list (list, optional):
        A list of q values to optimize. Defaults to [1, 2, 5, 10, 20, 50, 100].
        These values are used to evaluate the space-fillingness of the Latin
        hypercube. The best plan is selected based on the lowest mmphi value.

Returns:
    np.ndarray:
        A 2D array of shape (n, k) representing an optimized Latin hypercube.
"""
if n < 2:
    raise ValueError("Latin hypercubes require at least 2 points")
if k < 2:
    raise ValueError("Latin hypercubes are not defined for dim k < 2")

# A list of exponents (q) to optimize

# Start with a random Latin hypercube
X_start = rlh(n, k, edges=edges)
```

```
    # Allocate a 3D array to store the results for each q
    # (shape: (n, k, number_of_q_values))
    X3D = np.zeros((n, k, len(q_list)))

    # Evolve the plan for each q in q_list
    for i, q_val in enumerate(q_list):
        if verbosity > 0:
            print(f"Now optimizing for q={q_val}...")
        X3D[:, :, i] = mmlhs(X_start, population, iterations, q_val)

    # Sort the set of evolved plans according to the Morris-Mitchell criterion
    index_order = mmsort(X3D, p=p)

    # index_order is a 1-based array of plan indices; the first element is the best
    best_idx = index_order[0] - 1
    if verbosity > 0:
        print(f"Best lh found using q={q_list[best_idx]}...")

    # The best plan in 3D array order
    X = X3D[:, :, best_idx]

    # Plot the first two dimensions
    if plot and (k >= 2):
        plt.scatter(X[:, 0], X[:, 1], c="r", marker="o")
        plt.title(f"Morris-Mitchell optimum plan found using q={q_list[best_idx]}")
        plt.xlabel("x_1")
        plt.ylabel("x_2")
        plt.grid(True)
        plt.show()

    return X
```

Example 4.14 (The Function `bestlh`)**.** The `bestlh` function can be used to generate an optimized Latin hypercube sampling plan. The following code demonstrates how to use `bestlh` to create a 5x2 Latin hypercube with a population of 5 and 10 iterations:

```
Xbestlh= bestlh(n=5, k=2, population=5, iterations=10)
```

Figure 4.10 shows the best Latin hypercube sampling in 2D. The red points represent the optimized plan.

Sorting all candidate plans in ascending order is not strictly necessary - after all, only the best one is truly of interest. Nonetheless, the added computational complexity is

Figure 4.10.: Best Latin Hypercube Sampling

minimal (the vector will only ever contain as many elements as there are candidate q values, and only an index array is sorted, not the actual repository of plans). This sorting gives the reader the opportunity to compare, if desired, how different choices of q influence the resulting plans.

4.4. Experimental Analysis of the Morris-Mitchell Criterion

Morris-Mitchell Criterion Experimental Analysis

- Number of points: 16, Dimensions: 2
- mmphi parameters: q (exponent) = 2.0, p (distance norm) = 2.0 (1=Manhattan, 2=Euclidean)

```
N_POINTS = 16
N_DIM = 2
RANDOM_SEED = 42
q = 2.0
p = 2.0
```

4.4.1. Evaluation of Sampling Designs

We generate various sampling designs and evaluate their space-filling properties using the Morris-Mitchell criterion.

```
designs = {}
if int(np.sqrt(N_POINTS))**2 == N_POINTS:
    grid_design = Grid(k=N_DIM)
    designs["Grid (4x4)"] = grid_design.generate_grid_design(points_per_dim=int(np.sqrt(N_POINTS)))
else:
    print(f"Skipping grid design as N_POINTS={N_POINTS} is not a perfect square for a simple 2D grid

lhs_design = SpaceFilling(k=N_DIM, seed=42)
designs["LHS"] = lhs_design.generate_qms_lhs_design(n_points=N_POINTS)

sobol_design = Sobol(k=N_DIM, seed=42)
designs["Sobol"] = sobol_design.generate_sobol_design(n_points=N_POINTS)

random_design = Random(k=N_DIM)
designs["Random"] = random_design.uniform(n_points=N_POINTS)
```

```
poor_design = Poor(k=N_DIM)
designs["Collinear"] = poor_design.generate_collinear_design(n_points=N_POINTS)

clustered_design = Clustered(k=N_DIM)
designs["Clustered (3 clusters)"] = clustered_design.generate_clustered_design(n_point

results = {}

print("Calculating Morris-Mitchell metric (smaller is better):")
for name, X_design in designs.items():
    metric_val = mmphi(X_design, q=q, p=p)
    results[name] = metric_val
    print(f"  {name}: {metric_val:.4f}")
```

```
Calculating Morris-Mitchell metric (smaller is better):
  Grid (4x4): 20.2617
  LHS: 28.1868
  Sobol: 41.8196
  Random: 40.5840
  Collinear: 87.8829
  Clustered (3 clusters): 90.3702
```

```
if N_DIM == 2:
    num_designs = len(designs)
    cols = 2
    rows = int(np.ceil(num_designs / cols))
    fig, axes = plt.subplots(rows, cols, figsize=(5 * cols, 5 * rows))
    axes = axes.ravel() # Flatten axes array for easy iteration

    for i, (name, X_design) in enumerate(designs.items()):
        ax = axes[i]
        ax.scatter(X_design[:, 0], X_design[:, 1], s=50, edgecolors='k', alpha=0.7)
        ax.set_title(f"{name}\nmmphi = {results[name]:.3f}", fontsize=10)
        ax.set_xlabel("X1")
        ax.set_ylabel("X2")
        ax.set_xlim(-0.05, 1.05)
        ax.set_ylim(-0.05, 1.05)
        ax.set_aspect('equal', adjustable='box')
        ax.grid(True, linestyle='--', alpha=0.6)

    # Hide any unused subplots
    for j in range(i + 1, len(axes)):
        fig.delaxes(axes[j])
```

```
plt.tight_layout()
plt.suptitle(f"Comparison of 2D Sampling Designs ({N_POINTS} points each)", fontsize=14, y=1.02)
plt.show()
```

Comparison of 2D Sampling Designs (16 points each)
Grid (4x4)
mmphi = 20.262
LHS
mmphi = 28.187
Sobol
mmphi = 41.820
Random
mmphi = 40.584
Collinear
mmphi = 87.883
Clustered (3 clusters)
mmphi = 90.370
X1
X2
0.0
0.2
0.4
0.6
0.8
1.0

4.4.2. Demonstrate the Impact of mmphi Parameters

Demonstrating Impact of mmphi Parameters on 'LHS' Design

```
X_lhs = designs["LHS"]

# 1. Default parameters (already calculated)
print(f"  LHS (q={q}, p={p} Euclidean): {results['LHS']:.4f}")

# 2. Change q (main exponent, literature's p or k)
q_high = 15.0
metric_lhs_q_high = mmphi(X_lhs, q=q_high, p=p)
print(f"  LHS (q={q_high}, p={p} Euclidean): {metric_lhs_q_high:.4f} (Higher q penalizes small dista

# 3. Change p (distance norm, literature's q or m)
p_manhattan = 1.0
metric_lhs_p_manhattan = mmphi(X_lhs, q=q, p=p_manhattan)
print(f"  LHS (q={q}, p={p_manhattan} Manhattan): {metric_lhs_p_manhattan:.4f} (Using L1 distance)")
```

```
  LHS (q=2.0, p=2.0 Euclidean): 28.1868
  LHS (q=15.0, p=2.0 Euclidean): 8.1573 (Higher q penalizes small distances more)
  LHS (q=2.0, p=1.0 Manhattan): 22.0336 (Using L1 distance)
```

4.4.3. Morris-Mitchell Criterion: Impact of Adding Points

Impact of adding a point to a 2x2 grid design

```
# Initial 2x2 Grid Design
X_initial = np.array([[0.0, 0.0], [1.0, 0.0], [0.0, 1.0], [1.0, 1.0]])
mmphi_initial = mmphi(X_initial, q=q, p=p)

print(f"Parameters: q (exponent) = {q}, p (distance) = {p} (Euclidean)\n")
print(f"Initial 2x2 Grid Design (4 points):")
print(f"  Points:\n{X_initial}")
print(f"  Morris-Mitchell Criterion (Phi_q): {mmphi_initial:.4f}\n")
```

```
Parameters: q (exponent) = 2.0, p (distance) = 2.0 (Euclidean)

Initial 2x2 Grid Design (4 points):
  Points:
[[0. 0.]
 [1. 0.]
```

```
 [0. 1.]
 [1. 1.]]
  Morris-Mitchell Criterion (Phi_q): 2.2361
```

Scenarios for adding a 5th point:

```
scenarios = {
    "Scenario 1: Add to Center": {
        "new_point": np.array([[0.5, 0.5]]),
        "description": "Adding a point in the center of the grid."
    },
    "Scenario 2: Add Close to Existing (Cluster)": {
        "new_point": np.array([[0.1, 0.1]]),
        "description": "Adding a point very close to an existing point (0,0)."
    },
    "Scenario 3: Add on Edge": {
        "new_point": np.array([[0.5, 0.0]]),
        "description": "Adding a point on an edge between (0,0) and (1,0)."
    }
}

results_summary = []
augmented_designs_for_plotting = {"Initial Design": X_initial}

for name, scenario_details in scenarios.items():
    new_point = scenario_details["new_point"]
    X_augmented = np.vstack((X_initial, new_point))
    augmented_designs_for_plotting[name] = X_augmented

    mmphi_augmented = mmphi(X_augmented, q=q, p=p)
    change = mmphi_augmented - mmphi_initial

    print(f"{name}:")
    print(f"  Description: {scenario_details['description']}")
    print(f"  New Point Added: {new_point}")
    # print(f"  Augmented Design (5 points):\n{X_augmented}") # Optional: print full r
    print(f"  Morris-Mitchell Criterion (Phi_q): {mmphi_augmented:.4f}")
    print(f"  Change from Initial Phi_q: {change:+.4f}\n")

    results_summary.append({
        "Scenario": name,
        "Initial Phi_q": mmphi_initial,
        "Augmented Phi_q": mmphi_augmented,
        "Change": change
    })
```

```
Scenario 1: Add to Center:
  Description: Adding a point in the center of the grid.
  New Point Added: [[0.5 0.5]]
  Morris-Mitchell Criterion (Phi_q): 3.6056
  Change from Initial Phi_q: +1.3695

Scenario 2: Add Close to Existing (Cluster):
  Description: Adding a point very close to an existing point (0,0).
  New Point Added: [[0.1 0.1]]
  Morris-Mitchell Criterion (Phi_q): 7.6195
  Change from Initial Phi_q: +5.3834

Scenario 3: Add on Edge:
  Description: Adding a point on an edge between (0,0) and (1,0).
  New Point Added: [[0.5 0. ]]
  Morris-Mitchell Criterion (Phi_q): 3.8210
  Change from Initial Phi_q: +1.5849
```

```
num_designs = len(augmented_designs_for_plotting)
cols = 2
rows = int(np.ceil(num_designs / cols))

fig, axes = plt.subplots(rows, cols, figsize=(6 * cols, 5 * rows))
axes = axes.ravel()

plot_idx = 0
# Plot initial design first
ax = axes[plot_idx]
ax.scatter(X_initial[:, 0], X_initial[:, 1], s=100, edgecolors='k', alpha=0.7, label="Original Point
ax.set_title(f"Initial Design\nPhi_q = {mmphi_initial:.3f}", fontsize=10)
ax.set_xlabel("X1")
ax.set_ylabel("X2")
ax.set_xlim(-0.1, 1.1)
ax.set_ylim(-0.1, 1.1)
ax.set_aspect('equal', adjustable='box')
ax.grid(True, linestyle='--', alpha=0.6)
ax.legend(fontsize='small')
plot_idx +=1

# Plot augmented designs
for name, X_design in augmented_designs_for_plotting.items():
    if name == "Initial Design":
```

```
        continue # Already plotted

    ax = axes[plot_idx]
    # Highlight original vs new point
    original_points = X_design[:-1, :]
    new_point = X_design[-1, :].reshape(1,2)

    ax.scatter(original_points[:, 0], original_points[:, 1], s=100, edgecolors='k', al
    ax.scatter(new_point[:, 0], new_point[:, 1], s=150, color='red', edgecolors='k', r

    current_phi_q = next(item['Augmented Phi_q'] for item in results_summary if item['
    ax.set_title(f"{name}\nPhi_q = {current_phi_q:.3f}", fontsize=10)
    ax.set_xlabel("X1")
    ax.set_ylabel("X2")
    ax.set_xlim(-0.1, 1.1)
    ax.set_ylim(-0.1, 1.1)
    ax.set_aspect('equal', adjustable='box')
    ax.grid(True, linestyle='--', alpha=0.6)
    ax.legend(fontsize='small')
    plot_idx +=1

# Hide any unused subplots
for j in range(plot_idx, len(axes)):
    fig.delaxes(axes[j])

plt.tight_layout(rect=[0, 0, 1, 0.96]) # Adjust layout to make space for suptitle
plt.suptitle(f"Impact of Adding a Point to a 2x2 Grid Design (q={q}, p={p})", fontsize
plt.show()
```

Summary Table (Conceptual):

Scenario	Initial Phi_q	Augmented Phi_q	Change
Baseline (2x2 Grid)	2.236	—	—
Scenario 1: Add to Center	2.236	3.606	+1.369
Scenario 2: Add Close to Existing (Cluster)	2.236	7.619	+5.383
Scenario 3: Add on Edge	2.236	3.821	+1.585

4.5. A Sample-Size Invariant Version of the Morris-Mitchell Criterion

4.5.1. Comparison of `mmphi()` and `mmphi_intensive()`

The Morris-Mitchell criterion is a widely used metric for evaluating the space-filling properties of Latin hypercube sampling designs. However, it is sensitive to the number of points in the design, which can lead to misleading comparisons between designs with different sample sizes. To address this issue, a sample-size invariant version of the Morris-Mitchell criterion has been proposed. It is avaiable in the `spotpython` package as `mmphi_intensive()`, see [SOURCE].

The functions `mmphi()` and `mmphi_intensive()` both calculate a Morris-Mitchell criterion, but they differ in their normalization, which makes `mmphi_intensive()` invariant to the sample size.

Let X be a sampling plan with n points $\{x_1, x_2, \dots, x_n\}$ in a k-dimensional space. Let $d_{ij} = \|x_i - x_j\|_p$ be the p-norm distance between points x_i and x_j. Let J_l be the multiplicity of the l-th unique distance d_l among all pairs of points in X. Let m be the total number of unique distances.

1. `mmphi()` (Morris-Mitchell Criterion Φ_q)

The `mmphi()` function, as defined in the context and implemented in `sampling.py`, calculates the Morris-Mitchell criterion Φ_q as:

$$\Phi_q(X) = \left(\sum_{l=1}^{m} J_l d_l^{-q} \right)^{1/q},$$

where:

- J_l is the number of pairs of points separated by the unique distance d_l.
- d_l are the unique pairwise distances.
- q is a user-defined exponent (typically $q > 0$).

This formulation is directly based on the sum of inverse powers of distances. The value of Φ_q is generally dependent on the number of points n in the design X, as the sum $\sum J_l d_l^{-q}$ will typically increase with more points (and thus more pairs).

2. `mmphi_intensive()` (Intensive Morris-Mitchell Criterion)

The `mmphi_intensive()` function, as implemented in `sampling.py` calculates a sample-size invariant version of the Morris-Mitchell criterion, which will be referred to as Φ_q^I. The formula is:

$$\Phi_q^I(X) = \left(\frac{1}{M} \sum_{l=1}^{m} J_l d_l^{-q} \right)^{1/q}$$

where:

- $M = \binom{n}{2} = \frac{n(n-1)}{2}$ is the total number of unique pairs of points in the design X.
- The other terms J_l, d_l, q are the same as in `mmphi()`.

The key mathematical difference is the normalization factor $\frac{1}{M}$ inside the parentheses before the outer exponent $1/q$ is applied.

- **`mmphi()`**: Calculates $(\text{SumTerm})^{1/q}$, where $\text{SumTerm} = \sum J_l d_l^{-q}$.
- **`mmphi_intensive()`**: Calculates $\left(\frac{\text{SumTerm}}{M}\right)^{1/q}$.

By dividing the sum $\sum J_l d_l^{-q}$ by M (the total number of pairs), `mmphi_intensive()` effectively calculates an *average* contribution per pair to the $-q$-th power of distance, before taking the q-th root. This normalization makes the criterion less dependent on the absolute number of points n and allows for more meaningful comparisons of space-fillingness between designs of different sizes. A smaller value indicates a better (more space-filling) design for both criteria.

4.5.2. Plotting the Two Morris-Mitchell Criteria for Different Sample Sizes

Figure 4.11 shows the comparison of the two Morris-Mitchell criteria for different sample sizes using the `plot_mmphi_vs_n_lhs` function. The red line represents the standard Morris-Mitchell criterion, while the blue line represents the sample-size invariant version. Note the difference in the y-axis scales, which highlights how the sample-size invariant version remains consistent across varying sample sizes.

```
def plot_mmphi_vs_n_lhs(k_dim: int,
                        seed: int,
                        n_min: int = 10,
                        n_max: int = 100,
                        n_step: int = 5,
                        q_phi: float = 2.0,
                        p_phi: float = 2.0):
    """
    Generates LHS designs for varying n, calculates mmphi and mmphi_intensive,
    and plots them against the number of samples (n).

    Args:
        k_dim (int): Number of dimensions for the LHS design.
```

```
        seed (int): Random seed for reproducibility.
        n_min (int): Minimum number of samples.
        n_max (int): Maximum number of samples.
        n_step (int): Step size for increasing n.
        q_phi (float): Exponent q for the Morris-Mitchell criteria.
        p_phi (float): Distance norm p for the Morris-Mitchell criteria.
    """
    n_values = list(range(n_min, n_max + 1, n_step))
    if not n_values:
        print("Warning: n_values list is empty. Check n_min, n_max, and n_step.")
        return
    mmphi_results = []
    mmphi_intensive_results = []
    lhs_generator = SpaceFilling(k=k_dim, seed=seed)
    print(f"Calculating for n from {n_min} to {n_max} with step {n_step}...")
    for n_points in n_values:
        if n_points < 2 : # mmphi requires at least 2 points to calculate distances
            print(f"Skipping n={n_points} as it's less than 2.")
            mmphi_results.append(np.nan)
            mmphi_intensive_results.append(np.nan)
            continue
        try:
            X_design = lhs_generator.generate_qms_lhs_design(n_points=n_points)
            phi = mmphi(X_design, q=q_phi, p=p_phi)
            phi_intensive, _, _ = mmphi_intensive(X_design, q=q_phi, p=p_phi)
            mmphi_results.append(phi)
            mmphi_intensive_results.append(phi_intensive)
        except Exception as e:
            print(f"Error calculating for n={n_points}: {e}")
            mmphi_results.append(np.nan)
            mmphi_intensive_results.append(np.nan)

    fig, ax1 = plt.subplots(figsize=(9, 6))

    color = 'tab:red'
    ax1.set_xlabel('Number of Samples (n)')
    ax1.set_ylabel('mmphi (Phiq)', color=color)
    ax1.plot(n_values, mmphi_results, color=color, marker='o', linestyle='-', label='r
    ax1.tick_params(axis='y', labelcolor=color)
    ax1.grid(True, linestyle='--', alpha=0.7)

    ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis
    color = 'tab:blue'
    ax2.set_ylabel('mmphi_intensive (PhiqI)', color=color)  # we already handled the x
```

```
    ax2.plot(n_values, mmphi_intensive_results, color=color, marker='x', linestyle='--', label='mmph
    ax2.tick_params(axis='y', labelcolor=color)

    fig.tight_layout()  # otherwise the right y-label is slightly clipped
    plt.title(f'Morris-Mitchell Criteria vs. Number of Samples (n)\nLHS (k={k_dim}, q={q_phi}, p={p_
    # Add legends
    lines, labels = ax1.get_legend_handles_labels()
    lines2, labels2 = ax2.get_legend_handles_labels()
    ax2.legend(lines + lines2, labels + labels2, loc='best')
    plt.show()
```

```
N_DIM = 2
RANDOM_SEED = 42
plot_mmphi_vs_n_lhs(k_dim=N_DIM, seed=RANDOM_SEED, n_min=10, n_max=100, n_step=5)
```

```
Calculating for n from 10 to 100 with step 5...
```

Figure 4.11.: Comparison of the two Morris-Mitchell Criteria for Different Sample Sizes

4.6. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

5. Constructing a Surrogate

> **Note**
>
> This section is based on chapter 2 in Forrester, Sóbester, and Keane (2008).

Definition 5.1 (Black Box Problem)**.** We are trying to learn a mapping that converts the vector $\vec{x}$ into a scalar output y, i.e., we are trying to learn a function

$$y = f(x).$$

If function is hidden ("lives in a black box"), so that the physics of the problem is not known, the problem is called a black box problem.

This black box could take the form of either a physical or computer experiment, for example, a finite element code, which calculates the maximum stress (σ) for given product dimensions ($\vec{x}$).

Definition 5.2 (Generic Solution)**.** The generic solution method is to collect the output values $y^{(1)}$, $y^{(2)}$, ..., $y^{(n)}$ that result from a set of inputs $\vec{x}^{(1)}$, $\vec{x}^{(2)}$, ..., $\vec{x}^{(n)}$ and find a best guess $\hat{f}(\vec{x})$ for the black box mapping f, based on these known observations.

5.1. Stage One: Preparing the Data and Choosing a Modelling Approach

The first step is the identification, through a small number of observations, of the inputs that have a significant impact on f; that is the determination of the shortest design variable vector $\vec{x} = \{x_1, x_2, \dots, x_k\}^T$ that, by sweeping the ranges of all of its variables, can still elicit most of the behavior the black box is capable of. The ranges of the various design variables also have to be established at this stage.

The second step is to recruit n of these k-vectors into a list

$$X = \{\vec{x}^{(1)}, \vec{x}^{(2)}, \dots, \vec{x}^{(n)}\}^T,$$

where each $\vec{x}^{(i)}$ is a k-vector. The corresponding responses are collected in a vector such that this represents the design space as thoroughly as possible.

In the surrogate modeling process, the number of samples n is often limited, as it is constrained by the computational cost (money and/or time) associated with obtaining each observation.

It is advisable to scale $\vec{x}$ at this stage into the unit cube $[0,1]^k$, a step that can simplify the subsequent mathematics and prevent multidimensional scaling issues.

We now focus on the attempt to learn f through data pairs

$$\{(\vec{x}^{(1)}, y^{(1)}), (\vec{x}^{(2)}, y^{(2)}), \dots, (\vec{x}^{(n)}, y^{(n)})\}.$$

This supervised learning process essentially involves searching across the space of possible functions $\hat{f}$ that would replicate observations of f. This space of functions is infinite. Any number of hypersurfaces could be drawn to pass through or near the known observations, accounting for experimental error. However, most of these would generalize poorly; they would be practically useless at predicting responses at new sites, which is the ultimate goal.

Example 5.1 (The Needle(s) in the Haystack Function)**.** An extreme example is the 'needle(s) in the haystack' function:

$$f(x) = \begin{cases} y^{(1)}, & \text{if } x = \vec{x}^{(1)} \\ y^{(2)}, & \text{if } x = \vec{x}^{(2)} \\ \vdots & \\ y^{(n)}, & \text{if } x = \vec{x}^{(n)} \\ 0, & \text{otherwise.} \end{cases}$$

While this predictor reproduces all training data, it seems counter-intuitive and unsettling to predict 0 everywhere else for most engineering functions. Although there is a small chance that the function genuinely resembles the equation above and we sampled exactly where the needles are, it is highly unlikely.

There are countless other configurations, perhaps less contrived, that still generalize poorly. This suggests a need for systematic means to filter out nonsensical predictors. In our approach, we embed the structure of f into the model selection algorithm and search over its parameters to fine-tune the approximation to observations. For instance, consider one of the simplest models,

$$f(x, \vec{w}) = \vec{w}^T \vec{x} + v. \tag{5.1}$$

Learning f with this model implies that its structure—a hyperplane—is predetermined, and the fitting process involves finding the $k+1$ parameters (the slope vector $\vec{w}$ and the intercept v) that best fit the data. This will be accomplished in Stage Two.

Complicating this further is the noise present in observed responses (we assume design vectors $\vec{x}$ are not corrupted). Here, we focus on learning from such data, which sometimes risks overfitting.

Definition 5.3 (Overfitting)**.** Overfitting occurs when the model becomes too flexible and captures not only the underlying trend but also the noise in the data.

In the surrogate modeling process, the second stage as described in Section 5.2, addresses this issue of complexity control by estimating the parameters of the fixed structure model. However, foresight is necessary even at the model type selection stage.

Model selection often involves physics-based considerations, where the modeling technique is chosen based on expected underlying responses.

Example 5.2 (Model Selection)**.** Modeling stress in an elastically deformed solid due to small strains may justify using a simple linear approximation. Without insights into the physics, and if one fails to account for the simplicity of the data, a more complex and excessively flexible model may be incorrectly chosen. Although parameter estimation might still adjust the approximation to become linear, an opportunity to develop a simpler and robust model may be lost.

- Simple linear (or polynomial) models, despite their lack of flexibility, have advantages like applicability in further symbolic computations.
- Conversely, if we incorrectly assume a quadratic process when multiple peaks and troughs exist, the parameter estimation stage will not compensate for an unsuitable model choice. A quadratic model is too rigid to fit a multimodal function, regardless of parameter adjustments.

5.2. Stage Two: Parameter Estimation and Training

Assuming that Stage One helped identify the k critical design variables, acquire the learning data set, and select a generic model structure $f(\vec{x}, \vec{w})$, the task now is to estimate parameters $\vec{w}$ to ensure the model fits the data optimally. Among several estimation criteria, we will discuss two methods here.

Definition 5.4 (Maximum Likelihood Estimation)**.** Given a set of parameters $\vec{w}$, the model $f(\vec{x}, \vec{w})$ allows computation of the probability of the data set

$$\{(\vec{x}^{(1)}, y^{(1)} \pm \epsilon), (\vec{x}^{(2)}, y^{(2)} \pm \epsilon), \dots, (\vec{x}^{(n)}, y^{(n)} \pm \epsilon)\}$$

resulting from f (where ϵ is a small error margin around each data point).

Maximum Likelihood Estimation

Section 25.18 presents a more detailed discussion of the maximum likelihood estimation (MLE) method.

Taking Equation 25.36 and assuming errors ϵ are independently and normally distributed with standard deviation σ, the probability of the data set is given by:

$$P = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left[-\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(y^{(i)} - f(\vec{x}^{(i)}, \vec{w})\right)^2 \epsilon\right].$$

Intuitively, this is equivalent to the likelihood of the parameters given the data. Accepting this intuitive relationship as a mathematical one aids in model parameter estimation. This is achieved by maximizing the likelihood or, more conveniently, minimizing the negative of its natural logarithm:

$$\min_{\vec{w}} \sum_{i=1}^{n} \frac{[y^{(i)} - f(\vec{x}^{(i)}, \vec{w})]^2}{2\sigma^2} + \frac{n}{2}\ln\epsilon. \tag{5.2}$$

If we assume σ and ϵ are constants, Equation 5.2 simplifies to the well-known least squares criterion:

$$\min_{\vec{w}} \sum_{i=1}^{n} [y^{(i)} - f(\vec{x}^{(i)}, \vec{w})]^2.$$

Cross-validation is another method used to estimate model performance.

Definition 5.5 (Cross-Validation)**.** Cross-validation splits the data randomly into q roughly equal subsets, and then cyclically removing each subset and fitting the model to the remaining $q-1$ subsets. A loss function L is then computed to measure the error between the predictor and the withheld subset for each iteration, with contributions summed over all q iterations. More formally, if a mapping $\theta : \{1, \dots, n\} \to \{1, \dots, q\}$ describes the allocation of the n training points to one of the q subsets and $f^{(-\theta(i))}(\vec{x})$ is the predicted value by removing the subset $\theta(i)$ (i.e., the subset where observation i belongs), the cross-validation measure, used as an estimate of prediction error, is:

$$CV = \frac{1}{n}\sum_{i=1}^{n} L(y^{(i)}, f^{(-\theta(i))}(\vec{x}^{(i)})). \tag{5.3}$$

Introducing the squared error as the loss function and considering our generic model f still dependent on undetermined parameters, we write Equation 5.3 as:

$$CV = \frac{1}{n}\sum_{i=1}^{n} [y^{(i)} - f^{(-\theta(i))}(\vec{x}^{(i)})]^2. \tag{5.4}$$

The extent to which Equation 5.4 is an unbiased estimator of true risk depends on q. It is shown that if $q = n$, the leave-one-out cross-validation (LOOCV) measure is

almost unbiased. However, LOOCV can have high variance because subsets are very similar. Hastie, Tibshirani, and Friedman (2017)) suggest using compromise values like $q = 5$ or $q = 10$. Using fewer subsets also reduces the computational cost of the cross-validation process, see also Arlot, Celisse, et al. (2010) and Kohavi (1995).

5.3. Stage Three: Model Testing

If there is a sufficient amount of observational data, a random subset should be set aside initially for model testing. Hastie, Tibshirani, and Friedman (2017) recommend setting aside approximately $0.25n$ of $\vec{x} \to y$ pairs for testing purposes. These observations must remain untouched during Stages One and Two, as their sole purpose is to evaluate the testing error—the difference between true and approximated function values at the test sites—once the model has been built. Interestingly, if the main goal is to construct an initial surrogate for seeding a global refinement criterion-based strategy (as discussed in Section 3.2 in Forrester, Sóbester, and Keane (2008)), the model testing phase might be skipped.

It is noted that, ideally, parameter estimation (Stage Two) should also rely on a separate subset. However, observational data is rarely abundant enough to afford this luxury (if the function is cheap to evaluate and evaluation sites are selectable, a surrogate model might not be necessary).

When data are available for model testing and the primary objective is a globally accurate model, using either a root mean square error (RMSE) metric or the correlation coefficient (r^2) is recommended. To test the model, a test data set of size n_t is used alongside predictions at the corresponding locations to calculate these metrics.

The RMSE is defined as follows:

Definition 5.6 (Root Mean Square Error (RMSE))**.**

$$\text{RMSE} = \sqrt{\frac{1}{n_t}\sum_{i=1}^{n_t}(y^{(i)} - \hat{y}^{(i)})^2},$$

Ideally, the RMSE should be minimized, acknowledging its limitation by errors in the objective function f calculation. If the error level is known, like a standard deviation, the aim might be to achieve an RMSE within this value. Often, the target is an RMSE within a specific percentage of the observed data's objective value range.

The squared correlation coefficient r, see Equation 25.13, between the observed y and predicted $\hat{y}$ values can be computed as:

$$r^2 = \left(\frac{\text{cov}(y, \hat{y})}{\sqrt{\text{var}(y)\text{var}(\hat{y})}}\right)^2, \tag{5.5}$$

Equation 5.5 and can be expanded as:

$$r^2 = \left(\frac{n_t \sum_{i=1}^{n_t} y^{(i)} \hat{y}^{(i)} - \sum_{i=1}^{n_t} y^{(i)} \sum_{i=1}^{n_t} \hat{y}^{(i)}}{\sqrt{\left(n_t \sum_{i=1}^{n_t} (y^{(i)})^2 - \left(\sum_{i=1}^{n_t} y^{(i)} \right)^2 \right) \left(n_t \sum_{i=1}^{n_t} (\hat{y}^{(i)})^2 - \left(\sum_{i=1}^{n_t} \hat{y}^{(i)} \right)^2 \right)}} \right)^2 .$$

The correlation coefficient r^2 does not require scaling the data sets and only compares landscape shapes, not values. An $r^2 > 0.8$ typically indicates a surrogate with good predictive capability.

The methods outlined provide quantitative assessments of model accuracy, yet visual evaluations can also be insightful. In general, the RMSE will not reach zero but will stabilize around a low value. At this point, the surrogate model is saturated with data, and further additions do not enhance the model globally (though local improvements can occur at newly added points if using an interpolating model).

Example 5.3 (The Tea and Sugar Analogy)**.** Forrester, Sóbester, and Keane (2008) illustrates this saturation point using a comparison with a cup of tea and sugar. The tea represents the surrogate model, and sugar represents data. Initially, the tea is unsweetened, and adding sugar increases its sweetness. Eventually, a saturation point is reached where no more sugar dissolves, and the tea cannot get any sweeter. Similarly, a more flexible model, like one with additional parameters or employing interpolation rather than regression, can increase the saturation point—akin to making a hotter cup of tea for dissolving more sugar.

5.4. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

6. Response Surface Methods

This part deals with numerical implementations of optimization methods. The goal is to understand the implementation of optimization methods and to solve real-world problems numerically and efficiently. We will focus on the implementation of surrogate models, because they are the most efficient way to solve real-world problems.

Starting point is the well-established response surface methodology (RSM). It will be extended to the design and analysis of computer experiments (DACE). The DACE methodology is a modern extension of the response surface methodology. It is based on the use of surrogate models, which are used to replace the real-world problem with a simpler problem. The simpler problem is then solved numerically. The solution of the simpler problem is then used to solve the real-world problem.

! Numerical methods: Goals

- Understand implementation of optimization methods
- Solve real-world problems numerically and efficiently

6.1. What is RSM?

Response Surface Methods (RSM) refer to a collection of statistical and mathematical tools that are valuable for developing, improving, and optimizing processes. The overarching theme of RSM involves studying how input variables that control a product or process can potentially influence a response that measures performance or quality characteristics.

The advantages of RSM include a rich literature, well-established methods often used in manufacturing, the importance of careful experimental design combined with a well-understood model, and the potential to add significant value to scientific inquiry, process refinement, optimization, and more. However, there are also drawbacks to RSM, such as the use of simple and crude surrogates, the hands-on nature of the methods, and the limitation of local methods.

RSM is related to various fields, including Design of Experiments (DoE), quality management, reliability, and productivity. Its applications are widespread in industry and manufacturing, focusing on designing, developing, and formulating new products and improving existing ones, as well as from laboratory research. RSM is commonly

applied in domains such as materials science, manufacturing, applied chemistry, climate science, and many others.

An example of RSM involves studying the relationship between a response variable, such as yield (y) in a chemical process, and two process variables: reaction time (ξ_1) and reaction temperature (ξ_2). The provided code illustrates this scenario, following a variation of the so-called "banana function."

In the context of visualization, RSM offers the choice between 3D plots and contour plots. In a 3D plot, the independent variables ξ_1 and ξ_2 are represented, with y as the dependent variable.

```
import numpy as np
import matplotlib.pyplot as plt

def fun_rosen(x1, x2):
    b = 10
    return (x1-1)**2 + b*(x2-x1**2)**2

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x = np.arange(-2.0, 2.0, 0.05)
y = np.arange(-1.0, 3.0, 0.05)
X, Y = np.meshgrid(x, y)
zs = np.array(fun_rosen(np.ravel(X), np.ravel(Y)))
Z = zs.reshape(X.shape)

ax.plot_surface(X, Y, Z)

ax.set_xlabel('X1')
ax.set_ylabel('X2')
ax.set_zlabel('Y')

plt.show()
```

- contour plot example:
  - x_1 and x_2 are the independent variables
  - y is the dependent variable

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-1.0, 3.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_rosen(X1, X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y , 50)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title("Rosenbrock's Banana Function")
```

```
Text(0.5, 1.0, "Rosenbrock's Banana Function")
```

- Visual inspection: yield is optimized near $(\xi_1.\xi_2)$

6.1.1. Visualization: Problems in Practice

- True response surface is unknown in practice
- When yield evaluation is not as simple as a toy banana function, but a process requiring care to monitor, reconfigure and run, it's far too expensive to observe over a dense grid
- And, measuring yield may be a noisy/inexact process
- That's where stats (RSM) comes in

6.1.2. RSM: Strategies

- RSMs consist of experimental strategies for
- **exploring** the space of the process (i.e., independent/input) variables (above ξ_1 and $\xi 2$)
- empirical statistical **modeling** targeted toward development of an appropriate approximating relationship between the response (yield) and process variables local to a study region of interest

- **optimization** methods for sequential refinement in search of the levels or values of process variables that produce desirable responses (e.g., that maximize yield or explain variation)
- RSM used for fitting an Empirical Model
- True response surface driven by an unknown physical mechanism
- Observations corrupted by noise
- Helpful: fit an empirical model to output collected under different process configurations
- Consider response Y that depends on controllable input variables $\xi_1, \xi_2, \dots, \xi_m$
- RSM: Equations of the Empirical Model
  - $Y = f(\xi_1, \xi_2, \dots, \xi_m) + \epsilon$
  - $\mathbb{E}\{Y\} = \eta = f(\xi 1_1, \xi_2, \dots, \xi_m)$
  - ϵ is treated as zero mean idiosyncratic noise possibly representing
    - inherent variation, or
    - the effect of other systems or
    - variables not under our purview at this time

6.1.3. RSM: Noise in the Empirical Model

- Typical simplifying assumption: $\epsilon \sim N(0, \sigma^2)$
- We seek estimates for f and σ^2 from noisy observations Y at inputs ξ

6.1.4. RSM: Natural and Coded Variables

- Inputs $\xi_1, \xi_2, \dots, \xi_m$ called **natural variables**:
  - expressed in natural units of measurement, e.g., degrees Celsius, pounds per square inch (psi), etc.
- Transformed to **coded variables** $x_1, x_2, \dots, x_m$:
  - to mitigate hassles and confusion that can arise when working with a multitude of scales of measurement
- Typical **Transformations** offering dimensionless inputs $x_1, x_2, \dots, x_m$
  - in the unit cube, or
  - scaled to have a mean of zero and standard deviation of one, are common choices.
- Empirical model becomes $\eta = f(x_1, x_2, \dots, x_m)$

6.1.5. RSM Low-order Polynomials

- Low-order polynomial make the following simplifying Assumptions
  - Learning about f is lots easier if we make some simplifying approximations
  - Appealing to **Taylor's theorem**, a low-order polynomial in a small, localized region of the input (x) space is one way forward
  - Classical RSM:
    * disciplined application of **local analysis** and
    * **sequential refinement** of locality through conservative extrapolation
  - Inherently a **hands-on process**

6.2. First-Order Models (Main Effects Model)

- **First-order model** (sometimes called main effects model) useful in parts of the input space where it's believed that there's little curvature in f:

$$\eta = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

- For example:

$$\eta = 50 + 8x_1 + 3x_2$$

- In practice, such a surface would be obtained by fitting a model to the outcome of a designed experiment
- First-Order Model in python Evaluated on a Grid
- Evaluate model on a grid in a double-unit square centered at the origin
- Coded units are chosen arbitrarily, although one can imagine deploying this approximating function nearby $x^{(0)} = (0, 0)$

```
def fun_1(x1,x2):
    return 50 + 8*x1 + 3*x2
```

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-1.0, 1.0, delta)
x2 = np.arange(-1.0, 1.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_1(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('First Order Model: $50 + 8x_1 + 3x_2$')
```

```
Text(0.5, 1.0, 'First Order Model: $50 + 8x_1 + 3x_2$')
```

6.2.1. First-Order Model Properties

- First-order model in 2d traces out a **plane** in $y \times (x_1, x_2)$ space
- Only be appropriate for the most trivial of response surfaces, even when applied in a highly localized part of the input space
- Adding **curvature** is key to most applications:
  - First-order model with **interactions** induces limited degree of curvature via different rates of change of y as x_1 is varied for fixed x_2, and vice versa:

$$\eta = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_{12}$$

- For example $\eta = 50 + 8x_1 + 3x_2 - 4x_1 x_2$

6.2.2. First-order Model with Interactions in python

- Code below facilitates evaluations for pairs (x_1, x_2)
- Responses may be observed over a mesh in the same double-unit square

```
def fun_11(x1,x2):
    return 50 + 8 * x1 + 3 * x2 - 4 * x1 * x2
```

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_11(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('First Order Model with Interactions')
```

```
Text(0.5, 1.0, 'First Order Model with Interactions')
```

6.2.3. Observations: First-Order Model with Interactions

- Mean response η is increasing marginally in both x_1 and x_2, or conditional on a fixed value of the other until x_1 is 0.75
- Rate of increase slows as both coordinates grow simultaneously since the coefficient in front of the interaction term x_1x_2 is negative
- Compared to the first-order model (without interactions): surface is far more useful locally
- Least squares regressions often flag up significant interactions when fit to data collected on a design far from local optima

6.3. Second-Order Models

- Second-order model may be appropriate near local optima where f would have substantial curvature:

$$\eta = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{11} x_1^2 + \beta_{22} x^2 + \beta_{12} x_1 x_2$$

- For example

$$\eta = 50 + 8x_1 + 3x_2 - 7x_1^2 - 3x_2^2 - 4x_1x_2$$

- Implementation of the Second-Order Model as `fun_2()`.

```
def fun_2(x1,x2):
    return 50 + 8 * x1 + 3 * x2 - 7 * x1**2 - 3*x2**2 - 4 * x1 * x2
```

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_2(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Second Order Model with Interactions. Maximum near about $(0.6,0.2)$')
```

```
Text(0.5, 1.0, 'Second Order Model with Interactions. Maximum near about $(0.6,0.2)$')
```

6.3.1. Second-Order Models: Properties

- Not all second-order models would have a single stationary point (in RSM jargon called "a simple maximum")
- In "yield maximizing" setting we're presuming response surface is **concave** down from a global viewpoint
  - even though local dynamics may be more nuanced
- Exact criteria depend upon the eigenvalues of a certain matrix built from those coefficients
- Box and Draper (2007) provide a diagram categorizing all of the kinds of second-order surfaces in RSM analysis, where finding local maxima is the goal

6.3.2. Example: Stationary Ridge

- Example set of coefficients describing what's called a **stationary ridge** is provided by the code below

```
def fun_ridge(x1, x2):
    return 80 + 4*x1 + 8*x2 - 3*x1**2 - 12*x2**2 - 12*x1*x2
```

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_ridge(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Example of a stationary ridge')
```

```
Text(0.5, 1.0, 'Example of a stationary ridge')
```

6.3.3. Observations: Second-Order Model (Ridge)

- **Ridge**: a whole line of stationary points corresponding to maxima
- Situation means that the practitioner has some flexibility when it comes to optimizing:

- can choose the precise setting of (x_1, x_2) either arbitrarily or (more commonly) by consulting some tertiary criteria

6.3.4. Example: Rising Ridge

- An example of a rising ridge is implemented by the code below.

```
def fun_ridge_rise(x1, x2):
     return 80 - 4*x1 + 12*x2 - 3*x1**2 - 12*x2**2 - 12*x1*x2
```

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_ridge_rise(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Rising ridge: $\\eta = 80 + 4x_1 + 8x_2 - 3x_1^2 - 12x_2^2 - 12x_1x_2$')
```

```
Text(0.5, 1.0, 'Rising ridge: $\\eta = 80 + 4x_1 + 8x_2 - 3x_1^2 - 12x_2^2 - 12x_1x_2$
```

6.3.5. Summary: Rising Ridge

- The stationary point is remote to the study region
- Ccontinuum of (local) stationary points along any line going through the 2d space, excepting one that lies directly on the ridge
- Although estimated response will increase while moving along the axis of symmetry toward its stationary point, this situation indicates
  - either a poor fit by the approximating second-order function, or
  - that the study region is not yet precisely in the vicinity of a local optima—often both.

6.3.6. Falling Ridge

- Inversion of a rising ridge is a falling ridge
- Similarly indicating one is far from local optima, except that the response decreases as you move toward the stationary point
- Finding a falling ridge system can be a back-to-the-drawing-board affair.

6.3.7. Saddle Point

- Finally, we can get what's called a saddle or minimax system.

```
def fun_saddle(x1, x2):
    return 80 + 4*x1 + 8*x2 - 2*x2**2 - 12*x1*x2
```

```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_saddle(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Saddle Point: $\\eta = 80 + 4x_1 + 8x_2 - 2x_2^2 - 12x_1x_2$')
```

```
Text(0.5, 1.0, 'Saddle Point: $\\eta = 80 + 4x_1 + 8x_2 - 2x_2^2 - 12x_1x_2$')
```

6.3.8. Interpretation: Saddle Points

- Likely further data collection, and/or outside expertise, is needed before determining a course of action in this situation

6.3.9. Summary: Ridge Analysis

- Finding a simple maximum, or stationary ridge, represents ideals in the spectrum of second-order approximating functions
- But getting there can be a bit of a slog
- Using models fitted from data means uncertainty due to noise, and therefore uncertainty in the type of fitted second-order model
- A ridge analysis attempts to offer a principled approach to navigating uncertainties when one is seeking local maxima
- The two-dimensional setting exemplified above is convenient for visualization, but rare in practice
- Complications compound when studying the effect of more than two process variables

6.4. General RSM Models

- General **first-order model** on m process variables $x_1, x_2, \cdots, x_m$ is

$$\eta = \beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m$$

- General **second-order model** on m process variables

$$\eta = \beta_0 + \sum_{j=1}^{m} + \sum_{j=1}^{m} x_j^2 + \sum_{j=2}^{m} \sum_{k=1}^{j} \beta_{kj} x_k x_j.$$

6.4.1. Ordinary Least Squares

- Inference from data is carried out by **ordinary least squares** (OLS)
- For an excellent review including R examples, see Sheather (2009)
- OLS and maximum likelihood estimators (MLEs) are in the typical Gaussian linear modeling setup basically equivalent

6.5. General Linear Regression

We are considering a model, which can be written in the form

$$Y = X\beta + \epsilon,$$

where Y is an $(n \times 1)$ vector of observations (responses), X is an $(n \times p)$ matrix of known form, β is a $(1 \times p)$ vector of unknown parameters, and ϵ is an $(n \times 1)$ vector of errors. Furthermore, $E(\epsilon) = 0$, $Var(\epsilon) = \sigma^2 I$ and the ϵ_i are uncorrelated.

Using the normal equations

$$(X'X)b = X'Y,$$

the solution is given by

$$b = (X'X)^{-1}X'Y.$$

Example 6.1 (Linear Regression)**.**

```
import numpy as np
n = 8
X = np.linspace(0, 2*np.pi, n, endpoint=False).reshape(-1,1)
print(np.round(X, 2))
y = np.sin(X)
print(np.round(y, 2))
# fit an OLS model to the data, predict the response based on the 1ßß x values
m = 100
x = np.linspace(0, 2*np.pi, m, endpoint=False).reshape(-1,1)
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X, y)
y_pred = model.predict(x)
# visualize the data and the fitted model
import matplotlib.pyplot as plt
plt.scatter(X, y, color='black')
plt.plot(x, y_pred, color='blue', linewidth=1)
# add the ground truth (sine function) in orange
plt.plot(x, np.sin(x), color='orange', linewidth=1)
plt.show()
```

```
[[0.  ]
 [0.79]
 [1.57]
 [2.36]
```

```
 [3.14]
 [3.93]
 [4.71]
 [5.5 ]]
[[ 0.  ]
 [ 0.71]
 [ 1.  ]
 [ 0.71]
 [ 0.  ]
 [-0.71]
 [-1.  ]
 [-0.71]]
```

6.6. Designs

- Important: Organize the data collection phase of a response surface study carefully
- **Design**: choice of x's where we plan to observe y's, for the purpose of approximating f
- Analyses and designs need to be carefully matched
- When using a first-order model, some designs are preferred over others
- When using a second-order model to capture curvature, a different sort of design is appropriate

- Design choices often contain features enabling modeling assumptions to be challenged
  - e.g., to check if initial impressions are supported by the data ultimately collected

6.6.1. Different Designs

- **Screening desings**: determine which variables matter so that subsequent experiments may be smaller and/or more focused
- Then there are designs tailored to the form of model (first- or second-order, say) in the screened variables
- And then there are more designs still

6.7. RSM Experimentation

6.7.1. First Step

- RSM-based experimentation begins with a **first-order model**, possibly with interactions
- Presumption: current process operating **far from optimal** conditions
- Collect data and apply **method of steepest ascent** (gradient) on fitted surfaces to move to the optimum

6.7.2. Second Step

- Eventually, if all goes well after several such carefully iterated refinements, **second-order models** are used on appropriate designs in order to zero-in on ideal operating conditions
- Careful analysis of the fitted surface:
  - Ridge analysis with further refinement using gradients of, and
  - standard errors associated with, the fitted surfaces, and so on

6.7.3. Third Step

- Once the practitioner is satisfied with the full arc of
  - design(s),
  - fit(s), and
  - decision(s):

- A small experiment called **confirmation test** may be performed to check if the predicted optimal settings are realizable in practice

6.8. RSM: Review and General Considerations

- First Glimpse, RSM seems sensible, and pretty straightforward as quantitative statistics-based analysis goes
- But: RSM can get complicated, especially when input dimensions are not very low
- Design considerations are particularly nuanced, since the goal is to obtain reliable estimates of main effects, interaction, and curvature while minimizing sampling effort/expense
- RSM Downside: Inefficiency
  - Despite intuitive appeal, several RSM downsides become apparent upon reflection
  - Problems in practice
  - Stepwise nature of sequential decision making is inefficient:
    - Not obvious how to re-use or update analysis from earlier phases, or couple with data from other sources/related experiments
- RSM Downside: Locality
  - In addition to being local in experiment-time (stepwise approach), it's local in experiment-space
  - Balance between
    - exploration (maybe we're barking up the wrong tree) and
    - exploitation (let's make things a little better) is modest at best
- RSM Downside: Expert Knowledge
  - Interjection of expert knowledge is limited to hunches about relevant variables (i.e., the screening phase), where to initialize search, how to design the experiments
  - Yet at the same time classical RSMs rely heavily on constant examination throughout stages of modeling and design and on the instincts of seasoned practitioners
- RSM Downside: Replicability
  - Parallel analyses, conducted according to the same best intentions, rarely lead to the same designs, model fits and so on
  - Sometimes that means they lead to different conclusions, which can be cause for concern

6.8.1. Historical Considerations about RSM

- In spite of those criticisms, however, there was historically little impetus to revise the status quo
- Classical RSM was comfortable in its skin, consistently led to improvements or compelling evidence that none can reasonably be expected
- But then in the late 20th century came an explosive expansion in computational capability, and with it a means of addressing many of those downsides

6.8.2. Status Quo

- Nowadays, field experiments and statistical models, designs and optimizations are coupled with with mathematical models
- Simple equations are not regarded as sufficient to describe real-world systems anymore
- Physicists figured that out fifty years ago; industrial engineers followed, biologists, social scientists, climate scientists and weather forecasters, etc.
- Systems of equations are required, solved over meshes (e.g., finite elements), or stochastically interacting agents
- Goals for those simulation experiments are as diverse as their underlying dynamics
- Optimization of systems is common, e.g., to identify worst-case scenarios

6.8.3. The Role of Statistics

- Solving systems of equations, or interacting agents, requires computing
- Statistics involved at various stages:
  - choosing the mathematical model
  - solving by stochastic simulation (Monte Carlo)
  - designing the computer experiment
  - smoothing over idiosyncrasies or noise
  - finding optimal conditions, or
  - calibrating mathematical/computer models to data from field experiments

6.8.4. New RSM is needed: DACE

- Classical RSMs are not well-suited to any of those tasks, because
  - they lack the fidelity required to model these data
  - their intended application is too local
  - they're also too hands-on.

- Once computers are involved, a natural inclination is to automate—to remove humans from the loop and set the computer running on the analysis in order to maximize computing throughput, or minimize idle time
- **Design and Analysis of Computer Experiments** as a modern extension of RSM
- Experimentation is changing due to advances in machine learning
- **Gaussian process** (GP) regression is the canonical surrogate model
- Origins in geostatistics (gold mining)
- Wide applicability in contexts where prediction is king
- Machine learners exposed GPs as powerful predictors for all sorts of tasks:
- from regression to classification,
- active learning/sequential design,
- reinforcement learning and optimization,
- latent variable modeling, and so on

6.9. Exercises

1. Generate 3d Plots for the Contour Plots in this notebook.
2. Write a `plot_3d` function, that takes the objective function `fun` as an argument.

- It should provide the following interface: `plot_3d(fun)`.

3. Write a `plot_contour` function, that takes the objective function `fun` as an argument:

- It should provide the following interface: `plot_contour(fun)`.

4. Consider further arguments that might be useful for both function, e.g., ranges, size, etc.

6.10. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

7. Polynomial Models

> **Note**
>
> - This section is based on chapter 2.2 in Forrester, Sóbester, and Keane (2008).
> - The following Python packages are imported:

```
import numpy as np
import matplotlib.pyplot as plt
```

7.1. Fitting a Polynomial

We will consider one-variable cases, i.e., $k = 1$, first.

Let us consider the scalar-valued function $f : \mathbb{R} \to \mathbb{R}$ observed according to the sampling plan $X = \{x^{(1)}, x^{(2)} \dots, x^{(n)}\}^T$, yielding the responses $\vec{y} = \{y^{(1)}, y^{(2)}, \dots, y^{(n)}\}^T$.

A polynomial approximation of f of order m can be written as:

$$\hat{f}(x, m, \vec{w}) = \sum_{i=0}^{m} w_i x^i.$$

In the spirit of the earlier discussion of maximum likelihood parameter estimation, we seek to estimate $w = w_0, w_1, \dots, {w_m}^T$ through a least squares solution of:

$$\Phi \vec{w} = \vec{y}$$

where Φ is the Vandermonde matrix:

$$\Phi = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix}.$$

The maximum likelihood estimate of w is given by:

$$\vec{w} = (\Phi^T\Phi)^{-1}\Phi^T y,$$

where $\Phi^+ = (\Phi^T\Phi)^{-1}\Phi^T$ is the Moore-Penrose pseudo-inverse of Φ (see Section 10.3).

The polynomial approximation of order m is essentially a truncated Taylor series expansion. While higher values of m yield more accurate approximations, they risk overfitting the noise in the data.

To prevent this, we estimate m using cross-validation. This involves minimizing the cross-validation error over a discrete set of possible orders m (e.g., $m \in 1, 2, \ldots, 15$).

For each m, the data is split into q subsets. The model is trained on $q-1$ subsets, and the error is computed on the left-out subset. This process is repeated for all subsets, and the cross-validation error is summed. The order m with the smallest cross-validation error is chosen.

7.2. Polynomial Fitting in Python

7.2.1. Fitting the Polynomial

```
from sklearn.model_selection import KFold
def polynomial_fit(X, Y, max_order=15, q=5):
    """
    Fits a one-variable polynomial to one-dimensional data using cross-validation.

    Args:
        X (array-like): Training data vector (independent variable).
        Y (array-like): Training data vector (dependent variable).
        max_order (int): Maximum polynomial order to consider. Default is 15.
        q (int): Number of cross-validation folds. Default is 5.

    Returns:
        best_order (int): The optimal polynomial order.
        coeff (array): Coefficients of the best-fit polynomial.
        mnstd (tuple): Normalization parameters (mean, std) for X.
    """
    X = np.array(X)
    Y = np.array(Y)
    n = len(X)
    # Normalize X
    mnstd = (np.mean(X), np.std(X))
    X_norm = (X - mnstd[0]) / mnstd[1]
```

```
    # Cross-validation setup
    kf = KFold(n_splits=q, shuffle=True, random_state=42)
    cross_val_errors = np.zeros(max_order)
    for order in range(1, max_order + 1):
        fold_errors = []
        for train_idx, val_idx in kf.split(X_norm):
            X_train, X_val = X_norm[train_idx], X_norm[val_idx]
            Y_train, Y_val = Y[train_idx], Y[val_idx]
            # Fit polynomial
            coeff = np.polyfit(X_train, Y_train, order)
            # Predict on validation set
            Y_pred = np.polyval(coeff, X_val)
            # Compute mean squared error
            mse = np.mean((Y_val - Y_pred) ** 2)
            fold_errors.append(mse)
        cross_val_errors[order - 1] = np.mean(fold_errors)
    # Find the best order
    best_order = np.argmin(cross_val_errors) + 1
    # Fit the best polynomial on the entire dataset
    best_coeff = np.polyfit(X_norm, Y, best_order)
    return best_order, best_coeff, mnstd
```

7.2.2. Explaining the k-fold Cross-Validation

The line

```
kf = KFold(n_splits=q, shuffle=True, random_state=42)
```

initializes a k-Fold cross-validator object from the `sklearn.model_selection` library. The `n_splits` parameter specifies the number of folds. The data will be divided into `q` parts. In each iteration of the cross-validation, one part will be used as the validation set, and the remaining `q-1` parts will be used as the training set.

The `kf.split` method takes the dataset `X_norm` as input and yields pairs of index arrays for each fold: * `train_idx`: In each iteration, `train_idx` is an array containing the indices of the data points that belong to the training set for that specific fold. * `val_idx`: Similarly, `val_idx` is an array containing the indices of the data points that belong to the validation (or test) set for that specific fold.

The loop will run `q` times (the number of splits). In each iteration, a different fold serves as the validation set, while the other `q-1` folds form the training set.

Here's a Python example to demonstrate the values of `train_idx` and `val_idx`:

```python
# Sample data (e.g., X_norm)
X_norm = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
print(f"Original data indices: {np.arange(len(X_norm))}\n")
# Number of splits (folds)
q = 3 # Let's use 3 folds for this example
# Initialize KFold
kf = KFold(n_splits=q, shuffle=True, random_state=42)
# Iterate through the splits and print the indices
fold_number = 1
for train_idx, val_idx in kf.split(X_norm):
    print(f"--- Fold {fold_number} ---")
    print(f"Train indices: {train_idx}")
    print(f"Validation indices: {val_idx}")
    print(f"Training data for this fold: {X_norm[train_idx]}")
    print(f"Validation data for this fold: {X_norm[val_idx]}\n")
    fold_number += 1
```

```
Original data indices: [0 1 2 3 4 5 6 7 8 9]

--- Fold 1 ---
Train indices: [2 3 4 6 7 9]
Validation indices: [0 1 5 8]
Training data for this fold: [0.3 0.4 0.5 0.7 0.8 1. ]
Validation data for this fold: [0.1 0.2 0.6 0.9]

--- Fold 2 ---
Train indices: [0 1 3 4 5 6 8]
Validation indices: [2 7 9]
Training data for this fold: [0.1 0.2 0.4 0.5 0.6 0.7 0.9]
Validation data for this fold: [0.3 0.8 1. ]

--- Fold 3 ---
Train indices: [0 1 2 5 7 8 9]
Validation indices: [3 4 6]
Training data for this fold: [0.1 0.2 0.3 0.6 0.8 0.9 1. ]
Validation data for this fold: [0.4 0.5 0.7]
```

7.2.3. Making Predictions

To make predictions, we can use the coefficients. The data is standardized around its mean in the polynomial function, which is why the vector `mnstd` is required. The coefficient vector is computed based on the normalized data, and this must be taken into account if further analytical calculations are performed on the fitted model.

The polynomial approximation of C_D is:

$$C_D(x) = w_8 x^8 + w_7 x^7 + \cdots + w_1 x + w_0,$$

where x is normalized as:

$$\bar{x} = \frac{x - \mu(X)}{\sigma(X)}$$

```
def predict_polynomial_fit(X, coeff, mnstd):
    """
    Generates predictions for the polynomial fit.

    Args:
        X (array-like): Original independent variable data.
        coeff (array): Coefficients of the best-fit polynomial.
        mnstd (tuple): Normalization parameters (mean, std) for X.

    Returns:
        tuple: De-normalized predicted X values and corresponding Y predictions.
    """
    # Normalize X
    X_norm = (X - mnstd[0]) / mnstd[1]

    # Generate predictions
    X_pred = np.linspace(min(X_norm), max(X_norm), 100)
    Y_pred = np.polyval(coeff, X_pred)

    # De-normalize X for plotting
    X_pred_original = X_pred * mnstd[1] + mnstd[0]

    return X_pred_original, Y_pred
```

7.2.4. Plotting the Results

```
def plot_polynomial_fit(X, Y, X_pred_original, Y_pred, best_order, y_true=None):
    """
    Visualizes the polynomial fit.

    Args:
        X (array-like): Original independent variable data.
```

```
        Y (array-like): Original dependent variable data.
        X_pred_original (array): De-normalized predicted X values.
        Y_pred (array): Predicted Y values.
        y_true (array): True Y values.
        best_order (int): The optimal polynomial order.
    """
    plt.scatter(X, Y, label="Training Data", color="grey", marker="o")
    plt.plot(X_pred_original, Y_pred, label=f"Order {best_order} Polynomial", color="r
    if y_true is not None:
        plt.plot(X, y_true, label="True Function", color="blue", linestyle="--")
    plt.title(f"Polynomial Fit (Order {best_order})")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
    plt.show()
```

7.3. Example One: Aerofoil Drag

The circles in Figure 7.1 represent 101 drag coefficient values obtained through a numerical simulation by iterating each member of a family of aerofoils towards a target lift value (see the Appendix, Section A.3 in Forrester, Sóbester, and Keane (2008)). The members of the family have different shapes, as determined by the sampling plan:

$$X = x_1, x_2, \dots, x_{101}$$

The responses are:

$$C_D = \{C_D^{(1)}, C_D^{(2)}, \dots, C_D^{(101)}\}$$

These responses are corrupted by "noise," which are deviations of the systematic variety caused by small changes in the computational mesh from one design to the next.

The original data is measured in natural units, i.e., from -0.3 untion to 0.1 unit. The data is normalized to the range of 0 to 1 for the computation with the `aerofoilcd` function. The data is then fitted with a polynomial of order m. To obtain the best polynomial through this data, the following Python code can be used:

```
from spotpython.surrogate.functions.forr08a import aerofoilcd
import numpy as np
import matplotlib.pyplot as plt
X = np.linspace(-0.3, 0.1, 101)
# normalize the data so that it will be in the range of 0 to 1
```

```
a = np.min(X)
b = np.max(X)
X_cod = (X - a) / (b - a)
y = aerofoilcd(X_cod)
best_order, best_coeff, mnstd = polynomial_fit(X, y)
X_pred_original, Y_pred = predict_polynomial_fit(X, best_coeff, mnstd)
```

```
plot_polynomial_fit(X, y, X_pred_original, Y_pred, best_order)
```

Figure 7.1.: Aerofoil drag data

Figure 7.1 shows an eighth-order polynomial fitted through the aerofoil drag data. The order was selected via cross-validation, and the coefficients were determined through likelihood maximization. Results, i.e, the best polynomial order and coefficients, are printed in the console. The coefficients are stored in the vector `best_coeff`, which contains the coefficients of the polynomial in descending order. The first element is the coefficient of x^8, and the last element is the constant term. The vector `mnstd`, containing the mean and standard deviation of X, is:

```
print(f"Best polynomial order: {best_order}\n")
print(f"Coefficients (starting with w0):\n {best_coeff}\n")
print(f"Normalization parameters (mean, std):\n {mnstd}\n")
```

```
Best polynomial order: 8

Coefficients (starting with w0):
 [-0.00022964 -0.00014636  0.00116742  0.00052988 -0.0016912  -0.00047398
  0.00244373  0.00270342  0.03041508]

Normalization parameters (mean, std):
 (np.float64(-0.09999999999999999), np.float64(0.11661903789690602))
```

7.4. Example Two: A Multimodal Test Case

Let us consider the one-variable test function:

$$f(x) = (6x - 2)^2 \sin(12x - 4).$$

```
import numpy as np
from spotpython.surrogate.functions.forr08a import onevar
X = np.linspace(0, 1, 51)
y_true = onevar(X)
# initialize random seed
np.random.seed(42)
y = y_true + np.random.normal(0, 1, len(X))*1.1
best_order, best_coeff, mnstd = polynomial_fit(X, y)
X_pred_original, Y_pred = predict_polynomial_fit(X, best_coeff, mnstd)
```

```
plot_polynomial_fit(X, y, X_pred_original, Y_pred, best_order, y_true=y_true)
```

Figure 7.2.: Onevar function

This function, depicted by the dotted line in Figure 7.2, has local minima of different depths, which can be deceptive to some surrogate-based optimization procedures. Here, we use it as an example of a multimodal function for polynomial fitting.

We generate the training data (depicted by circles in Figure 7.2) by adding normally distributed noise to the function. Figure 7.2 shows a seventh-order polynomial fitted through the noisy data. This polynomial was selected as it minimizes the cross-validation metric.

7.5. Extending to Multivariable Polynomial Models

While the examples above focus on the one-variable case, real-world engineering problems typically involve multiple input variables. For k-variable problems, polynomial approximation becomes significantly more complex but follows the same fundamental principles. For a k-dimensional input space, the polynomial approximation can be expressed as:

$$\hat{f}(\vec{x}) = \sum_{i=1}^{N} w_i \phi_i(\vec{x}),$$

where $\phi_i(\vec{x})$ represents multivariate basis functions, and N is the total number of terms in the polynomial. Unlike the univariate case, these basis functions include all possible combinations of variables up to the selected polynomial order m, which might result in a "basis function explosion" as the number of variables increases.

For a third-order polynomial ($m = 3$) with three variables ($k = 3$), the complete set of basis functions would include 20 terms:

$$\text{Constant term: } 1 \tag{7.1}$$
$$\text{First-order terms: } x_1, x_2, x_3 \tag{7.2}$$
$$\text{Second-order terms: } x_1^2, x_2^2, x_3^2, x_1x_2, x_1x_3, x_2x_3 \tag{7.3}$$
$$\text{Third-order terms: } x_1^3, x_2^3, x_3^3, x_1^2x_2, x_1^2x_3, x_2^2x_1, x_2^2x_3, x_3^2x_1, x_3^2x_2, x_1x_2x_3 \tag{7.4}$$

The total number of terms grows combinatorially as $N = \binom{k+m}{m}$, which quickly becomes prohibitive as dimensionality increases. For example, a 10-variable cubic polynomial requires $\binom{13}{3} = 286$ coefficients! This exponential growth creates three interrelated challenges:

- Model Selection: Determining the appropriate polynomial order m that balances complexity with generalization ability
- Coefficient Estimation: Computing the potentially large number of weights $\vec{w}$ while avoiding numerical instability
- Term Selection: Identifying which specific basis functions should be included, as many may be irrelevant to the response

Several techniques have been developed to address these challenges:

- Regularization methods (LASSO, ridge regression) that penalize model complexity
- Stepwise regression algorithms that incrementally add or remove terms
- Dimension reduction techniques that project the input space to lower dimensions
- Orthogonal polynomials that improve numerical stability for higher-order models

These limitations of polynomial models in higher dimensions motivate the exploration of more flexible surrogate modeling approaches like Radial Basis Functions and Kriging, which we'll examine in subsequent sections.

7.6. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

8. Radial Basis Function Models

> **Note**
>
> - This section is based on chapter 2.3 in Forrester, Sóbester, and Keane (2008).
> - The following Python packages are imported:

```
import numpy as np
import matplotlib.pyplot as plt
```

8.1. Radial Basis Function Models

Scientists and engineers frequently tackle complex functions by decomposing them into a "vocabulary" of simpler, well-understood basic functions. These fundamental building blocks possess properties that make them easier to analyze mathematically and implement computationally. We explored this concept earlier with multivariable polynomials, where complex behaviors were modeled using combinations of polynomial terms such as 1, x_1, x_2, x_1^2, and x_1x_2. This approach is not limited to polynomials; it extends to various function classes, including trigonometric functions, exponential functions, and even more complex structures.

While Fourier analysis—perhaps the most widely recognized example of this approach—excels at representing periodic phenomena through sine and cosine functions, the focus in Forrester, Sóbester, and Keane (2008) is broader. They aim to approximate arbitrary smooth, continuous functions using strategically positioned basis functions. Specifically, radial basis function (RBF) models employ symmetrical basis functions centered at selected points distributed throughout the design space. These basis functions have the unique property that their output depends only on the distance from their center point.

First, we give a definition of the Euclidean distance, which is the most common distance measure used in RBF models.

Definition 8.1 (Euclidean Distance)**.** The Euclidean distance between two points in a k-dimensional space is defined as:

$$\|\vec{x} - \vec{c}\| = \sqrt{\sum_{i=1}^{k} (x_i - c_i)^2}, \tag{8.1}$$

where:

- $\vec{x} = (x_1, x_2, \dots, x_d)$ is the first point,
- $\vec{c} = (c_1, c_2, \dots, c_d)$ is the second point, and
- k is the number of dimensions.

The Euclidean distance measure represents the straight-line distance between two points in Euclidean space.

Using the Euclidean distance, we can define the radial basis function (RBF) model.

Definition 8.2 (Radial Basis Function (RBF))**.** Mathematically, a radial basis function ψ can be expressed as:

$$\psi(\vec{x}) = \psi(\|\vec{x} - \vec{c}\|), \tag{8.2}$$

where $\vec{x}$ is the input vector, $\vec{c}$ is the center of the function, and $\|\vec{x} - \vec{c}\|$ denotes the Euclidean distance between $\vec{x}$ and $\vec{c}$.

In the context of RBFs, the Euclidean distance calculation determines how much influence a particular center point $\vec{c}$ has on the prediction at point $\vec{x}$.

We will first examine interpolating RBF models, which assume noise-free data and pass exactly through all training points. This approach provides an elegant mathematical foundation before we consider more practical scenarios where data contains measurement or process noise.

8.1.1. Fitting Noise-Free Data

Let us consider the scalar valued function f observed without error, according to the sampling plan $X = \{\vec{x}^{(1)}, \vec{x}^{(2)}, \dots, \vec{x}^{(n)}\}^T$, yielding the responses $\vec{y} = \{y^{(1)}, y^{(2)}, \dots, y^{(n)}\}^T$.

For a given set of n_c centers $\vec{c}^{(i)}$, we would like to express the RBF model as a linear combination of the basis functions centered at these points. The goal is to find the weights $\vec{w}$ that minimize the error between the predicted and observed values. Thus, we seek a radial basis function approximation to f of the fixed form:

$$\hat{f}(\vec{x}) = \sum_{i=1}^{n_c} w_i \psi(||\vec{x} - \vec{c}^{(i)}||), \tag{8.3}$$

where

- w_i are the weights of the n_c basis functions,
- $\vec{c}^{(i)}$ are the n_c centres of the basis functions, and
- ψ is a radial basis function.

The notation $||\cdot||$ denotes the Euclidean distance between two points in the design space as defined in Equation 8.1.

8.1.1.1. Selecting Basis Functions: From Fixed to Parametric Forms

When implementing a radial basis function model, we initially have one undetermined parameter per basis function: the weight applied to each function's output. This simple parameterization remains true when we select from several standard fixed-form basis functions, such as:

- Linear ($\psi(r) = r$): The simplest form, providing a response proportional to distance
- Cubic ($\psi(r) = r^3$): Offers stronger emphasis on points farther from the center
- Thin plate spline ($\psi(r) = r^2 \ln r$): Models the physical bending of a thin sheet, providing excellent smoothness properties

While these fixed basis functions are computationally efficient, they offer limited flexibility in how they generalize across the design space. For more adaptive modeling power, we can employ parametric basis functions that introduce additional tunable parameters:

- Gaussian ($\psi(r) = e^{-r^2/(2\sigma^2)}$): Produces bell-shaped curves with σ controlling the width of influence
- Multiquadric ($\psi(r) = (r^2 + \sigma^2)^{1/2}$): Provides broader coverage with less localized effects
- Inverse multiquadric ($\psi(r) = (r^2 + \sigma^2)^{-1/2}$): Offers sharp peaks near centers with asymptotic behavior

The parameter σ in these functions serves as a shape parameter that controls how rapidly the function's influence decays with distance. This added flexibility enables significantly better generalization, particularly when modeling complex responses, though at the cost of a more involved parameter estimation process requiring optimization of both weights and shape parameters.

Example 8.1 (Gaussian RBF)**.** Using the general definition of a radial basis function (Equation 8.2), we can express the Gaussian RBF as:

$$\psi(\vec{x}) = \exp\left(-\frac{\|\vec{x} - \vec{c}\|^2}{2\sigma^2}\right) = \exp\left(-\frac{\sum_{j=1}^{k}(x_j - c_j)^2}{2\sigma^2}\right) \tag{8.4}$$

where:

- $\vec{x}$ is the input vector,
- $\vec{c}$ is the center vector,
- $\|\vec{x} - \vec{c}\|$ is the Euclidean distance between the input and center, and
- σ is the width parameter that controls how quickly the function's response diminishes with distance from the center.

The Gaussian RBF produces a bell-shaped response that reaches its maximum value of 1 when $\vec{x} = \vec{c}$ and asymptotically approaches zero as the distance increases. The parameter σ determines how "localized" the response is—smaller values create a narrower peak with faster decay, while larger values produce a broader, more gradual response across the input space. Figure 8.1 shows the Gaussian RBF for different values of σ in an one-dimensional space. The center of the RBF is set at 0, and the width parameter σ varies to illustrate how it affects the shape of the function.

```
def gaussian_rbf(x, center, sigma):
    """
    Compute the Gaussian Radial Basis Function.

    Args:
        x (ndarray): Input points
        center (float): Center of the RBF
        sigma (float): Width parameter

    Returns:
        ndarray: RBF values
    """
    return np.exp(-((x - center)**2) / (2 * sigma**2))
```

The sum of Gaussian RBFs can be visualized by summing the individual Gaussian RBFs centered at different points as shown in Figure 8.2. The following code snippet demonstrates how to create this plot showing the sum of three Gaussian RBFs with different centers and a common width parameter σ.

8.1.1.2. The Interpolation Condition: Elegant Solutions Through Linear Systems

A remarkable property of radial basis function models is that regardless of which basis functions we choose—parametric or fixed—determining the weights $\vec{w}$ remains straightforward through interpolation. The core principle is elegantly simple: we require our model to exactly reproduce the observed data points:

$$\hat{f}(\vec{x}^{(i)}) = y^{(i)}, \quad i = 1, 2, \dots, n. \tag{8.5}$$

This constraint produces one of the most powerful aspects of RBF modeling: while the system in Equation 8.5 is linear with respect to the weights $\vec{w}$, the resulting predictor $\hat{f}$

Figure 8.1.: Gaussian RBF

can capture highly nonlinear relationships in the data. The RBF approach transforms a complex nonlinear modeling problem into a solvable linear algebra problem.

For a unique solution to exist, we require that the number of basis functions equals the number of data points ($n_c = n$). The standard practice, which greatly simplifies implementation, is to center each basis function at a training data point, setting $\vec{c}^{(i)} = \vec{x}^{(i)}$ for all $i = 1, 2, \dots, n$. This choice allows us to express the interpolation condition as a compact matrix equation:

$$\Psi \vec{w} = \vec{y}.$$

Here, Ψ represents the Gram matrix (also called the design matrix or kernel matrix), whose elements measure the similarity between data points:

$$\Psi_{i,j} = \psi(||\vec{x}^{(i)} - \vec{x}^{(j)}||), \quad i, j = 1, 2, \dots, n.$$

The solution for the weight vector becomes:

$$\vec{w} = \Psi^{-1} \vec{y}.$$

Figure 8.2.: Sum of Gaussian RBFs

This matrix inversion step is the computational core of the RBF model fitting process, and the numerical properties of this operation depend critically on the chosen basis function. Different basis functions produce Gram matrices with distinct conditioning properties, directly affecting both computational stability and the model's generalization capabilities.

8.1.2. Numerical Stability Through Positive Definite Matrices

A significant advantage of Gaussian and inverse multiquadric basis functions lies in their mathematical guarantees. Vapnik (1998) demonstrated that these functions always produce symmetric positive definite Gram matrices when using strictly positive definite kernels (see Section 10.4), which is a critical property for numerical reliability. Unlike other basis functions that may lead to ill-conditioned systems, these functions ensure the existence of unique, stable solutions.

This positive definiteness enables the use of Cholesky factorization, which offers substantial computational advantages over standard matrix inversion techniques. The Cholesky approach reduces the computational cost (reducing from $O(n^3)$ to roughly $O(n^3/3)$) while significantly improving numerical stability when handling the inevitable rounding errors in floating-point arithmetic. This robustness to numerical issues explains why

Gaussian and inverse multiquadric basis functions remain the preferred choice in many practical RBF implementations.

Furthermore, the positive definiteness guarantee provides theoretical assurances about the model's interpolation properties—ensuring that the RBF interpolant exists and is unique for any distinct set of centers. This mathematical foundation gives practitioners confidence in the method's reliability, particularly for complex engineering applications where model stability is paramount.

The computational advantage stems from how a symmetric positive definite matrix Ψ can be efficiently decomposed into the product of an upper triangular matrix U and its transpose:

$$\Psi = U^T U.$$

This decomposition transforms the system

$$\Psi \vec{w} = \vec{y}$$

into

$$U^T U \vec{w} = \vec{y},$$

which can be solved through two simpler triangular systems:

- First solve $U^T \vec{v} = \vec{y}$ for the intermediate vector $\vec{v}$
- Then solve $U\vec{w} = \vec{v}$ for the desired weights $\vec{w}$

In Python implementations, this process is elegantly handled using NumPy's or SciPy's Cholesky decomposition functions, followed by specialized solvers that exploit the triangular structure:

```
from scipy.linalg import cholesky, cho_solve
# Compute the Cholesky factorization
L = cholesky(Psi, lower=True)  # L is the lower triangular factor
weights = cho_solve((L, True), y)  # Efficient solver for (L L^T)w = y
```

8.1.3. Ill-Conditioning

An important numerical consideration in RBF modeling is that points positioned extremely close to each other in the input space X can lead to severe ill-conditioning of the Gram matrix (Micchelli 1986). This ill-conditioning manifests as nearly linearly dependent rows and columns in Ψ, potentially causing the Cholesky factorization to fail.

While this problem rarely arises with initial space-filling experimental designs (such as Latin Hypercube or quasi-random sequences), it frequently emerges during sequential

optimization processes that adaptively add infill points in promising regions. As these clusters of points concentrate in areas of high interest, the condition number of the Gram matrix deteriorates, jeopardizing numerical stability.

Several mitigation strategies exist: regularization through ridge-like penalties (modifying the standard RBF interpolation problem by adding a penalty term to the diagonal of the Gram matrix. This creates a literal "ridge" along the diagonal of the matrix), removing nearly coincident points, clustering, or applying more sophisticated approaches. One theoretically elegant solution involves augmenting non-conditionally positive definite basis functions with polynomial terms (Keane and Nair 2005). This technique not only improves conditioning but also ensures polynomial reproduction properties, enhancing the approximation quality for certain function classes while maintaining numerical stability.

Beyond determining $\vec{w}$, there is, of course, the additional task of estimating any other parameters introduced via the basis functions. A typical example is the σ of the Gaussian basis function, usually taken to be the same for all basis functions, though a different one can be selected for each centre, as is customary in the case of the Kriging basis function, to be discussed shortly (once again, we trade additional parameter estimation complexity versus increased flexibility and, hopefully, better generalization).

8.1.4. Parameter Optimization: A Two-Level Approach

When building RBF models, we face two distinct parameter estimation challenges:

- Determining the weights ($\vec{w}$): These parameters ensure our model precisely reproduces the training data. For any fixed basis function configuration, we can calculate these weights directly through linear algebra as shown earlier.
- Optimizing shape parameters (like σ in Gaussian RBF): These parameters control how the model generalizes to new, unseen data. Unlike weights, there's no direct formula to find their optimal values.

To address this dual challenge, we employ a nested optimization strategy (inner and outer levels):

8.1.4.1. Inner Level ($\vec{w}$)

For each candidate value of shape parameters (e.g., σ), we determine the corresponding optimal weights $\vec{w}$ by solving the linear system. The `estim_weights()` method implements the inner level optimization by calculating the optimal weights $\vec{w}$ for a given shape parameter (σ):

```
def estim_weights(self):
    # [...]

    # Construct the Phi (Psi) matrix
    self.Phi = np.zeros((n, n))
    for i in range(n):
        for j in range(i+1):
            self.Phi[i, j] = self.basis(d[i, j], self.sigma)
            self.Phi[j, i] = self.Phi[i, j]

    # Calculate weights using appropriate method
    if self.code == 4 or self.code == 6:
        # Use Cholesky factorization for Gaussian or inverse multiquadric
        try:
            L = cholesky(self.Phi, lower=True)
            self.weights = cho_solve((L, True), self.y)
            self.success = True
        except np.linalg.LinAlgError:
            # Error handling...
    else:
        # Use direct solve for other basis functions
        try:
            self.weights = np.linalg.solve(self.Phi, self.y)
            self.success = True
        except np.linalg.LinAlgError:
            # Error handling...

    return self
```

This method:

- Creates the Gram matrix (Phi) based on distances between points
- Solves the linear system $\Psi\vec{w} = \vec{y}$ for weights
- Uses appropriate numerical methods based on the basis function type (Cholesky factorization or direct solve)

8.1.4.2. Outer Level (σ)

We use cross-validation to evaluate how well the model generalizes with different shape parameter values. The outer level optimization is implemented within the `fit()` method, where cross-validation is used to evaluate different σ values:

```python
def fit(self):
    if self.code < 4:
        # Fixed basis function, only w needs estimating
        self.estim_weights()
    else:
        # Basis function requires a sigma, estimate first using cross-validation
        # [...]

        # Generate candidate sigma values
        sigmas = np.logspace(-2, 2, 30)

        # Setup cross-validation (determine number of folds)
        # [...]

        cross_val = np.zeros(len(sigmas))

        # For each candidate sigma value
        for sig_index, sigma in enumerate(sigmas):
            print(f"Computing cross-validation metric for Sigma={sigma:.4f}...")

            # Perform k-fold cross-validation
            for j in range(len(from_idx)):
                # Create and fit model on training subset
                temp_model = Rbf(
                    X=X_orig[xs_temp],
                    y=y_orig[xs_temp],
                    code=self.code
                )
                temp_model.sigma = sigma

                # Call inner level optimization
                temp_model.estim_weights()

                # Evaluate on held-out data
                # [...]

        # Select best sigma based on cross-validation performance
        min_cv_index = np.argmin(cross_val)
        best_sig = sigmas[min_cv_index]

        # Use the best sigma for final model
        self.sigma = best_sig
        self.estim_weights()  # Call inner level again with optimal sigma
```

The outer level:

- Generates a range of candidate σ values
- For each σ, performs k-fold cross-validation:
  - Creates models on subsets of the data
  - Calls the inner level method (estim_weights()) to determine weights
  - Evaluates prediction quality on held-out data
- Selects the σ that minimizes cross-validation error
- Performs a final call to the inner level method with the optimal σ

This two-level approach is particularly critical for parametric basis functions (Gaussian, multiquadric, etc.), where the wrong choice of shape parameter could lead to either overfitting (too much flexibility) or underfitting (too rigid). Cross-validation provides an unbiased estimate of how well different parameter choices will perform on new data, helping us balance the trade-off between fitting the training data perfectly and generalizing well.

8.2. Python Implementation of the RBF Model

Section 8.2 shows a Python implementation of this parameter estimation process (based on a cross-validation routine), which will represent the surrogate, once its parameters have been estimated. The model building process is very simple.

Instead of using a dictionary for bookkeeping, we implement a Python class Rbf that encapsulates all the necessary data and functionality. The class stores the sampling plan X as the X attribute and the corresponding n-vector of responses y as the y attribute. The code attribute specifies the type of basis function to be used. After fitting the model, the class will also contain the estimated parameter values $\vec{w}$ and, if a parametric basis function is used, σ. These are stored in the weights and sigma attributes respectively.

Finally, a note on prediction error estimation. We have already indicated that the guarantee of a positive definite Ψ is one of the advantages of Gaussian radial basis functions. They also possess another desirable feature: it is relatively easy to estimate their prediction error at any $\vec{x}$ in the design space. Additionally, the expectation function of the improvement in minimum (or maximum) function value with respect to the minimum (or maximum) known so far can also be calculated quite easily, both of these features being very useful when the optimization of f is the goal of the surrogate modelling process.

8.2.1. The Rbf Class

The Rbf class implements the Radial Basis Function model. It encapsulates all the data and methods needed for fitting the model and making predictions.

```
import numpy as np
from scipy.linalg import cholesky, cho_solve
import numpy.random as rnd

class Rbf:
    """Radial Basis Function model implementation.

    Attributes:
        X (ndarray): The sampling plan (input points).
        y (ndarray): The response vector.
        code (int): Type of basis function to use.
        weights (ndarray, optional): The weights vector (set after fitting).
        sigma (float, optional): Parameter for parametric basis functions.
        Phi (ndarray, optional): The Gram matrix (set during fitting).
        success (bool, optional): Flag indicating successful fitting.
    """

    def __init__(self, X=None, y=None, code=3):
        """Initialize the RBF model.

        Args:
            X (ndarray, optional):
                The sampling plan.
            y (ndarray, optional):
                The response vector.
            code (int, optional):
                Type of basis function.
                Default is 3 (thin plate spline).
        """
        self.X = X
        self.y = y
        self.code = code
        self.weights = None
        self.sigma = None
        self.Phi = None
        self.success = None

    def basis(self, r, sigma=None):
        """Compute the value of the basis function.
```

```
        Args:
            r (float): Radius (distance)
            sigma (float, optional): Parameter for parametric basis functions

        Returns:
            float: Value of the basis function
        """
        # Use instance sigma if not provided
        if sigma is None and hasattr(self, 'sigma'):
            sigma = self.sigma

        if self.code == 1:
            # Linear function
            return r
        elif self.code == 2:
            # Cubic
            return r**3
        elif self.code == 3:
            # Thin plate spline
            if r < 1e-200:
                return 0
            else:
                return r**2 * np.log(r)
        elif self.code == 4:
            # Gaussian
            return np.exp(-(r**2)/(2*sigma**2))
        elif self.code == 5:
            # Multi-quadric
            return (r**2 + sigma**2)**0.5
        elif self.code == 6:
            # Inverse Multi-Quadric
            return (r**2 + sigma**2)**(-0.5)
        else:
            raise ValueError("Invalid basis function code")

    def estim_weights(self):
        """Estimates the basis function weights if sigma is known or not required.

        Returns:
            self: The updated model instance
        """
        # Check if sigma is required but not provided
        if self.code > 3 and self.sigma is None:
            raise ValueError("The basis function requires a sigma parameter")
```

```
        # Number of points
        n = len(self.y)

        # Build distance matrix
        d = np.zeros((n, n))
        for i in range(n):
            for j in range(i+1):
                d[i, j] = np.linalg.norm(self.X[i] - self.X[j])
                d[j, i] = d[i, j]

        # Construct the Phi (Psi) matrix
        self.Phi = np.zeros((n, n))
        for i in range(n):
            for j in range(i+1):
                self.Phi[i, j] = self.basis(d[i, j], self.sigma)
                self.Phi[j, i] = self.Phi[i, j]

        # Calculate weights using appropriate method
        if self.code == 4 or self.code == 6:
            # Use Cholesky factorization for Gaussian or inverse multiquadric
            try:
                L = cholesky(self.Phi, lower=True)
                self.weights = cho_solve((L, True), self.y)
                self.success = True
            except np.linalg.LinAlgError:
                print("Cholesky factorization failed.")
                print("Two points may be too close together.")
                self.weights = None
                self.success = False
        else:
            # Use direct solve for other basis functions
            try:
                self.weights = np.linalg.solve(self.Phi, self.y)
                self.success = True
            except np.linalg.LinAlgError:
                self.weights = None
                self.success = False

        return self

    def fit(self):
        """Estimates the parameters of the Radial Basis Function model.

        Returns:
```

```
        self: The updated model instance
    """
    if self.code < 4:
        # Fixed basis function, only w needs estimating
        self.estim_weights()
    else:
        # Basis function also requires a sigma, estimate first
        # Save original model data
        X_orig = self.X.copy()
        y_orig = self.y.copy()

        # Direct search between 10^-2 and 10^2
        sigmas = np.logspace(-2, 2, 30)

        # Number of cross-validation subsets
        if len(self.X) < 6:
            q = 2
        elif len(self.X) < 15:
            q = 3
        elif len(self.X) < 50:
            q = 5
        else:
            q = 10

        # Number of sample points
        n = len(self.X)

        # X split into q randomly selected subsets
        xs = rnd.permutation(n)
        full_xs = xs.copy()

        # The beginnings of the subsets...
        from_idx = np.arange(0, n, n//q)
        if from_idx[-1] >= n:
            from_idx = from_idx[:-1]

        # ...and their ends
        to_idx = np.zeros_like(from_idx)
        for i in range(len(from_idx) - 1):
            to_idx[i] = from_idx[i+1] - 1
        to_idx[-1] = n - 1

        cross_val = np.zeros(len(sigmas))
```

```
        # Cycling through the possible values of Sigma
        for sig_index, sigma in enumerate(sigmas):
            print(f"Computing cross-validation metric for Sigma={sigma:.4f}...")

            cross_val[sig_index] = 0

            # Model fitting to subsets of the data
            for j in range(len(from_idx)):
                removed = xs[from_idx[j]:to_idx[j]+1]
                xs_temp = np.delete(xs, np.arange(from_idx[j], to_idx[j]+1))

                # Create a temporary model for CV
                temp_model = Rbf(
                    X=X_orig[xs_temp],
                    y=y_orig[xs_temp],
                    code=self.code
                )
                temp_model.sigma = sigma

                # Sigma and subset chosen, now estimate w
                temp_model.estim_weights()

                if temp_model.weights is None:
                    cross_val[sig_index] = 1e20
                    xs = full_xs.copy()
                    break

                # Compute vector of predictions at the removed sites
                pr = np.zeros(len(removed))
                for jj, idx in enumerate(removed):
                    pr[jj] = temp_model.predict(X_orig[idx])

                # Calculate cross-validation error
                cross_val[sig_index] += np.sum((y_orig[removed] - pr)**2) / len(re

                xs = full_xs.copy()

            # Now attempt Cholesky on the full set, in case the subsets could
            # be fitted correctly, but the complete X could not
            temp_model = Rbf(
                X=X_orig,
                y=y_orig,
                code=self.code
            )
```

```
                temp_model.sigma = sigma
                temp_model.estim_weights()

                if temp_model.weights is None:
                    cross_val[sig_index] = 1e20
                    print("Failed to fit complete sample data.")

            # Find the best sigma
            min_cv_index = np.argmin(cross_val)
            best_sig = sigmas[min_cv_index]

            # Set the best sigma and recompute weights
            print(f"Selected sigma={best_sig:.4f}")
            self.sigma = best_sig
            self.estim_weights()

        return self

    def predict(self, x):
        """Calculates the value of the Radial Basis Function surrogate model at x.

        Args:
            x (ndarray): Point at which to make prediction

        Returns:
            float: Predicted value
        """
        # Calculate distances to all sample points
        d = np.zeros(len(self.X))
        for k in range(len(self.X)):
            d[k] = np.linalg.norm(x - self.X[k])

        # Calculate basis function values
        phi = np.zeros(len(self.X))
        for k in range(len(self.X)):
            phi[k] = self.basis(d[k], self.sigma)

        # Calculate prediction
        y = np.dot(phi, self.weights)
        return y
```

8.3. RBF Example: The One-Dimensional `sin` Function

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import cholesky, cho_solve

# Define the data points for fitting
x_centers = np.array([np.pi/2, np.pi, 3*np.pi/2]).reshape(-1, 1)  # Centers for RBFs
y_values = np.sin(x_centers.flatten())  # Sine values at these points

# Create and fit the RBF model
rbf_model = Rbf(X=x_centers, y=y_values, code=4)  # Code 4 is Gaussian RBF
rbf_model.sigma = 1.0  # Set sigma parameter directly
rbf_model.estim_weights()  # Calculate optimal weights

# Print the weights
print("RBF model weights:", rbf_model.weights)

# Create a grid for visualization
x_grid = np.linspace(0, 2*np.pi, 1000).reshape(-1, 1)
y_true = np.sin(x_grid.flatten())  # True sine function

# Generate predictions using the RBF model
y_pred = np.zeros(len(x_grid))
for i in range(len(x_grid)):
    y_pred[i] = rbf_model.predict(x_grid[i])

# Calculate individual basis functions for visualization
basis_funcs = np.zeros((len(x_grid), len(x_centers)))
for i in range(len(x_grid)):
    for j in range(len(x_centers)):
        # Calculate distance
        distance = np.linalg.norm(x_grid[i] - x_centers[j])
        # Compute basis function value scaled by its weight
        basis_funcs[i, j] = rbf_model.basis(distance, rbf_model.sigma) * rbf_model.wei

# Plot the results
plt.figure(figsize=(6, 4))

# Plot the true sine function
plt.plot(x_grid, y_true, 'k-', label='True sine function', linewidth=2)

# Plot individual basis functions
```

```
for i in range(len(x_centers)):
    plt.plot(x_grid, basis_funcs[:, i], '--',
             label=f'Basis function at x={x_centers[i][0]:.2f}')

# Plot the RBF fit (sum of basis functions)
plt.plot(x_grid, y_pred, 'r-', label='RBF fit', linewidth=2)

# Plot the sample points
plt.scatter(x_centers, y_values, color='blue', s=100, label='Sample points')

# Add horizontal line at y=0
plt.axhline(y=0, color='gray', linestyle='--', alpha=0.3)

plt.title('RBF Approximation of Sine Function')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

```
RBF model weights: [ 1.00724398e+00  2.32104414e-16 -1.00724398e+00]
```

8.4. RBF Example: The Two-Diemnsional `dome` Function

The `dome` function is an example of a test function that can be used to evaluate the performance of the Radial Basis Function model. It is a simple mathematical function defined over a two-dimensional space.

```
def dome(x) -> float:
  """
  Dome test function.

  Args:
      x (ndarray): Input vector (1D array of length 2)

  Returns:
      float: Function value

  Examples:
      dome(np.array([0.5, 0.5]))
  """
  return np.sum(1 - (2*x - 1)**2) / len(x)
```

The following code demonstrates how to use the Radial Basis Function model to approximate a function. It generates a Latin Hypercube sample, computes the objective function values, estimates the model parameters, and plots the results.

```
def generate_rbf_data(n_samples=10, grid_points=41):
    """
    Generates data for RBF visualization.

    Args:
        n_samples (int): Number of samples for the RBF model
        grid_points (int): Number of grid points for prediction

    Returns:
        tuple: (rbf_model, X, Y, Z, Z_0) - Model and grid data for plotting
    """
    from spotpython.utils.sampling import bestlh as best_lh
    # Generate sampling plan
    X_samples = best_lh(n_samples, 2, population=10, iterations=100)
    # Compute objective function values
    y_samples = np.zeros(len(X_samples))
    for i in range(len(X_samples)):
        y_samples[i] = dome(X_samples[i])
    # Create and fit RBF model
```

```
    rbf_model = Rbf(X=X_samples, y=y_samples, code=3)  # Thin plate spline
    rbf_model.fit()
    # Generate grid for prediction
    x = np.linspace(0, 1, grid_points)
    y = np.linspace(0, 1, grid_points)
    X, Y = np.meshgrid(x, y)
    Z_0 = np.zeros_like(X)
    Z = np.zeros_like(X)

    # Evaluate model at grid points
    for i in range(len(x)):
        for j in range(len(y)):
            Z_0[j, i] = dome(np.array([x[i], y[j]]))
            Z[j, i] = rbf_model.predict(np.array([x[i], y[j]]))

    return rbf_model, X, Y, Z, Z_0

def plot_rbf_results(rbf_model, X, Y, Z, Z_0=None, n_contours=10):
    """
    Plots RBF approximation results.

    Args:
        rbf_model (Rbf): Fitted RBF model
        X (ndarray): Grid X-coordinates
        Y (ndarray): Grid Y-coordinates
        Z (ndarray): RBF model predictions
        Z_0 (ndarray, optional): True function values for comparison
        n_contours (int): Number of contour levels to plot
    """
    import matplotlib.pyplot as plt

    plt.figure(figsize=(10, 8))

    # Plot the contour
    contour = plt.contour(X, Y, Z, n_contours)

    if Z_0 is not None:
        contour_0 = plt.contour(X, Y, Z_0, n_contours, colors='k', linestyles='dashed')

    # Plot the sample points
    plt.scatter(rbf_model.X[:, 0], rbf_model.X[:, 1],
                c='r', marker='o', s=50)

    plt.title('RBF Approximation (Thin Plate Spline)')
```

```
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.colorbar(label='f(x1, x2)')
    plt.show()
```

Figure 8.3 shows the contour plots of the underlying function $f(x_1, x_2) = 0.5[-(2x_1 - 1)^2 - (2x_2 - 1)^2]$ and its thin plate spline radial basis function approximation, along with the 10 points of a Morris-Mitchell optimal Latin hypercube sampling plan (obtained via best_lh()).

```
rbf_model, X, Y, Z, Z_0 = generate_rbf_data(n_samples=10, grid_points=41)
plot_rbf_results(rbf_model, X, Y, Z, Z_0)
```

Figure 8.3.: RBF Approximation.

8.4.1. The Connection Between RBF Models and Neural Networks

Radial basis function models share a profound architectural similarity with artificial neural networks, specifically with what's known as RBF networks. This connection provides valuable intuition about how RBF models function. A radial basis function model can be viewed as a specialized neural network with the following structure:

- Input Layer: Receives the feature vector $\vec{x}$
- Hidden Layer: Contains neurons (basis functions) that compute radial distances
- Output Layer: Produces a weighted sum of the hidden unit activations

Unlike traditional neural networks that use dot products followed by nonlinear activation functions, RBF networks measure the distance between inputs and learned center points. This distance is then transformed by the radial basis function.

Mathematically, the equivalence between RBF models and RBF networks can be expressed as follows:

The RBF model equation:

$$\hat{f}(\vec{x}) = \sum_{i=1}^{n_c} w_i \psi(||\vec{x} - \vec{c}^{(i)}||)$$

directly maps to the following neural network components:

- $\vec{x}$: Input vector
- $\vec{c}^{(i)}$: Center vectors for each hidden neuron
- $\psi(\cdot)$: Activation function (Gaussian, inverse multiquadric, etc.)
- w_i: Output weights
- $\hat{f}(\vec{x})$: Network output

Example 8.2 (Comparison of RBF Networks and Traditional Neural Networks)**.** Consider approximating a simple 1D function $f(x) = \sin(2\pi x)$ over the interval $[0, 1]$:

The neral network approach would use multiple layers with neurons computing $\sigma(w \cdot x + b)$. It would require a large number of neurons and layers to capture the sine wave's complexity. The network would learn both weights and biases, making it less interpretable.

The RBF network approach, on the other hand, places basis functions at strategic points (e.g., 5 evenly spaced centers). Each neuron computes $\psi(||x - c_i||)$ (e.g., using Gaussian RBF). The output layer combines these values with learned weights. If we place Gaussian RBFs with $\sigma = 0.15$ at $0.1, 0.3, 0.5, 0.7, 0.9$, each neuron responds strongly when the input is close to its center and weakly otherwise. The network can then learn weights that, when multiplied by these response patterns and summed, closely approximate the sine function.

This locality property gives RBF networks a notable advantage: they offer more interpretable internal representations and often require fewer neurons for certain types of function approximation compared to traditional multilayer perceptrons.

The key insight is that while standard neural networks create complex decision boundaries through compositions of hyperplanes, RBF networks directly model functions using a set of overlapping "bumps" positioned strategically in the input space.

8.5. Radial Basis Function Models for Noisy Data

When the responses $\vec{y} = y^{(1)}, y^{(2)}, \dots, y^{(n)}{}^T$ contain measurement or simulation noise, the standard RBF interpolation approach can lead to overfitting—the model captures both the underlying function and the random noise. This compromises generalization performance on new data points. Two principal strategies address this challenge:

8.5.1. Ridge Regularization Approach

The most straightforward solution involves introducing regularization through the parameter λ (Poggio and Girosi 1990). This is implemented by adding λ to the diagonal elements of the Gram matrix, creating a "ridge" that improves numerical stability. Mathematically, the weights are determined by:

$$\vec{w} = (\Psi + \lambda I)^{-1}\vec{y},$$

where I is an $n \times n$ identity matrix. This regularized solution balances two competing objectives:

- fitting the training data accurately versus
- keeping the magnitude of weights controlled to prevent overfitting.

Theoretically, optimal performance is achieved when λ equals the variance of the noise in the response data $\vec{y}$ (Keane and Nair 2005). Since this information is rarely available in practice, λ is typically estimated through cross-validation alongside other model parameters.

8.5.2. Reduced Basis Approach

An alternative strategy involves reducing m, the number of basis functions. This might result in a non-square Ψ matrix. With a non-square Ψ matrix, the weights are found through least squares minimization:

$$\vec{w} = (\Psi^T\Psi)^{-1}\Psi^T\vec{y}$$

This approach introduces an important design decision: which subset of points should serve as basis function centers? Several selection strategies exist:

- Clustering methods that identify representative points
- Greedy algorithms that sequentially select influential centers
- Support vector regression techniques (discussed elsewhere in the literature)

Additional parameters such as the width parameter σ in Gaussian bases can be optimized through cross-validation to minimize generalization error estimates.

The ridge regularization and reduced basis approaches can be combined, allowing for a flexible modeling framework, though at the cost of a more complex parameter estimation process. This hybrid approach often yields superior results for highly noisy datasets or when the underlying function has varying complexity across the input space.

The broader challenge of building accurate models from noisy observations is examined comprehensively in the context of Kriging models, which provide a statistical framework for explicitly modeling both the underlying function and the noise process.

8.6. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

9. Kriging (Gaussian Process Regression)

> **Note**
>
> - This section is based on chapter 2.4 in Forrester, Sóbester, and Keane (2008).
> - The following Python packages are imported:

```
import matplotlib.pyplot as plt
import numpy as np
from numpy import (array, zeros, power, ones, exp, multiply,
                   eye, linspace, spacing, sqrt, arange,
                   append, ravel)
from numpy.linalg import cholesky, solve
from scipy.spatial.distance import squareform, pdist, cdist
```

9.1. From Gaussian RBF to Kriging Basis Functions

Kriging can be explained using the concept of radial basis functions (RBFs), which were introduced in Chapter 8. An RBF is a real-valued function whose value depends only on the distance from a certain point, called the center, usually in a multidimensional space. The basis function is a function of the distance between the point $\vec{x}$ and the center $\vec{x}^{(i)}$. Other names for basis functions are *kernel* or *covariance* functions.

Definition 9.1 (The Kriging Basis Functions)**.** Kriging (also known as Gaussian Process Regression) uses k-dimensional basis functions of the form

$$\psi^{(i)}(\vec{x}) = \psi(\vec{x}^{(i)}, \vec{x}) = \exp\left(-\sum_{j=1}^{k} \theta_j |x_j^{(i)} - x_j|^{p_j}\right), \tag{9.1}$$

where $\vec{x}$ and $\vec{x}^{(i)}$ denote the k-dim vector $\vec{x} = (x_1, \dots, x_k)^T$ and $\vec{x}^{(i)} = (x_1^{(i)}, \dots, x_k^{(i)})^T$, respectively.

□

Kriging uses a specialized basis function that offers greater flexibility than standard RBFs. Examining Equation 9.1, we can observe how Kriging builds upon and extends the Gaussian basis concept. The key enhancements of Kriging over Gaussian RBF can be summarized as follows:

- Dimension-specific width parameters: While a Gaussian RBF uses a single width parameter $1/\sigma^2$, Kriging employs a vector $\vec{\theta} = (\theta_1, \theta_2, \dots, \theta_k)^T$. This allows the model to automatically adjust its sensitivity to each input dimension, effectively performing automatic feature relevance determination.
- Flexible smoothness control: The Gaussian RBF fixes the exponent at 2, producing uniformly smooth functions. In contrast, Kriging's dimension-specific exponents $\vec{p} = (p_1, p_2, \dots, p_k)^T$ (typically with $p_j \in [1, 2]$) enable precise control over smoothness properties in each dimension.
- Unifying framework: When all exponents are set to $p_j = 2$ and all width parameters are equal ($\theta_j = 1/\sigma^2$ for all j), the Kriging basis function reduces exactly to the Gaussian RBF. This makes Gaussian RBF a special case within the more general Kriging framework.

These enhancements make Kriging particularly well-suited for engineering problems where variables may operate at different scales or exhibit varying degrees of smoothness across dimensions. For now, we will only consider Kriging interpolation. We will cover Kriging regression later.

9.2. Building the Kriging Model

Consider sample data X and $\vec{y}$ from n locations that are available in matrix form: X is a $(n \times k)$ matrix, where k denotes the problem dimension and $\vec{y}$ is a $(n \times 1)$ vector. We want to find an expression for a predicted values at a new point $\vec{x}$, denoted as $\hat{y}$.

We start with an abstract, not really intuitive concept: The observed responses $\vec{y}$ are considered as if they are from a stochastic process, which will be denoted as

$$\begin{pmatrix} Y(\vec{x}^{(1)}) \\ \vdots \\ Y(\vec{x}^{(n)}) \end{pmatrix}. \tag{9.2}$$

The set of random vectors from Equation 9.2 (also referred to as a *random field*) has a mean of $\vec{1}\mu$, which is a $(n \times 1)$ vector. The random vectors are correlated with each other using the basis function expression from Equation 9.1:

$$\operatorname{cor}\left(Y(\vec{x}^{(i)}), Y(\vec{x}^{(l)})\right) = \exp\left(-\sum_{j=1}^{k} \theta_j |x_j^{(i)} - x_j^{(l)}|^{p_j}\right). \tag{9.3}$$

Using Equation 9.3, we can compute the $(n \times n)$ correlation matrix Ψ of the observed sample data as shown in Equation 9.4,

$$\Psi = \begin{pmatrix} \text{cor}\left(Y(\vec{x}^{(1)}), Y(\vec{x}^{(1)})\right) & \dots & \text{cor}\left(Y(\vec{x}^{(1)}), Y(\vec{x}^{(n)})\right) \\ \vdots & \vdots & \vdots \\ \text{cor}\left(Y(\vec{x}^{(n)}), Y(\vec{x}^{(1)})\right) & \dots & \text{cor}\left(Y(\vec{x}^{(n)}), Y(\vec{x}^{(n)})\right) \end{pmatrix}, \tag{9.4}$$

and a covariance matrix as shown in Equation 9.5,

$$\text{Cov}(Y, Y) = \sigma^2 \Psi. \tag{9.5}$$

This assumed correlation between the sample data reflects our expectation that an engineering function will behave in a certain way and it will be smoothly and continuous.

Remark 9.1 (Note on Stochastic Processes). See Section D.4 for a more detailed discussion on realizations of stochastic processes.

□

We now have a set of n random variables ($\mathbf{Y}$) that are correlated with each other as described in the $(n \times n)$ correlation matrix Ψ, see Equation 9.4. The correlations depend on the absolute distances in dimension j between the i-th and the l-th sample point $|x_j^{(i)} - x_j^{(l)}|$ and the corresponding parameters p_j and θ_j for dimension j. The correlation is intuitive, because when

- two points move close together, then $|x_j^{(i)} - x_j| \to 0$ and $\exp\left(-|x_j^{(i)} - x_j|^{p_j}\right) \to 1$ (these points show very close correlation and $Y(x_j^{(i)}) = Y(x_j)$).
- two points move far apart, then $|x_j^{(i)} - x_j| \to \infty$ and $\exp\left(-|x_j^{(i)} - x_j|^{p_j}\right) \to 0$ (these points show very low correlation).

Example 9.1 (Correlations for different p_j)**.** Three different correlations are shown in Figure 9.1: $p_j = 0.1, 1, 2$. The smoothness parameter p_j affects the correlation:

- With $p_j = 0.1$, there is basicaly no immediate correlation between the points and there is a near discontinuity between the points $Y(\vec{x}_j^{(i)})$ and $Y(\vec{x}_j)$.
- With $p_j = 2$, the correlation is more smooth and we have a continuous gradient through $x_j^{(i)} - x_j$.

Reducing p_j increases the rate at which the correlation initially drops with distance. This is shown in Figure 9.1.

□

Example 9.2 (Correlations for different θ)**.** Figure 9.2 visualizes the correlation between two points $Y(\vec{x}_j^{(i)})$ and $Y(\vec{x}_j)$ for different values of θ. The parameter θ can be seen as a *width* parameter:

Figure 9.1.: Correlations with varying p. θ set to 1.

- low θ_j means that all points will have a high correlation, with $Y(x_j)$ being similar across the sample.
- high θ_j means that there is a significant difference between the $Y(x_j)$'s.
- θ_j is a measure of how *active* the function we are approximating is.
- High θ_j indicate important parameters, see Figure 9.2.

□

Considering the activity parameter θ is useful in high-dimensional problems where it is difficult to visualize the design landscape and the effect of the variable is unknown. By examining the elements of the vector $\vec{\theta}$, we can identify the most important variables and focus on them. This is a crucial step in the optimization process, as it allows us to reduce the dimensionality of the problem and focus on the most important variables.

Example 9.3 (The Correlation Matrix (Detailed Computation))**.** Let $n = 4$ and $k = 3$. The sample plan is represented by the following matrix X:

$$X = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \end{pmatrix}$$

To compute the elements of the matrix Ψ, the following k (one for each of the k dimensions) (n, n)-matrices have to be computed:

Figure 9.2.: Correlations with varying θ. p set to 2.

- For $k = 1$, i.e., the first column of X:

$$D_1 = \begin{pmatrix} x_{11} - x_{11} & x_{11} - x_{21} & x_{11} - x_{31} & x_{11} - x_{41} \\ x_{21} - x_{11} & x_{21} - x_{21} & x_{21} - x_{31} & x_{21} - x_{41} \\ x_{31} - x_{11} & x_{31} - x_{21} & x_{31} - x_{31} & x_{31} - x_{41} \\ x_{41} - x_{11} & x_{41} - x_{21} & x_{41} - x_{31} & x_{41} - x_{41} \end{pmatrix}$$

- For $k = 2$, i.e., the second column of X:

$$D_2 = \begin{pmatrix} x_{12} - x_{12} & x_{12} - x_{22} & x_{12} - x_{32} & x_{12} - x_{42} \\ x_{22} - x_{12} & x_{22} - x_{22} & x_{22} - x_{32} & x_{22} - x_{42} \\ x_{32} - x_{12} & x_{32} - x_{22} & x_{32} - x_{32} & x_{32} - x_{42} \\ x_{42} - x_{12} & x_{42} - x_{22} & x_{42} - x_{32} & x_{42} - x_{42} \end{pmatrix}$$

- For $k = 3$, i.e., the third column of X:

$$D_3 = \begin{pmatrix} x_{13} - x_{13} & x_{13} - x_{23} & x_{13} - x_{33} & x_{13} - x_{43} \\ x_{23} - x_{13} & x_{23} - x_{23} & x_{23} - x_{33} & x_{23} - x_{43} \\ x_{33} - x_{13} & x_{33} - x_{23} & x_{33} - x_{33} & x_{33} - x_{43} \\ x_{43} - x_{13} & x_{43} - x_{23} & x_{43} - x_{33} & x_{43} - x_{43} \end{pmatrix}$$

Since the matrices are symmetric and the main diagonals are zero, it is sufficient to

compute the following matrices:

$$D_1 = \begin{pmatrix} 0 & x_{11}-x_{21} & x_{11}-x_{31} & x_{11}-x_{41} \\ 0 & 0 & x_{21}-x_{31} & x_{21}-x_{41} \\ 0 & 0 & 0 & x_{31}-x_{41} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & x_{12}-x_{22} & x_{12}-x_{32} & x_{12}-x_{42} \\ 0 & 0 & x_{22}-x_{32} & x_{22}-x_{42} \\ 0 & 0 & 0 & x_{32}-x_{42} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & x_{13}-x_{23} & x_{13}-x_{33} & x_{13}-x_{43} \\ 0 & 0 & x_{23}-x_{33} & x_{23}-x_{43} \\ 0 & 0 & 0 & x_{33}-x_{43} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

We will consider $p_l = 2$. The differences will be squared and multiplied by θ_i, i.e.:

$$D_1 = \theta_1 \begin{pmatrix} 0 & (x_{11}-x_{21})^2 & (x_{11}-x_{31})^2 & (x_{11}-x_{41})^2 \\ 0 & 0 & (x_{21}-x_{31})^2 & (x_{21}-x_{41})^2 \\ 0 & 0 & 0 & (x_{31}-x_{41})^2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_2 = \theta_2 \begin{pmatrix} 0 & (x_{12}-x_{22})^2 & (x_{12}-x_{32})^2 & (x_{12}-x_{42})^2 \\ 0 & 0 & (x_{22}-x_{32})^2 & (x_{22}-x_{42})^2 \\ 0 & 0 & 0 & (x_{32}-x_{42})^2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_3 = \theta_3 \begin{pmatrix} 0 & (x_{13}-x_{23})^2 & (x_{13}-x_{33})^2 & (x_{13}-x_{43})^2 \\ 0 & 0 & (x_{23}-x_{33})^2 & (x_{23}-x_{43})^2 \\ 0 & 0 & 0 & (x_{33}-x_{43})^2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The sum of the three matrices $D = D_1 + D_2 + D_3$ will be calculated next:

$$\begin{pmatrix} 0 & \theta_1(x_{11}-x_{21})^2 + \theta_2(x_{12}-x_{22})^2 + \theta_3(x_{13}-x_{23})^2 & \theta_1(x_{11}-x_{31})^2 + \theta_2(x_{12}-x_{32})^2 + \theta_3(x_{13}-x_{33} \\ 0 & 0 & \theta_1(x_{21}-x_{31})^2 + \theta_2(x_{22}-x_{32})^2 + \theta_3(x_{23}-x_{33} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Finally,

$$\Psi = \exp(-D)$$

is computed.

Next, we will demonstrate how this computation can be implemented in Python. We will consider four points in three dimensions and compute the correlation matrix Ψ using the basis function from Equation 9.1. These points are placed at the origin, at the unit vectors, and at the points $(100, 100, 100)$ and $(101, 100, 100)$. So, they form two clusters: one at the origin and one at $(100, 100, 100)$.

```
theta = np.array([1,2,3])
X = np.array([ [1,0,0], [0,1,0], [100, 100, 100], [101, 100, 100]])
X
```

```
array([[  1,   0,   0],
       [  0,   1,   0],
       [100, 100, 100],
       [101, 100, 100]])
```

```
def build_Psi(X, theta):
    n = X.shape[0]
    k = X.shape[1]
    D = zeros((k, n, n))
    for l in range(k):
        for i in range(n):
            for j in range(i, n):
                D[l, i, j] = theta[l]*(X[i,l] - X[j,l])**2
    D = sum(D)
    D = D + D.T
    return exp(-D)
```

```
Psi = build_Psi(X, theta)
Psi
```

```
array([[1.        , 0.04978707, 0.        , 0.        ],
       [0.04978707, 1.        , 0.        , 0.        ],
       [0.        , 0.        , 1.        , 0.36787944],
       [0.        , 0.        , 0.36787944, 1.        ]])
```

□

Example 9.4 (Example: The Correlation Matrix (Using Existing Functions))**.** The same result as computed in Example 9.3 can be obtained with existing python functions, e.g., from the package `scipy`.

Figure 9.3.: Correlation matrix Ψ.

```
def build_Psi(X, theta, eps=sqrt(spacing(1))):
    return exp(- squareform(pdist(X,
                            metric='sqeuclidean',
                            out=None,
                            w=theta))) +  multiply(eye(X.shape[0]),
                                                   eps)

Psi = build_Psi(X, theta, eps=.0)
Psi
```

```
array([[1.        , 0.04978707, 0.        , 0.        ],
       [0.04978707, 1.        , 0.        , 0.        ],
       [0.        , 0.        , 1.        , 0.36787944],
       [0.        , 0.        , 0.36787944, 1.        ]])
```

The condition number of the correlation matrix Ψ is a measure of how well the matrix can be inverted. A high condition number indicates that the matrix is close to singular, which can lead to numerical instability in computations involving the inverse of the matrix, see Section 10.2.

```
np.linalg.cond(Psi)
```

```
np.float64(2.163953413738652)
```

□

9.3. MLE to estimate θ and p

9.3.1. The Log-Likelihood

Until now, the observed data $\vec{y}$ was not used. We know what the correlations mean, but how do we estimate the values of θ_j and where does our observed data y come in? To estimate the values of $\vec{\theta}$ and $\vec{p}$, they are chosen to maximize the likelihood of $\vec{y}$,

$$L = L\left(Y(\vec{x}^{(1)}), \dots, Y(\vec{x}^{(n)}) | \mu, \sigma\right) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left[-\frac{\sum_{i=1}^{n}(Y(\vec{x}^{(i)}) - \mu)^2}{2\sigma^2}\right], \tag{9.6}$$

where μ is the mean of the observed data $\vec{y}$ and σ is the standard deviation of the errors ϵ, which can be expressed in terms of the sample data

$$L = \frac{1}{(2\pi\sigma^2)^{n/2}|\vec{\Psi}|^{1/2}} \exp\left[-\frac{(\vec{y} - \vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)}{2\sigma^2}\right]. \tag{9.7}$$

Remark 9.2. The transition from Equation 9.6 to Equation 9.7 reflects a shift from assuming independent errors in the observed data to explicitly modeling the *correlation structure* between the observed responses, which is a key aspect of the stochastic process framework used in methods like Kriging. It can be explained as follows:

1. **Initial Likelihood Expression (assuming independent errors):** Equation 9.6 is an expression for the likelihood of the data set, which is based on the assumption that the errors ϵ are *independently randomly distributed according to a normal distribution with standard deviation* σ. This form is characteristic of the likelihood of n independent observations $Y(\vec{x}^{(i)})$, each following a normal distribution with mean μ and variance σ^2.

2. **Using Vector Notation.** The sum in the exponent, i.e.,

$$\sum_{i=1}^{n}(Y(\vec{x}^{(i)}) - \mu)^2$$

is equivalent to

$$(\vec{y} - \vec{1}\mu)^T(\vec{y} - \vec{1}\mu),$$

assuming $Y(\vec{x}^{(i)}) = y^{(i)}$ and using vector notation for $\vec{y}$ and $\vec{1}\mu$.

3. **Assuming Independent Observations:** Equation 9.6 assumes that the observations are independent, which means that the covariance between any two observations $Y(\vec{x}^{(i)})$ and $Y(\vec{x}^{(l)})$ is zero for $i \neq l$. In this case, the covariance matrix of the observations would be a diagonal matrix with σ^2 along the diagonal (i.e., $\sigma^2 I$), where I is the identity matrix.

4. **Stochastic Process and Correlation:** In the context of Kriging, the observed responses $\vec{y}$ are considered as if they are from a *stochastic process* or *random field*. This means the random variables $Y(\vec{x}^{(i)})$ at different locations $\vec{x}^{(i)}$ are not independent, but they correlated with each other. This correlation is described by an $(n \times n)$ **correlation matrix** Ψ, which is used instead of $\sigma^2 I$. The strength of the correlation between two points $Y(\vec{x}^{(i)})$ and $Y(\vec{x}^{(l)})$ depends on the distance between them and model parameters θ_j and p_j.

5. **Multivariate Normal Distribution:** When random variables are correlated, their joint probability distribution is generally described by a *multivariate distribution*. Assuming the stochastic process follows a Gaussian process, the joint distribution of the observed responses $\vec{y}$ is a **multivariate normal distribution**. A multivariate normal distribution for a vector $\vec{Y}$ with mean vector $\vec{\mu}$ and covariance matrix Σ has a probability density function given by:

$$p(\vec{y}) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} \exp\left[-\frac{1}{2}(\vec{y} - \vec{\mu})^T \Sigma^{-1} (\vec{y} - \vec{\mu})\right].$$

6. **Connecting the Expressions:** In the stochastic process framework, the following holds:

   - The mean vector of the observed data $\vec{y}$ is $\vec{1}\mu$.
   - The covariance matrix Σ is constructed by considering both the variance σ^2 and the correlations Ψ.
   - The covariance between $Y(\vec{x}^{(i)})$ and $Y(\vec{x}^{(l)})$ is $\sigma^2 \mathrm{cor}(Y(\vec{x}^{(i)}), Y(\vec{x}^{(l)}))$.
   - Therefore, the covariance matrix is $\Sigma = \sigma^2 \vec{\Psi}$.
   - Substituting $\vec{\mu} = \vec{1}\mu$ and $\Sigma = \sigma^2 \vec{\Psi}$ into the multivariate normal PDF formula, we get:

     $$\Sigma^{-1} = (\sigma^2 \vec{\Psi})^{-1} = \frac{1}{\sigma^2} \vec{\Psi}^{-1}$$

     and

     $$|\Sigma| = |\sigma^2 \vec{\Psi}| = (\sigma^2)^n |\vec{\Psi}|.$$

     The PDF becomes:

     $$p(\vec{y}) = \frac{1}{\sqrt{(2\pi)^n (\sigma^2)^n |\vec{\Psi}|}} \exp\left[-\frac{1}{2}(\vec{y} - \vec{1}\mu)^T \left(\frac{1}{\sigma^2} \vec{\Psi}^{-1}\right) (\vec{y} - \vec{1}\mu)\right]$$

     and simplifies to:

     $$p(\vec{y}) = \frac{1}{(2\pi\sigma^2)^{n/2} |\vec{\Psi}|^{1/2}} \exp\left[-\frac{1}{2\sigma^2}(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1} (\vec{y} - \vec{1}\mu)\right].$$

This is the **likelihood of the sample data** $\vec{y}$ given the parameters μ, σ, and the correlation structure defined by the parameters within $\vec{\Psi}$ (i.e., $\vec{\theta}$ and $\vec{p}$).

In summary, the Equation 9.6 represents the likelihood under a simplified assumption of independent errors, whereas Equation 9.7 is the likelihood derived from the assumption that the observed data comes from a **multivariate normal distribution** where observations are correlated according to the matrix $\vec{\Psi}$. Equation 9.7, using the sample data vector $\vec{y}$ and the correlation matrix $\vec{\Psi}$, properly accounts for the dependencies between data points inherent in the stochastic process model. Maximizing this likelihood is how the correlation parameters $\vec{\theta}$ and $\vec{p}$ are estimated in Kriging.

□

Equation 9.7 can be formulated as the log-likelihood:

$$\ln(L) = -\frac{n}{2}\ln(2\pi\sigma) - \frac{1}{2}\ln|\vec{\Psi}| - \frac{(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{2\sigma^2}. \tag{9.8}$$

9.3.2. Differentiation with Respect to μ

Looking at the log-likelihood function, only the last term depends on μ:

$$\frac{1}{2\sigma^2}(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)$$

To differentiate this with respect to the scalar μ, we can use matrix calculus rules.

Let $\mathbf{v} = \vec{y} - \vec{1}\mu$. $\vec{y}$ is a constant vector with respect to μ, and $\vec{1}\mu$ is a vector whose derivative with respect to the scalar μ is $\vec{1}$. So, $\frac{\partial \mathbf{v}}{\partial \mu} = -\vec{1}$.

The term is in the form $\mathbf{v}^T\mathbf{A}\mathbf{v}$, where $\mathbf{A} = \vec{\Psi}^{-1}$ is a symmetric matrix. The derivative of $\mathbf{v}^T\mathbf{A}\mathbf{v}$ with respect to $\mathbf{v}$ is $2\mathbf{A}\mathbf{v}$ as explained in Remark 9.3.

Remark 9.3 (Derivative of a Quadratic Form). Consider the derivative of $\mathbf{v}^T\mathbf{A}\mathbf{v}$ with respect to $\mathbf{v}$:

- The derivative of a scalar function $f(\mathbf{v})$ with respect to a vector $\mathbf{v}$ is a vector (the gradient).
- For a quadratic form $\mathbf{v}^T\mathbf{A}\mathbf{v}$, where $\mathbf{A}$ is a matrix and $\mathbf{v}$ is a vector, the general formula for the derivative with respect to $\mathbf{v}$ is $\frac{\partial}{\partial \mathbf{v}}(\mathbf{v}^T\mathbf{A}\mathbf{v}) = \mathbf{A}\mathbf{v} + \mathbf{A}^T\mathbf{v}$. (This is a standard result in matrix calculus and explained in Equation 10.1).
- Since $\mathbf{A} = \vec{\Psi}^{-1}$ is a symmetric matrix, its transpose $\mathbf{A}^T$ is equal to $\mathbf{A}$.
- Substituting $\mathbf{A}^T = \mathbf{A}$ into the general derivative formula, we get $\mathbf{A}\mathbf{v} + \mathbf{A}\mathbf{v} = 2\mathbf{A}\mathbf{v}$.

□

Using the chain rule for differentiation with respect to the scalar μ:

$$\frac{\partial}{\partial \mu}(\mathbf{v}^T \mathbf{A} \mathbf{v}) = 2 \left(\frac{\partial \mathbf{v}}{\partial \mu} \right)^T \mathbf{A} \mathbf{v}$$

Substituting $\frac{\partial \mathbf{v}}{\partial \mu} = -\vec{1}$ and $\mathbf{v} = \vec{y} - \vec{1}\mu$:

$$\frac{\partial}{\partial \mu}(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu) = 2(-\vec{1})^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu) = -2\vec{1}^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)$$

Now, differentiate the full log-likelihood term depending on μ:

$$\frac{\partial}{\partial \mu}\left(-\frac{1}{2\sigma^2}(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu) \right) = -\frac{1}{2\sigma^2}\left(-2\vec{1}^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu) \right) = \frac{1}{\sigma^2}\vec{1}^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)$$

Setting this to zero for maximization gives:

$$\frac{1}{\sigma^2}\vec{1}^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu) = 0.$$

Rearranging gives:

$$\vec{1}^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu) = 0.$$

Solving for μ gives:

$$\vec{1}^T \vec{\Psi}^{-1}\vec{y} = \mu \vec{1}^T \vec{\Psi}^{-1}\vec{1}.$$

9.3.3. Differentiation with Respect to σ

Let $\nu = \sigma^2$ for simpler differentiation notation. The log-likelihood becomes:

$$\ln(L) = C_1 - \frac{n}{2}\ln(\nu) - \frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)}{2\nu},$$

where $C_1 = -\frac{n}{2}\ln(2\pi) - \frac{1}{2}\ln|\vec{\Psi}|$ is a constant with respect to $\nu = \sigma^2$.
We differentiate with respect to ν:

$$\frac{\partial \ln(L)}{\partial \nu} = \frac{\partial}{\partial \nu}\left(-\frac{n}{2}\ln(\nu) \right) + \frac{\partial}{\partial \nu}\left(-\frac{(\vec{y} - \vec{1}\mu)^T \vec{\Psi}^{-1}(\vec{y} - \vec{1}\mu)}{2\nu} \right).$$

The first term's derivative is straightforward:

$$\frac{\partial}{\partial \nu}\left(-\frac{n}{2}\ln(\nu) \right) = -\frac{n}{2} \cdot \frac{1}{\nu} = -\frac{n}{2\sigma^2}.$$

For the second term, let $C_2 = (\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)$. This term is constant with respect to σ^2. The derivative is:

$$\frac{\partial}{\partial\nu}\left(-\frac{C_2}{2\nu}\right) = -\frac{C_2}{2}\frac{\partial}{\partial\nu}(\nu^{-1}) = -\frac{C_2}{2}(-\nu^{-2}) = \frac{C_2}{2\nu^2} = \frac{(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{2(\sigma^2)^2}.$$

Combining the derivatives, the gradient of the log-likelihood with respect to σ^2 is:

$$\frac{\partial \ln(L)}{\partial\sigma^2} = -\frac{n}{2\sigma^2} + \frac{(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{2(\sigma^2)^2}.$$

Setting this to zero for maximization gives:

$$-\frac{n}{2\sigma^2} + \frac{(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{2(\sigma^2)^2} = 0.$$

9.3.4. Results of the Optimizations

Optimization of the log-likelihood by taking derivatives with respect to μ and σ results in

$$\hat{\mu} = \frac{\vec{1}^T\vec{\Psi}^{-1}\vec{y}^T}{\vec{1}^T\vec{\Psi}^{-1}\vec{1}^T} \tag{9.9}$$

and

$$\hat{\sigma}^2 = \frac{(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{n}. \tag{9.10}$$

9.3.5. The Concentrated Log-Likelihood Function

Combining the equations, i.e., substituting Equation 9.9 and Equation 9.10 into Equation 9.8 leads to the concentrated log-likelihood function:

$$\ln(L) \approx -\frac{n}{2}\ln(\hat{\sigma}) - \frac{1}{2}\ln|\vec{\Psi}|. \tag{9.11}$$

Remark 9.4 (The Concentrated Log-Likelihood).

- The first term in Equation 9.11 requires information about the measured point (observations) y_i.
- To maximize $\ln(L)$, optimal values of $\vec{\theta}$ and $\vec{p}$ are determined numerically, because the function (Equation 9.11) is not differentiable.

□

9.3.6. Optimizing the Parameters $\vec{\theta}$ and $\vec{p}$

The concentrated log-likelihood function is very quick to compute. We do not need a statistical model, because we are only interested in the maximum likelihood estimate (MLE) of θ and p. Optimizers such as Nelder-Mead, Conjugate Gradient, or Simulated Annealing can be used to determine optimal values for θ and p. After the optimization, the correlation matrix Ψ is build with the optimized θ and p values. This is best (most likely) Kriging model for the given data y.

Observing Figure 9.2, there's significant change between $\theta = 0.1$ and $\theta = 1$, just as there is between $\theta = 1$ and $\theta = 10$. Hence, it is sensible to search for θ on a logarithmic scale. Suitable search bounds typically range from 10^{-3} to 10^2, although this is not a stringent requirement. Importantly, the scaling of the observed data does not affect the values of $\hat{\theta}$, but the scaling of the design space does. Therefore, it is advisable to consistently scale variable ranges between zero and one to ensure consistency in the degree of activity $\hat{\theta}_j$ represents across different problems.

9.3.7. Correlation and Covariance Matrices Revisited

The covariance matrix Σ is constructed by considering both the variance σ^2 and the correlation matrix Ψ. They are related as follows:

1. **Covariance vs. Correlation:** Covariance is a measure of the joint variability of two random variables, while correlation is a standardized measure of this relationship, ranging from -1 to 1. The relationship between covariance and correlation for two random variables X and Y is given by $\text{cor}(X, Y) = \text{cov}(X, Y)/(\sigma_X \sigma_Y)$, where σ_X and σ_Y are their standard deviations.
2. **The Covariance Matrix Σ:** The **covariance matrix** Σ (or $\text{Cov}(Y, Y)$ for the vector $\vec{Y}$) captures the **pairwise covariances** between all elements of the vector of observed responses.
3. **Connecting σ^2 and Ψ to Σ:** In the Kriging framework described, the variance of each observation is often assumed to be constant, σ^2. The covariance between any two observations $Y(\vec{x}^{(i)})$ and $Y(\vec{x}^{(l)})$ is given by σ^2 multiplied by their correlation. That is,
$$\text{cov}(Y(\vec{x}^{(i)}), Y(\vec{x}^{(l)})) = \sigma^2 \text{cor}(Y(\vec{x}^{(i)}), Y(\vec{x}^{(l)})).$$
This relationship holds for *all* pairs of points. When expressed in matrix form, the covariance matrix Σ is the product of the variance σ^2 (a scalar) and the correlation matrix Ψ:
$$\Sigma = \sigma^2 \Psi.$$

In essence, the correlation matrix Ψ defines the *structure* or *shape* of the dependencies between the data points based on their locations. The parameter σ^2 acts as a **scaling factor** that converts these unitless correlation values (which are between -1 and 1) into

actual covariance values with units of variance, setting the overall level of variability in the system.

So, σ^2 tells us about the general spread or variability of the underlying process, while Ψ tells you *how* that variability is distributed and how strongly points are related to each other based on their positions. Together, they completely define the covariance structure of your observed data in the multivariate normal distribution used in Kriging.

9.4. Implementing an MLE of the Model Parameters

The matrix algebra necessary for calculating the likelihood is the most computationally intensive aspect of the Kriging process. It is crucial to ensure that the code implementation is as efficient as possible.

Given that Ψ (our correlation matrix) is symmetric, only half of the matrix needs to be computed before adding it to its transpose. When calculating the log-likelihood, several matrix inversions are required. The fastest approach is to conduct one Cholesky factorization and then apply backward and forward substitution for each inverse.

The Cholesky factorization is applicable only to positive-definite matrices, which Ψ generally is. However, if Ψ becomes nearly singular, such as when the $\vec{x}^{(i)}$'s are densely packed, the Cholesky factorization might fail. In these cases, one could employ an LU-decomposition, though the result might be unreliable. When Ψ is near singular, the best course of action is to either use regression techniques or, as we do here, assign a poor likelihood value to parameters generating the near singular matrix, thus diverting the MLE search towards better-conditioned Ψ matrices.

When working with correlation matrices, increasing the values on the main diagonal of a matrix will increase the absolute value of its determinant. A critical numerical consideration in calculating the concentrated log-likelihood is that for poorly conditioned matrices, $\det(\Psi)$ approaches zero, leading to potential numerical instability. To address this issue, it is advisable to calculate $\ln(|\Psi|)$ in Equation 9.11 using twice the sum of the logarithms of the diagonal elements of the Cholesky factorization. This approach provides a more numerically stable method for computing the log-determinant, as the Cholesky decomposition $\Psi = LL^T$ allows us to express $\ln(|\Psi|) = 2\sum_{i=1}^{n} \ln(L_{ii})$, avoiding the direct computation of potentially very small determinant values.

9.5. Kriging Prediction

We will use the Kriging correlation Ψ to predict new values based on the observed data. The presentation follows the approach described in Forrester, Sóbester, and Keane (2008) and Bartz et al. (2022).

Main idea for prediction is that the new $Y(\vec{x})$ should be consistent with the old sample data X. For a new prediction $\hat{y}$ at $\vec{x}$, the value of $\hat{y}$ is chosen so that it maximizes the likelihood of the sample data X and the prediction, given the (optimized) correlation parameter $\vec{\theta}$ and $\vec{p}$ from above. The observed data $\vec{y}$ is augmented with the new prediction $\hat{y}$ which results in the augmented vector $\tilde{\vec{y}} = (\vec{y}^T, \hat{y})^T$. A vector of correlations between the observed data and the new prediction is defined as

$$\vec{\psi} = \begin{pmatrix} \operatorname{cor}\left(Y(\vec{x}^{(1)}), Y(\vec{x})\right) \\ \vdots \\ \operatorname{cor}\left(Y(\vec{x}^{(n)}), Y(\vec{x})\right) \end{pmatrix} = \begin{pmatrix} \vec{\psi}^{(1)} \\ \vdots \\ \vec{\psi}^{(n)} \end{pmatrix}.$$

Definition 9.2 (The Augmented Correlation Matrix)**.** The augmented correlation matrix is constructed as

$$\tilde{\vec{\Psi}} = \begin{pmatrix} \vec{\Psi} & \vec{\psi} \\ \vec{\psi}^T & 1 \end{pmatrix}.$$

□

The log-likelihood of the augmented data is

$$\ln(L) = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln(\hat{\sigma}^2) - \frac{1}{2}\ln|\tilde{\vec{\Psi}}| - \frac{(\tilde{\vec{y}} - \vec{1}\hat{\mu})^T \tilde{\vec{\Psi}}^{-1} (\tilde{\vec{y}} - \vec{1}\hat{\mu})}{2\hat{\sigma}^2}, \tag{9.12}$$

where $\vec{1}$ is a vector of ones and $\hat{\mu}$ and $\hat{\sigma}^2$ are the MLEs from Equation 9.9 and Equation 9.10. Only the last term in Equation 9.12 depends on $\hat{y}$, so we need only consider this term in the maximization. Details can be found in Forrester, Sóbester, and Keane (2008). Finally, the MLE for $\hat{y}$ can be calculated as

$$\hat{y}(\vec{x}) = \hat{\mu} + \vec{\psi}^T \tilde{\vec{\Psi}}^{-1} (\vec{y} - \vec{1}\hat{\mu}). \tag{9.13}$$

Equation 9.13 reveals two important properties of the Kriging predictor:

- Basis functions: The basis function impacts the vector $\vec{\psi}$, which contains the n correlations between the new point $\vec{x}$ and the observed locations. Values from the n basis functions are added to a mean base term μ with weightings

$$\vec{w} = \tilde{\vec{\Psi}}^{(-1)} (\vec{y} - \vec{1}\hat{\mu}).$$

- Interpolation: The predictions interpolate the sample data. When calculating the prediction at the ith sample point, $\vec{x}^{(i)}$, the ith column of $\vec{\Psi}^{-1}$ is $\vec{\psi}$, and $\vec{\psi}\vec{\Psi}^{-1}$ is the ith unit vector. Hence,

$$\hat{y}(\vec{x}^{(i)}) = y^{(i)}.$$

9.6. Kriging Example: Sinusoid Function

Toy example in 1d where the response is a simple sinusoid measured at eight equally spaced x-locations in the span of a single period of oscillation.

9.6.1. Calculating the Correlation Matrix Ψ

The correlation matrix Ψ is based on the pairwise squared distances between the input locations. Here we will use $n = 8$ sample locations and θ is set to 1.0.

```
n = 8
X = np.linspace(0, 2*np.pi, n, endpoint=False).reshape(-1,1)
print(np.round(X, 2))
```

```
[[0.  ]
 [0.79]
 [1.57]
 [2.36]
 [3.14]
 [3.93]
 [4.71]
 [5.5 ]]
```

Evaluate at sample points

```
y = np.sin(X)
print(np.round(y, 2))
```

```
[[ 0.  ]
 [ 0.71]
 [ 1.  ]
 [ 0.71]
 [ 0.  ]
 [-0.71]
 [-1.  ]
 [-0.71]]
```

We have the data points shown in Table 9.1.

Table 9.1.: Data points for the sinusoid function

x	y
0.0	0.0
0.79	0.71
1.57	1.0
2.36	0.71
3.14	0.0
3.93	-0.71
4.71	-1.0
5.5	-0.71

The data points are visualized in Figure 9.4.

```
plt.plot(X, y, "bo")
plt.title(f"Sin(x) evaluated at {n} points")
plt.grid()
plt.show()
```

Figure 9.4.: Sin(x) evaluated at 8 points.

9.6.2. Computing the Ψ Matrix

We will use the `build_Psi` function from Example 9.4 to compute the correlation matrix Ψ. θ should be an array of one value, because we are only working in one dimension ($k = 1$).

```
theta = np.array([1.0])
Psi = build_Psi(X, theta)
print(np.round(Psi, 2))
```

```
[[1.   0.54 0.08 0.   0.   0.   0.   0.  ]
 [0.54 1.   0.54 0.08 0.   0.   0.   0.  ]
 [0.08 0.54 1.   0.54 0.08 0.   0.   0.  ]
 [0.   0.08 0.54 1.   0.54 0.08 0.   0.  ]
 [0.   0.   0.08 0.54 1.   0.54 0.08 0.  ]
 [0.   0.   0.   0.08 0.54 1.   0.54 0.08]
 [0.   0.   0.   0.   0.08 0.54 1.   0.54]
 [0.   0.   0.   0.   0.   0.08 0.54 1.  ]]
```

Figure 9.5 visualizes the $(8, 8)$ correlation matrix Ψ.

Figure 9.5.: Correlation matrix Ψ for the sinusoid function.

9.6.3. Selecting the New Locations

We would like to predict at $m = 100$ new locations (or testign locations) in the interval $[0, 2\pi]$. The new locations are stored in the variable `x`.

```
m = 100
x = np.linspace(0, 2*np.pi, m, endpoint=False).reshape(-1,1)
```

9.6.4. Computing the ψ Vector

Distances between testing locations x and training data locations X.

```
def build_psi(X, x, theta, eps=sqrt(spacing(1))):
    n = X.shape[0]
    k = X.shape[1]
    m = x.shape[0]
    psi = zeros((n, m))
    theta = theta * ones(k)
    D = zeros((n, m))
    D = cdist(x.reshape(-1, k),
              X.reshape(-1, k),
              metric='sqeuclidean',
              out=None,
              w=theta)
    psi = exp(-D)
    # return psi transpose to be consistent with the literature
    print(f"Dimensions of psi: {psi.T.shape}")
    return(psi.T)

psi = build_psi(X, x, theta)
```

```
Dimensions of psi: (8, 100)
```

Figure 9.6 visualizes the $(8, 100)$ prediction matrix ψ.

Figure 9.6.: Visualization of the predition matrix ψ

9.6.5. Predicting at New Locations

Computation of the predictive equations.

```
U = cholesky(Psi).T
one = np.ones(n).reshape(-1,1)
mu = (one.T.dot(solve(U, solve(U.T, y)))) / one.T.dot(solve(U, solve(U.T, one)))
f = mu * ones(m).reshape(-1,1) + psi.T.dot(solve(U, solve(U.T, y - one * mu)))
print(f"Dimensions of f: {f.shape}")
```

```
Dimensions of f: (100, 1)
```

To compute f, Equation 9.13 is used.

9.6.6. Visualization

```
plt.plot(x, f, color = "orange", label="Fitted")
plt.plot(x, np.sin(x), color = "grey", label="Original")
plt.plot(X, y, "bo", label="Measurements")
plt.title("Kriging prediction of sin(x) with {} points.\n theta: {}".format(n, theta[0]))
plt.legend(loc='upper right')
plt.show()
```

9.6.7. The Complete Python Code for the Example

Here is the self-contained Python code for direct use in a notebook:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import (array, zeros, power, ones, exp, multiply, eye, linspace, spacing, s
from numpy.linalg import cholesky, solve
from scipy.spatial.distance import squareform, pdist, cdist

# --- 1. Kriging Basis Functions (Defining the Correlation) ---
# The core of Kriging uses a specialized basis function for correlation:
# psi(x^(i), x) = exp(- sum_{j=1}^k theta_j |x_j^(i) - x_j|^p_j)
# For this 1D example (k=1), and with p_j=2 (squared Euclidean distance implicit from
# and theta_j = theta (a single value), it simplifies.

def build_Psi(X, theta, eps=sqrt(spacing(1))):
    """
    Computes the correlation matrix Psi based on pairwise squared Euclidean distances
    between input locations, scaled by theta.
    Adds a small epsilon to the diagonal for numerical stability (nugget effect).
```

```
    """
    # Calculate pairwise squared Euclidean distances (D) between points in X
    D = squareform(pdist(X, metric='sqeuclidean', out=None, w=theta))
    # Compute Psi = exp(-D)
    Psi = exp(-D)
    # Add a small value to the diagonal for numerical stability (nugget)
    # This is often done in Kriging implementations, though a regression method
    # with a 'nugget' parameter (Lambda) is explicitly mentioned for noisy data later.
    # The source code snippet for build_Psi explicitly includes `multiply(eye(X.shape), eps)`.
    # FIX: Use X.shape to get the number of rows for the identity matrix
    Psi += multiply(eye(X.shape[0]), eps) # Corrected line
    return Psi

def build_psi(X_train, x_predict, theta):
    """
    Computes the correlation vector (or matrix) psi between new prediction locations
    and training data locations.
    """
    # Calculate pairwise squared Euclidean distances (D) between prediction points (x_predict)
    # and training points (X_train).
    # `cdist` computes distances between each pair of the two collections of inputs.
    D = cdist(x_predict, X_train, metric='sqeuclidean', out=None, w=theta)
    # Compute psi = exp(-D)
    psi = exp(-D)
    return psi.T # Return transpose to be consistent with literature (n x m or n x 1)

# --- 2. Data Points for the Sinusoid Function Example ---
# The example uses a 1D sinusoid measured at eight equally spaced x-locations [153, Table 9.1].
n = 8 # Number of sample locations
X_train = np.linspace(0, 2 * np.pi, n, endpoint=False).reshape(-1, 1) # Generate x-locations
y_train = np.sin(X_train) # Corresponding y-values (sine of x)

print("--- Training Data (X_train, y_train) ---")
print("x values:\n", np.round(X_train, 2))
print("y values:\n", np.round(y_train, 2))
print("-" * 40)

# Visualize the data points
plt.figure(figsize=(8, 5))
plt.plot(X_train, y_train, "bo", label=f"Measurements ({n} points)")
plt.title(f"Sin(x) evaluated at {n} points")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.grid(True)
```

```
plt.legend()
plt.show()

# --- 3. Calculating the Correlation Matrix (Psi) ---
# Psi is based on pairwise squared distances between input locations.
# theta is set to 1.0 for this 1D example.
theta = np.array([1.0])
Psi = build_Psi(X_train, theta)

print("\n--- Computed Correlation Matrix (Psi) ---")
print("Dimensions of Psi:", Psi.shape) # Should be (8, 8)
print("First 5x5 block of Psi:\n", np.round(Psi[:5,:5], 2))
print("-" * 40)

# --- 4. Selecting New Locations (for Prediction) ---
# We want to predict at m = 100 new locations in the interval [0, 2*pi].
m = 100 # Number of new locations
x_predict = np.linspace(0, 2 * np.pi, m, endpoint=True).reshape(-1, 1)

print("\n--- New Locations for Prediction (x_predict) ---")
print(f"Number of prediction points: {m}")
print("First 5 prediction points:\n", np.round(x_predict[:5], 2).flatten())
print("-" * 40)

# --- 5. Computing the psi Vector ---
# This vector contains correlations between each of the n observed data points
# and each of the m new prediction locations.
psi = build_psi(X_train, x_predict, theta)

print("\n--- Computed Prediction Correlation Matrix (psi) ---")
print("Dimensions of psi:", psi.shape) # Should be (8, 100)
print("First 5x5 block of psi:\n", np.round(psi[:5,:5], 2))
print("-" * 40)

# --- 6. Predicting at New Locations (Kriging Prediction) ---
# The Maximum Likelihood Estimate (MLE) for y_hat is calculated using the formula:
# y_hat(x) = mu_hat + psi.T @ Psi_inv @ (y - 1 * mu_hat) [p. 2 of previous response, a
# Matrix inversion is efficiently performed using Cholesky factorization.

# Step 6a: Cholesky decomposition of Psi
U = cholesky(Psi).T # Note: `cholesky` in numpy returns lower triangular L, we need U

# Step 6b: Calculate mu_hat (estimated mean)
# mu_hat = (one.T @ Psi_inv @ y) / (one.T @ Psi_inv @ one) [p. 2 of previous response]
```

```
one = np.ones(n).reshape(-1, 1) # Vector of ones
mu_hat = (one.T @ solve(U, solve(U.T, y_train))) / (one.T @ solve(U, solve(U.T, one)))
mu_hat = mu_hat.item() # Extract scalar value

print("\n--- Kriging Prediction Calculation ---")
print(f"Estimated mean (mu_hat): {np.round(mu_hat, 4)}")

# Step 6c: Calculate predictions f (y_hat) at new locations
# f = mu_hat * ones(m) + psi.T @ Psi_inv @ (y - one * mu_hat)
f_predict = mu_hat * np.ones(m).reshape(-1, 1) + psi.T @ solve(U, solve(U.T, y_train - one * mu_hat)

print(f"Dimensions of predicted values (f_predict): {f_predict.shape}") # Should be (100, 1)
print("First 5 predicted f values:\n", np.round(f_predict[:5], 2).flatten())
print("-" * 40)

# --- 7. Visualization ---
# Plot the original sinusoid function, the measured points, and the Kriging predictions.

plt.figure(figsize=(10, 6))
plt.plot(x_predict, f_predict, color="orange", label="Kriging Prediction")
plt.plot(x_predict, np.sin(x_predict), color="grey", linestyle='--', label="True Sinusoid Function")
plt.plot(X_train, y_train, "bo", markersize=8, label="Measurements")
plt.title(f"Kriging prediction of sin(x) with {n} points. (theta: {theta})")
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```

```
--- Training Data (X_train, y_train) ---
x values:
 [[0.  ]
 [0.79]
 [1.57]
 [2.36]
 [3.14]
 [3.93]
 [4.71]
 [5.5 ]]
y values:
 [[ 0.  ]
 [ 0.71]
 [ 1.  ]
 [ 0.71]
```

```
 [ 0.  ]
 [-0.71]
 [-1.  ]
 [-0.71]]
----------------------------------------
```

```
--- Computed Correlation Matrix (Psi) ---
Dimensions of Psi: (8, 8)
First 5x5 block of Psi:
 [[1.   0.54 0.08 0.   0.  ]
 [0.54 1.   0.54 0.08 0.  ]
 [0.08 0.54 1.   0.54 0.08]
 [0.   0.08 0.54 1.   0.54]
 [0.   0.   0.08 0.54 1.  ]]
----------------------------------------

--- New Locations for Prediction (x_predict) ---
Number of prediction points: 100
First 5 prediction points:
 [0.   0.06 0.13 0.19 0.25]
----------------------------------------
```

```
--- Computed Prediction Correlation Matrix (psi) ---
Dimensions of psi: (8, 100)
First 5x5 block of psi:
 [[1.   1.   0.98 0.96 0.94]
 [0.54 0.59 0.65 0.7  0.75]
 [0.08 0.1  0.12 0.15 0.18]
 [0.   0.01 0.01 0.01 0.01]
 [0.   0.   0.   0.   0.  ]]
----------------------------------------

--- Kriging Prediction Calculation ---
Estimated mean (mu_hat): -0.0499
Dimensions of predicted values (f_predict): (100, 1)
First 5 predicted f values:
 [0.   0.05 0.1  0.15 0.21]
----------------------------------------
```

9.7. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

10. Matrices

10.1. Derivatives of Quadratic Forms

We present a step-by-step derivation of the general formula

$$\frac{\partial}{\partial \mathbf{v}}(\mathbf{v}^T\mathbf{A}\mathbf{v}) = \mathbf{A}\mathbf{v} + \mathbf{A}^T\mathbf{v}. \tag{10.1}$$

1. Define the components. Let $\mathbf{v}$ be a vector of size $n \times 1$, and let $\mathbf{A}$ be a matrix of size $n \times n$.

2. Write out the quadratic form in summation notation. The product $\mathbf{v}^T\mathbf{A}\mathbf{v}$ is a scalar. It can be expanded and be rewritten as a double summation:

$$\mathbf{v}^T\mathbf{A}\mathbf{v} = \sum_{i=1}^{n}\sum_{j=1}^{n} v_i a_{ij} v_j.$$

3. Calculate the partial derivative with respect to a component v_k: The derivative of the scalar $\mathbf{v}^T\mathbf{A}\mathbf{v}$ with respect to the vector $\mathbf{v}$ is the gradient vector, whose k-th component is $\frac{\partial}{\partial v_k}(\mathbf{v}^T\mathbf{A}\mathbf{v})$. We need to find $\frac{\partial}{\partial v_k}\left(\sum_{i=1}^{n}\sum_{j=1}^{n} v_i a_{ij} v_j\right)$. Consider the terms in the summation that involve v_k. A term $v_i a_{ij} v_j$ involves v_k if $i = k$ or $j = k$ (or both).

   - Terms where $i = k$: $v_k a_{kj} v_j$. The derivative with respect to v_k is $a_{kj} v_j$.
   - Terms where $j = k$: $v_i a_{ik} v_k$. The derivative with respect to v_k is $v_i a_{ik}$.
   - The term where $i = k$ and $j = k$: $v_k a_{kk} v_k = a_{kk} v_k^2$. Its derivative with respect to v_k is $2a_{kk}v_k$. Notice this term is included in both cases above when $i = k$ and $j = k$. When $i = k$, the term is $v_k a_{kk} v_k$, derivative is $a_{kk} v_k$. When $j = k$, the term is $v_k a_{kk} v_k$, derivative is $v_k a_{kk}$. Summing these two gives $2a_{kk}v_k$.

4. Let's differentiate the sum $\sum_{i=1}^{n}\sum_{j=1}^{n} v_i a_{ij} v_j$ with respect to v_k:

$$\frac{\partial}{\partial v_k}\left(\sum_{i=1}^{n}\sum_{j=1}^{n} v_i a_{ij} v_j\right) = \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{\partial}{\partial v_k}(v_i a_{ij} v_j).$$

5. The partial derivative $\frac{\partial}{\partial v_k}(v_i a_{ij} v_j)$ is non-zero only if $i = k$ or $j = k$.

- If $i = k$ and $j \neq k$: $\frac{\partial}{\partial v_k}(v_k a_{kj} v_j) = a_{kj} v_j$.
- If $i \neq k$ and $j = k$: $\frac{\partial}{\partial v_k}(v_i a_{ik} v_k) = v_i a_{ik}$.
- If $i = k$ and $j = k$: $\frac{\partial}{\partial v_k}(v_k a_{kk} v_k) = \frac{\partial}{\partial v_k}(a_{kk} v_k^2) = 2a_{kk} v_k$.

6. So, the partial derivative is the sum of derivatives of all terms involving v_k: $\frac{\partial}{\partial v_k}(\mathbf{v}^T \mathbf{A} \mathbf{v}) = \sum_{j \neq k}(a_{kj} v_j) + \sum_{i \neq k}(v_i a_{ik}) + (2a_{kk} v_k)$.

7. We can rewrite this by including the $i = k, j = k$ term back into the summations: $\sum_{j \neq k}(a_{kj} v_j) + a_{kk} v_k + \sum_{i \neq k}(v_i a_{ik}) + v_k a_{kk}$ (since $v_k a_{kk} = a_{kk} v_k$) $= \sum_{j=1}^{n} a_{kj} v_j + \sum_{i=1}^{n} v_i a_{ik}$.

8. Convert back to matrix/vector notation: The first summation $\sum_{j=1}^{n} a_{kj} v_j$ is the k-th component of the matrix-vector product $\mathbf{A}\mathbf{v}$.The second summation $\sum_{i=1}^{n} v_i a_{ik}$ can be written as $\sum_{i=1}^{n} a_{ik} v_i$. Recall that the element in the k-th row and i-th column of the transpose matrix $\mathbf{A}^T$ is $(A^T)_{ki} = a_{ik}$. So, $\sum_{i=1}^{n} a_{ik} v_i = \sum_{i=1}^{n} (A^T)_{ki} v_i$, which is the k-th component of the matrix-vector product $\mathbf{A}^T \mathbf{v}$.

9. Assemble the gradient vector: The k-th component of the gradient $\frac{\partial}{\partial \mathbf{v}}(\mathbf{v}^T \mathbf{A} \mathbf{v})$ is $(\mathbf{A}\mathbf{v})_k + (\mathbf{A}^T \mathbf{v})_k$. Since this holds for all $k = 1, \ldots, n$, the gradient vector is the sum of the two vectors $\mathbf{A}\mathbf{v}$ and $\mathbf{A}^T \mathbf{v}$. Therefore, the general formula for the derivative is $\frac{\partial}{\partial \mathbf{v}}(\mathbf{v}^T \mathbf{A} \mathbf{v}) = \mathbf{A}\mathbf{v} + \mathbf{A}^T \mathbf{v}$.

10.2. The Condition Number

A small value, `eps`, can be passed to the function `build_Psi` to improve the condition number. For example, `eps=sqrt(spacing(1))` can be used. The numpy function `spacing()` returns the distance between a number and its nearest adjacent number.

The condition number of a matrix is a measure of its sensitivity to small changes in its elements. It is used to estimate how much the output of a function will change if the input is slightly altered.

A matrix with a low condition number is well-conditioned, which means its behavior is relatively stable, while a matrix with a high condition number is ill-conditioned, meaning its behavior is unstable with respect to numerical precision.

```
import numpy as np

# Define a well-conditioned matrix (low condition number)
A = np.array([[1, 0.1], [0.1, 1]])
print("Condition number of A: ", np.linalg.cond(A))

# Define an ill-conditioned matrix (high condition number)
```

```
B = np.array([[1, 0.99999999], [0.99999999, 1]])
print("Condition number of B: ", np.linalg.cond(B))
```

```
Condition number of A:  1.2222222222222225
Condition number of B:  200000000.57495335
```

10.3. The Moore-Penrose Pseudoinverse

10.3.1. Definitions

The Moore-Penrose pseudoinverse is a generalization of the inverse matrix for non-square or singular matrices. It is computed as

$$A^+ = (A^*A)^{-1}A^*,$$

where A^* is the conjugate transpose of A.

It satisfies the following properties:

1. $AA^+A = A$
2. $A^+AA^+ = A^+$
3. $(AA^+)^* = AA^+$.
4. $(A^+A)^* = A^+A$
5. $A^+ = (A^*)^+$
6. $A^+ = A^T$ if A is a square matrix and A is invertible.

The pseudoinverse can be computed using Singular Value Decomposition (SVD).

10.3.2. Implementation in Python

```
import numpy as np
from numpy.linalg import pinv
A = np.array([[1, 2], [3, 4], [5, 6]])
print(f"Matrix A:\n {A}")
A_pseudo_inv = pinv(A)
print(f"Moore-Penrose Pseudoinverse:\n {A_pseudo_inv}")
```

```
Matrix A:
 [[1 2]
 [3 4]
 [5 6]]
Moore-Penrose Pseudoinverse:
 [[-1.33333333 -0.33333333  0.66666667]
 [ 1.08333333  0.33333333 -0.41666667]]
```

10.4. Strictly Positive Definite Kernels

10.4.1. Definition

Definition 10.1 (Strictly Positive Definite Kernel)**.** A kernel function $k(x, y)$ is called strictly positive definite if for any finite collection of distinct points $x_1, x_2, \dots, x_n$ in the input space and any non-zero vector of coefficients $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n)$, the following inequality holds:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j k(x_i, x_j) > 0. \tag{10.2}$$

In contrast, a kernel function $k(x, y)$ is called positive definite (but not strictly) if the "$>$" sign is replaced by "$\geq$" in the above inequality.

10.4.2. Connection to Positive Definite Matrices

The connection between strictly positive definite kernels and positive definite matrices lies in the Gram matrix construction:

- When we evaluate a kernel function $k(x, y)$ at all pairs of data points in our sample, we construct the Gram matrix K where $K_{ij} = k(x_i, x_j)$.
- If the kernel function k is strictly positive definite, then for any set of distinct points, the resulting Gram matrix will be symmetric positive definite.

A symmetric matrix is positive definite if and only if for any non-zero vector α, the quadratic form $\alpha^T K \alpha > 0$, which directly corresponds to the kernel definition above.

10.4.3. Connection to RBF Models

For RBF models, the kernel function is the radial basis function itself:

$$k(x, y) = \psi(||x - y||).$$

The Gaussian RBF kernel $\psi(r) = e^{-r^2/(2\sigma^2)}$ is strictly positive definite in $\mathbb{R}^n$ for any dimension n. The inverse multiquadric kernel $\psi(r) = (r^2 + \sigma^2)^{-1/2}$ is also strictly positive definite in any dimension.

This mathematical property guarantees that the interpolation problem has a unique solution (the weight vector $\vec{w}$ is uniquely determined). The linear system $\Psi\vec{w} = \vec{y}$ can be solved reliably using Cholesky decomposition. The RBF interpolant exists and is unique for any distinct set of centers.

10.5. Cholesky Decomposition and Positive Definite Matrices

We consider the definiteness of a matrix, before discussing the Cholesky decomposition.

Definition 10.2 (Positive Definite Matrix)**.** A symmetric matrix A is positive definite if all its eigenvalues are positive.

Example 10.1 (Positive Definite Matrix)**.** Given a symmetric matrix $A = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$, the eigenvalues of A are $\lambda_1 = 13$ and $\lambda_2 = 5$. Since both eigenvalues are positive, the matrix A is positive definite.

Definition 10.3 (Negative Definite, Positive Semidefinite, and Negative Semidefinite Matrices)**.** Similarily, a symmetric matrix A is negative definite if all its eigenvalues are negative. It is positive semidefinite if all its eigenvalues are non-negative, and negative semidefinite if all its eigenvalues are non-positive.

The covariance matrix must be positive definite for a multivariate normal distribution for a couple of reasons:

- Semidefinite vs Definite: A covariance matrix is always symmetric and positive semidefinite. However, for a multivariate normal distribution, it must be positive definite, not just semidefinite. This is because a positive semidefinite matrix can have zero eigenvalues, which would imply that some dimensions in the distribution have zero variance, collapsing the distribution in those dimensions. A positive definite matrix has all positive eigenvalues, ensuring that the distribution has positive variance in all dimensions.

- Invertibility: The multivariate normal distribution's probability density function involves the inverse of the covariance matrix. If the covariance matrix is not positive definite, it may not be invertible, and the density function would be undefined.

In summary, the covariance matrix being positive definite ensures that the multivariate normal distribution is well-defined and has positive variance in all dimensions.

The definiteness of a matrix can be checked by examining the eigenvalues of the matrix. If all eigenvalues are positive, the matrix is positive definite.

```
import numpy as np

def is_positive_definite(matrix):
    return np.all(np.linalg.eigvals(matrix) > 0)

matrix = np.array([[9, 4], [4, 9]])
print(is_positive_definite(matrix))  # Outputs: True
```

```
True
```

However, a more efficient way to check the definiteness of a matrix is through the Cholesky decomposition.

Definition 10.4 (Cholesky Decomposition)**.** For a given symmetric positive-definite matrix $A \in \mathbb{R}^{n \times n}$, there exists a unique lower triangular matrix $L \in \mathbb{R}^{n \times n}$ with positive diagonal elements such that:

$$A = LL^T.$$

Here, L^T denotes the transpose of L.

Example 10.2 (Cholesky decomposition using `numpy`)**.** `linalg.cholesky` computes the Cholesky decomposition of a matrix, i.e., it computes a lower triangular matrix L such that $LL^T = A$. If the matrix is not positive definite, an error (`LinAlgError`) is raised.

```
import numpy as np

# Define a Hermitian, positive-definite matrix
A = np.array([[9, 4], [4, 9]])

# Compute the Cholesky decomposition
L = np.linalg.cholesky(A)
```

```
print("L = \n", L)
print("L*LT = \n", np.dot(L, L.T))
```

```
L =
 [[3.         0.        ]
 [1.33333333 2.68741925]]
L*LT =
 [[9. 4.]
 [4. 9.]]
```

Example 10.3 (Cholesky Decomposition)**.** Given a symmetric positive-definite matrix $A = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$, the Cholesky decomposition computes the lower triangular matrix L such that $A = LL^T$. The matrix L is computed as:

$$L = \begin{pmatrix} 3 & 0 \\ 4/3 & 2 \end{pmatrix},$$

so that

$$LL^T = \begin{pmatrix} 3 & 0 \\ 4/3 & \sqrt{65}/3 \end{pmatrix} \begin{pmatrix} 3 & 4/3 \\ 0 & \sqrt{65}/3 \end{pmatrix} = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix} = A.$$

An efficient implementation of the definiteness-check based on Cholesky is already available in the `numpy` library. It provides the `np.linalg.cholesky` function to compute the Cholesky decomposition of a matrix. This more efficient `numpy`-approach can be used as follows:

```
import numpy as np

def is_pd(K):
    try:
        np.linalg.cholesky(K)
        return True
    except np.linalg.linalg.LinAlgError as err:
        if 'Matrix is not positive definite' in err.message:
            return False
        else:
            raise
matrix = np.array([[9, 4], [4, 9]])
print(is_pd(matrix))  # Outputs: True
```

```
True
```

10.5.1. Example of Cholesky Decomposition

We consider dimension $k = 1$ and $n = 2$ sample points. The sample points are located at $x_1 = 1$ and $x_2 = 5$. The response values are $y_1 = 2$ and $y_2 = 10$. The correlation parameter is $\theta = 1$ and p is set to 1. Using Equation 9.1, we can compute the correlation matrix Ψ:

$$\Psi = \begin{pmatrix} 1 & e^{-1} \\ e^{-1} & 1 \end{pmatrix}.$$

To determine MLE as in Equation 9.13, we need to compute Ψ^{-1}:

$$\Psi^{-1} = \frac{e}{e^2 - 1} \begin{pmatrix} e & -1 \\ -1 & e \end{pmatrix}.$$

Cholesky-decomposition of Ψ is recommended to compute Ψ^{-1}. Cholesky decomposition is a decomposition of a positive definite symmetric matrix into the product of a lower triangular matrix L, a diagonal matrix D and the transpose of L, which is denoted as L^T. Consider the following example:

$$LDL^T = \begin{pmatrix} 1 & 0 \\ l_{21} & 1 \end{pmatrix} \begin{pmatrix} d_{11} & 0 \\ 0 & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix} =$$

$$\begin{pmatrix} d_{11} & 0 \\ d_{11}l_{21} & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} d_{11} & d_{11}l_{21} \\ d_{11}l_{21} & d_{11}l_{21}^2 + d_{22} \end{pmatrix}. \tag{10.3}$$

Using Equation 10.3, we can compute the Cholesky decomposition of Ψ:

1. $d_{11} = 1$,
2. $l_{21}d_{11} = e^{-1} \Rightarrow l_{21} = e^{-1}$, and
3. $d_{11}l_{21}^2 + d_{22} = 1 \Rightarrow d_{22} = 1 - e^{-2}$.

The Cholesky decomposition of Ψ is

$$\Psi = \begin{pmatrix} 1 & 0 \\ e^{-1} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 - e^{-2} \end{pmatrix} \begin{pmatrix} 1 & e^{-1} \\ 0 & 1 \end{pmatrix} = LDL^T$$

Some programs use U instead of L. The Cholesky decomposition of Ψ is

$$\Psi = LDL^T = U^TDU.$$

Using

$$\sqrt{D} = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1 - e^{-2}} \end{pmatrix},$$

we can write the Cholesky decomposition of Ψ without a diagonal matrix D as

$$\Psi = \begin{pmatrix} 1 & 0 \\ e^{-1} & \sqrt{1-e^{-2}} \end{pmatrix} \begin{pmatrix} 1 & e^{-1} \\ 0 & \sqrt{1-e^{-2}} \end{pmatrix} = U^T U.$$

10.5.2. Inverse Matrix Using Cholesky Decomposition

To compute the inverse of a matrix using the Cholesky decomposition, you can follow these steps:

1. Decompose the matrix A into L and L^T, where L is a lower triangular matrix and L^T is the transpose of L.
2. Compute L^{-1}, the inverse of L.
3. The inverse of A is then $(L^{-1})^T L^-1$.

Please note that this method only applies to symmetric, positive-definite matrices.

The inverse of the matrix Ψ from above is:

$$\Psi^{-1} = \frac{e}{e^2-1} \begin{pmatrix} e & -1 \\ -1 & e \end{pmatrix}.$$

Here's an example of how to compute the inverse of a matrix using Cholesky decomposition in Python:

```
import numpy as np
from scipy.linalg import cholesky, inv
E = np.exp(1)

# Psi is a symmetric, positive-definite matrix
Psi = np.array([[1, 1/E], [1/E, 1]])
L = cholesky(Psi, lower=True)
L_inv = inv(L)
# The inverse of A is (L^-1)^T * L^-1
Psi_inv = np.dot(L_inv.T, L_inv)

print("Psi:\n", Psi)
print("Psi Inverse:\n", Psi_inv)
```

```
Psi:
 [[1.         0.36787944]
 [0.36787944 1.        ]]
Psi Inverse:
 [[ 1.15651764 -0.42545906]
 [-0.42545906  1.15651764]]
```

10.6. Nyström Approximation

10.6.1. What's the Big Idea?

Imagine you have a huge, detailed map of a country. Working with the full, high-resolution map is slow and takes up a lot of computer memory. The Nyström method is like creating a smaller-scale summary map by only looking at a few key, representative locations.

In machine learning, we often work with a **kernel matrix** (or Gram matrix), which tells us how similar every pair of data points is to each other. For very large datasets, this matrix can become massive, making it computationally expensive to store and process.

The Nyström method provides an efficient way to create a **low-rank approximation** of this large kernel matrix. In simple terms, it finds a "simpler" version of the matrix that captures its most important properties without needing to compute or store the whole thing.

10.6.2. How Does It Work?

The core idea is to select a small, random subset of the columns of the full kernel matrix and use them to reconstruct the entire matrix. Let's say our full kernel matrix is K.

1. **Sample:** Randomly select l columns from the n total columns of K. Let C be the $n \times l$ matrix of these sampled columns.
2. **Intersect:** Take the rows of C corresponding to the sampled column indices to form the $l \times l$ matrix W.
3. **Approximate:** Using C and W, calculate the Nyström approximation $\tilde{K}$ of K:

$$\tilde{K} \approx CW^{+}C^{T}$$

   where W^{+} is the pseudoinverse of W.

10.6.3. Example

Suppose we have 4 data points and the full kernel matrix K is:

$$K = \begin{pmatrix} 9 & 6 & 3 & 1 \\ 6 & 4 & 2 & 0.5 \\ 3 & 2 & 1 & 0.25 \\ 1 & 0.5 & 0.25 & 0.1 \end{pmatrix}$$

Let's approximate it by sampling 2 columns ($l = 2$):

1. **Sample:** Pick the 1st and 3rd columns:

$$C = \begin{pmatrix} 9 & 3 \\ 6 & 2 \\ 3 & 1 \\ 1 & 0.25 \end{pmatrix}$$

2. **Intersect:** Take the 1st and 3rd rows from C to form W:

$$W = \begin{pmatrix} 9 & 3 \\ 3 & 1 \end{pmatrix}$$

3. **Approximate:** Suppose the pseudoinverse of W is:

$$W^+ = \begin{pmatrix} 0.09 & -0.27 \\ -0.27 & 0.81 \end{pmatrix}$$

   Then,

$$\tilde{K} = CW^+C^T = \begin{pmatrix} 9 & 6 & 3 & 0.675 \\ 6 & 4 & 2 & 0.45 \\ 3 & 2 & 1 & 0.225 \\ 0.675 & 0.45 & 0.225 & 0.05 \end{pmatrix}$$

$\tilde{K}$ is a good approximation of the original K, especially in the top-left portion.

10.6.4. Why Is This Useful?

- **Speed:** The Nyström method is much faster than computing the full kernel matrix. The complexity is roughly $O(l^2 n)$ instead of $O(n^2 d)$ (where d is the number of features).
- **Scalability:** It allows kernel methods (like SVM or Kernel PCA) to be used on much larger datasets.

- **Feature Mapping:** The method can be used to project new data points into the same feature space for prediction tasks.

The quality of the approximation depends on the columns you sample. Uniform random sampling is common and often effective, but more advanced techniques exist to select more informative columns.

10.6.5. Applying the Nyström Approximation: How Nyström Approximation Helps Kriging

Kriging can significantly benefit from the Nyström approximation, especially when dealing with large datasets. Kriging is a spatial interpolation method used to estimate values at unmeasured locations based on observed points. It relies on a **covariance matrix** (often denoted as $\mathbf{K}$) that describes the spatial correlation between all observed data points.

The Problem with Standard Kriging:

The main computational challenge in Kriging is solving for the weights needed for prediction, which requires **inverting the covariance matrix $\mathbf{K}$**. For `n` data points, $\mathbf{K}$ is an `n x n` matrix, and inverting it has computational complexity $O(n^3)$. This becomes impractical for large datasets.

The Nyström Solution:

Since the covariance matrix in Kriging is a type of kernel matrix, we can use the Nyström method to create a low-rank approximation, $\tilde{K}$. Instead of inverting the full matrix, we use the **Woodbury matrix identity** on the Nyström approximation, allowing us to efficiently compute $\tilde{K}^{-1}$ without forming the full matrix. This reduces computational complexity to roughly $O(l^2 n)$, where `l` is the number of sampled columns.

In summary, Nyström makes Kriging feasible for large-scale problems by replacing expensive matrix inversion with a faster, memory-efficient approximation.

10.6.6. Example: Predicting Temperature with Nyström-Kriging

Suppose we have temperature readings from 100 weather stations (`n=100`) and want to predict the temperature at a new location.

Data:

- Observed Locations (X): 100 coordinate pairs
- Observed Temperatures (y): 100 values
- Prediction Location (x*): Coordinates of the new location

10.6.6.1. Step 1: Nyström Approximation of the Covariance Matrix

1. **Sample Representative Points:** Randomly select `l=10` stations as landmarks.
2. **Compute C and W:**
   - **C:** Covariance between all 100 stations and the 10 landmarks (`100x10` matrix)
   - **W:** Covariance among the 10 landmarks (`10x10` matrix)

Nyström approximation: $\tilde{K} = CW^{+}C^{T}$

10.6.6.2. Step 2: Modeling and Prediction

Standard Kriging prediction:

$$y(x^*) = \mathbf{k}^{*T}\mathbf{K}^{-1}\mathbf{y}$$

where $\mathbf{k}^{*T}$ is the covariance vector between the prediction location and all observed locations.

Nyström-Kriging prediction:

$$y(x^*) \approx \mathbf{k}^{*T}(\text{fast_approx_inverse}(\mathbf{C}, \mathbf{W}))\mathbf{y}$$

Prediction Steps:

1. Calculate $\mathbf{k}^{*T}$: Covariance between new location and all stations.
2. Approximate the inverse term using the Woodbury identity with $\mathbf{C}$ and $\mathbf{W}$.
3. Make the prediction: Take the dot product of $\mathbf{k}^{*T}$ and the weights vector.

This yields an accurate prediction efficiently, enabling rapid mapping for large regions.

10.6.7. Details: Woodbury Matrix Identity for Avoiding the Big Inversion

First, what is the **Woodbury matrix identity**? It's a mathematical rule that tells you how to find the inverse of a matrix that's been modified slightly. Its most useful form is for a "low-rank update":

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}$$

This looks complicated, but the core idea is simple:

- If you have a matrix A that is **easy to invert** (like a diagonal matrix).
- And you add a low-rank matrix to it (the UCV part, where C is small).

- You can find the new inverse without directly inverting the big $(A+UCV)$ matrix. Instead, you only need to invert the much smaller matrix in the middle of the formula: $(C^{-1} + VA^{-1}U)$.

How does this apply to the Nyström approximation?

In many machine learning and Kriging applications, we don't just need the kernel matrix $\tilde{K}$, but a "regularized" version, $(\lambda I + \tilde{K})$, where λI is a diagonal matrix that helps prevent overfitting. We need to find the inverse of this:

$$(\lambda I + \tilde{K})^{-1}$$

Substituting the Nyström formula $\tilde{K} = CW^{+}C^{T}$, we get:

$$(\lambda I + CW^{+}C^{T})^{-1}$$

This expression fits the Woodbury identity perfectly!

- $A = \lambda I$ (very easy to invert: $A^{-1} = \frac{1}{\lambda} I$)
- $U = C$ (our $n \times l$ matrix)
- C (middle matrix) $= W^{+}$ (our small $l \times l$ matrix)
- $V = C^{T}$ (our $l \times n$ matrix)

By plugging these into the Woodbury formula, we get an expression for the inverse that only requires inverting a small $l \times l$ matrix. This means we never have to build the full $n \times n$ matrix $\tilde{K}$ or invert it directly. This is the source of the massive speed-up.

10.6.8. The Example: Step-by-Step

Let's reuse our 4-point example and show both the slow way and the fast Woodbury way.

Recall our matrices:

- $C = \begin{pmatrix} 9 & 3 \\ 6 & 2 \\ 3 & 1 \\ 1 & 0.25 \end{pmatrix}$
- $W^{+} = \begin{pmatrix} 0.09 & -0.27 \\ -0.27 & 0.81 \end{pmatrix}$
- Let's use a regularization value $\lambda = 0.1$.

10.6.8.1. Method 1: The Slow Way (Forming the full matrix)

1. **Construct $\tilde{K}$:** First, we explicitly calculate the full 4×4 Nyström approximation $\tilde{K} = CW^{+}C^{T}$.

$$\tilde{K} = \begin{pmatrix} 9 & 6 & 3 & 0.675 \\ 6 & 4 & 2 & 0.45 \\ 3 & 2 & 1 & 0.225 \\ 0.675 & 0.45 & 0.225 & 0.05 \end{pmatrix}$$

2. **Add the regularization:** Now we compute $(\lambda I + \tilde{K})$.

$$(\lambda I + \tilde{K}) = \begin{pmatrix} 9.1 & 6 & 3 & 0.675 \\ 6 & 4.1 & 2 & 0.45 \\ 3 & 2 & 1.1 & 0.225 \\ 0.675 & 0.45 & 0.225 & 0.15 \end{pmatrix}$$

3. **Invert the 4×4 matrix:** This is the expensive step. The result is:

$$(\lambda I + \tilde{K})^{-1} \approx \begin{pmatrix} 9.85 & -14.78 & -0.07 & 0.27 \\ -14.78 & 22.22 & 0.09 & -0.41 \\ -0.07 & 0.09 & 0.91 & -0.03 \\ 0.27 & -0.41 & -0.03 & 6.67 \end{pmatrix}$$

This works for our tiny 4×4 example, but it would be computationally infeasible if n was 10,000.

10.6.8.2. Method 2: The Fast Way (Using Woodbury Identity)

We use the Woodbury formula to get the same result without ever creating a 4×4 matrix. The formula simplifies to:

$$(\lambda I + \tilde{K})^{-1} = \frac{1}{\lambda} I - \frac{1}{\lambda^2} C \left(W + \frac{1}{\lambda} C^T C \right)^{-1} C^T$$

1. **Compute the small 2×2 pieces:**

   - $C^T C = \begin{pmatrix} 127 & 42.25 \\ 42.25 & 14.0625 \end{pmatrix}$
   - $W = \begin{pmatrix} 9 & 3 \\ 3 & 1 \end{pmatrix}$

- The matrix to invert is $W + \frac{1}{0.1}C^TC = W + 10 \cdot (C^TC)$, which is:

$$\begin{pmatrix} 9 & 3 \\ 3 & 1 \end{pmatrix} + \begin{pmatrix} 1270 & 422.5 \\ 422.5 & 140.625 \end{pmatrix} = \begin{pmatrix} 1279 & 425.5 \\ 425.5 & 141.625 \end{pmatrix}$$

2. **Invert the small 2×2 matrix:** This is the only inversion we need, and it's extremely fast.

$$(W + \frac{1}{\lambda}C^TC)^{-1} \approx \begin{pmatrix} 0.22 & -0.66 \\ -0.66 & 1.99 \end{pmatrix}$$

3. **Combine the results:** Now we plug this small inverse back into the full formula. The rest is just matrix multiplication, no more inversions.
   - First, calculate the middle term: $M = \frac{1}{\lambda^2}C(\dots)^{-1}C^T$. This will result in a 4×4 matrix.
   - Then, calculate the final result: $\frac{1}{\lambda}I - M$.

After performing these multiplications, you will get the **exact same 4×4 inverse matrix** as in the slow method.

The crucial difference is that the most expensive operation—the matrix inversion—was performed on a tiny 2×2 matrix instead of a 4×4 one. For a large-scale problem, this is the difference between a calculation that takes seconds and one that could take hours or even be impossible.

10.7. Extending spotpython's Kriging Surrogate with Nyström Approximation for Enhanced Scalability

10.7.1. Introduction: Overcoming the Scalability Challenge in Kriging for Sequential Optimization

The Sequential Parameter Optimization Toolbox (spotpython) is a framework for hyperparameter tuning and black-box optimization based on Sequential Model-Based Optimization (SMBO). At the core of SMBO lies a surrogate model that approximates the true, expensive objective. Kriging (Gaussian Process regression) is a premier choice because it provides both predictions and a principled measure of uncertainty. This uncertainty enables a balance between exploration and exploitation. In each SMBO iteration, the Kriging model is updated with new evaluations, refining its approximation and proposing the next points.

Standard Kriging requires constructing and inverting an $n \times n$ covariance matrix, where n is the number of data points. Matrix inversion scales as $O(n^3)$. During SMBO, n can reach hundreds or thousands; refitting the surrogate each iteration becomes

prohibitively expensive. This cubic scaling is the key obstacle to applying Kriging at larger scales.

We integrate the Nyström method into the spotpython Kriging class. The Nyström method yields a low-rank approximation of a symmetric positive semidefinite (SPSD) kernel matrix by selecting $l \ll n$ "landmark" points. It approximates the full $n \times n$ covariance while requiring inversion of only an $l \times l$ matrix, reducing fitting cost from $O(n^3)$ to $O(n\,l^2)$. This makes Kriging viable even when the number of function evaluations is large.

10.7.2. Report Objectives and Structure

- Review theoretical foundations of Kriging and Nyström approximation
- Present documented Python code updates for Kriging (as in kriging.py)
- Explain changes to `__init__`, `fit`, and `predict`
- Show how mixed variable types are preserved via `build_Psi` and `build_psi_vec`
- Provide practical usage guidance and a formal complexity analysis

---

10.8. Theoretical Foundations: The Nyström–Kriging Framework

10.8.1. A Primer on Kriging (Gaussian Process Regression)

Kriging models $f(x)$ as a Gaussian Process with mean function $m(\cdot)$ and covariance (kernel) $k(\cdot,\cdot)$. For training inputs $X = \{x_1, \dots, x_n\}$ and observations $y = \{y_1, \dots, y_n\}$:

$$y \sim \mathcal{N}(m(X),\, K(X,X) + \sigma_n^2 I)$$

For a new point x_*:

$$\mu(x_*) = k(x_*, X)\,[K(X,X) + \sigma_n^2 I]^{-1} y$$

$$\sigma^2(x_*) = k(x_*, x_*) - k(x_*, X)\,[K(X,X) + \sigma_n^2 I]^{-1} k(X, x_*)$$

The challenge is inverting the $n \times n$ matrix $K(X,X) + \sigma_n^2 I$.

10.8.2. The Nyström Method for Low-Rank Kernel Approximation

Select l landmark points $X_m \subset X$. Let: - $C = K_{nm} = K(X, X_m) \in \mathbb{R}^{n \times l}$ - $W = K_{mm} = K(X_m, X_m) \in \mathbb{R}^{l \times l}$ Then the Nyström approximation is:

$$\tilde{K}_{nn} = C W^{+} C^{\top} = K_{nm} K_{mm}^{+} K_{mn}$$

where W^{+} is the pseudoinverse of W. The approximation has rank $\leq l$.

10.8.3. Justification for Landmark Selection

Uniform sampling without replacement is an effective and inexpensive strategy for selecting landmarks across varied datasets and kernels.

10.9. Implementation: A Scalable Kriging Class for spotpython

10.9.1. Updated kriging.py with Nyström Approximation (excerpt)

```
"""
Kriging surrogate with optional Nyström approximation.
"""
import numpy as np
from scipy.spatial.distance import cdist
from scipy.linalg import cholesky, cho_solve, solve_triangular

class Kriging:
    def __init__(self, fun_control, n_theta=None, theta=None, p=2.0,
                 corr="squared_exponential", isotropic=False,
                 approximation="None", n_landmarks=100):
        self.fun_control = fun_control
        self.dim = self.fun_control["lower"].shape
        self.p = p
        self.corr = corr
        self.isotropic = isotropic
        self.approximation = approximation
        self.n_landmarks = n_landmarks
        self.factor_mask = self.fun_control["var_type"] == "factor"
```

```
        self.ordered_mask = ~self.factor_mask
        self.n_theta = 1 if isotropic else (n_theta or self.dim)
        self.theta = np.full(self.n_theta, 0.1) if theta is None else theta
        self.X_, self.y_, self.L_, self.alpha_ = None, None, None, None
        self.landmarks_, self.W_cho_, self.nystrom_alpha_ = None, None, None

    def fit(self, X, y):
        self.X_, self.y_ = X, y
        n_samples = X.shape[0]
        if self.approximation.lower() == "nystroem" and n_samples > self.n_landmarks:
            return self._fit_nystrom(X, y)
        return self._fit_standard(X, y)

    def _fit_standard(self, X, y):
        Psi = self.build_Psi(X, X)
        Psi[np.diag_indices_from(Psi)] += 1e-8
        try:
            self.L_ = cholesky(Psi, lower=True)
            self.alpha_ = cho_solve((self.L_, True), y)
        except np.linalg.LinAlgError:
            self.L_ = None
            self.alpha_ = np.linalg.pinv(Psi) @ y

    def _fit_nystrom(self, X, y):
        n_samples = X.shape[0]
        idx = np.random.choice(n_samples, self.n_landmarks, replace=False)
        self.landmarks_ = X[idx, :]
        W = self.build_Psi(self.landmarks_, self.landmarks_) + 1e-8 * np.eye(self.n_landmarks)
        C = self.build_Psi(X, self.landmarks_)
        try:
            self.W_cho_ = cholesky(W, lower=True)
            self.nystrom_alpha_ = cho_solve((self.W_cho_, True), C.T @ y)
        except np.linalg.LinAlgError:
            self.W_cho_ = None
            self._fit_standard(X, y)

    def predict(self, X_star):
        if self.approximation.lower() == "nystroem" and self.landmarks_ is not None:
            return self._predict_nystrom(X_star)
        return self._predict_standard(X_star)

    def _predict_standard(self, X_star):
        psi = self.build_Psi(X_star, self.X_)
        y_pred = psi @ self.alpha_
```

```
        if self.L_ is not None:
            v = solve_triangular(self.L_, psi.T, lower=True)
            y_mse = 1.0 - np.sum(v**2, axis=0)
        else:
            Psi = self.build_Psi(self.X_, self.X_) + 1e-8 * np.eye(self.X_.shape[0])
            pi_Psi = np.linalg.pinv(Psi)
            y_mse = 1.0 - np.sum((psi @ pi_Psi) * psi, axis=1)
        y_mse[y_mse < 0] = 0
        return y_pred, y_mse.reshape(-1, 1)

    def _predict_nystrom(self, X_star):
        psi_star_m = self.build_Psi(X_star, self.landmarks_)
        y_pred = psi_star_m @ self.nystrom_alpha_
        if self.W_cho_ is not None:
            v = cho_solve((self.W_cho_, True), psi_star_m.T)
            quad = np.sum(psi_star_m * v.T, axis=1)
            y_mse = 1.0 - quad
        else:
            y_mse = np.ones(X_star.shape[0])
        y_mse[y_mse < 0] = 0
        return y_pred, y_mse.reshape(-1, 1)

    def build_Psi(self, X1, X2):
        n1 = X1.shape[0]
        Psi = np.zeros((n1, X2.shape[0]))
        for i in range(n1):
            Psi[i, :] = self.build_psi_vec(X1[i, :], X2)
        return Psi

    def build_psi_vec(self, x, X_):
        theta10 = np.full(self.dim, 10**self.theta) if self.isotropic else 10**self.th
        D = np.zeros(X_.shape[0])
        if self.ordered_mask.any():
            Xo = X_[:, self.ordered_mask]
            xo = x[self.ordered_mask]
            D += cdist(xo.reshape(1, -1), Xo, metric="sqeuclidean",
                       w=theta10[self.ordered_mask]).ravel()
        if self.factor_mask.any():
            Xf = X_[:, self.factor_mask]
            xf = x[self.factor_mask]
            D += cdist(xf.reshape(1, -1), Xf, metric="hamming",
                       w=theta10[self.factor_mask]).ravel() * self.factor_mask.sum()
        return np.exp(-D) if self.corr == "squared_exponential" else np.exp(-(D**self
```

10.10. Implementation Details

10.10.1. Architectural Enhancements to `init`

- New argument `approximation="None"` for backward-compatible selection between exact Kriging and Nyström
- New argument `n_landmarks` (default 100) controls the number of inducing points when using Nyström
- State attributes for both exact and Nyström paths are maintained separately

10.10.2. The `fit()` Method: A Dual-Pathway Approach

- Dispatcher selecting exact or Nyström pathway
- The Nyström fit Pathway (`_fit_nystrom`):
  - Landmark selection via uniform sampling without replacement
  - Core matrices:
    - $W = K_{mm}$ (landmark-landmark)
    - $C = K_{nm}$ (data-landmark)
  - Cholesky factorization of W (with jitter) for stability
  - Pre-computation: $\alpha_{nys} = W^{-1}C^T y$ via `cho_solve`
- The Standard fit Pathway (`_fit_standard`):
  - Full Ψ construction, Cholesky decomposition, and solve for α
  - Fallback to pseudoinverse if Cholesky fails

10.10.3. The `predict()` Method: Conditional Prediction Logic

- Routes to Nyström or standard prediction path based on fitted model state
- The Nyström predict Pathway (`_predict_nystrom`):
  - Cross-covariance ψ between test points and landmarks
  - Mean: $\psi \cdot \alpha_{nys}$
  - Variance: uses `cho_solve` with W Cholesky; non-negative clipping
- The Standard predict Pathway (`_predict_standard`):
  - Cross-covariance with all training points
  - Mean from α; variance via triangular solves or pseudoinverse fallback

10.10.4. Critical Detail: Preserving Mixed Variable Type Functionality

The Significance of `build_psi_vec`:

- Mixed spaces: continuous (ordered) and categorical (factor) variables
- Distances:
  - Weighted squared Euclidean for ordered variables
  - Weighted Hamming for factors
- Anisotropic kernel via per-dimension length-scales θ
- Nyström path reuses `build_Psi` → `build_psi_vec`, preserving mixed-type handling

10.10.5. Seamless Integration into the Nyström Workflow

All covariance computations (W, C, predictive cross-covariance) use `build_Psi`, ensuring identical handling for mixed variable types in both standard and Nyström modes.

10.11. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

11. Infill Criteria

In the context of computer experiments and surrogate modeling, a sampling plan refers to the set of input values, often denoted as X, at which a computer code is evaluated. The primary objective of a sampling plan is to efficiently explore the input space to understand the behavior of the computer code and to construct a surrogate model that accurately represents that behavior. Historically, Response Surface Methodology (RSM) provided methods for designing such plans, often based on rectangular grids or factorial designs. More recently, Design and Analysis of Computer Experiments (DACE) has emerged as a more flexible and powerful approach for this purpose.

A surrogate model, or $\hat{f}$, is built to approximate the expensive response of a black-box function $f(x)$. Since evaluating f is costly, only a sparse set of samples is used to construct $\hat{f}$, which can then provide inexpensive predictions for any point in the design space. However, as a surrogate model is inherently an approximation of the true function, its accuracy and predictive capabilities can be significantly improved by incorporating new data points, known as infill points. Infill points are strategically chosen to either reduce uncertainty, improve predictions in specific regions of interest, or enhance the model's ability to identify optima or trends.

The process of updating a surrogate model with infill points is iterative. It typically involves:

- **Identifying Regions of Interest**: Analyzing the current surrogate model to determine areas where it is inaccurate, has high uncertainty, or predicts promising results (e.g., potential optima).
- **Selecting Infill Points**: Choosing new data points based on specific criteria that balance different objectives.
- **Evaluating the True Function**: Running the actual simulation or experiment at the selected infill points to obtain their corresponding outputs.
- **Updating the Surrogate Model**: Retraining or updating the surrogate model using the new, augmented dataset.
- **Repeating**: Iterating this process until the model meets predefined accuracy criteria or the computational budget is exhausted.

11.1. Balancing Exploitation and Exploration

A crucial aspect of selecting infill points is navigating the inherent trade-off between exploitation and exploration.

Definition 11.1 (Exploitation)**.** Exploitation refers to sampling near predicted optima to refine the solution. This strategy aims to rapidly converge on a good solution by focusing computational effort where the surrogate model suggests the best values might lie.

Definition 11.2 (Exploration)**.** Exploration involves sampling in regions of high uncertainty to improve the global accuracy of the model. This approach ensures that the model is well-informed across the entire design space, preventing it from getting stuck in local optima.

Forrester, Sóbester, and Keane (2008) emphasizes that effective infill criteria are designed to combine both exploitation and exploration.

11.2. Expected Improvement (EI)

Expected Improvement (EI) is one of the most influential and widely-used infill criteria. Formalized by Jones, Schonlau, and Welch (1998) and building upon the work of Močkus (1974), EI provides a mathematically elegant framework that naturally balances exploitation and exploration. Rather than simply picking the point with the best predicted value (pure exploitation) or the point with the highest uncertainty (pure exploration), EI asks a more nuanced question: "How much improvement over the current best solution can we *expect* to gain by evaluating the true function at a new point x?".

The Expected Improvement, $EI(x)$, can be calculated using the following formula:

$$EI(x) = \sigma(x)\left[Z\Phi(Z) + \phi(Z)\right]$$

where:

- $\mu(x)$ (or $\hat{y}(x)$) is the Kriging prediction (mean of the stochastic process) at a new, unobserved point x.
- $\sigma(x)$ (or $\hat{s}(x)$) is the estimated standard deviation (square root of the variance $\hat{s}^2(x)$) of the prediction at point x.
- f_{best} (or y_{min}) is the best (minimum, for minimization problems) observed function value found so far.
- $Z = \frac{f_{best} - \mu(x)}{\sigma(x)}$ is the standardized improvement.

- $\Phi(Z)$ is the cumulative distribution function (CDF) of the standard normal distribution.
- $\phi(Z)$ is the probability density function (PDF) of the standard normal distribution.

If $\sigma(x) = 0$ (meaning there is no uncertainty at point x, typically because it's an already sampled point), then $EI(x) = 0$, reflecting the intuition that no further improvement can be expected at a known point. A maximization of Expected Improvement as an infill criterion will eventually lead to the global optimum.

The elegance of the EI formula lies in its combination of two distinct terms:

- **Exploitation Term**: $(f_{best} - \mu(x))\Phi(Z)$. This part of the formula contributes more when the predicted value $\mu(x)$ is significantly lower (better) than the current best observed value f_{best}. It is weighted by the probability $\Phi(Z)$ that the true function value at x will indeed be an improvement over f_{best}.
- **Exploration Term**: $\sigma(x)\phi(Z)$. This term becomes larger when there is high uncertainty ($\sigma(x)$ is large) in the model's prediction at x. It accounts for the potential of discovering unexpectedly good values in areas that have not been thoroughly explored, even if the current mean prediction there is not the absolute best.

Expected Improvement offers several significant practical benefits:

- **Automatic Balance**: It inherently balances exploitation and exploration without requiring any manual adjustment of weights or parameters.
- **Scale Invariance**: EI is relatively insensitive to the scaling of the objective function, making it robust across various problem types.
- **Theoretical Foundation**: It is underpinned by a strong theoretical basis derived from decision theory and information theory.
- **Efficient Optimization**: The smooth and differentiable nature of the EI function allows for efficient optimization using gradient-based algorithms to find the next infill point.
- **Proven Performance**: EI has demonstrated consistent and strong performance in numerous real-world applications across various domains.

11.3. Expected Improvement in the Hyperparameter Tuning Cookbook (Python Implementation)

Within the context of the Hyperparameter Tuning Cookbook, Expected Improvement serves a critical role in Sequential Model-Based Optimization. It systematically guides the selection of which hyperparameter configurations to evaluate next, facilitating the efficient utilization of computational resources. By intelligently balancing the need to exploit promising regions and explore uncertain areas, EI helps identify optimal

hyperparameters with a reduced number of expensive model training runs. This provides a principled and automated method for navigating complex hyperparameter spaces without extensive manual intervention.

While the foundational concepts in Forrester, Sóbester, and Keane (2008) are often illustrated with MATLAB code, the Hyperparameter Tuning Cookbook emphasizes and provides implementations in Python. The `spotpython` package, consistent with the Cookbook's approach, provides a Python implementation of Expected Improvement within its Kriging class. For minimization problems, `spotpython` typically calculates and returns the negative Expected Improvement, aligning with standard optimization algorithm conventions. Furthermore, to enhance numerical stability and mitigate issues when EI values are very small, `spotpython` often works with a logarithmic transformation of EI and incorporates a small epsilon value.

11.4. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

Part III.

Sequential Parameter Optimization Toolbox (SPOT)

12. Introduction to Sequential Parameter Optimization

The following libraries are used in this document:

```
import numpy as np
from math import inf
from scipy.optimize import shgo
from scipy.optimize import direct
from scipy.optimize import differential_evolution
import matplotlib.pyplot as plt
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.utils.init import (fun_control_init, design_control_init, surrogate_control_init, op
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.eda import print_exp_table, print_res_table
```

This presents an introduction to `spotpythons`'s `Spot` class. The official `spotpython` documentation can be found here: https://sequential-parameter-optimization.github.io/spotpython/.

12.1. Surrogate Model Based Optimization

Surrogate model based optimization methods are common approaches in simulation and optimization. The sequential parameter optimization toolbox (SPOT) was developed because there is a great need for sound statistical analysis of simulation and optimization algorithms (Bartz-Beielstein, Lasarczyk, and Preuss 2005). SPOT includes methods for tuning based on classical regression and analysis of variance techniques. It presents tree-based models such as classification and regression trees and random forests as well as Bayesian optimization (Gaussian process models, also known as Kriging). Combinations of different meta-modeling approaches are possible. SPOT comes with a sophisticated surrogate model based optimization method, that can handle discrete and continuous inputs. Furthermore, any model implemented in `scikit-learn` can be used out-of-the-box as a surrogate in `spotpython`.

SPOT implements key techniques such as exploratory fitness landscape analysis and sensitivity analysis. It can be used to understand the performance of various algorithms, while simultaneously giving insights into their algorithmic behavior.

The `spot` loop consists of the following steps:

1. Init: Build initial design X using the `design` method, e.g., Latin Hypercube Sampling (LHS).
2. Evaluate initial design on real objective f: $y = f(X)$ using the `fun` method, e.g., `fun_sphere`.
3. Build surrogate: $S = S(X, y)$, using the `fit` method of the surrogate model, e.g., `Kriging`.
4. Optimize on surrogate: $X_0 = \text{optimize}(S)$, using the `optimizer` method, e.g., `differential_evolution`.
5. Evaluate on real objective: $y_0 = f(X_0)$, using the `fun` method from above.
6. Impute (Infill) new points: $X = X \cup X_0$, $y = y \cup y_0$, using `Spot`'s `infill` method.
7. Goto 3.

12.2. Advantages of the `spotpython` Approach

- Neural networks and many ML algorithms are non-deterministic, so results are noisy (i.e., depend on the the initialization of the weights). Enhanced noise handling strategies, OCBA (description from HPT-book).
- Optimal Computational Budget Allocation (OCBA) is a very efficient solution to solve the "general ranking and selection problem" if the objective function is noisy. It allocates function evaluations in an uneven manner to identify the best solutions and to reduce the total optimization costs. [Chen10a, Bart11b] Given a total number of optimization samples N to be allocated to k competing solutions whose performance is depicted by random variables with means $\bar{y}_i$ $(i = 1, 2, \ldots, k)$, and finite variances σ_i^2, respectively, as $N \to \infty$, the Approximate Probability of Correct Selection (APCS) can be asymptotically maximized when

$$\frac{N_i}{N_j} = \left(\frac{\sigma_i/\delta_{b,i}}{\sigma_j/\delta_{b,j}}\right)^2, i, j \in \{1, 2, \ldots, k\}, \text{ and } i \neq j \neq b, \tag{12.1}$$

$$N_b = \sigma_b \sqrt{\sum_{i=1, i\neq b}^{k} \frac{N_i^2}{\sigma_i^2}}, \tag{12.2}$$

  where N_i is the number of replications allocated to solution i, $\delta_{b,i} = \bar{y}_b - \bar{y}_i$, and $\bar{y}_b \leq \min_{i\neq b} \bar{y}_i$ Bartz-Beielstein and Friese (2011).
- Surrogate-based optimization: Better than grid search and random search (Reference to HPT-book)
- Visualization

- Importance based on the Kriging model
- Sensitivity analysis. Exploratory fitness landscape analysis. Provides XAI methods (feature importance, integrated gradients, etc.)
- Uncertainty quantification
- Flexible, modular meta-modeling handling. `spotpython` comes with a Kriging model, which can be replaced by any model implemented in `scikit-learn`.
- Enhanced metric handling, especially for categorical hyperparameters (any `sklearn` metric can be used). Default is..
- Integration with `TensorBoard`: Visualization of the hyperparameter tuning process, of the training steps, the model graph. Parallel coordinates plot, scatter plot matrix, and more.
- Reproducibility. Results are stored as pickle files. The results can be loaded and visualized at any time and be transferred between different machines and operating systems.
- Handles `scikit-learn` models and `PyTorch` models out-of-the-box. The user has to add a simple wrapper for passing the hyperparameters to use a `PyTorch` model in `spotpython`.
- Compatible with `Lightning`.
- User can add own models as plain python code.
- User can add own data sets in various formats.
- Flexible data handling and data preprocessing.
- Many examples online (hyperparameter-tuning-cookbook).
- `spotpython` uses a robust optimizer that can even deal with hyperparameter-settings that cause crashes of the algorithms to be tuned.
- even if the optimum is not found, HPT with `spotpython` prevents the user from choosing bad hyperparameters in a systematic way (design of experiments).

12.3. Disadvantages of the spotpython Approach

- Time consuming
- Surrogate can be misguiding

12.4. An Initial Example: Optimization of the One-dimensional Sphere Function

12.4.1. The Objective Function: Sphere

The `spotpython` package provides several classes of objective functions. We will use the `sphere`-function, which is an analytical objective function. Analytical functions can be described by a (closed) formula:

$$f(x) = x^2.$$

```
fun = Analytical().fun_sphere
x = np.linspace(-1, 1, 100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x,y, "k")
plt.grid()
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Sphere Function")
plt.show()
```

Figure 12.1.: The sphere function $f(x) = x^2$ in the range [-1, 1].

The optimal solution of the sphere function is $x^* = 0$ with $f(x^*) = 0$.

12.4.2. The `Spot` Method as an Optimization Algorithm Using a Surrogate Model

`pyspot` implements the `Spot` method. The `Spot` method uses a surrogate model to approximate the objective function and to find the optimal solution by performing an optimization on the surrogate model. In this example, `Spot` uses ten initial design points, which are sampled from a Latin Hypercube Sampling (LHS) design. The `Spot` method will then build a surrogate model based on these initial design points and the corresponding objective function values. The surrogate model is then used to find the optimal solution. As a default, `Spot` uses a Kriging surrogate model, which is a Gaussian process model. As a default, ten initial design points are sampled from a Latin Hypercube Sampling (LHS) design. Also as a default, 15 function evaluations are performed, i.e., the final surrogate model is built based on 15 points.

The specification of the `lower` and `upper` bounds of the input space is mandatory via the `fun_control` dictionary, is passed to the `Spot` method. For convenience, `spotpython` provides an initialization method `fun_control_init()` to create the `fun_control`

dictionary. After the `Spot` method is initialized, the `run()` method is called to start the optimization process. The `run()` method will perform the optimization on the surrogate model and return the optimal solution. Finally, the `print_results()` method is called to print the results of the optimization process, i.e., the best objective function value ("min y") and the corresponding input value ("x0").

```
fun_control=fun_control_init(lower = np.array([-1]),
                             upper = np.array([1]))
S = Spot(fun=fun, fun_control=fun_control)
S.run()
S.print_results()
```

```
spotpython tuning: 2.4412172266370036e-09 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 2.032431480352692e-09 [########--] 80.00%. Success rate: 100.00%
spotpython tuning: 2.032431480352692e-09 [#########-] 86.67%. Success rate: 66.67%
spotpython tuning: 2.032431480352692e-09 [#########-] 93.33%. Success rate: 50.00%
spotpython tuning: 1.2228307315407816e-10 [##########] 100.00%. Success rate: 60.00% I

Experiment saved to 000_res.pkl
min y: 1.2228307315407816e-10
x0: 1.1058167712332733e-05

[['x0', np.float64(1.1058167712332733e-05)]]
```

`spotpython` provides a method `plot_progress()` to visualize the search progress. The parameter `log_y` is used to plot the objective function values on a logarithmic scale. The black points represent the ten initial design points, whereas the remainin g five red points represent the points found by the surrogate model based optimization.

```
S.plot_progress(log_y=True)
```

```
S.surrogate_control
```

```
{'log_level': 50,
 'method': 'regression',
 'model_optimizer': <function scipy.optimize._differentialevolution.differential_evolution(func, bou
 'model_fun_evals': 10000,
 'min_theta': -3.0,
 'max_theta': 2.0,
 'isotropic': False,
 'p_val': 2.0,
 'n_p': 1,
 'optim_p': False,
 'min_Lambda': -9,
 'max_Lambda': 0,
 'seed': 124,
 'theta_init_zero': False,
 'var_type': ['num'],
 'metric_factorial': 'canberra',
 'use_nystrom': False,
 'nystrom_m': 20,
 'nystrom_seed': 1234,
 'max_surrogate_points': 30}
```

12.5. A Second Example: Optimization of a User-Specified Multi-dimensional Function

Users can easily specify their own objective functions. The following example shows how to use the `Spot` method to optimize a user-specified two-dimensional function. Here we will use the 2-dimensional Branin function, which is a well-known test function in optimization. The Branin function is defined as follows:

$$f(x) = a(x2 - bx1^2 + cx1 - r)^2 + s(1 - t)\cos(x1) + s$$

where:

- $a = 1$,
- $b = 5.1/(4\pi^2)$,
- $c = 5/\pi$,
- $r = 6$,
- $s = 10$, and
- $t = 1/(8\pi)$.

The input space is defined as $x_1 \in [-5, 10]$ and $x_2 \in [0, 15]$. The user specified Branin function can be implemented as follows (note, that the function is vectorized, i.e., it can handle multiple input points at once and that the `**kwargs` argument is mandatory for compatibility with `spotpython` to pass additional parameters to the function, if needed):

```
def user_fun(X, **kwargs):
    x1 = X[:, 0]
    x2 = X[:, 1]
    a = 1
    b = 5.1 / (4 * np.pi**2)
    c = 5 / np.pi
    r = 6
    s = 10
    t = 1 / (8 * np.pi)
    y = a * (x2 - b * x1**2 + c * x1 - r)**2 + s * (1 - t) * np.cos(x1) + s
    return y
```

The Branin function has its global minimum at three points: $f(x^*) = 0.397887$, at $x^* = (-\pi, 12.275)$, $(\pi, 2.275)$ and $(9.42478, 2.475)$. It can be visualized as shown in Figure 12.2.

```
x1 = np.linspace(-5, 10, 100)
x2 = np.linspace(0, 15, 100)
X1, X2 = np.meshgrid(x1, x2)
Z = user_fun(np.array([X1.ravel(), X2.ravel()]).T)
Z = Z.reshape(X1.shape)
plt.figure(figsize=(8, 6))
plt.contourf(X1, X2, Z, levels=30, cmap='jet')
plt.colorbar(label='f(x)')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Branin Function')
plt.scatter([-np.pi, np.pi, 9.42478], [12.275, 2.275, 2.475], color='white', label='Gl
plt.legend()
plt.show()
```

Figure 12.2.: The Branin function $f(x)$ in the range $x1 \in [-5, 10]$ and $x2 \in [0, 15]$.

The default `Spot` class assumes a noisy objective function, because many real-world applications have noisy objective functions. Here, we will set the `noise` parameter to `False` in the `fun_control` dictionary, because the Branin function is a deterministic function. Accordingly, the `interpolation` surrogate method is used, which is suitable for deterministic functions. Therefore, we need to specify the `interpolation` method in the `surrogate_control` dictionary. Furthermore, since the Branin is harder to optimize than the sphere function, we will increase the number of function evaluations to 25 and the number of initial design points to 15.

```
fun_control = fun_control_init(
              lower = np.array( [-5, 0]),
              upper = np.array([10, 15]),
              fun_evals = 25,
              noise = False,
              seed=123
)
```

```
design_control=design_control_init(init_size=15)
surrogate_control=surrogate_control_init(method="interpolation")

S_2 = Spot(fun=user_fun,
                 fun_control=fun_control,
                 design_control=design_control,
                 surrogate_control=surrogate_control)
S_2.run()
S_2.print_results()
S_2.plot_progress(log_y=True)
```

```
spotpython tuning: 3.1165078710223124 [######----] 64.00%. Success rate: 0.00%
spotpython tuning: 2.323465946406328 [#######---] 68.00%. Success rate: 50.00%
spotpython tuning: 0.7636114164021386 [#######---] 72.00%. Success rate: 66.67%
spotpython tuning: 0.7636114164021386 [########--] 76.00%. Success rate: 50.00%
spotpython tuning: 0.7134874467492036 [########--] 80.00%. Success rate: 60.00%
spotpython tuning: 0.5278720689655838 [########--] 84.00%. Success rate: 66.67%
spotpython tuning: 0.4032724486704513 [#########-] 88.00%. Success rate: 71.43%
spotpython tuning: 0.39848208189117607 [#########-] 92.00%. Success rate: 75.00%
spotpython tuning: 0.3984371737124537 [##########] 96.00%. Success rate: 77.78%
spotpython tuning: 0.3984371737124537 [##########] 100.00%. Success rate: 70.00% Done

Experiment saved to 000_res.pkl
min y: 0.3984371737124537
x0: -3.144600358136924
x1: 12.30473238732154
```

The resulting surrogate used by `Spot` can be visualized as shown in Figure 12.3.

```
S_2.plot_contour()
```

Figure 12.3.: Visualization of the surrogate model of the Branin function.

```
S_2.surrogate_control
```

```
{'log_level': 50,
 'method': 'interpolation',
 'model_optimizer': <function scipy.optimize._differentialevolution.differential_evolution(func, bou
 'model_fun_evals': 10000,
 'min_theta': -3.0,
 'max_theta': 2.0,
 'isotropic': False,
 'p_val': 2.0,
```

```
'n_p': 1,
'optim_p': False,
'min_Lambda': -9,
'max_Lambda': 0,
'seed': 124,
'theta_init_zero': False,
'var_type': ['num', 'num'],
'metric_factorial': 'canberra',
'use_nystrom': False,
'nystrom_m': 20,
'nystrom_seed': 1234,
'max_surrogate_points': 30}
```

12.6. A Third Example: Hyperparameter Tuning of a Neural Network

`spotpython` provides several `PyTorch` neural networks, which can be used out-of-the-box for hyperparameter tuning. We will use the `Diabetes` dataset, which is a regression dataset with 10 input features and one output feature.

Similar to the steps from above, we define the `fun_control` dictionary. The hyperparameter tuning of neural networks requires a few additional parameters in the `fun_control` dictionary, e.g., for selecting the neural network model, the hyperparameter dictionary, and the number of input and output features of the neural network. The `fun_control` dictionary is initialized using the `fun_control_init()` method. The following parameters are used:

- The `core_model_name` parameter specifies the neural network model to be used. In this case, we will use the `NNLinearRegressor`, which is a simple linear regression model implemented in `spotpython`.
- The `hyperdict` parameter specifies the hyperparameter dictionary to be used. In this case, we will use the `LightHyperDict`, which is a hyperparameter dictionary for light-weight models.
- `_L_in` and `_L_out` specify the number of input and output features of the neural network, respectively. In this case, we will use 10 input features and 1 output feature, which are required for the `Diabetes` dataset.

Sometimes, very bad configurations appear in the initial design, leading to an unnecessarily long optimization process. An example is illustrated in Figure 12.4. To avoid this, we can set a divergence threshold in the `fun_control` dictionary. The `divergence_threshold` parameter is used to stop the optimization process if the objective function value exceeds a certain threshold. This is useful to avoid long optimization processes if the objective function is not well-behaved. We have set the `divergence_threshold=5_000`.

Figure 12.4.: TensorBoard visualization of the spotpython process.

```
fun_control = fun_control_init(
    PREFIX="S_3",
    max_time=1,
    data_set = Diabetes(),
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    divergence_threshold=5_000,
    _L_in=10,
    _L_out=1)

S_3 = Spot(fun=HyperLight().fun,
         fun_control=fun_control)
S_3.surrogate_control
```

```
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
```

```
{'log_level': 50,
 'method': 'regression',
 'model_optimizer': <function scipy.optimize._differentialevolution.differential_evolution(func, bou
 'model_fun_evals': 10000,
 'min_theta': -3.0,
 'max_theta': 2.0,
 'isotropic': False,
 'p_val': 2.0,
 'n_p': 1,
```

```
 'optim_p': False,
 'min_Lambda': -9,
 'max_Lambda': 0,
 'seed': 124,
 'theta_init_zero': False,
 'var_type': ['int',
  'int',
  'int',
  'factor',
  'factor',
  'float',
  'float',
  'int',
  'factor',
  'factor'],
 'metric_factorial': 'canberra',
 'use_nystrom': False,
 'nystrom_m': 20,
 'nystrom_seed': 1234,
 'max_surrogate_points': 30}
```

```
S_3.run()
S_3.print_results()
S_3.plot_progress(log_y=True)
```

```
     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   12.7 M   train   407 M   [16, 10]     [16, 1]


Trainable params: 12.7 M
Non-trainable params: 0
Total params: 12.7 M
Total estimated model params size (MB): 50
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 407 M

train_model result: {'val_loss': 13860.1416015625, 'hp_metric': 13860.1416015625}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes
```

```
  0   layers   Sequential   51.9 K   train   409 K    [4, 10]      [4, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 409 K

train_model result: {'val_loss': 5115.263671875, 'hp_metric': 5115.263671875}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0   layers   Sequential   50.9 M   train   813 M    [8, 10]      [8, 1]


Trainable params: 50.9 M
Non-trainable params: 0
Total params: 50.9 M
Total estimated model params size (MB): 203
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 813 M

train_model result: {'val_loss': 5342.6416015625, 'hp_metric': 5342.6416015625}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0   layers   Sequential    807 K   train   12.8 M    [8, 10]      [8, 1]


Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.8 M
```

```
train_model result: {'val_loss': 14175.4755859375, 'hp_metric': 14175.4755859375}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    3.2 M   train   25.5 M    [4, 10]      [4, 1]


Trainable params: 3.2 M
Non-trainable params: 0
Total params: 3.2 M
Total estimated model params size (MB): 12
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.5 M


train_model result: {'val_loss': nan, 'hp_metric': nan}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential   12.7 M   train   50.9 M    [2, 10]      [2, 1]


Trainable params: 12.7 M
Non-trainable params: 0
Total params: 12.7 M
Total estimated model params size (MB): 50
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 50.9 M


train_model result: {'val_loss': 5.328983764888453e+18, 'hp_metric': 5.32898376488845


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   3.2 M    [8, 10]      [8, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.2 M

train_model result: {'val_loss': nan, 'hp_metric': nan}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    807 K   train   3.2 M    [2, 10]      [2, 1]


Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.2 M

train_model result: {'val_loss': 22902.03125, 'hp_metric': 22902.03125}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   1.6 M    [4, 10]      [4, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 1.6 M

train_model result: {'val_loss': 11929.5634765625, 'hp_metric': 11929.5634765625}
```

```
     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    3.2 M   train   102 M   [16, 10]     [16, 1]


Trainable params: 3.2 M
Non-trainable params: 0
Total params: 3.2 M
Total estimated model params size (MB): 12
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 102 M

train_model(): trainer.fit failed with exception: SparseAdam does not support dense gr
train_model result: {'val_loss': 24054.533203125, 'hp_metric': 24054.533203125}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   12.8 M   train   407 M   [16, 10]     [16, 1]


Trainable params: 12.8 M
Non-trainable params: 0
Total params: 12.8 M
Total estimated model params size (MB): 51
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 407 M

train_model result: {'val_loss': 17948.6796875, 'hp_metric': 17948.6796875}
spotpython tuning: 5115.263671875 [######----] 60.00%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   12.8 M   train   101 M    [4, 10]      [4, 1]


Trainable params: 12.8 M
Non-trainable params: 0
Total params: 12.8 M
```

```
Total estimated model params size (MB): 51
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 101 M

train_model(): trainer.fit failed with exception: SparseAdam does not support dense gradients, pleas
train_model result: {'val_loss': 104114438144.0, 'hp_metric': 104114438144.0}
spotpython tuning: 5115.263671875 [#######---] 66.67%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   12.8 M   train   407 M   [16, 10]     [16, 1]


Trainable params: 12.8 M
Non-trainable params: 0
Total params: 12.8 M
Total estimated model params size (MB): 51
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 407 M

train_model(): trainer.fit failed with exception: SparseAdam does not support dense gradients, pleas
train_model result: {'val_loss': 3014937856.0, 'hp_metric': 3014937856.0}
spotpython tuning: 5115.263671875 [#######---] 73.33%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   50.9 M   train   1.6 B   [16, 10]     [16, 1]


Trainable params: 50.9 M
Non-trainable params: 0
Total params: 50.9 M
Total estimated model params size (MB): 203
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 1.6 B

train_model result: {'val_loss': 15521.505859375, 'hp_metric': 15521.505859375}
spotpython tuning: 5115.263671875 [########--] 80.00%. Success rate: 0.00%
```

```
     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   1.6 M    [4, 10]      [4, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 1.6 M

train_model result: {'val_loss': 17394.609375, 'hp_metric': 17394.609375}
spotpython tuning: 5115.263671875 [##########-] 86.67%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   12.8 M   train   203 M    [8, 10]      [8, 1]


Trainable params: 12.8 M
Non-trainable params: 0
Total params: 12.8 M
Total estimated model params size (MB): 51
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 203 M

train_model result: {'val_loss': 5266.83984375, 'hp_metric': 5266.83984375}
spotpython tuning: 5115.263671875 [##########-] 93.33%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   12.8 M   train   407 M   [16, 10]     [16, 1]


Trainable params: 12.8 M
Non-trainable params: 0
Total params: 12.8 M
```

```
Total estimated model params size (MB): 51
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 407 M


train_model result: {'val_loss': 23922.74609375, 'hp_metric': 23922.74609375}
spotpython tuning: 5115.263671875 [##########] 100.00%. Success rate: 0.00% Done...

Experiment saved to S_3_res.pkl
min y: 5115.263671875
l1: 3.0
epochs: 7.0
batch_size: 2.0
act_fn: 2.0
optimizer: 2.0
dropout_prob: 0.184251494885258
lr_mult: 3.1418668140600845
patience: 6.0
batch_norm: 0.0
initialization: 2.0
```

Results can be printed using `Spot`'s `print_results()` method. The results include the best objective function value, the corresponding input values, and the number of function evaluations.

```
results = S_3.print_results()
```

```
min y: 5115.263671875
l1: 3.0
epochs: 7.0
batch_size: 2.0
```

```
act_fn: 2.0
optimizer: 2.0
dropout_prob: 0.184251494885258
lr_mult: 3.1418668140600845
patience: 6.0
batch_norm: 0.0
initialization: 2.0
```

```
S_3.surrogate_control
```

```
{'log_level': 50,
 'method': 'regression',
 'model_optimizer': <function scipy.optimize._differentialevolution.differential_evolu
 'model_fun_evals': 10000,
 'min_theta': -3.0,
 'max_theta': 2.0,
 'isotropic': False,
 'p_val': 2.0,
 'n_p': 1,
 'optim_p': False,
 'min_Lambda': -9,
 'max_Lambda': 0,
 'seed': 124,
 'theta_init_zero': False,
 'var_type': ['int',
  'int',
  'int',
  'factor',
  'factor',
  'float',
  'float',
  'int',
  'factor',
  'factor'],
 'metric_factorial': 'canberra',
 'use_nystrom': False,
 'nystrom_m': 20,
 'nystrom_seed': 1234,
 'max_surrogate_points': 30}
```

A formatted table of the results can be printed using the `print_res_table()` method:

```
print_res_table(S_3)
```

```
| name           | type   | default   |   lower |   upper | tuned               | transform
|----------------|--------|-----------|---------|---------|---------------------
|------------------------|--------------|---------|
| l1             | int    | 3         |     3.0 |     8.0 | 3.0                 | transform_power_2_i
| epochs         | int    | 4         |     4.0 |     9.0 | 7.0                 | transform_power_2_i
| batch_size     | int    | 4         |     1.0 |     4.0 | 2.0                 | transform_power_2_i
| act_fn         | factor | ReLU      |     0.0 |     5.0 | ReLU                | None
| optimizer      | factor | SGD       |     0.0 |    11.0 | Adam                | None
| dropout_prob   | float  | 0.01      |     0.0 |    0.25 | 0.184251494885258   | None
| lr_mult        | float  | 1.0       |     0.1 |    10.0 | 3.1418668140600845  | None
| patience       | int    | 2         |     2.0 |     6.0 | 6.0                 | transform_power_2_i
| batch_norm     | factor | 0         |     0.0 |     1.0 | 0                   | None
| initialization | factor | Default   |     0.0 |     4.0 | kaiming_normal      | None
```

The fitness landscape can be visualized using the `plot_important_hyperparameter_contour()` method:

```
S_3.plot_important_hyperparameter_contour(max_imp=3)
```

```
l1:  0.001217746922356549
epochs:  100.0
batch_size:  0.001217746922356549
act_fn:  0.001217746922356549
optimizer:  0.001217746922356549
dropout_prob:  0.001217746922356549
lr_mult:  0.001217746922356549
patience:  0.001217746922356549
batch_norm:  0.028897599107519437
initialization:  0.001217746922356549
```

Prediction Surface
Prediction 1e18
epochs
batch_norm
Prediction Error Surface
Error 1e17
epochs
batch_norm
Prediction Contour
1e18
batch_norm
epochs
Error Contour
1e17
batch_norm
epochs

Prediction Surface
Prediction 1e18
epochs
l1
Prediction Error Surface
Error 1e17
epochs
l1
Prediction Contour
1e18
epochs
l1
Error Contour
1e17
epochs
l1

12.7. Organization

`Spot` organizes the surrogate based optimization process in four steps:

1. Selection of the objective function: `fun`.
2. Selection of the initial design: `design`.
3. Selection of the optimization algorithm: `optimizer`.
4. Selection of the surrogate model: `surrogate`.

For each of these steps, the user can specify an object:

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical().fun_sphere
from spotpython.design.spacefilling import SpaceFilling
design = SpaceFilling(2)
```

```
from scipy.optimize import differential_evolution
optimizer = differential_evolution
from spotpython.surrogate.kriging import Kriging
surrogate = Kriging()
```

For each of these steps, the user can specify a dictionary of control parameters.

1. `fun_control`
2. `design_control`
3. `optimizer_control`
4. `surrogate_control`

Each of these dictionaries has an initialization method, e.g., `fun_control_init()`. The initialization methods set the default values for the control parameters.

> **!** Important:
>
> - The specification of an lower bound in `fun_control` is mandatory.

12.8. Summary: Basic Attributes and Methods of the `Spot` Object

The `Spot` class organizes the surrogate based optimization process into the

- `fun` object with dictionry `fun_control`,
- `design` object with dictionary `design_control`,
- `optimizer` object with dictionary `optimizer_control`, and
- `surrogate` object with dictionary `surrogate_control`.

Object	Description	Default
`fun`	Can be one of the following: (1) any function from `spotpython.fun.*`, e.g., fun_rosen, (2) a user-specified function (vectorized and accepting the `**kwargs` argument as, e.g., in Section 12.5), or, (3) if `PyTorch` hyperparameter tuning with `Lightning` is desired, the `HyperLight` class from `spotpython`, see HyperLight.	There is no default, a simple example is `fun = Analytical().fun_sphere`
`design`	Can be any design generating function from `spotpython.design:*`, e.g., Clustered, Factorial, Grid, Poor, Random, Sobol, or SpaceFilling.	`design = SpaceFilling(k=15)`
`optimizer`	Can be any of `dual_annealing`, `direct`, `shgo`, `basinhopping`, or `differential_evolution`, see, e.g., differential_evolution.	`optimizer=differential_evolution`
`surrogate`	Can be any `sklearn` regressor, e.g., GaussianProcessRegressor or, the Kriging class from `spotpython`.	`surrogate=Kriging()`

Dictionary	Code Reference	Example
`fun_control`	fun_control_init	The `lower` and `upper` arguments are mandatory, e.g., `fun_control=fun_control_init(lower=np 1, - 1]),upper=np.array([1, 1])).`

Dictionary	Code Reference	Example
design_control	design_control_init()	design_control=design_control_init()
optimizer_control	optimizer_control_init	optimizer_control_init()
surrogate_control	surrogate_control_init	surrogate_control=surrogate_control_init()

Based on the definition of the fun, design, optimizer, and surrogate objects, and their corresponding control parameter dictionaries, fun_control, design_control, optimizer_control, and surrogate_control, the Spot object can be build as follows:

```
fun_control=fun_control_init(lower=np.array([-1, -1]),
                             upper=np.array([1, 1]))
design_control=design_control_init()
optimizer_control=optimizer_control_init()
surrogate_control=surrogate_control_init()
spot_tuner = Spot(fun=fun,
                        fun_control=fun_control,
                        design_control=design_control,
                        optimizer_control=optimizer_control,
                        surrogate_control=surrogate_control)
```

12.9. Run

```
spot_tuner.run()
```

```
spotpython tuning: 6.611333241687761e-06 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 6.611333241687761e-06 [########--] 80.00%. Success rate: 50.00%
spotpython tuning: 6.611333241687761e-06 [#########-] 86.67%. Success rate: 33.33%
spotpython tuning: 6.611333241687761e-06 [#########-] 93.33%. Success rate: 25.00%
spotpython tuning: 3.1531067682958562e-06 [##########] 100.00%. Success rate: 40.00% Done...

Experiment saved to 000_res.pkl

<spotpython.spot.spot.Spot at 0x39761a850>
```

12.10. Print the Results

```
spot_tuner.print_results()
```

```
min y: 3.1531067682958562e-06
x0: -0.0012459503825024937
x1: -0.0012651934289418935
```

```
[['x0', np.float64(-0.0012459503825024937)],
 ['x1', np.float64(-0.0012651934289418935)]]
```

12.11. Show the Progress

```
spot_tuner.plot_progress()
```

12.12. Visualize the Surrogate

- The plot method of the `kriging` surrogate is used.
- Note: the plot uses the interval defined by the ranges of the natural variables.

```
spot_tuner.surrogate.plot()
```

12.13. Init: Build Initial Design

```
from spotpython.design.spacefilling import SpaceFilling
from spotpython.surrogate.kriging import Kriging
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init
gen = SpaceFilling(2)
rng = np.random.RandomState(1)
lower = np.array([-5,-0])
upper = np.array([10,15])
fun = Analytical().fun_branin

fun_control = fun_control_init(sigma=0)

X = gen.scipy_lhd(10, lower=lower, upper = upper)
```

```
print(X)
y = fun(X, fun_control=fun_control)
print(y)
```

```
[[ 8.97647221 13.41926847]
 [ 0.66946019  1.22344228]
 [ 5.23614115 13.78185824]
 [ 5.6149825  11.5851384 ]
 [-1.72963184  1.66516096]
 [-4.26945568  7.1325531 ]
 [ 1.26363761 10.17935555]
 [ 2.88779942  8.05508969]
 [-3.39111089  4.15213772]
 [ 7.30131231  5.22275244]]
[128.95676449  31.73474356 172.89678121 126.71295908  64.34349975
  70.16178611  48.71407916  31.77322887  76.91788181  30.69410529]
```

12.14. Replicability

Seed

```
gen = SpaceFilling(2, seed=123)
X0 = gen.scipy_lhd(3)
gen = SpaceFilling(2, seed=345)
X1 = gen.scipy_lhd(3)
X2 = gen.scipy_lhd(3)
gen = SpaceFilling(2, seed=123)
X3 = gen.scipy_lhd(3)
X0, X1, X2, X3
```

```
(array([[0.77254938, 0.31539299],
        [0.59321338, 0.93854273],
        [0.27469803, 0.3959685 ]]),
 array([[0.78373509, 0.86811887],
        [0.06692621, 0.6058029 ],
        [0.41374778, 0.00525456]]),
 array([[0.121357  , 0.69043832],
        [0.41906219, 0.32838498],
        [0.86742658, 0.52910374]]),
 array([[0.77254938, 0.31539299],
        [0.59321338, 0.93854273],
        [0.27469803, 0.3959685 ]]))
```

12.15. Surrogates

12.15.1. A Simple Predictor

The code below shows how to use a simple model for prediction. Assume that only two (very costly) measurements are available:

1. f(0) = 0.5
2. f(2) = 2.5

We are interested in the value at $x_0 = 1$, i.e., $f(x_0 = 1)$, but cannot run an additional, third experiment.

```
from sklearn import linear_model
X = np.array([[0], [2]])
y = np.array([0.5, 2.5])
S_lm = linear_model.LinearRegression()
S_lm = S_lm.fit(X, y)
X0 = np.array([[1]])
y0 = S_lm.predict(X0)
print(y0)
```

```
[1.5]
```

Central Idea: Evaluation of the surrogate model `S_lm` is much cheaper (or / and much faster) than running the real-world experiment f.

12.16. Tensorboard Setup

12.16.1. Tensorboard Configuration

The `TENSORBOARD_CLEAN` argument can be set to `True` in the `fun_control` dictionary to archive the TensorBoard folder if it already exists. This is useful if you want to start a hyperparameter tuning process from scratch. If you want to continue a hyperparameter tuning process, set `TENSORBOARD_CLEAN` to `False`. Then the TensorBoard folder will not be archived and the old and new TensorBoard files will shown in the TensorBoard dashboard.

12.16.2. Starting TensorBoard

`TensorBoard` can be started as a background process with the following command, where `./runs` is the default directory for the TensorBoard log files:

```
tensorboard --logdir="./runs"
```

> **TENSORBOARD_PATH**
>
> The TensorBoard path can be printed with the following command (after a `fun_control` object has been created):
>
> ```
> from spotpython.utils.init import get_tensorboard_path
> get_tensorboard_path(fun_control)
> ```

12.17. Demo/Test: Objective Function Fails

SPOT expects `np.nan` values from failed objective function values. These are handled. Note: SPOT's counter considers only successful executions of the objective function.

```
import numpy as np
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
import numpy as np
from math import inf
# number of initial points:
ni = 20
# number of points
n = 30

fun = Analytical().fun_random_error
fun_control=fun_control_init(
    lower = np.array([-1]),
    upper= np.array([1]),
    fun_evals = n,
    show_progress=False)
design_control=design_control_init(init_size=ni)

spot_1 = Spot(fun=fun,
                     fun_control=fun_control,
                     design_control=design_control)
```

```
# assert value error from the run method
try:
    spot_1.run()
except ValueError as e:
    print(e)
```

```
Using spacefilling design as fallback.
Using spacefilling design as fallback.
Using spacefilling design as fallback.
Using spacefilling design as fallback.
Using spacefilling design as fallback.
Using spacefilling design as fallback.
Using spacefilling design as fallback.
Using spacefilling design as fallback.
Using spacefilling design as fallback.
Using spacefilling design as fallback.
Using spacefilling design as fallback.
Experiment saved to 000_res.pkl
```

12.18. Handling Results: Printing, Saving, and Loading

The results can be printed with the following command:

```
spot_tuner.print_results(print_screen=False)
```

The tuned hyperparameters can be obtained as a dictionary with the following command:

```
from spotpython.hyperparameters.values import get_tuned_hyperparameters
get_tuned_hyperparameters(spot_tuner, fun_control)
```

The results can be saved and reloaded with the following commands:

```
from spotpython.utils.file import save_pickle, load_pickle
from spotpython.utils.init import get_experiment_name
experiment_name = get_experiment_name("024")
SAVE_AND_LOAD = False
if SAVE_AND_LOAD == True:
    save_pickle(spot_tuner, experiment_name)
    spot_tuner = load_pickle(experiment_name)
```

12.19. spotpython as a Hyperparameter Tuner

12.19.1. Modifying Hyperparameter Levels

spotpython distinguishes between different types of hyperparameters. The following types are supported:

- int (integer)
- float (floating point number)
- boolean (boolean)
- factor (categorical)

12.19.1.1. Integer Hyperparameters

Integer hyperparameters can be modified with the set_int_hyperparameter_values() [SOURCE] function. The following code snippet shows how to modify the n_estimators hyperparameter of a random forest model:

```
from spotriver.hyperdict.river_hyper_dict import RiverHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import set_int_hyperparameter_values
from spotpython.utils.eda import print_exp_table
fun_control = fun_control_init(
    core_model_name="forest.AMFRegressor",
    hyperdict=RiverHyperDict,
)
print("Before modification:")
print_exp_table(fun_control)
set_int_hyperparameter_values(fun_control, "n_estimators", 2, 5)
print("After modification:")
print_exp_table(fun_control)
```

```
Before modification:
| name            | type    |   default |   lower |   upper | transform             |
|-----------------|---------|-----------|---------|---------|-----------------------
|
| n_estimators    | int     |         3 |     2   |       5 | transform_power_2_int |
| step            | float   |         1 |     0.1 |      10 | None                  |
| use_aggregation | factor  |         1 |     0   |       1 | None                  |
Setting hyperparameter n_estimators to value [2, 5].
Variable type is int.
Core type is None.
Calling modify_hyper_parameter_bounds().
```

```
After modification:
| name            | type    |   default |   lower |   upper | transform             |
|-----------------|---------|-----------|---------|---------|-----------------------
|
| n_estimators    | int     |         3 |     2   |       5 | transform_power_2_int |
| step            | float   |         1 |     0.1 |      10 | None                  |
| use_aggregation | factor  |         1 |     0   |       1 | None                  |
```

12.19.1.2. Float Hyperparameters

Float hyperparameters can be modified with the `set_float_hyperparameter_values()` [SOURCE] function. The following code snippet shows how to modify the `step` hyperparameter of a hyperparameter of a Mondrian Regression Tree model:

```
from spotriver.hyperdict.river_hyper_dict import RiverHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import set_float_hyperparameter_values
from spotpython.utils.eda import print_exp_table
fun_control = fun_control_init(
    core_model_name="forest.AMFRegressor",
    hyperdict=RiverHyperDict,
)
print("Before modification:")
print_exp_table(fun_control)
set_float_hyperparameter_values(fun_control, "step", 0.2, 5)
print("After modification:")
print_exp_table(fun_control)
```

```
Before modification:
| name            | type    |   default |   lower |   upper | transform             |
|-----------------|---------|-----------|---------|---------|-----------------------
|
| n_estimators    | int     |         3 |     2   |       5 | transform_power_2_int |
| step            | float   |         1 |     0.1 |      10 | None                  |
| use_aggregation | factor  |         1 |     0   |       1 | None                  |
Setting hyperparameter step to value [0.2, 5].
Variable type is float.
Core type is None.
Calling modify_hyper_parameter_bounds().
After modification:
| name            | type    |   default |   lower |   upper | transform             |
|-----------------|---------|-----------|---------|---------|-----------------------
|
```

```
| n_estimators    | int    |         3 |     2   |       5 | transform_power_2_int |
| step            | float  |         1 |     0.2 |       5 | None                  |
| use_aggregation | factor |         1 |     0   |       1 | None                  |
```

12.19.1.3. Boolean Hyperparameters

Boolean hyperparameters can be modified with the `set_boolean_hyperparameter_values()` [SOURCE] function. The following code snippet shows how to modify the `use_aggregation` hyperparameter of a Mondrian Regression Tree model:

```python
from spotriver.hyperdict.river_hyper_dict import RiverHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import set_boolean_hyperparameter_values
from spotpython.utils.eda import print_exp_table
fun_control = fun_control_init(
    core_model_name="forest.AMFRegressor",
    hyperdict=RiverHyperDict,
)
print("Before modification:")
print_exp_table(fun_control)
set_boolean_hyperparameter_values(fun_control, "use_aggregation", 0, 0)
print("After modification:")
print_exp_table(fun_control)
```

```
Before modification:
| name            | type   |   default |   lower |   upper | transform             |
|-----------------|--------|-----------|---------|---------|-----------------------
|
| n_estimators    | int    |         3 |     2   |       5 | transform_power_2_int |
| step            | float  |         1 |     0.1 |      10 | None                  |
| use_aggregation | factor |         1 |     0   |       1 | None                  |
Setting hyperparameter use_aggregation to value [0, 0].
Variable type is factor.
Core type is bool.
Calling modify_boolean_hyper_parameter_levels().
After modification:
| name            | type   |   default |   lower |   upper | transform             |
|-----------------|--------|-----------|---------|---------|-----------------------
|
| n_estimators    | int    |         3 |     2   |       5 | transform_power_2_int |
| step            | float  |         1 |     0.1 |      10 | None                  |
| use_aggregation | factor |         1 |     0   |       0 | None                  |
```

12.19.1.4. Factor Hyperparameters

Factor hyperparameters can be modified with the `set_factor_hyperparameter_values()` [SOURCE] function. The following code snippet shows how to modify the `leaf_model` hyperparameter of a Hoeffding Tree Regressor model:

```
from spotriver.hyperdict.river_hyper_dict import RiverHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import set_factor_hyperparameter_values
from spotpython.utils.eda import print_exp_table
fun_control = fun_control_init(
    core_model_name="tree.HoeffdingTreeRegressor",
    hyperdict=RiverHyperDict,
)
print("Before modification:")
print_exp_table(fun_control)
set_factor_hyperparameter_values(fun_control, "leaf_model", ['LinearRegression',
                                                             'Perceptron'])
print("After modification:")
```

```
Before modification:
| name                    | type    | default           |   lower |    upper | transform              |
|-------------------------|---------|-------------------|---------|----------
|-------------------------|
| grace_period            | int     | 200               |  10     | 1000     | None                   |
| max_depth               | int     | 20                |   2     |   20     | transform_power_2_int  |
| delta                   | float   | 1e-07             |   1e-08 |    1e-
06 | None                    |
| tau                     | float   | 0.05              |   0.01  |    0.1   | None                   |
| leaf_prediction         | factor  | mean              |   0     |    2     | None                   |
| leaf_model              | factor  | LinearRegression  |   0     |    2     | None                   |
| model_selector_decay    | float   | 0.95              |   0.9   |    0.99  | None                   |
| splitter                | factor  | EBSTSplitter      |   0     |    2     | None                   |
| min_samples_split       | int     | 5                 |   2     |   10     | None                   |
| binary_split            | factor  | 0                 |   0     |    1     | None                   |
| max_size                | float   | 500.0             | 100     | 1000     | None                   |
| memory_estimate_period  | int     | 6                 |   3     |    8     | transform_power_10_int |
| stop_mem_management     | factor  | 0                 |   0     |    1     | None                   |
| remove_poor_attrs       | factor  | 0                 |   0     |    1     | None                   |
| merit_preprune          | factor  | 1                 |   0     |    1     | None                   |
After modification:
```

12.20. Sampling in spotpython

`spotpython` uses a class for generating space-filling designs using Latin Hypercube Sampling (LHS) and maximin distance criteria. It is based on `scipy`'s `LatinHypercube` class. The following example demonstrates how to generate a Latin Hypercube Sampling design using `spotpython`. The result is shown in Figure 12.5. As can seen in the figure, a Latin hypercube sample generates n points in $[0,1)^d$. Each univariate marginal distribution is stratified, placing exactly one point in $[j/n,(j+1)/n)$ for $j=0,1,...,n-1$.

```
import matplotlib.pyplot as plt
import numpy as np
from spotpython.design.spacefilling import SpaceFilling
lhd = SpaceFilling(k=2, seed=123)
X = lhd.scipy_lhd(n=10, repeats=1, lower=np.array([0, 0]), upper=np.array([10, 10]))
plt.scatter(X[:, 0], X[:, 1])
plt.xlabel('x1')
plt.ylabel('x2')
plt.grid()
```

Figure 12.5.: Latin Hypercube Sampling design (sampling plan)

12.21. Example: Spot and the Sphere Function

Central Idea: Evaluation of the surrogate model S is much cheaper (or / and much faster) than running the real-world experiment f. We start with a small example.

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, design_control_init
from spotpython.hyperparameters.values import set_control_key_value
from spotpython.spot import Spot
import matplotlib.pyplot as plt
```

12.21.1. The Objective Function: Sphere

The spotpython package provides several classes of objective functions. We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = x^2$$

```
fun = Analytical().fun_sphere
```

We can apply the function fun to input values and plot the result:

```
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x, y, "k")
plt.show()
```

12.21.2. The Spot Method as an Optimization Algorithm Using a Surrogate Model

We initialize the `fun_control` dictionary. The `fun_control` dictionary contains the parameters for the objective function. The `fun_control` dictionary is passed to the `Spot` method.

```
fun_control=fun_control_init(lower = np.array([-1]),
                     upper = np.array([1]))
spot_0 = Spot(fun=fun,
                   fun_control=fun_control)
spot_0.run()
```

```
spotpython tuning: 2.4412172266370036e-09 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 2.032431480352692e-09 [########--] 80.00%. Success rate: 100.00%
spotpython tuning: 2.032431480352692e-09 [#########-] 86.67%. Success rate: 66.67%
spotpython tuning: 2.032431480352692e-09 [#########-] 93.33%. Success rate: 50.00%
spotpython tuning: 1.2228307315407816e-10 [##########] 100.00%. Success rate: 60.00% I

Experiment saved to 000_res.pkl


<spotpython.spot.spot.Spot at 0x39f58c7d0>
```

The method `print_results()` prints the results, i.e., the best objective function value ("min y") and the corresponding input value ("x0").

```
spot_0.print_results()
```

```
min y: 1.2228307315407816e-10
x0: 1.1058167712332733e-05
```

```
[['x0', np.float64(1.1058167712332733e-05)]]
```

To plot the search progress, the method `plot_progress()` can be used. The parameter `log_y` is used to plot the objective function values on a logarithmic scale.

```
spot_0.plot_progress(log_y=True)
```

Figure 12.6.: Visualization of the search progress of the `Spot` method. The black elements (points and line) represent the initial design, before the surrogate is build. The red elements represent the search on the surrogate.

If the dimension of the input space is one, the method `plot_model()` can be used to visualize the model and the underlying objective function values.

```
spot_0.plot_model()
```

Figure 12.7.: Visualization of the model and the underlying objective function values.

12.22. Spot Parameters: fun_evals, init_size and show_models

We will modify three parameters:

1. The number of function evaluations (`fun_evals`) will be set to `10` (instead of 15, which is the default value) in the `fun_control` dictionary.
2. The parameter `show_models`, which visualizes the search process for each single iteration for 1-dim functions, in the `fun_control` dictionary.
3. The size of the initial design (`init_size`) in the `design_control` dictionary.

The full list of the `Spot` parameters is shown in code reference on GitHub, see Spot.

```
fun_control=fun_control_init(lower = np.array([-1]),
                     upper = np.array([1]),
                     fun_evals = 10,
                     show_models = True)
design_control = design_control_init(init_size=9)
spot_1 = Spot(fun=fun,
```

```
                    fun_control=fun_control,
                    design_control=design_control)
spot_1.run()
```

```
spotpython tuning: 3.7772791809054267e-10 [##########] 100.00%. Success rate: 100.00%

Experiment saved to 000_res.pkl
```

12.23. Print the Results

```
spot_1.print_results()
```

```
min y: 3.7772791809054267e-10
x0: -1.9435223643954877e-05

[['x0', np.float64(-1.9435223643954877e-05)]]
```

12.24. Show the Progress

```
spot_1.plot_progress()
```

12.25. Visualizing the Optimization and Hyperparameter Tuning Process with TensorBoard

`spotpython` supports the visualization of the hyperparameter tuning process with TensorBoard. The following example shows how to use TensorBoard with `spotpython`.

First, we define an "PREFIX" to identify the hyperparameter tuning process. The PREFIX is used to create a directory for the TensorBoard files.

```
fun_control = fun_control_init(
    PREFIX = "01",
    lower = np.array([-1]),
    upper = np.array([2]),
    fun_evals=100,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True)
design_control = design_control_init(init_size=5)
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_11_04_20_33_06_0
Created spot_tensorboard_path: runs/spot_logs/01_maans08_2025-11-04_20-
33-06 for SummaryWriter()
```

Since the `tensorboard_log` is `True`, `spotpython` will log the optimization process in the TensorBoard files. The argument `TENSORBOARD_CLEAN=True` will move the TensorBoard files from the previous run to a backup folder, so that TensorBoard

files from previous runs are not overwritten and a clean start in the `runs` folder is guaranteed.

```
spot_tuner = Spot(fun=fun,
                  fun_control=fun_control,
                  design_control=design_control)
spot_tuner.run()
spot_tuner.print_results()
```

```
spotpython tuning: 6.428084284497196e-06 [#---------] 6.00%. Success rate: 100.00%
spotpython tuning: 1.0018985789303623e-07 [#---------] 7.00%. Success rate: 100.00%
spotpython tuning: 7.156564048814216e-08 [#---------] 8.00%. Success rate: 100.00%
spotpython tuning: 7.4015092835768465e-09 [#---------] 9.00%. Success rate: 100.00%
spotpython tuning: 7.4015092835768465e-09 [#---------] 10.00%. Success rate: 80.00%
spotpython tuning: 7.4015092835768465e-09 [#---------] 11.00%. Success rate: 66.67%
spotpython tuning: 7.4015092835768465e-09 [#---------] 12.00%. Success rate: 57.14%
spotpython tuning: 7.4015092835768465e-09 [#---------] 13.00%. Success rate: 50.00%
spotpython tuning: 7.4015092835768465e-09 [#---------] 14.00%. Success rate: 44.44%
spotpython tuning: 7.4015092835768465e-09 [##--------] 15.00%. Success rate: 40.00%
spotpython tuning: 7.4015092835768465e-09 [##--------] 16.00%. Success rate: 36.36%
spotpython tuning: 7.4015092835768465e-09 [##--------] 17.00%. Success rate: 33.33%
spotpython tuning: 7.4015092835768465e-09 [##--------] 18.00%. Success rate: 30.77%
spotpython tuning: 7.4015092835768465e-09 [##--------] 19.00%. Success rate: 28.57%
spotpython tuning: 7.4015092835768465e-09 [##--------] 20.00%. Success rate: 26.67%
spotpython tuning: 7.4015092835768465e-09 [##--------] 21.00%. Success rate: 25.00%
spotpython tuning: 7.4015092835768465e-09 [##--------] 22.00%. Success rate: 23.53%
spotpython tuning: 7.4015092835768465e-09 [##--------] 23.00%. Success rate: 22.22%
spotpython tuning: 7.4015092835768465e-09 [##--------] 24.00%. Success rate: 21.05%
spotpython tuning: 7.4015092835768465e-09 [##--------] 25.00%. Success rate: 20.00%
spotpython tuning: 7.4015092835768465e-09 [###-------] 26.00%. Success rate: 19.05%
spotpython tuning: 7.4015092835768465e-09 [###-------] 27.00%. Success rate: 18.18%
spotpython tuning: 7.4015092835768465e-09 [###-------] 28.00%. Success rate: 17.39%
spotpython tuning: 7.4015092835768465e-09 [###-------] 29.00%. Success rate: 16.67%
spotpython tuning: 7.4015092835768465e-09 [###-------] 30.00%. Success rate: 16.00%
spotpython tuning: 7.4015092835768465e-09 [###-------] 31.00%. Success rate: 15.38%
spotpython tuning: 7.4015092835768465e-09 [###-------] 32.00%. Success rate: 14.81%
spotpython tuning: 7.4015092835768465e-09 [###-------] 33.00%. Success rate: 14.29%
spotpython tuning: 7.4015092835768465e-09 [###-------] 34.00%. Success rate: 13.79%
spotpython tuning: 7.4015092835768465e-09 [####------] 35.00%. Success rate: 13.33%
spotpython tuning: 7.4015092835768465e-09 [####------] 36.00%. Success rate: 12.90%
spotpython tuning: 7.4015092835768465e-09 [####------] 37.00%. Success rate: 12.50%
Using spacefilling design as fallback.
spotpython tuning: 7.4015092835768465e-09 [####------] 38.00%. Success rate: 12.12%
spotpython tuning: 7.4015092835768465e-09 [####------] 39.00%. Success rate: 11.76%
```

```
spotpython tuning: 7.4015092835768465e-09 [####------] 40.00%. Success rate: 11.43%
spotpython tuning: 7.4015092835768465e-09 [####------] 41.00%. Success rate: 11.11%
spotpython tuning: 7.4015092835768465e-09 [####------] 42.00%. Success rate: 10.81%
spotpython tuning: 7.4015092835768465e-09 [####------] 43.00%. Success rate: 10.53%
Using spacefilling design as fallback.
spotpython tuning: 7.4015092835768465e-09 [####------] 44.00%. Success rate: 10.26%
spotpython tuning: 7.4015092835768465e-09 [####------] 45.00%. Success rate: 10.00%
spotpython tuning: 4.312084739261867e-09 [#####-----] 46.00%. Success rate: 12.20%
spotpython tuning: 4.312084739261867e-09 [#####-----] 47.00%. Success rate: 11.90%
spotpython tuning: 4.2846000859350335e-09 [#####-----] 48.00%. Success rate: 13.95%
spotpython tuning: 4.254083847765371e-09 [#####-----] 49.00%. Success rate: 15.91%
spotpython tuning: 4.254083847765371e-09 [#####-----] 50.00%. Success rate: 15.56%
Using spacefilling design as fallback.
spotpython tuning: 4.254083847765371e-09 [#####-----] 51.00%. Success rate: 15.22%
spotpython tuning: 9.802119461374754e-10 [#####-----] 52.00%. Success rate: 17.02%
spotpython tuning: 9.802119461374754e-10 [#####-----] 53.00%. Success rate: 16.67%
spotpython tuning: 9.802119461374754e-10 [#####-----] 54.00%. Success rate: 16.33%
spotpython tuning: 9.802119461374754e-10 [######----] 55.00%. Success rate: 16.00%
spotpython tuning: 9.802119461374754e-10 [######----] 56.00%. Success rate: 15.69%
spotpython tuning: 9.802119461374754e-10 [######----] 57.00%. Success rate: 15.38%
spotpython tuning: 9.802119461374754e-10 [######----] 58.00%. Success rate: 15.09%
spotpython tuning: 9.802119461374754e-10 [######----] 59.00%. Success rate: 14.81%
spotpython tuning: 9.421493769530215e-10 [######----] 60.00%. Success rate: 16.36%
spotpython tuning: 9.421493769530215e-10 [######----] 61.00%. Success rate: 16.07%
Using spacefilling design as fallback.
spotpython tuning: 9.421493769530215e-10 [######----] 62.00%. Success rate: 15.79%
spotpython tuning: 9.421493769530215e-10 [######----] 63.00%. Success rate: 15.52%
spotpython tuning: 9.421493769530215e-10 [######----] 64.00%. Success rate: 15.25%
spotpython tuning: 9.421493769530215e-10 [######----] 65.00%. Success rate: 15.00%
spotpython tuning: 9.421493769530215e-10 [#######---] 66.00%. Success rate: 14.75%
spotpython tuning: 9.421493769530215e-10 [#######---] 67.00%. Success rate: 14.52%
spotpython tuning: 9.421493769530215e-10 [#######---] 68.00%. Success rate: 14.29%
spotpython tuning: 9.421493769530215e-10 [#######---] 69.00%. Success rate: 14.06%
Using spacefilling design as fallback.
spotpython tuning: 9.421493769530215e-10 [#######---] 70.00%. Success rate: 13.85%
spotpython tuning: 9.421493769530215e-10 [#######---] 71.00%. Success rate: 13.64%
spotpython tuning: 9.421493769530215e-10 [#######---] 72.00%. Success rate: 13.43%
Using spacefilling design as fallback.
spotpython tuning: 9.421493769530215e-10 [#######---] 73.00%. Success rate: 13.24%
spotpython tuning: 9.421493769530215e-10 [#######---] 74.00%. Success rate: 13.04%
spotpython tuning: 9.421493769530215e-10 [########--] 75.00%. Success rate: 12.86%
spotpython tuning: 9.421493769530215e-10 [########--] 76.00%. Success rate: 12.68%
spotpython tuning: 9.421493769530215e-10 [########--] 77.00%. Success rate: 12.50%
spotpython tuning: 9.421493769530215e-10 [########--] 78.00%. Success rate: 12.33%
Using spacefilling design as fallback.
```

```
spotpython tuning: 9.421493769530215e-10 [########--] 79.00%. Success rate: 12.16%
spotpython tuning: 2.670314613623362e-10 [########--] 80.00%. Success rate: 13.33%
spotpython tuning: 1.0264793385366614e-10 [########--] 81.00%. Success rate: 14.47%
spotpython tuning: 1.0264793385366614e-10 [########--] 82.00%. Success rate: 14.29%
spotpython tuning: 1.0264793385366614e-10 [########--] 83.00%. Success rate: 14.10%
spotpython tuning: 1.0264793385366614e-10 [########--] 84.00%. Success rate: 13.92%
spotpython tuning: 1.0264793385366614e-10 [########--] 85.00%. Success rate: 13.75%
spotpython tuning: 1.0264793385366614e-10 [#########-] 86.00%. Success rate: 13.58%
spotpython tuning: 1.0264793385366614e-10 [#########-] 87.00%. Success rate: 13.41%
spotpython tuning: 1.0264793385366614e-10 [#########-] 88.00%. Success rate: 13.25%
spotpython tuning: 1.0264793385366614e-10 [#########-] 89.00%. Success rate: 13.10%
spotpython tuning: 6.105961713788403e-11 [#########-] 90.00%. Success rate: 14.12%
spotpython tuning: 6.105961713788403e-11 [#########-] 91.00%. Success rate: 13.95%
spotpython tuning: 6.105961713788403e-11 [#########-] 92.00%. Success rate: 13.79%
spotpython tuning: 6.105961713788403e-11 [#########-] 93.00%. Success rate: 13.64%
spotpython tuning: 6.105961713788403e-11 [#########-] 94.00%. Success rate: 13.48%
Using spacefilling design as fallback.
spotpython tuning: 6.105961713788403e-11 [##########] 95.00%. Success rate: 13.33%
spotpython tuning: 4.607935942920989e-14 [##########] 96.00%. Success rate: 14.29%
spotpython tuning: 4.607935942920989e-14 [##########] 97.00%. Success rate: 14.13%
spotpython tuning: 4.607935942920989e-14 [##########] 98.00%. Success rate: 13.98%
spotpython tuning: 4.607935942920989e-14 [##########] 99.00%. Success rate: 13.83%
spotpython tuning: 4.607935942920989e-14 [##########] 100.00%. Success rate: 13.68% Do

Experiment saved to 01_res.pkl
min y: 4.607935942920989e-14
x0: 2.1466103379330375e-07


[['x0', np.float64(2.1466103379330375e-07)]]
```

Now we can start TensorBoard in the background. The TensorBoard process will read the TensorBoard files and visualize the hyperparameter tuning process. From the terminal, we can start TensorBoard with the following command:

```
tensorboard --logdir="./runs"
```

`logdir` is the directory where the TensorBoard files are stored. In our case, the TensorBoard files are stored in the directory `./runs`.

TensorBoard will start a web server on port 6006. We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The first TensorBoard visualization shows the objective function values plotted against the wall time. The wall time is the time that has passed since the start of the hyperparameter tuning process. The five initial design points are shown in the upper left region of the plot. The line visualizes the optimization process.

The second TensorBoard visualization shows the input values, i.e., x_0, plotted against the wall time.

The third TensorBoard plot illustrates how `spotpython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

Figure 12.8.: TensorBoard visualization of the spotpython process.

12.26. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

13. Multi-dimensional Functions

This chapter illustrates how high-dimensional functions can be optimized and analyzed. For reasons of illustration, we will use the three-dimensional Sphere function, which is a simple and well-known function. The problem dimension is $k = 3$, but can be easily adapted to other, higher dimensions.

```
import numpy as np
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, surrogate_control_init, design_control_init
from spotpython.spot import Spot
```

13.1. The Objective Function: 3-dim Sphere

The `spotpython` package provides several classes of objective functions. We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = \sum_{i}^{k} x_i^2.$$

The Sphere function is continuous, convex and unimodal. The plot shows its two-dimensional form. The global minimum is

$$f(x) = 0, \text{at } x = (0, 0, \dots, 0).$$

It is available as `fun_sphere` in the `Analytical` class [SOURCE].

```
fun = Analytical().fun_sphere
```

Here we will use problem dimension $k = 3$, which can be specified by the `lower` bound arrays. The size of the `lower` bound array determines the problem dimension. If we select `-1.0 * np.ones(3)`, a three-dimensional function is created.

In contrast to the one-dimensional case (Section 12.25), where only one `theta` value was used, we will use three different `theta` values (one for each dimension). This is done automatically, because the setting `isotropic=False` is the default in the `surrogate_control`. As default, `spotpython` uses separate theta values for each

problem dimension. More specifically, if `isotropic` is set to `True`, only one theta value is used for all dimensions. If `isotropic` is set to `False`, then the k theta values are used, where k is the problem dimension. The meaning of "isotropic" is explained in @#sec-iso-aniso-kriging.

The prefix is set to `"03"` to distinguish the results from the one-dimensional case. Again, TensorBoard can be used to monitor the progress of the optimization.

We can also add interpretable labels to the dimensions, which will be used in the plots. Therefore, we set `var_name=["Pressure", "Temp", "Lambda"]` instead of the default `var_name=None`, which would result in the labels `x_0`, `x_1`, and `x_2`.

```
fun_control = fun_control_init(
              PREFIX="03",
              lower = -1.0*np.ones(3),
              upper = np.ones(3),
              var_name=["Pressure", "Temp", "Lambda"],
              TENSORBOARD_CLEAN=True,
              tensorboard_log=True)
surrogate_control = surrogate_control_init()
spot_3 = Spot(fun=fun,
                 fun_control=fun_control,
                 surrogate_control=surrogate_control)
spot_3.run()
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_11_04_20_33
Created spot_tensorboard_path: runs/spot_logs/03_maans08_2025-11-04_20-
33-53 for SummaryWriter()
spotpython tuning: 0.03443805738918038 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 0.03134407473407909 [########--] 80.00%. Success rate: 100.00%
spotpython tuning: 0.0009627751650306065 [#########-] 86.67%. Success rate: 100.00%
spotpython tuning: 8.337474234130422e-05 [#########-] 93.33%. Success rate: 100.00%
spotpython tuning: 3.8172596925679106e-05 [##########] 100.00%. Success rate: 100.00%

Experiment saved to 03_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x1082d7e00>
```

> **Note**
>
> Now we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

and can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

13.1.1. Results

13.1.1.1. Best Objective Function Values

The best objective function value and its corresponding input values are printed as follows:

```
_ = spot_3.print_results()
```

```
min y: 3.8172596925679106e-05
Pressure: 0.0036952312636787847
Temp: 0.0009377053161100289
Lambda: 0.004861951416226718
```

The method `plot_progress()` plots current and best found solutions versus the number of iterations as shown in Figure 13.1.

```
spot_3.plot_progress()
```

Figure 13.1.: Progress of the optimization process for the 3-dim Sphere function. The initial design points are shown in black, whereas the points that were found by the search on the surrogate are plotted in red.

13.1.1.2. A Contour Plot

We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows. Note, we have specified identical `min_z` and `max_z` values to generate comparable plots.

```
spot_3.plot_contour(i=0, j=1, min_z=0, max_z=2.25)
```

- In a similar manner, we can plot dimension $i = 0$ and $j = 2$:

```
spot_3.plot_contour(i=0, j=2, min_z=0, max_z=2.25)
```

- The final combination is $i = 1$ and $j = 2$:

```
spot_3.plot_contour(i=1, j=2, min_z=0, max_z=2.25)
```

- The three plots look very similar, because the `fun_sphere` is symmetric.
- This can also be seen from the variable importance:

```
_ = spot_3.print_importance()
```

```
Pressure:  99.99999999999999
Temp:  99.17928367751998
Lambda:  91.43655243996746
```

```
spot_3.plot_importance()
```

13.1.2. TensorBoard

Figure 13.2.: TensorBoard visualization of the spotpython process. Objective function values plotted against wall time.

The second TensorBoard visualization shows the input values, i.e., $x_0, \dots, x_2$, plotted against the wall time.

The third TensorBoard plot illustrates how `spotpython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

Figure 13.3.: TensorBoard visualization of the spotpython surrogate model.

13.1.3. Conclusion

Based on this quick analysis, we can conclude that all three dimensions are equally important (as expected, because the Analytical function is known).

13.2. Exercises

Exercise 13.1 (The Three Dimensional `fun_cubed`)**.** The `spotpython` package provides several classes of objective functions.

We will use the `fun_cubed` in the `Analytical` class [SOURCE]. The input dimension is `3`. The search range is $-1 \leq x \leq 1$ for all dimensions.

Tasks: * Generate contour plots * Calculate the variable importance. * Discuss the variable importance: * Are all variables equally important? * If not: * Which is the most important variable? * Which is the least important variable?

Exercise 13.2 (The Ten Dimensional `fun_wing_wt`).

- The input dimension is `10`. The search range is $0 \leq x \leq 1$ for all dimensions.
- Calculate the variable importance.
- Discuss the variable importance:
  - Are all variables equally important?
  - If not:
    - Which is the most important variable?
    - Which is the least important variable?
  - Generate contour plots for the three most important variables. Do they confirm your selection?

Exercise 13.3 (The Three Dimensional `fun_runge`).

- The input dimension is `3`. The search range is $-5 \leq x \leq 5$ for all dimensions.
- Generate contour plots
- Calculate the variable importance.
- Discuss the variable importance:
  - Are all variables equally important?
  - If not:
    - Which is the most important variable?
    - Which is the least important variable?

Exercise 13.4 (The Three Dimensional `fun_linear`).

- The input dimension is `3`. The search range is $-5 \leq x \leq 5$ for all dimensions.
- Generate contour plots
- Calculate the variable importance.
- Discuss the variable importance:
  - Are all variables equally important?
  - If not:
    - Which is the most important variable?
    - Which is the least important variable?

Exercise 13.5 (The Two Dimensional Rosenbrock Function `fun_rosen`).

- The input dimension is `2`. The search range is $-5 \leq x \leq 10$ for all dimensions.

- See Rosenbrock function and Rosenbrock Function for details.
- Generate contour plots
- Calculate the variable importance.
- Discuss the variable importance:
  - Are all variables equally important?
  - If not:
    - Which is the most important variable?
    - Which is the least important variable?

13.3. Selected Solutions

Solution 13.1 (Solution to Exercise 13.1: The Three-dimensional Cubed Function `fun_cubed`). We instanciate the `fun_cubed` function from the `Analytical` class.

```
from spotpython.fun.objectivefunctions import Analytical
fun_cubed = Analytical().fun_cubed
```

- Here we will use problem dimension $k = 3$, which can be specified by the `lower` bound arrays. The size of the `lower` bound array determines the problem dimension. If we select `-1.0 * np.ones(3)`, a three-dimensional function is created.
- In contrast to the one-dimensional case, where only one `theta` value was used, we will use three different `theta` values (one for each dimension).
- The prefix is set to `"03"` to distinguish the results from the one-dimensional case.
- We will set the `fun_evals=20` to limit the number of function evaluations to 20 for this example.
- The size of the initial design is set to `10` by default. It can be changed by setting `init_size=10` via `design_control_init` in the `design_control` dictionary.
- Again, TensorBoard can be used to monitor the progress of the optimization.
- We can also add interpretable labels to the dimensions, which will be used in the plots. Therefore, we set `var_name=["Pressure", "Temp", "Lambda"]` instead of the default `var_name=None`, which would result in the labels `x_0`, `x_1`, and `x_2`.

Here is the link to the documentation of the fun_control_init function: [DOC]. The documentation of the `design_control_init` function can be found here: [DOC].

The setup can be done as follows:

```
fun_control = fun_control_init(
              PREFIX="cubed",
              fun_evals=20,
              lower = -1.0*np.ones(3),
```

```
               upper = np.ones(3),
               var_name=["Pressure", "Temp", "Lambda"],
               TENSORBOARD_CLEAN=True,
               tensorboard_log=True
               )

surrogate_control = surrogate_control_init()
design_control = design_control_init(init_size=10)
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_11_04_20_33_59_0
Created spot_tensorboard_path: runs/spot_logs/cubed_maans08_2025-11-
04_20-33-59 for SummaryWriter()
```

- After the setup, we can pass the dictionaries to the `Spot` class and run the optimization process.

```
spot_cubed = Spot(fun=fun_cubed,
                  fun_control=fun_control,
                  surrogate_control=surrogate_control)
spot_cubed.run()
```

```
spotpython tuning: -1.4616828410586926 [######----] 55.00%. Success rate: 100.00%
spotpython tuning: -1.4616828410586926 [######----] 60.00%. Success rate: 50.00%
spotpython tuning: -2.0973782471243143 [######----] 65.00%. Success rate: 66.67%
spotpython tuning: -2.0973782471243143 [#######---] 70.00%. Success rate: 50.00%
spotpython tuning: -2.0973782471243143 [########--] 75.00%. Success rate: 40.00%
spotpython tuning: -2.122182907736537 [########--] 80.00%. Success rate: 50.00%
spotpython tuning: -3.0 [########--] 85.00%. Success rate: 57.14%
Using spacefilling design as fallback.
spotpython tuning: -3.0 [#########-] 90.00%. Success rate: 50.00%
Using spacefilling design as fallback.
spotpython tuning: -3.0 [##########] 95.00%. Success rate: 44.44%
spotpython tuning: -3.0 [##########] 100.00%. Success rate: 40.00% Done...

Experiment saved to cubed_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x157d87110>
```

- Results

```
_ = spot_cubed.print_results()
```

```
min y: -3.0
Pressure: -1.0
Temp: -1.0
Lambda: -1.0
```

```
spot_cubed.plot_progress()
```

- Contour Plots

We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows.

We can specify identical `min_z` and `max_z` values to generate comparable plots. The default values are `min_z=None` and `max_z=None`, which will be replaced by the minimum and maximum values of the objective function.

```
min_z = -3
max_z = 1
spot_cubed.plot_contour(i=0, j=1, min_z=min_z, max_z=max_z)
```

- In a similar manner, we can plot dimension $i = 0$ and $j = 2$:

```
spot_cubed.plot_contour(i=0, j=2, min_z=min_z, max_z=max_z)
```

- The final combination is $i = 1$ and $j = 2$:

```
spot_cubed.plot_contour(i=1, j=2, min_z=min_z, max_z=max_z)
```

- The variable importance can be printed and visualized as follows:

```
_ = spot_cubed.print_importance()
```

```
Pressure:  30.019890618668967
Temp:  100.0
Lambda:  93.41582927565221
```

```
spot_cubed.plot_importance()
```

Solution 13.2 (Solution to Exercise 13.5: The Two-dimensional Rosenbrock Function `fun_rosen`).

```
import numpy as np
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, surrogate_control_init
from spotpython.spot import Spot
```

- The Objective Function: 2-dim `fun_rosen`

The `spotpython` package provides several classes of objective functions. We will use the `fun_rosen` in the `Analytical` class [SOURCE].

```
fun_rosen = Analytical().fun_rosen
```

- Here we will use problem dimension $k = 2$, which can be specified by the `lower` bound arrays.
- The size of the `lower` bound array determines the problem dimension. If we select `-5.0 * np.ones(2)`, a two-dimensional function is created.
- In contrast to the one-dimensional case, where only one `theta` value is used, we will use k different `theta` values (one for each dimension).
- The prefix is set to `"ROSEN"`.
- Again, TensorBoard can be used to monitor the progress of the optimization.

```
fun_control = fun_control_init(
              PREFIX="ROSEN",
              lower = -5.0*np.ones(2),
              upper = 10*np.ones(2),
              fun_evals=25)
surrogate_control = surrogate_control_init()
spot_rosen = Spot(fun=fun_rosen,
                  fun_control=fun_control,
                  surrogate_control=surrogate_control)
spot_rosen.run()
```

```
spotpython tuning: 90.77788110350957 [####------] 44.00%. Success rate: 100.00%
spotpython tuning: 1.0173064877521703 [#####-----] 48.00%. Success rate: 100.00%
spotpython tuning: 1.0173064877521703 [#####-----] 52.00%. Success rate: 66.67%
spotpython tuning: 1.0173064877521703 [######----] 56.00%. Success rate: 50.00%
spotpython tuning: 0.7867277329036355 [######----] 60.00%. Success rate: 60.00%
spotpython tuning: 0.7867277329036355 [######----] 64.00%. Success rate: 50.00%
spotpython tuning: 0.7867277329036355 [#######---] 68.00%. Success rate: 42.86%
spotpython tuning: 0.7867277329036355 [#######---] 72.00%. Success rate: 37.50%
spotpython tuning: 0.7867277329036355 [########--] 76.00%. Success rate: 33.33%
spotpython tuning: 0.7867277329036355 [########--] 80.00%. Success rate: 30.00%
spotpython tuning: 0.7867277329036355 [########--] 84.00%. Success rate: 27.27%
spotpython tuning: 0.7867277329036355 [#########-] 88.00%. Success rate: 25.00%
spotpython tuning: 0.7867277329036355 [#########-] 92.00%. Success rate: 23.08%
spotpython tuning: 0.7867277329036355 [##########] 96.00%. Success rate: 21.43%
spotpython tuning: 0.7867277329036355 [##########] 100.00%. Success rate: 20.00% Done...

Experiment saved to ROSEN_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x16a8f6c10>
```

Now we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

and can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

- Results

```
_ = spot_rosen.print_results()
```

```
min y: 0.7867277329036355
x0: 0.13953655588708272
x1: 0.08753688433642533
```

```
spot_rosen.plot_progress(log_y=True)
```

- A Contour Plot: We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows.
- Note: For higher dimensions, it might be useful to have identical `min_z` and `max_z` values to generate comparable plots. The default values are `min_z=None` and `max_z=None`, which will be replaced by the minimum and maximum values of the objective function.

```
min_z = None
max_z = None
spot_rosen.plot_contour(i=0, j=1, min_z=min_z, max_z=max_z)
```

- The variable importance can be calculated as follows:

```
_ = spot_rosen.print_importance()
```

```
x0:  99.99999999999999
x1:  2.2596151128068156
```

```
spot_rosen.plot_importance()
```

13.4. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

14. Isotropic and Anisotropic Kriging

This chapter illustrates the difference between isotropic and anisotropic Kriging models. The difference is illustrated with the help of the `spotpython` package. Isotropic Kriging models use the same `theta` value for every dimension. Anisotropic Kriging models use different `theta` values for each dimension.

14.1. Example: Isotropic `Spot` Surrogate and the 2-dim Sphere Function

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.utils.init import fun_control_init, surrogate_control_init
PREFIX="003"
```

14.1.1. The Objective Function: 2-dim Sphere

The `spotpython` package provides several classes of objective functions. We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x, y) = x^2 + y^2$$

The size of the `lower` bound vector determines the problem dimension. Here we will use `np.array([-1, -1])`, i.e., a two-dimensional function.

```
fun = Analytical().fun_sphere
fun_control = fun_control_init(PREFIX=PREFIX,
                               lower = np.array([-1, -1]),
                               upper = np.array([1, 1]))
```

The default `Spot` surrogate model is an anisotropic Kriging model. We will explicitly set the `isotropic` parameter to a value of `False`, so that the same theta value is used for both dimensions. This is done to illustrate the difference between isotropic and anisotropic Kriging models.

```
surrogate_control=surrogate_control_init(isotropic=True)
```

```
spot_2 = Spot(fun=fun,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)

spot_2.run()
```

```
spotpython tuning: 0.0025690995161869214 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 0.0025690995161869214 [########--] 80.00%. Success rate: 50.00%
spotpython tuning: 0.0025690995161869214 [#########-] 86.67%. Success rate: 33.33%
spotpython tuning: 0.0025690995161869214 [#########-] 93.33%. Success rate: 25.00%
spotpython tuning: 0.0025690995161869214 [##########] 100.00%. Success rate: 20.00% Do

Experiment saved to 003_res.pkl


<spotpython.spot.spot.Spot at 0x1040d8050>
```

14.1.2. Results

```
spot_2.print_results()
```

```
min y: 0.0025690995161869214
x0: 0.03166420094465901
x1: 0.03957875559846692


[['x0', np.float64(0.03166420094465901)],
 ['x1', np.float64(0.03957875559846692)]]
```

```
spot_2.plot_progress(log_y=True)
```

14.2. Example With Anisotropic Kriging

As described in Section 14.1, the default parameter setting of `spotpython`'s Kriging surrogate uses different `theta` value for every dimension. This is referred to as "using an anisotropic kernel". To enable isotropic models in `spotpython`, the command `surrogate_control=surrogate_control_init(isotropic=True)` can be used. In this case, the same `theta` value is used for every dimension.

```
spot_2_anisotropic = Spot(fun=fun,
                    fun_control=fun_control)
spot_2_anisotropic.run()
```

```
spotpython tuning: 6.611333241687761e-06 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 6.611333241687761e-06 [########--] 80.00%. Success rate: 50.00%
spotpython tuning: 6.611333241687761e-06 [#########-] 86.67%. Success rate: 33.33%
spotpython tuning: 6.611333241687761e-06 [#########-] 93.33%. Success rate: 25.00%
spotpython tuning: 3.1531067682958562e-06 [##########] 100.00%. Success rate: 40.00% Done...

Experiment saved to 003_res.pkl


<spotpython.spot.spot.Spot at 0x133f99950>
```

The search progress of the optimization with the anisotropic model can be visualized:

```
spot_2_anisotropic.plot_progress(log_y=True)
```

```
spot_2_anisotropic.print_results()
```

```
min y: 3.1531067682958562e-06
x0: -0.0012459503825024937
x1: -0.0012651934289418935
```

```
[['x0', np.float64(-0.0012459503825024937)],
 ['x1', np.float64(-0.0012651934289418935)]]
```

```
spot_2_anisotropic.surrogate.plot()
```

14.2.1. Taking a Look at the `theta` Values

14.2.1.1. `theta` Values from the `spot` Model

We can check, whether one or several `theta` values were used. The `theta` values from the surrogate can be printed as follows:

```
spot_2_anisotropic.surrogate.theta
```

```
array([-0.35825091, -0.1916896 ])
```

- Since the surrogate from the isotropic setting was stored as `spot_2`, we can also take a look at the `theta` value from this model:

```
spot_2.surrogate.theta
```

```
array([-0.07437973])
```

14.2.1.2. TensorBoard

Now we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The TensorBoard plot illustrates how `spotpython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

Figure 14.1.: TensorBoard visualization of the spotpython surrogate model.

14.3. Exercises

14.3.1. 1. The Branin Function `fun_branin`

- Describe the function.
  - The input dimension is 2. The search range is $-5 \leq x_1 \leq 10$ and $0 \leq x_2 \leq 15$.
- Compare the results from `spotpython` run a) with isotropic and b) anisotropic surrogate models.

- Modify the termination criterion: instead of the number of evaluations (which is specified via `fun_evals`), the time should be used as the termination criterion. This can be done as follows (`max_time=1` specifies a run time of one minute):

```
from math import inf
fun_control = fun_control_init(
              fun_evals=inf,
              max_time=1)
```

14.3.2. 2. The Two-dimensional Sin-Cos Function `fun_sin_cos`

- Describe the function.
  - The input dimension is 2. The search range is $-2\pi \leq x_1 \leq 2\pi$ and $-2\pi \leq x_2 \leq 2\pi$.
- Compare the results from `spotpython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

14.3.3. 3. The Two-dimensional Runge Function `fun_runge`

- Describe the function.
  - The input dimension is 2. The search range is $-5 \leq x_1 \leq 5$ and $-5 \leq x_2 \leq 5$.
- Compare the results from `spotpython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

14.3.4. 4. The Ten-dimensional Wing-Weight Function `fun_wingwt`

- Describe the function.
  - The input dimension is 10. The search ranges are between 0 and 1 (values are mapped internally to their natural bounds).
- Compare the results from `spotpython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

14.3.5. 5. The Two-dimensional Rosenbrock Function `fun_rosen`

- Describe the function.
  - The input dimension is 2. The search ranges are between -5 and 10.
- Compare the results from `spotpython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

14.4. Selected Solutions

14.4.1. Solution to Exercise Section 14.3.5: The Two-dimensional Rosenbrock Function `fun_rosen`

14.4.1.1. The Two Dimensional `fun_rosen`: The Isotropic Case

```
import numpy as np
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, surrogate_control_init
from spotpython.spot import Spot
```

The `spotpython` package provides several classes of objective functions. We will use the `fun_rosen` in the `analytical` class [SOURCE].

```
fun_rosen = Analytical().fun_rosen
```

Here we will use problem dimension $k = 2$, which can be specified by the `lower` bound arrays. The size of the `lower` bound array determines the problem dimension.

The prefix is set to `"ROSEN"` to distinguish the results from the one-dimensional case. Again, TensorBoard can be used to monitor the progress of the optimization.

```
fun_control = fun_control_init(
              PREFIX="ROSEN",
              lower = np.array([-5, -5]),
              upper = np.array([10, 10]),
              show_progress=True)
surrogate_control = surrogate_control_init(isotropic=True)
spot_rosen = Spot(fun=fun_rosen,
                  fun_control=fun_control,
```

```
                    surrogate_control=surrogate_control)
spot_rosen.run()
```

```
spotpython tuning: 53.73873931238049 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 49.866300631713166 [########--] 80.00%. Success rate: 100.00%
spotpython tuning: 49.866300631713166 [#########-] 86.67%. Success rate: 66.67%
spotpython tuning: 12.621648163047155 [#########-] 93.33%. Success rate: 75.00%
spotpython tuning: 12.621648163047155 [##########] 100.00%. Success rate: 60.00% Done...

Experiment saved to ROSEN_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x134762e90>
```

> **Note**
>
> Now we can start TensorBoard in the background with the following command:
>
> ```
> tensorboard --logdir="./runs"
> ```
>
> and can access the TensorBoard web server with the following URL:
>
> ```
> http://localhost:6006/
> ```

14.4.1.1.1. Results

```
_ = spot_rosen.print_results()
```

```
min y: 12.621648163047155
x0: -0.5398890007511357
x1: -0.7209619653808784
```

```
spot_rosen.plot_progress()
```

14.4.1.1.2. A Contour Plot

We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows.

```
min_z = None
max_z = None
spot_rosen.plot_contour(i=0, j=1, min_z=min_z, max_z=max_z)
```

- The variable importance cannot be calculated, because only one `theta` value was used.

14.4.1.1.3. TensorBoard

TBD

14.4.1.2. The Two Dimensional `fun_rosen`: The Anisotropic Case

```
import numpy as np
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, surrogate_control_init
from spotpython.spot import Spot
```

The `spotpython` package provides several classes of objective functions. We will use the `fun_rosen` in the `analytical` class [SOURCE].

```
fun_rosen = Analytical().fun_rosen
```

Here we will use problem dimension $k = 2$, which can be specified by the `lower` bound arrays. The size of the `lower` bound array determines the problem dimension. The default anisotropic kernel is used, so no `isotropic` parameter is set.

We can also add interpretable labels to the dimensions, which will be used in the plots.

```
fun_control = fun_control_init(
              PREFIX="ROSEN",
              lower = np.array([-5, -5]),
              upper = np.array([10, 10]),
              show_progress=True)
spot_rosen = Spot(fun=fun_rosen,
                  fun_control=fun_control)
spot_rosen.run()
```

```
spotpython tuning: 90.77788110350957 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 1.0173064877521703 [########--] 80.00%. Success rate: 100.00%
spotpython tuning: 1.0173064877521703 [#########-] 86.67%. Success rate: 66.67%
spotpython tuning: 1.0173064877521703 [#########-] 93.33%. Success rate: 50.00%
spotpython tuning: 0.7867277329036355 [##########] 100.00%. Success rate: 60.00% Done

Experiment saved to ROSEN_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x13225cf50>
```

> **Note**
>
> Now we can start TensorBoard in the background with the following command:
>
> ```
> tensorboard --logdir="./runs"
> ```
>
> and can access the TensorBoard web server with the following URL:
>
> ```
> http://localhost:6006/
> ```

14.4.1.2.1. Results

```
_ = spot_rosen.print_results()
```

```
min y: 0.7867277329036355
x0: 0.13953655588708272
x1: 0.08753688433642533
```

```
spot_rosen.plot_progress()
```

14.4.1.2.2. A Contour Plot

We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows.

```
min_z = None
max_z = None
spot_rosen.plot_contour(i=0, j=1, min_z=min_z, max_z=max_z)
```

- The variable importance can be calculated as follows:

```
_ = spot_rosen.print_importance()
```

```
x0:  100.0
x1:  2.8374127215640144
```

```
spot_rosen.plot_importance()
```

14.4.1.2.3. TensorBoard

TBD

14.5. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

15. Sequential Parameter Optimization: Using `scipy` Optimizers

As a default optimizer, `spotpython` uses `differential_evolution` from the `scipy.optimize` package. Alternatively, any other optimizer from the `scipy.optimize` package can be used. This chapter describes how different optimizers from the `scipy optimize` package can be used on the surrogate. The optimization algorithms are available from https://docs.scipy.org/doc/scipy/reference/optimize.html

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from scipy.optimize import shgo
from scipy.optimize import direct
from scipy.optimize import differential_evolution
from scipy.optimize import dual_annealing
from scipy.optimize import basinhopping
from spotpython.utils.init import fun_control_init, design_control_init, optimizer_control_init
```

15.1. The Objective Function Branin

The `spotpython` package provides several classes of objective functions. We will use an analytical objective function, i.e., a function that can be described by a (closed) formula. Here we will use the Branin function. The 2-dim Branin function is

$$y = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1-t)\cos(x_1) + s,$$

where values of a, b, c, r, s and t are: $a = 1$, $b = 5.1/(4\pi^2)$, $c = 5/\pi$, $r = 6$, $s = 10$ and $t = 1/(8\pi)$.

It has three global minima: $f(x) = 0.397887$ at $(-\pi, 12.275)$, $(\pi, 2.275)$, and $(9.42478, 2.475)$.

Input Domain: This function is usually evaluated on the square $x_1 \in [-5, 10] \times x_2 \in [0, 15]$.

```
from spotpython.fun.objectivefunctions import Analytical
lower = np.array([-5,-0])
upper = np.array([10,15])
fun = Analytical(seed=123).fun_branin
```

15.2. The Optimizer

Differential Evolution (DE) from the `scikit.optimize` package, see https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html#scipy.optimize.differential_evolution is the default optimizer for the search on the surrogate. Other optimiers that are available in `spotpython`, see https://docs.scipy.org/doc/scipy/reference/optimize.html#global-optimization.

- `dual_annealing`
- `direct`
- `shgo`
- `basinhopping`

These optimizers can be selected as follows:

```
from scipy.optimize import differential_evolution
optimizer = differential_evolution
```

As noted above, we will use `differential_evolution`. The optimizer can use `1000` evaluations. This value will be passed to the `differential_evolution` method, which has the argument `maxiter` (int). It defines the maximum number of generations over which the entire differential evolution population is evolved, see https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html#scipy.optimize.differential_evolution

TensorBoard

Similar to the one-dimensional case, which is discussed in Section 12.25, we can use TensorBoard to monitor the progress of the optimization. We will use a similar code, only the prefix is different:

```
fun_control=fun_control_init(
                    lower = lower,
                    upper = upper,
                    fun_evals = 20,
                    PREFIX = "04_DE_"
                    )
```

```
spot_de = Spot(fun=fun,
                    fun_control=fun_control)
spot_de.run()
```

```
spotpython tuning: 3.8004525569254524 [######----] 55.00%. Success rate: 100.00%
spotpython tuning: 3.8004525569254524 [######----] 60.00%. Success rate: 50.00%
spotpython tuning: 3.1644860748767325 [######----] 65.00%. Success rate: 66.67%
spotpython tuning: 3.1405013986990866 [#######---] 70.00%. Success rate: 75.00%
spotpython tuning: 3.0703215884465367 [########--] 75.00%. Success rate: 80.00%
spotpython tuning: 2.934924209020851 [########--] 80.00%. Success rate: 83.33%
spotpython tuning: 0.4523710119285411 [########--] 85.00%. Success rate: 85.71%
spotpython tuning: 0.4335054313511062 [#########-] 90.00%. Success rate: 87.50%
spotpython tuning: 0.39841709590731966 [##########] 95.00%. Success rate: 88.89%
spotpython tuning: 0.39796888979727996 [##########] 100.00%. Success rate: 90.00% Done...

Experiment saved to 04_DE__res.pkl
```

```
<spotpython.spot.spot.Spot at 0x1066bc050>
```

15.2.1. TensorBoard

If the `prefix` argument in `fun_control_init()`is not `None` (as above, where the `prefix` was set to `04_DE_`) , we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The TensorBoard plot illustrates how `spotpython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

Figure 15.1.: TensorBoard visualization of the spotpython optimization process and the surrogate model.

15.3. Print the Results

```
spot_de.print_results()
```

```
min y: 0.39796888979727996
x0: 3.140132672900477
x1: 2.26769501627711
```

```
[['x0', np.float64(3.140132672900477)], ['x1', np.float64(2.26769501627711)]]
```

15.4. Show the Progress

```
spot_de.plot_progress(log_y=True)
```

```
spot_de.surrogate.plot()
```

15.5. Exercises

15.5.1. dual_annealing

- Describe the optimization algorithm, see scipy.optimize.dual_annealing.
- Use the algorithm as an optimizer on the surrogate.

> **Tip: Selecting the Optimizer for the Surrogate**
>
> We can run spotpython with the `dual_annealing` optimizer as follows:

```
spot_da = Spot(fun=fun,
                    fun_control=fun_control,
                    optimizer=dual_annealing)
spot_da.run()
spot_da.print_results()
spot_da.plot_progress(log_y=True)
spot_da.surrogate.plot()
```

```
spotpython tuning: 3.800454182091854 [######----
] 55.00%. Success rate: 100.00%
spotpython tuning: 3.800454182091854 [######----
] 60.00%. Success rate: 50.00%
spotpython tuning: 3.159404256311033 [######----
] 65.00%. Success rate: 66.67%
spotpython tuning: 3.1344112546586977 [#######---
] 70.00%. Success rate: 75.00%
spotpython tuning: 3.0492833293320416 [########--
] 75.00%. Success rate: 80.00%
spotpython tuning: 2.9070231530180237 [########--
] 80.00%. Success rate: 83.33%
spotpython tuning: 0.44694718755548735 [########--
] 85.00%. Success rate: 85.71%
spotpython tuning: 0.4275929669767766 [#########-
] 90.00%. Success rate: 87.50%
spotpython tuning: 0.3999639038471443 [##########] 95.00%. Success rate: 88.89%
spotpython tuning: 0.3981150855246529 [##########] 100.00%. Success rate: 90.00% Done...

Experiment saved to 04_DE__res.pkl
min y: 0.3981150855246529
x0: 3.135596675425474
x1: 2.2722562497899212
```

15.5.2. `direct`

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

Tip: Selecting the Optimizer for the Surrogate

We can run spotpython with the `direct` optimizer as follows:

```
spot_di = Spot(fun=fun,
                    fun_control=fun_control,
                    optimizer=direct)
spot_di.run()
spot_di.print_results()
spot_di.plot_progress(log_y=True)
spot_di.surrogate.plot()
```

```
spotpython tuning: 3.808603529901438 [######----
] 55.00%. Success rate: 100.00%
spotpython tuning: 3.808603529901438 [######----
] 60.00%. Success rate: 50.00%
spotpython tuning: 3.19804562480188 [######----
] 65.00%. Success rate: 66.67%
spotpython tuning: 3.17767194117126 [#######---
] 70.00%. Success rate: 75.00%
spotpython tuning: 3.165751373773567 [########--
] 75.00%. Success rate: 80.00%
spotpython tuning: 3.133265047041581 [########--
] 80.00%. Success rate: 83.33%
spotpython tuning: 3.076588713032807 [########--
] 85.00%. Success rate: 85.71%
spotpython tuning: 0.5028918360432506 [#########-
] 90.00%. Success rate: 87.50%
spotpython tuning: 0.45130981703175976 [##########] 95.00%. Success rate: 88.89%
spotpython tuning: 0.3989943920885928 [##########] 100.00%. Success rate: 90.00% Done...

Experiment saved to 04_DE__res.pkl
min y: 0.3989943920885928
x0: 3.1264288980338364
x1: 2.28509373571102
```

15.5.3. shgo

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

Tip: Selecting the Optimizer for the Surrogate

We can run spotpython with the `direct` optimizer as follows:

```
spot_sh = Spot(fun=fun,
                    fun_control=fun_control,
                    optimizer=shgo)
spot_sh.run()
spot_sh.print_results()
spot_sh.plot_progress(log_y=True)
spot_sh.surrogate.plot()
```

```
spotpython tuning: 3.8004550272025437 [######----
] 55.00%. Success rate: 100.00%
spotpython tuning: 3.8004550272025437 [######----
] 60.00%. Success rate: 50.00%
spotpython tuning: 3.1593634068529015 [######----
] 65.00%. Success rate: 66.67%
spotpython tuning: 3.1343990956358345 [#######---
] 70.00%. Success rate: 75.00%
spotpython tuning: 3.0512879835062634 [########--
] 75.00%. Success rate: 80.00%
spotpython tuning: 2.904985437740331 [########--
] 80.00%. Success rate: 83.33%
spotpython tuning: 0.4470562902385673 [########--
] 85.00%. Success rate: 85.71%
spotpython tuning: 0.42244233288071875 [#########-
] 90.00%. Success rate: 87.50%
spotpython tuning: 0.3999188404619609 [##########] 95.00%. Success rate: 88.89%
spotpython tuning: 0.3981002885577922 [##########] 100.00%. Success rate: 90.00% Done...

Experiment saved to 04_DE__res.pkl
min y: 0.3981002885577922
x0: 3.1357673421870387
x1: 2.272475458942683
```

15.5.4. basinhopping

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

Tip: Selecting the Optimizer for the Surrogate

We can run spotpython with the `direct` optimizer as follows:

```
spot_bh = Spot(fun=fun,
                    fun_control=fun_control,
                    optimizer=basinhopping)
spot_bh.run()
spot_bh.print_results()
spot_bh.plot_progress(log_y=True)
spot_bh.surrogate.plot()
```

```
spotpython tuning: 3.8004406841429414 [######----
] 55.00%. Success rate: 100.00%
spotpython tuning: 3.8004406841429414 [######----
] 60.00%. Success rate: 50.00%
spotpython tuning: 3.1591903142692317 [######----
] 65.00%. Success rate: 66.67%
spotpython tuning: 3.128117693244988 [#######---
] 70.00%. Success rate: 75.00%
spotpython tuning: 3.0449391022276853 [########--
] 75.00%. Success rate: 80.00%
spotpython tuning: 2.9085008044527525 [########--
] 80.00%. Success rate: 83.33%
spotpython tuning: 0.44636813870454084 [########--
] 85.00%. Success rate: 85.71%
spotpython tuning: 0.4203418629420792 [#########-
] 90.00%. Success rate: 87.50%
spotpython tuning: 0.3999877661836013 [##########] 95.00%. Success rate: 88.89%
spotpython tuning: 0.39825689415986787 [##########] 100.00%. Success rate: 90.00% Done...

Experiment saved to 04_DE__res.pkl
min y: 0.39825689415986787
x0: 3.133951975326067
x1: 2.271518851806762
```

15.5.5. Performance Comparison

Compare the performance and run time of the 5 different optimizers:

- `differential_evolution`
- `dual_annealing`
- `direct`
- `shgo`
- `basinhopping`.

The Branin function has three global minima:

- $f(x) = 0.397887$ at
  - $(-\pi, 12.275)$,
  - $(\pi, 2.275)$, and
  - $(9.42478, 2.475)$.

- Which optima are found by the optimizers?
- Does the `seed` argument in `fun = Analytical(seed=123).fun_branin` change this behavior?

15.6. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this chapter is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

16. Using sklearn Surrogates in spotpython

Besides the internal kriging surrogate, which is used as a default by spotpython, any surrogate model from scikit-learn can be used as a surrogate in spotpython. This chapter explains how to use scikit-learn surrogates in spotpython.

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
```

16.1. Example: Branin Function with spotpython's Internal Kriging Surrogate

16.1.1. The Objective Function Branin

- The spotpython package provides several classes of objective functions.
- We will use an analytical objective function, i.e., a function that can be described by a (closed) formula.
- Here we will use the Branin function:

  ```
  y = a * (x2 - b * x1**2 + c * x1 - r) ** 2 + s * (1 - t) * np.cos(x1) + s,
  where values of a, b, c, r, s and t are: a = 1, b = 5.1 / (4*pi**2),
  c = 5 / pi, r = 6, s = 10 and t = 1 / (8*pi).
  ```

- It has three global minima:

  ```
  f(x) = 0.397887 at (-pi, 12.275), (pi, 2.275), and (9.42478, 2.475).
  ```

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical().fun_branin
```

TensorBoard

Similar to the one-dimensional case, which was introduced in Section Section 12.25, we can use TensorBoard to monitor the progress of the optimization. We will use the same code, only the prefix is different:

```
from spotpython.utils.init import fun_control_init, design_control_init
PREFIX = "04"
fun_control = fun_control_init(
    PREFIX=PREFIX,
    lower = np.array([-5,-0]),
    upper = np.array([10,15]),
    fun_evals=20,
    max_time=inf)

design_control = design_control_init(
    init_size=10)
```

16.1.2. Running the surrogate model based optimizer Spot:

```
spot_2 = Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
```

```
spot_2.run()
```

```
spotpython tuning: 3.8004525569254524 [######----] 55.00%. Success rate: 100.00%
spotpython tuning: 3.8004525569254524 [######----] 60.00%. Success rate: 50.00%
spotpython tuning: 3.1644860748767325 [######----] 65.00%. Success rate: 66.67%
spotpython tuning: 3.1405013986990866 [#######---] 70.00%. Success rate: 75.00%
spotpython tuning: 3.0703215884465367 [########--] 75.00%. Success rate: 80.00%
spotpython tuning: 2.934924209020851 [########--] 80.00%. Success rate: 83.33%
spotpython tuning: 0.4523710119285411 [########--] 85.00%. Success rate: 85.71%
spotpython tuning: 0.4335054313511062 [#########-] 90.00%. Success rate: 87.50%
spotpython tuning: 0.39841709590731966 [##########] 95.00%. Success rate: 88.89%
spotpython tuning: 0.39796888979727996 [##########] 100.00%. Success rate: 90.00% Done

Experiment saved to 04_res.pkl


<spotpython.spot.spot.Spot at 0x106ac8050>
```

16.1.3. TensorBoard

Now we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The TensorBoard plot illustrates how `spotpython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

Figure 16.1.: TensorBoard visualization of the spotpython optimization process and the surrogate model.

16.1.4. Print the Results

```
spot_2.print_results()
```

```
min y: 0.39796888979727996
x0: 3.140132672900477
x1: 2.26769501627711
```

```
[['x0', np.float64(3.140132672900477)], ['x1', np.float64(2.26769501627711)]]
```

16.1.5. Show the Progress and the Surrogate

```
spot_2.plot_progress(log_y=True)
```

```
spot_2.surrogate.plot()
```

16.2. Example: Using Surrogates From scikit-learn

- Default is the `spotpython` (i.e., the internal) `kriging` surrogate.
- It can be called explicitely and passed to `Spot`.

```
from spotpython.surrogate.kriging import Kriging
S_0 = Kriging(name='kriging', seed=123)
```

- Alternatively, models from `scikit-learn` can be selected, e.g., Gaussian Process, RBFs, Regression Trees, etc.

```
# Needed for the sklearn surrogates:
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeRegressor
```

```
from sklearn.ensemble import RandomForestRegressor
from sklearn import linear_model
from sklearn import tree
import pandas as pd
```

- Here are some additional models that might be useful later:

```
S_Tree = DecisionTreeRegressor(random_state=0)
S_LM = linear_model.LinearRegression()
S_Ridge = linear_model.Ridge()
S_RF = RandomForestRegressor(max_depth=2, random_state=0)
```

16.2.1. GaussianProcessRegressor as a Surrogate

- To use a Gaussian Process model from `sklearn`, that is similar to `spotpython`'s `Kriging`, we can proceed as follows:

```
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
S_GP = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
```

- The scikit-learn GP model `S_GP` is selected for `Spot` as follows:

  `surrogate = S_GP`

- We can check the kind of surogate model with the command `isinstance`:

```
isinstance(S_GP, GaussianProcessRegressor)
```

```
True
```

```
isinstance(S_0, Kriging)
```

```
True
```

- Similar to the `Spot` run with the internal `Kriging` model, we can call the run with the `scikit-learn` surrogate:

```
fun = Analytical(seed=123).fun_branin
spot_2_GP = Spot(fun=fun,
                     fun_control=fun_control,
                     design_control=design_control,
                     surrogate = S_GP)
spot_2_GP.run()
```

```
spotpython tuning: 18.865094417928255 [######----] 55.00%. Success rate: 100.00%
spotpython tuning: 4.066951764659748 [######----] 60.00%. Success rate: 100.00%
spotpython tuning: 3.4618268978622053 [######----] 65.00%. Success rate: 100.00%
spotpython tuning: 3.4618268978622053 [#######---] 70.00%. Success rate: 75.00%
spotpython tuning: 1.328427129373333 [########--] 75.00%. Success rate: 80.00%
spotpython tuning: 0.9549984699881495 [########--] 80.00%. Success rate: 83.33%
spotpython tuning: 0.9351546756190547 [########--] 85.00%. Success rate: 85.71%
spotpython tuning: 0.3997498980904961 [#########-] 90.00%. Success rate: 87.50%
spotpython tuning: 0.398303602593181 [##########] 95.00%. Success rate: 88.89%


spotpython tuning: 0.3982354139382931 [##########] 100.00%. Success rate: 90.00% Done...

Experiment saved to 04_res.pkl


<spotpython.spot.spot.Spot at 0x15fefec10>
```

```
spot_2_GP.plot_progress()
```

```
spot_2_GP.print_results()
```

```
min y: 0.3982354139382931
x0: 3.149733645254801
x1: 2.274124565582247


[['x0', np.float64(3.149733645254801)], ['x1', np.float64(2.274124565582247)]]
```

16.3. Example: One-dimensional Sphere Function With spotpython's Kriging

- In this example, we will use an one-dimensional function, which allows us to visualize the optimization process.
  - show_models= True is added to the argument list.

```
from spotpython.fun.objectivefunctions import Analytical
fun_control = fun_control_init(
    lower = np.array([-1]),
    upper = np.array([1]),
    fun_evals=10,
    max_time=inf,
    show_models= True,
    tolerance_x = np.sqrt(np.spacing(1)))
fun = Analytical(seed=123).fun_sphere
design_control = design_control_init(
    init_size=3)
```

```
spot_1 = Spot(fun=fun,
                    fun_control=fun_control,
                    design_control=design_control)
spot_1.run()
```

```
Using spacefilling design as fallback.
```

```
spotpython tuning: 0.03475493366922229 [####------] 40.00%. Success rate: 0.00%
```

```
spotpython tuning: 0.03475493366922229 [#####-----] 50.00%. Success rate: 0.00%
```

```
spotpython tuning: 0.0002493578103496971 [######----] 60.00%. Success rate: 33.33%
```

```
spotpython tuning: 3.737005778245563e-07 [#######---] 70.00%. Success rate: 50.00%
```

```
spotpython tuning: 9.313071592223354e-08 [########--] 80.00%. Success rate: 60.00%
```

```
spotpython tuning: 2.9997585469544003e-11 [#########-] 90.00%. Success rate: 66.67%
```

```
spotpython tuning: 2.9997585469544003e-11 [##########] 100.00%. Success rate: 57.14% I

Experiment saved to 000_res.pkl
```

16.3.1. Results

```
spot_1.print_results()
```

```
min y: 2.9997585469544003e-11
x0: -5.477005155150395e-06
```

```
[['x0', np.float64(-5.477005155150395e-06)]]
```

```
spot_1.plot_progress(log_y=True)
```

- The method `plot_model` plots the final surrogate:

```
spot_1.plot_model()
```

16.4. Example: Sklearn Model GaussianProcess

- This example visualizes the search process on the `GaussianProcessRegression` surrogate from `sklearn`.

- Therefore `surrogate = S_GP` is added to the argument list.

```
fun = Analytical(seed=123).fun_sphere
spot_1_GP = Spot(fun=fun,
                      fun_control=fun_control,
                      design_control=design_control,
                      surrogate = S_GP)
spot_1_GP.run()
```

```
spotpython tuning: 0.004925671398319439 [####------] 40.00%. Success rate: 100.00%
```

```
spotpython tuning: 0.00261206235804241 [#####-----] 50.00%. Success rate: 100.00%
```

```
spotpython tuning: 6.30724685184118e-07 [######----] 60.00%. Success rate: 100.00%
```

```
spotpython tuning: 8.015822937486393e-08 [#######---] 70.00%. Success rate: 100.00%
```

```
spotpython tuning: 8.015822937486393e-08 [########--] 80.00%. Success rate: 80.00%
```

```
spotpython tuning: 6.369177036405373e-09 [#########-] 90.00%. Success rate: 83.33%
```

```
spotpython tuning: 2.0638361689231364e-09 [##########] 100.00%. Success rate: 85.71% Done...

Experiment saved to 000_res.pkl
```

```
spot_1_GP.print_results()
```

```
min y: 2.0638361689231364e-09
x0: -4.5429463665369596e-05
```

```
[['x0', np.float64(-4.5429463665369596e-05)]]
```

```
spot_1_GP.plot_progress(log_y=True)
```

```
spot_1_GP.plot_model()
```

16.5. Exercises

16.5.1. 1. A decision tree regressor: DecisionTreeRegressor

- Describe the surrogate model. Use the information from the scikit-learn documentation.

- Use the surrogate as the model for optimization.

16.5.2. 2. A random forest regressor: `RandomForestRegressor`

- Describe the surrogate model. Use the information from the scikit-learn documentation.
- Use the surrogate as the model for optimization.

16.5.3. 3. Ordinary least squares Linear Regression: `LinearRegression`

- Describe the surrogate model. Use the information from the scikit-learn documentation.
- Use the surrogate as the model for optimization.

16.5.4. 4. Linear least squares with l2 regularization: `Ridge`

- Describe the surrogate model. Use the information from the scikit-learn documentation.
- Use the surrogate as the model for optimization.

16.5.5. 5. Gradient Boosting: `HistGradientBoostingRegressor`

- Describe the surrogate model. Use the information from the scikit-learn documentation.
- Use the surrogate as the model for optimization.

16.5.6. 6. Comparison of Surrogates

- Use the following two objective functions

  1. the 1-dim sphere function `fun_sphere` and
  2. the two-dim Branin function `fun_branin`:

  for a comparison of the performance of the five different surrogates:

  - `spotpython`'s internal Kriging
  - `DecisionTreeRegressor`
  - `RandomForestRegressor`
  - `linear_model.LinearRegression`
  - `linear_model.Ridge`.

- Generate a table with the results (number of function evaluations, best function value, and best parameter vector) for each surrogate and each function as shown in Table 16.1.

Table 16.1.: Result table

surrogate	fun	fun_evals	max_time	x_0	min_y	Comments
Kriging	fun_sphere	10	inf			
Kriging	fun_branin	10	inf			
DecisionTreeRegressor	fun_sphere	10	inf			
…	…	…	…			
Ridge	fun_branin	10	inf			

- Discuss the results. Which surrogate is the best for which function? Why?

16.6. Selected Solutions

16.6.1. Solution to Exercise Section 16.5.5: Gradient Boosting

16.6.1.1. Branin: Using SPOT

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, design_control_init
from spotpython.spot import Spot
```

- The Objective Function Branin

```
fun = Analytical().fun_branin
PREFIX = "BRANIN"
fun_control = fun_control_init(
    PREFIX=PREFIX,
    lower = np.array([-5,-0]),
    upper = np.array([10,15]),
    fun_evals=20,
    max_time=inf)

design_control = design_control_init(
    init_size=10)
```

- Running the surrogate model based optimizer `Spot`:

```
spot_2 = Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
spot_2.run()
```

```
spotpython tuning: 3.8004525569254524 [######----] 55.00%. Success rate: 100.00%
spotpython tuning: 3.8004525569254524 [######----] 60.00%. Success rate: 50.00%
spotpython tuning: 3.1644860748767325 [######----] 65.00%. Success rate: 66.67%
spotpython tuning: 3.1405013986990866 [#######---] 70.00%. Success rate: 75.00%
spotpython tuning: 3.0703215884465367 [########--] 75.00%. Success rate: 80.00%
spotpython tuning: 2.934924209020851 [########--] 80.00%. Success rate: 83.33%
spotpython tuning: 0.4523710119285411 [########--] 85.00%. Success rate: 85.71%
spotpython tuning: 0.4335054313511062 [#########-] 90.00%. Success rate: 87.50%
spotpython tuning: 0.39841709590731966 [##########] 95.00%. Success rate: 88.89%
spotpython tuning: 0.39796888979727996 [##########] 100.00%. Success rate: 90.00% Done...

Experiment saved to BRANIN_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x16035e5d0>
```

- Print the results

```
spot_2.print_results()
```

```
min y: 0.39796888979727996
x0: 3.140132672900477
x1: 2.26769501627711
```

```
[['x0', np.float64(3.140132672900477)], ['x1', np.float64(2.26769501627711)]]
```

- Show the optimization progress:

```
spot_2.plot_progress(log_y=True)
```

- Generate a surrogate model plot:

```
spot_2.surrogate.plot()
```

16.6.1.2. Branin: Using Surrogates From scikit-learn

- The `HistGradientBoostingRegressor` model from `scikit-learn` is selected:

```
# Needed for the sklearn surrogates:
from sklearn.ensemble import HistGradientBoostingRegressor
import pandas as pd
S_XGB = HistGradientBoostingRegressor()
```

- The scikit-learn XGB model `S_XGB` is selected for `Spot` as follows: `surrogate = S_XGB`.
- Similar to the `Spot` run with the internal `Kriging` model, we can call the run with the `scikit-learn` surrogate:

```
fun = Analytical(seed=123).fun_branin
spot_2_XGB = Spot(fun=fun,
                     fun_control=fun_control,
                     design_control=design_control,
                     surrogate = S_XGB)
spot_2_XGB.run()
```

```
spotpython tuning: 30.69410528614059 [######----] 55.00%. Success rate: 0.00%
Using spacefilling design as fallback.
spotpython tuning: 30.69410528614059 [######----] 60.00%. Success rate: 0.00%
Using spacefilling design as fallback.
spotpython tuning: 30.69410528614059 [######----] 65.00%. Success rate: 0.00%
Using spacefilling design as fallback.
spotpython tuning: 30.69410528614059 [#######---] 70.00%. Success rate: 0.00%
Using spacefilling design as fallback.
spotpython tuning: 1.3263745845108854 [########--] 75.00%. Success rate: 20.00%
Using spacefilling design as fallback.
spotpython tuning: 1.3263745845108854 [########--] 80.00%. Success rate: 16.67%
Using spacefilling design as fallback.
spotpython tuning: 1.3263745845108854 [########--] 85.00%. Success rate: 14.29%
Using spacefilling design as fallback.
spotpython tuning: 1.3263745845108854 [#########-] 90.00%. Success rate: 12.50%
Using spacefilling design as fallback.
spotpython tuning: 1.3263745845108854 [##########] 95.00%. Success rate: 11.11%
Using spacefilling design as fallback.
spotpython tuning: 1.3263745845108854 [##########] 100.00%. Success rate: 10.00% Done...

Experiment saved to BRANIN_res.pkl

<spotpython.spot.spot.Spot at 0x15ed33250>
```

- Print the Results

```
spot_2_XGB.print_results()
```

```
min y: 1.3263745845108854
x0: -2.872730773493426
x1: 10.874313833535739

[['x0', np.float64(-2.872730773493426)],
 ['x1', np.float64(10.874313833535739)]]
```

- Show the Progress

```
spot_2_XGB.plot_progress(log_y=True)
```

- Since the `sklearn` model does not provide a `plot` method, we cannot generate a surrogate model plot.

16.6.1.3. One-dimensional Sphere Function With `spotpython`'s Kriging

- In this example, we will use an one-dimensional function, which allows us to visualize the optimization process.
  - `show_models= True` is added to the argument list.

```
from spotpython.fun.objectivefunctions import Analytical
fun_control = fun_control_init(
    lower = np.array([-1]),
    upper = np.array([1]),
    fun_evals=10,
```

```
    max_time=inf,
    show_models= True,
    tolerance_x = np.sqrt(np.spacing(1)))
fun = Analytical(seed=123).fun_sphere
design_control = design_control_init(
    init_size=3)
```

```
spot_1 = Spot(fun=fun,
                    fun_control=fun_control,
                    design_control=design_control)
spot_1.run()
```

```
Using spacefilling design as fallback.
```

```
spotpython tuning: 0.03475493366922229 [####------] 40.00%. Success rate: 0.00%
```

```
spotpython tuning: 0.03475493366922229 [#####-----] 50.00%. Success rate: 0.00%
```

```
spotpython tuning: 0.0002493578103496971 [######----] 60.00%. Success rate: 33.33%
```

```
spotpython tuning: 3.737005778245563e-07 [#######---] 70.00%. Success rate: 50.00%
```

```
spotpython tuning: 9.313071592223354e-08 [########--] 80.00%. Success rate: 60.00%
```

```
spotpython tuning: 2.9997585469544003e-11 [#########-] 90.00%. Success rate: 66.67%
```

```
spotpython tuning: 2.9997585469544003e-11 [##########] 100.00%. Success rate: 57.14% I

Experiment saved to 000_res.pkl
```

- Print the Results

```
spot_1.print_results()
```

```
min y: 2.9997585469544003e-11
x0: -5.477005155150395e-06
```

```
[['x0', np.float64(-5.477005155150395e-06)]]
```

- Show the Progress

```
spot_1.plot_progress(log_y=True)
```

- The method `plot_model` plots the final surrogate:

```
spot_1.plot_model()
```

16.6.1.4. One-dimensional Sphere Function With `Sklearn` Model HistGradientBoostingRegressor

- This example visualizes the search process on the `HistGradientBoostingRegressor` surrogate from `sklearn`.

- Therefore `surrogate = S_XGB` is added to the argument list.

```
fun_control = fun_control_init(
    lower = np.array([-1]),
    upper = np.array([1]),
    fun_evals=10,
    max_time=inf,
    show_models= True,
    tolerance_x = np.sqrt(np.spacing(1)))
fun = Analytical(seed=123).fun_sphere
design_control = design_control_init(
    init_size=3)
spot_1_XGB = Spot(fun=fun,
                      fun_control=fun_control,
                      design_control=design_control,
                      surrogate = S_XGB)
spot_1_XGB.run()
```

```
spotpython tuning: 0.03475493366922229 [####------] 40.00%. Success rate: 0.00%
Using spacefilling design as fallback.
```

```
spotpython tuning: 0.03475493366922229 [#####-----] 50.00%. Success rate: 0.00%
Using spacefilling design as fallback.
```

```
spotpython tuning: 0.03475493366922229 [######----] 60.00%. Success rate: 0.00%
Using spacefilling design as fallback.
```

```
spotpython tuning: 0.03475493366922229 [#######---] 70.00%. Success rate: 0.00%
Using spacefilling design as fallback.
```

```
spotpython tuning: 0.008730885505764131 [########--] 80.00%. Success rate: 20.00%
Using spacefilling design as fallback.
```

```
spotpython tuning: 0.008730885505764131 [#########-] 90.00%. Success rate: 16.67%
Using spacefilling design as fallback.
```

```
spotpython tuning: 0.008730885505764131 [##########] 100.00%. Success rate: 14.29% Done...

Experiment saved to 000_res.pkl
```

```
spot_1_XGB.print_results()
```

```
min y: 0.008730885505764131
x0: 0.09343920754032609
```

```
[['x0', np.float64(0.09343920754032609)]]
```

```
spot_1_XGB.plot_progress(log_y=True)
```

```
spot_1_XGB.plot_model()
```

16.7. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

17. Sequential Parameter Optimization: Gaussian Process Models

This chapter analyzes differences between the `Kriging` implementation in `spotpython` and the `GaussianProcessRegressor` in `scikit-learn`.

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.design.spacefilling import SpaceFilling
from spotpython.spot import Spot
from spotpython.surrogate.kriging import Kriging
from scipy.optimize import shgo
from scipy.optimize import direct
from scipy.optimize import differential_evolution
import matplotlib.pyplot as plt
import math as m
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
```

17.1. Gaussian Processes Regression: Basic Introductory `scikit-learn` Example

- This is the example from scikit-learn: https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_noisy_targets.html
- After fitting our model, we see that the hyperparameters of the kernel have been optimized.
- Now, we will use our kernel to compute the mean prediction of the full dataset and plot the 95% confidence interval.

17.1.1. Train and Test Data

```
X = np.linspace(start=0, stop=10, num=1_000).reshape(-1, 1)
y = np.squeeze(X * np.sin(X))
rng = np.random.RandomState(1)
training_indices = rng.choice(np.arange(y.size), size=6, replace=False)
X_train, y_train = X[training_indices], y[training_indices]
```

17.1.2. Building the Surrogate With `Sklearn`

- The model building with `sklearn` consisits of three steps:
  1. Instantiating the model, then
  2. fitting the model (using `fit`), and
  3. making predictions (using `predict`)

```
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
gaussian_process = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
gaussian_process.fit(X_train, y_train)
mean_prediction, std_prediction = gaussian_process.predict(X, return_std=True)
```

17.1.3. Plotting the `Sklearn`Model

```
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
    mean_prediction - 1.96 * std_prediction,
    mean_prediction + 1.96 * std_prediction,
    alpha=0.5,
    label=r"95% confidence interval",
)
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("sk-learn Version: Gaussian process regression on noise-free dataset")
```

17.1.4. The spotpython Version

- The spotpython version is very similar:
  1. Instantiating the model, then
  2. fitting the model and
  3. making predictions (using predict).

```
S = Kriging(name='kriging',  seed=123, log_level=50, cod_type="norm")
S.fit(X_train, y_train)
S_mean_prediction, S_std_prediction, S_ei = S.predict(X, return_val="all")
```

```
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, S_mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
    S_mean_prediction - 1.96 * S_std_prediction,
    S_mean_prediction + 1.96 * S_std_prediction,
    alpha=0.5,
    label=r"95% confidence interval",
)
plt.legend()
plt.xlabel("$x$")
```

```
plt.ylabel("$f(x)$")
_ = plt.title("spotpython Version: Gaussian process regression on noise-free dataset")
```

17.1.5. Visualizing the Differences Between the `spotpython` and the `sklearn` Model Fits

```
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, S_mean_prediction, label="spotpython Mean prediction")
plt.plot(X, mean_prediction, label="Sklearn Mean Prediction")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Comparing Mean Predictions")
```

17.2. Exercises

17.2.1. Schonlau Example Function

- The Schonlau Example Function is based on sample points only (there is no analytical function description available):

```
X = np.linspace(start=0, stop=13, num=1_000).reshape(-1, 1)
X_train = np.array([1., 2., 3., 4., 12.]).reshape(-1,1)
y_train = np.array([0., -1.75, -2, -0.5, 5.])
```

- Describe the function.
- Compare the two models that were build using the `spotpython` and the `sklearn` surrogate.
- Note: Since there is no analytical function available, you might be interested in adding some points and describe the effects.

17.2.2. Forrester Example Function

- The Forrester Example Function is defined as follows:

```
f(x) = (6x- 2)^2 sin(12x-4) for x in [0,1].
```

- Data points are generated as follows:

```
from spotpython.utils.init import fun_control_init
X = np.linspace(start=-0.5, stop=1.5, num=1_000).reshape(-1, 1)
X_train = np.array([0.0, 0.175, 0.225, 0.3, 0.35, 0.375, 0.5,1]).reshape(-1,1)
fun = Analytical().fun_forrester
fun_control = fun_control_init(sigma = 0.1)
y = fun(X, fun_control=fun_control)
y_train = fun(X_train, fun_control=fun_control)
```

- Describe the function.
- Compare the two models that were build using the `spotpython` and the `sklearn` surrogate.
- Note: Modify the noise level (`"sigma"`), e.g., use a value of `0.2`, and compare the two models.

```
fun_control = fun_control_init(sigma = 0.2)
```

17.2.3. `fun_runge` Function (1-dim)

- The Runge function is defined as follows:

```
f(x) = 1/ (1 + sum(x_i))^2
```

- Data points are generated as follows:

```
gen = SpaceFilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_runge
fun_control = fun_control_init(sigma = 0.025)
X_train = gen.scipy_lhd(10, lower=lower, upper = upper).reshape(-1,1)
y_train = fun(X, fun_control=fun_control)
X = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
y = fun(X, fun_control=fun_control)
```

- Describe the function.
- Compare the two models that were build using the `spotpython` and the `sklearn` surrogate.
- Note: Modify the noise level (`"sigma"`), e.g., use a value of `0.05`, and compare the two models.

```
fun_control = fun_control_init(sigma = 0.5)
```

17.2.4. fun_cubed (1-dim)

- The Cubed function is defined as follows:

  `np.sum(X[i]** 3)`

- Data points are generated as follows:

```
gen = SpaceFilling(1)
rng = np.random.RandomState(1)
fun_control = fun_control_init(sigma = 0.025,
                lower = np.array([-10]),
                upper = np.array([10]))
fun = Analytical().fun_cubed
X_train = gen.scipy_lhd(10, lower=lower, upper = upper).reshape(-1,1)
y_train = fun(X, fun_control=fun_control)
X = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
y = fun(X, fun_control=fun_control)
```

- Describe the function.
- Compare the two models that were build using the `spotpython` and the `sklearn` surrogate.
- Note: Modify the noise level (`"sigma"`), e.g., use a value of `0.05`, and compare the two models.

```
fun_control = fun_control_init(sigma = 0.025)
```

17.2.5. The Effect of Noise

How does the behavior of the `spotpython` fit changes when the argument `noise` is set to `True`, i.e.,

`S = Kriging(name='kriging',  seed=123, isotropic=true, method="regression")`

is used?

18. Infill Criteria

This chapter describes, analyzes, and compares different infill criterion. An infill criterion defines how the next point x_{n+1} is selected from the surrogate model S. Expected improvement is a popular infill criterion in Bayesian optimization.

18.1. Expected Improvement

Expected Improvement (EI) is one of the most influential and widely-used infill criteria in surrogate-based optimization, particularly in Bayesian optimization. An infill criterion defines how the next evaluation point x_{n+1} is selected from the surrogate model S, balancing the fundamental trade-off between **exploitation** (sampling where the surrogate predicts good values) and **exploration** (sampling where the surrogate is uncertain).

The concept of Expected Improvement was formalized by Jones, Schonlau, and Welch (1998) and builds upon the theoretical foundation established by Močkus (1974). It provides an elegant mathematical framework that naturally combines both exploitation and exploration in a single criterion, making it particularly well-suited for expensive black-box optimization problems.

18.1.1. The Philosophy Behind Expected Improvement

The core idea of Expected Improvement is deceptively simple yet mathematically sophisticated. Rather than simply choosing the point where the surrogate model predicts the best value (pure exploitation) or the point with the highest uncertainty (pure exploration), EI asks a more nuanced question:

> "What is the expected value of improvement over the current best observation if we evaluate the objective function at point x?"

This approach naturally balances exploitation and exploration because:

- Points near the current best solution have a reasonable chance of improvement (exploitation)
- Points in unexplored regions with high uncertainty may yield surprising improvements (exploration)

- The mathematical expectation provides a principled way to combine these considerations

18.1.2. Mathematical Definition

18.1.2.1. Setup and Notation

Consider a Gaussian Process (Kriging) surrogate model fitted to n observations $\{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$, where $y^{(i)} = f(x^{(i)})$ are the expensive function evaluations. Let $f_{best} = \min_{i=1,\dots,n} y^{(i)}$ be the best (minimum) observed value so far.

At any unobserved point x, the Gaussian Process provides:

- A predictive mean: $\hat{f}(x) = \mu(x)$
- A predictive standard deviation: $s(x) = \sigma(x)$

The GP assumes that the true function value $f(x)$ follows a normal distribution:

$$f(x) \sim \mathcal{N}(\mu(x), \sigma^2(x))$$

18.1.2.2. The Improvement Function

The **improvement** at point x is defined as:

$$I(x) = \max(f_{best} - f(x), 0)$$

This represents how much better the function value at x is compared to the current best. Note that $I(x) = 0$ if $f(x) \geq f_{best}$ (no improvement).

Definition 18.1 (Expected Improvement Formula)**.** The Expected Improvement is the expectation of the improvement function:

$$EI(x) = \mathbb{E}[I(x)] = \mathbb{E}[\max(f_{best} - f(x), 0)]$$

Since $f(x)$ is normally distributed under the GP model, this expectation has a closed-form solution:

$$EI(x) = \begin{cases} (f_{best} - \mu(x))\Phi\left(\frac{f_{best}-\mu(x)}{\sigma(x)}\right) + \sigma(x)\phi\left(\frac{f_{best}-\mu(x)}{\sigma(x)}\right) & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases}$$

where:

- $\Phi(\cdot)$ is the cumulative distribution function (CDF) of the standard normal distribution
- $\phi(\cdot)$ is the probability density function (PDF) of the standard normal distribution
- $Z = \frac{f_{best}-\mu(x)}{\sigma(x)}$ is the standardized improvement

18.1.2.3. Alternative Formulation

The Expected Improvement can also be written as:

$$EI(x) = \sigma(x)\left[Z\Phi(Z) + \phi(Z)\right]$$

where $Z = \frac{f_{best} - \mu(x)}{\sigma(x)}$ is the standardized improvement.

18.1.3. Understanding the Components

The EI formula elegantly combines two terms:

1. **Exploitation Term**: $(f_{best} - \mu(x))\Phi(Z)$
   - Larger when $\mu(x)$ is small (good predicted value)
   - Weighted by the probability that $f(x) < f_{best}$
2. **Exploration Term**: $\sigma(x)\phi(Z)$
   - Larger when $\sigma(x)$ is large (high uncertainty)
   - Represents the potential for discovering unexpectedly good values

18.2. EI: Implementation in spotpython

The spotpython package implements Expected Improvement in its Kriging class. Here's how it works in practice:

18.2.1. Key Implementation Details

1. **Negative Expected Improvement**: In optimization contexts, spotpython often returns the **negative** Expected Improvement because many optimization algorithms are designed to minimize rather than maximize objectives.
2. **Logarithmic Transformation**: To handle numerical issues and improve optimization stability, spotpython often works with $\log(EI)$:

```
ExpImp = np.log10(EITermOne + EITermTwo + self.eps)
return float(-ExpImp)  # Negative for minimization
```

3. **Numerical Stability**: A small epsilon value (`self.eps`) is added to prevent numerical issues when EI becomes very small.

18.2.2. Code Example from the Kriging Class

```
def _pred(self, x: np.ndarray) -> Tuple[float, float, float]:
    """Computes Kriging prediction including Expected Improvement."""
    # ... [prediction calculations] ...

    # Compute Expected Improvement
    if self.return_ei:
        yBest = np.min(y)  # Current best observation

        # First term: (f_best - mu) * Phi(Z)
        EITermOne = (yBest - f) * (0.5 + 0.5 * erf((1 / np.sqrt(2)) * ((yBest - f) / s

        # Second term: sigma * phi(Z)
        EITermTwo = s * (1 / np.sqrt(2 * np.pi)) * np.exp(-0.5 * ((yBest - f) ** 2 / S

        # Expected Improvement (in log scale)
        ExpImp = np.log10(EITermOne + EITermTwo + self.eps)

        return float(f), float(s), float(-ExpImp)  # Return negative EI
```

18.3. Practical Advantages of Expected Improvement

1. **Automatic Balance**: EI naturally balances exploitation and exploration without requiring manual tuning of weights or parameters.
2. **Scale Invariance**: EI is relatively invariant to the scale of the objective function.
3. **Theoretical Foundation**: EI has strong theoretical justification from decision theory and information theory.
4. **Efficient Optimization**: The smooth, differentiable nature of EI makes it suitable for gradient-based optimization of the acquisition function.
5. **Proven Performance**: EI has been successfully applied across numerous domains and consistently performs well in practice.

18.4. Connection to the Hyperparameter Tuning Cookbook

In the context of hyperparameter tuning, Expected Improvement plays a crucial role in:

- **Sequential Model-Based Optimization**: EI guides the selection of which hyperparameter configurations to evaluate next
- **Efficient Resource Utilization**: By balancing exploration and exploitation, EI helps find good hyperparameters with fewer expensive model training runs
- **Automated Optimization**: EI provides a principled, automatic way to navigate the hyperparameter space without manual intervention

The implementation in `spotpython` makes Expected Improvement accessible for practical hyperparameter optimization tasks, providing both the theoretical rigor of Bayesian optimization and the computational efficiency needed for real-world applications.

18.5. Example: Spot and the 1-dim Sphere Function

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.utils.init import fun_control_init, surrogate_control_init, design_control_init
import matplotlib.pyplot as plt
```

18.5.1. The Objective Function: 1-dim Sphere

- The `spotpython` package provides several classes of objective functions.
- We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = x^2$$

```
fun = Analytical().fun_sphere
```

- The size of the `lower` bound vector determines the problem dimension.
- Here we will use `np.array([-1])`, i.e., a one-dim function.

> **TensorBoard**
>
> Similar to the one-dimensional case, which was introduced in Section Section 12.25, we can use TensorBoard to monitor the progress of the optimization. We will use the same code, only the prefix is different:

```
from spotpython.utils.init import fun_control_init
PREFIX = "07_Y"
fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals = 25,
    lower = np.array([-1]),
    upper = np.array([1]),
    tolerance_x = np.sqrt(np.spacing(1)),)
design_control = design_control_init(init_size=10)
```

```
spot_1 = Spot(
            fun=fun,
            fun_control=fun_control,
            design_control=design_control)
spot_1.run()
```

```
spotpython tuning: 2.4412172266370036e-09 [####------] 44.00%. Success rate: 100.00%
spotpython tuning: 2.032431480352692e-09 [#####-----] 48.00%. Success rate: 100.00%
spotpython tuning: 2.032431480352692e-09 [#####-----] 52.00%. Success rate: 66.67%
spotpython tuning: 2.032431480352692e-09 [######----] 56.00%. Success rate: 50.00%
spotpython tuning: 1.2228307315407816e-10 [######----] 60.00%. Success rate: 60.00%
spotpython tuning: 1.2228307315407816e-10 [######----] 64.00%. Success rate: 50.00%
spotpython tuning: 1.2228307315407816e-10 [#######---] 68.00%. Success rate: 42.86%
spotpython tuning: 1.2228307315407816e-10 [#######---] 72.00%. Success rate: 37.50%
spotpython tuning: 1.2228307315407816e-10 [########--] 76.00%. Success rate: 33.33%
spotpython tuning: 1.2228307315407816e-10 [########--] 80.00%. Success rate: 30.00%
spotpython tuning: 1.2228307315407816e-10 [########--] 84.00%. Success rate: 27.27%
spotpython tuning: 1.2228307315407816e-10 [#########-] 88.00%. Success rate: 25.00%
Using spacefilling design as fallback.
spotpython tuning: 1.2228307315407816e-10 [#########-] 92.00%. Success rate: 23.08%
spotpython tuning: 1.2228307315407816e-10 [##########] 96.00%. Success rate: 21.43%
spotpython tuning: 1.2228307315407816e-10 [##########] 100.00%. Success rate: 20.00% I

Experiment saved to 07_Y_res.pkl


<spotpython.spot.spot.Spot at 0x1083f4050>
```

18.5.2. Results

```
spot_1.print_results()
```

```
min y: 1.2228307315407816e-10
x0: 1.1058167712332733e-05

[['x0', np.float64(1.1058167712332733e-05)]]
```

```
spot_1.plot_progress(log_y=True)
```

Figure 18.1.: TensorBoard visualization of the spotpython optimization process and the surrogate model.

18.6. Same, but with EI as infill_criterion

```
PREFIX = "07_EI_ISO"
fun_control = fun_control_init(
    PREFIX=PREFIX,
    lower = np.array([-1]),
    upper = np.array([1]),
```

```
    fun_evals = 25,
    tolerance_x = np.sqrt(np.spacing(1)),
    infill_criterion = "ei")
```

```
spot_1_ei = Spot(fun=fun,
                     fun_control=fun_control)
spot_1_ei.run()
```

```
spotpython tuning: 1.0756606897092693e-07 [####------] 44.00%. Success rate: 100.00%
spotpython tuning: 3.3424229306960255e-09 [#####-----] 48.00%. Success rate: 100.00%
spotpython tuning: 3.3424229306960255e-09 [#####-----] 52.00%. Success rate: 66.67%
Using spacefilling design as fallback.
spotpython tuning: 3.3424229306960255e-09 [######----] 56.00%. Success rate: 50.00%
spotpython tuning: 3.323200616032585e-09 [######----] 60.00%. Success rate: 60.00%
spotpython tuning: 3.323200616032585e-09 [######----] 64.00%. Success rate: 50.00%
spotpython tuning: 3.323200616032585e-09 [#######---] 68.00%. Success rate: 42.86%
spotpython tuning: 3.323200616032585e-09 [#######---] 72.00%. Success rate: 37.50%
spotpython tuning: 3.323200616032585e-09 [########--] 76.00%. Success rate: 33.33%
Using spacefilling design as fallback.
spotpython tuning: 3.323200616032585e-09 [########--] 80.00%. Success rate: 30.00%
spotpython tuning: 3.323200616032585e-09 [########--] 84.00%. Success rate: 27.27%
Using spacefilling design as fallback.
spotpython tuning: 3.323200616032585e-09 [#########-] 88.00%. Success rate: 25.00%
Using spacefilling design as fallback.
spotpython tuning: 3.323200616032585e-09 [#########-] 92.00%. Success rate: 23.08%
spotpython tuning: 3.323200616032585e-09 [##########] 96.00%. Success rate: 21.43%
spotpython tuning: 1.3692913843728344e-10 [##########] 100.00%. Success rate: 26.67% I

Experiment saved to 07_EI_ISO_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x157ac0410>
```

```
spot_1_ei.plot_progress(log_y=True)
```

```
spot_1_ei.print_results()
```

```
min y: 1.3692913843728344e-10
x0: 1.1701672463254277e-05

[['x0', np.float64(1.1701672463254277e-05)]]
```

Figure 18.2.: TensorBoard visualization of the spotpython optimization process and the surrogate model. Expected improvement, isotropic Kriging.

18.7. Non-isotropic Kriging

```
PREFIX = "07_EI_NONISO"
fun_control = fun_control_init(
    PREFIX=PREFIX,
    lower = np.array([-1, -1]),
    upper = np.array([1, 1]),
    fun_evals = 25,
    tolerance_x = np.sqrt(np.spacing(1)),
```

```
    infill_criterion = "ei")
surrogate_control = surrogate_control_init(
    method="interpolation",
    )
```

```
spot_2_ei_noniso = Spot(fun=fun,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)
spot_2_ei_noniso.run()
```

```
spotpython tuning: 1.9002681213719912e-05 [####------] 44.00%. Success rate: 100.00%
spotpython tuning: 1.9002681213719912e-05 [#####-----] 48.00%. Success rate: 50.00%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [#####-----] 52.00%. Success rate: 33.33%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [######----] 56.00%. Success rate: 25.00%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [######----] 60.00%. Success rate: 20.00%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [######----] 64.00%. Success rate: 16.67%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [#######---] 68.00%. Success rate: 14.29%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [#######---] 72.00%. Success rate: 12.50%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [########--] 76.00%. Success rate: 11.11%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [########--] 80.00%. Success rate: 10.00%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [########--] 84.00%. Success rate: 9.09%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [#########-] 88.00%. Success rate: 8.33%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [#########-] 92.00%. Success rate: 7.69%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [##########] 96.00%. Success rate: 7.14%
Using spacefilling design as fallback.
spotpython tuning: 1.9002681213719912e-05 [##########] 100.00%. Success rate: 6.67% Do

Experiment saved to 07_EI_NONISO_res.pkl


<spotpython.spot.spot.Spot at 0x157bc1950>
```

```
spot_2_ei_noniso.plot_progress(log_y=True)
```

```
spot_2_ei_noniso.print_results()
```

```
min y: 1.9002681213719912e-05
x0: 0.0016094687577244535
x1: 0.004051208650715094
```

```
[['x0', np.float64(0.0016094687577244535)],
 ['x1', np.float64(0.004051208650715094)]]
```

```
spot_2_ei_noniso.surrogate.plot()
```

Figure 18.3.: TensorBoard visualization of the spotpython optimization process and the surrogate model. Expected improvement, isotropic Kriging.

18.8. Using `sklearn` Surrogates

18.8.1. The spot Loop

The `spot` loop consists of the following steps:

1. Init: Build initial design X
2. Evaluate initial design on real objective f: $y = f(X)$
3. Build surrogate: $S = S(X, y)$
4. Optimize on surrogate: $X_0 = \text{optimize}(S)$
5. Evaluate on real objective: $y_0 = f(X_0)$
6. Impute (Infill) new points: $X = X \cup X_0$, $y = y \cup y_0$.
7. Got 3.

The `spot` loop is implemented in `R` as follows:

Figure 18.4.: Visual representation of the model based search with SPOT. Taken from: Bartz-Beielstein, T., and Zaefferer, M. Hyperparameter tuning approaches. In Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide, E. Bartz, T. Bartz-Beielstein, M. Zaefferer, and O. Mersmann, Eds. Springer, 2022, ch. 4, pp. 67–114.

18.8.2. spot: The Initial Model

18.8.2.1. Example: Modifying the initial design size

This is the "Example: Modifying the initial design size" from Chapter 4.5.1 in [bart21i].

```
spot_ei = Spot(fun=fun,
               fun_control=fun_control_init(
               lower = np.array([-1,-1]),
               upper= np.array([1,1])),
```

```
                 design_control = design_control_init(init_size=5))
spot_ei.run()
```

```
spotpython tuning: 0.005235029089139776 [####------] 40.00%. Success rate: 100.00%
spotpython tuning: 0.005235029089139776 [#####-----] 46.67%. Success rate: 50.00%
spotpython tuning: 0.0007954275397760479 [#####-----] 53.33%. Success rate: 66.67%
spotpython tuning: 0.00047655813124564703 [######----] 60.00%. Success rate: 75.00%
spotpython tuning: 6.229052080428518e-05 [#######---] 66.67%. Success rate: 80.00%
spotpython tuning: 1.5782437351402495e-05 [#######---] 73.33%. Success rate: 83.33%
spotpython tuning: 3.3613071700767093e-06 [########--] 80.00%. Success rate: 85.71%
spotpython tuning: 8.155872607289737e-07 [#########-] 86.67%. Success rate: 87.50%
spotpython tuning: 2.1006222800202367e-07 [#########-] 93.33%. Success rate: 88.89%
spotpython tuning: 9.342669539119086e-08 [##########] 100.00%. Success rate: 90.00% D

Experiment saved to 000_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x17414f610>
```

```
spot_ei.plot_progress()
```

```
np.min(spot_1.y), np.min(spot_ei.y)
```

```
(np.float64(1.2228307315407816e-10), np.float64(9.342669539119086e-08))
```

18.8.3. Init: Build Initial Design

```
from spotpython.design.spacefilling import SpaceFilling
from spotpython.surrogate.kriging import Kriging
from spotpython.fun.objectivefunctions import Analytical
gen = SpaceFilling(2)
rng = np.random.RandomState(1)
lower = np.array([-5,-0])
upper = np.array([10,15])
fun = Analytical().fun_branin

X = gen.scipy_lhd(10, lower=lower, upper = upper)
print(X)
y = fun(X, fun_control=fun_control)
print(y)
```

```
[[ 8.97647221 13.41926847]
 [ 0.66946019  1.22344228]
 [ 5.23614115 13.78185824]
 [ 5.6149825  11.5851384 ]
 [-1.72963184  1.66516096]
 [-4.26945568  7.1325531 ]
 [ 1.26363761 10.17935555]
 [ 2.88779942  8.05508969]
 [-3.39111089  4.15213772]
 [ 7.30131231  5.22275244]]
[128.95676449  31.73474356 172.89678121 126.71295908  64.34349975
  70.16178611  48.71407916  31.77322887  76.91788181  30.69410529]
```

```
S = Kriging(name='kriging',  seed=123)
S.fit(X, y)
S.plot()
```

```
gen = SpaceFilling(2, seed=123)
X0 = gen.scipy_lhd(3)
gen = SpaceFilling(2, seed=345)
X1 = gen.scipy_lhd(3)
X2 = gen.scipy_lhd(3)
gen = SpaceFilling(2, seed=123)
X3 = gen.scipy_lhd(3)
X0, X1, X2, X3
```

```
(array([[0.77254938, 0.31539299],
        [0.59321338, 0.93854273],
        [0.27469803, 0.3959685 ]]),
 array([[0.78373509, 0.86811887],
        [0.06692621, 0.6058029 ],
        [0.41374778, 0.00525456]]),
 array([[0.121357  , 0.69043832],
```

```
        [0.41906219, 0.32838498],
        [0.86742658, 0.52910374]]),
 array([[0.77254938, 0.31539299],
        [0.59321338, 0.93854273],
        [0.27469803, 0.3959685 ]]))
```

18.8.4. Evaluate

18.8.5. Build Surrogate

18.8.6. A Simple Predictor

The code below shows how to use a simple model for prediction.

- Assume that only two (very costly) measurements are available:

  1. f(0) = 0.5
  2. f(2) = 2.5

- We are interested in the value at $x_0 = 1$, i.e., $f(x_0 = 1)$, but cannot run an additional, third experiment.

```
from sklearn import linear_model
X = np.array([[0], [2]])
y = np.array([0.5, 2.5])
S_lm = linear_model.LinearRegression()
S_lm = S_lm.fit(X, y)
X0 = np.array([[1]])
y0 = S_lm.predict(X0)
print(y0)
```

```
[1.5]
```

- Central Idea:

  – Evaluation of the surrogate model `S_lm` is much cheaper (or / and much faster) than running the real-world experiment f.

18.9. Gaussian Processes regression: basic introductory example

This example was taken from scikit-learn. After fitting our model, we see that the hyperparameters of the kernel have been optimized. Now, we will use our kernel to compute the mean prediction of the full dataset and plot the 95% confidence interval.

```
import numpy as np
import matplotlib.pyplot as plt
import math as m
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF

X = np.linspace(start=0, stop=10, num=1_000).reshape(-1, 1)
y = np.squeeze(X * np.sin(X))
rng = np.random.RandomState(1)
training_indices = rng.choice(np.arange(y.size), size=6, replace=False)
X_train, y_train = X[training_indices], y[training_indices]

kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
gaussian_process = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
gaussian_process.fit(X_train, y_train)
gaussian_process.kernel_

mean_prediction, std_prediction = gaussian_process.predict(X, return_std=True)

plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
    mean_prediction - 1.96 * std_prediction,
    mean_prediction + 1.96 * std_prediction,
    alpha=0.5,
    label=r"95% confidence interval",
)
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("sk-learn Version: Gaussian process regression on noise-free dataset")
```

```
from spotpython.surrogate.kriging import Kriging
import numpy as np
import matplotlib.pyplot as plt
rng = np.random.RandomState(1)
X = np.linspace(start=0, stop=10, num=1_000).reshape(-1, 1)
y = np.squeeze(X * np.sin(X))
training_indices = rng.choice(np.arange(y.size), size=6, replace=False)
X_train, y_train = X[training_indices], y[training_indices]


S = Kriging(name='kriging',  seed=123, log_level=50, cod_type="norm")
S.fit(X_train, y_train)

mean_prediction, std_prediction, ei = S.predict(X, return_val="all")

std_prediction

plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
    mean_prediction - 1.96 * std_prediction,
    mean_prediction + 1.96 * std_prediction,
    alpha=0.5,
```

```
    label=r"95% confidence interval",
)
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("spotpython Version: Gaussian process regression on noise-free dataset")
```

18.10. The Surrogate: Using scikit-learn models

Default is the internal `kriging` surrogate.

```
S_0 = Kriging(name='kriging', seed=123)
```

Models from `scikit-learn` can be selected, e.g., Gaussian Process:

```
# Needed for the sklearn surrogates:
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn import linear_model
from sklearn import tree
import pandas as pd
```

```
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
S_GP = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
```

- and many more:

```
S_Tree = DecisionTreeRegressor(random_state=0)
S_LM = linear_model.LinearRegression()
S_Ridge = linear_model.Ridge()
S_RF = RandomForestRegressor(max_depth=2, random_state=0)
```

- The scikit-learn GP model `S_GP` is selected.

```
S = S_GP
```

```
isinstance(S, GaussianProcessRegressor)
```

```
True
```

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical().fun_branin
fun_control = fun_control_init(
    lower = np.array([-5,-0]),
    upper = np.array([10,15]),
    fun_evals = 15)
design_control = design_control_init(init_size=5)
spot_GP = Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate=S,
                    design_control=design_control)
spot_GP.run()
```

```
spotpython tuning: 24.51465459019188 [####------] 40.00%. Success rate: 0.00%
spotpython tuning: 11.00309804905928 [#####-----] 46.67%. Success rate: 50.00%
spotpython tuning: 11.00309804905928 [#####-----] 53.33%. Success rate: 33.33%
spotpython tuning: 7.281531327496661 [######----] 60.00%. Success rate: 50.00%
spotpython tuning: 7.281531327496661 [#######---] 66.67%. Success rate: 40.00%
spotpython tuning: 7.281531327496661 [#######---] 73.33%. Success rate: 33.33%
spotpython tuning: 2.951979854778794 [########--] 80.00%. Success rate: 42.86%
spotpython tuning: 2.951979854778794 [#########-] 86.67%. Success rate: 37.50%
spotpython tuning: 2.1049810988944007 [#########-] 93.33%. Success rate: 44.44%
spotpython tuning: 1.9431600176593857 [##########] 100.00%. Success rate: 50.00% Done...

Experiment saved to 000_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x16a7b4f50>
```

```
spot_GP.y
```

```
array([ 69.32459936, 152.38491454, 107.92560483,  24.51465459,
        76.73500031,  86.30427428,  11.00309805,  16.11758746,
         7.28153133,  21.82310106,  10.96088904,   2.95197985,
         3.02910096,   2.1049811 ,   1.94316002])
```

```
spot_GP.plot_progress()
```

```
spot_GP.print_results()
```

```
min y: 1.9431600176593857
x0: 10.0
x1: 2.998557199177113

[['x0', np.float64(10.0)], ['x1', np.float64(2.998557199177113)]]
```

18.11. Additional Examples

```
# Needed for the sklearn surrogates:
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn import linear_model
```

```
from sklearn import tree
import pandas as pd
```

```
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
S_GP = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
```

```
from spotpython.surrogate.kriging import Kriging
import numpy as np
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot

S_K = Kriging(name='kriging',
              seed=123,
              log_level=50,
              infill_criterion = "y",
              isotropic=True,  # Use isotropic kernel
              method="interpolation",
              cod_type="norm")
fun = Analytical().fun_sphere

fun_control = fun_control_init(
    lower = np.array([-1,-1]),
    upper = np.array([1,1]),
    fun_evals = 25)

spot_S_K = Spot(fun=fun,
                     fun_control=fun_control,
                     surrogate=S_K,
                     design_control=design_control,
                     surrogate_control=surrogate_control)
spot_S_K.run()
```

```
spotpython tuning: 0.13771720249971786 [##--------] 24.00%. Success rate: 100.00%
spotpython tuning: 0.008758924098141154 [###-------] 28.00%. Success rate: 100.00%
spotpython tuning: 0.0028330055253615745 [###-------] 32.00%. Success rate: 100.00%
spotpython tuning: 0.0008163667213763957 [####------] 36.00%. Success rate: 100.00%
spotpython tuning: 0.0003648612873732834 [####------] 40.00%. Success rate: 100.00%
spotpython tuning: 0.00035952467170408163 [####------] 44.00%. Success rate: 100.00%
spotpython tuning: 0.00035952467170408163 [#####-----] 48.00%. Success rate: 85.71%
spotpython tuning: 0.00033242195170392255 [#####-----] 52.00%. Success rate: 87.50%
spotpython tuning: 0.000277658313657234 [######----] 56.00%. Success rate: 88.89%
spotpython tuning: 0.00016145264878065344 [######----] 60.00%. Success rate: 90.00%
```

```
spotpython tuning: 1.8753804818916804e-05 [######----] 64.00%. Success rate: 90.91%
spotpython tuning: 1.6850631244024777e-06 [#######---] 68.00%. Success rate: 91.67%
spotpython tuning: 7.325057368601842e-07 [#######---] 72.00%. Success rate: 92.31%
spotpython tuning: 4.3739809858009297e-07 [########--] 76.00%. Success rate: 92.86%
spotpython tuning: 3.9904274412067933e-07 [########--] 80.00%. Success rate: 93.33%
spotpython tuning: 3.7985542542754437e-07 [########--] 84.00%. Success rate: 93.75%
spotpython tuning: 3.720583681780579e-07 [#########-] 88.00%. Success rate: 94.12%
spotpython tuning: 3.720583681780579e-07 [#########-] 92.00%. Success rate: 88.89%
spotpython tuning: 3.720583681780579e-07 [##########] 96.00%. Success rate: 84.21%
spotpython tuning: 3.720583681780579e-07 [##########] 100.00%. Success rate: 80.00% Do

Experiment saved to 000_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x1743cd1d0>
```

```
spot_S_K.plot_progress(log_y=True)
```

```
spot_S_K.surrogate.plot()
```

```
spot_S_K.print_results()
```

```
min y: 3.720583681780579e-07
x0: -0.0006065092770223268
x1: 6.484493090375132e-05
```

```
[['x0', np.float64(-0.0006065092770223268)],
 ['x1', np.float64(6.484493090375132e-05)]]
```

18.11.1. Optimize on Surrogate

18.11.2. Evaluate on Real Objective

18.11.3. Impute / Infill new Points

18.12. Tests

```
import numpy as np
from spotpython.spot import Spot
from spotpython.fun.objectivefunctions import Analytical

fun_sphere = Analytical().fun_sphere

fun_control = fun_control_init(
                    lower=np.array([-1, -1]),
                    upper=np.array([1, 1]),
                    n_points = 2)
spot_1 = Spot(
    fun=fun_sphere,
    fun_control=fun_control,
)

# (S-2) Initial Design:
spot_1.X = spot_1.design.scipy_lhd(
    spot_1.design_control["init_size"], lower=spot_1.lower, upper=spot_1.upper
)
print(spot_1.X)

# (S-3): Eval initial design:
spot_1.y = spot_1.fun(spot_1.X)
print(spot_1.y)

spot_1.fit_surrogate()
X0 = spot_1.suggest_new_X()
print(X0)
assert X0.size == spot_1.n_points * spot_1.k
```

```
[[ 0.86352963  0.7892358 ]
 [-0.24407197 -0.83687436]
 [ 0.36481882  0.8375811 ]
 [ 0.415331    0.54468512]
```

```
 [-0.56395091 -0.77797854]
 [-0.90259409 -0.04899292]
 [-0.16484832  0.35724741]
 [ 0.05170659  0.07401196]
 [-0.78548145 -0.44638164]
 [ 0.64017497 -0.30363301]]
[1.36857656 0.75992983 0.83463487 0.46918172 0.92329124 0.8170764
 0.15480068 0.00815134 0.81623768 0.502017  ]
[[0.00068402 0.0024786 ]
 [0.00071693 0.00220831]]
```

18.13. EI: The Famous Schonlau Example

```
X_train0 = np.array([1, 2, 3, 4, 12]).reshape(-1,1)
X_train = np.linspace(start=0, stop=10, num=5).reshape(-1, 1)
```

```
from spotpython.surrogate.kriging import Kriging
import numpy as np
import matplotlib.pyplot as plt

X_train = np.array([1., 2., 3., 4., 12.]).reshape(-1,1)
y_train = np.array([0., -1.75, -2, -0.5, 5.])

S = Kriging(name='kriging',  seed=123, log_level=50, isotropic=True, method="interpolation", cod_typ
S.fit(X_train, y_train)

X = np.linspace(start=0, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
if True:
    plt.fill_between(
        X.ravel(),
        mean_prediction - 2 * std_prediction,
        mean_prediction + 2 * std_prediction,
        alpha=0.5,
        label=r"95% confidence interval",
    )
plt.legend()
plt.xlabel("$x$")
```

```
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")
```

```
#plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
# plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, -ei, label="Expected Improvement")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")
```

```
S.get_model_params()
```

```
{'n': 5,
 'k': 1,
 'logtheta_loglambda_p_': array([-0.99002508]),
 'U': array([[1.00000001e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00],
        [9.02737561e-01, 4.30191714e-01, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00],
        [6.64119239e-01, 7.04830372e-01, 2.49318667e-01, 0.00000000e+00,
         0.00000000e+00],
        [3.98156347e-01, 7.08262259e-01, 5.57958729e-01, 1.68873230e-
01,
         0.00000000e+00],
        [4.19704343e-06, 7.48472504e-05, 7.85846087e-04, 5.55936571e-
03,
         9.99984242e-01]]),
 'X': array([[ 1.],
        [ 2.],
        [ 3.],
        [ 4.],
        [12.]]),
```

```
 'y': array([ 0.  , -1.75, -2.  , -0.5 ,  5.  ]),
 'negLnLike': np.float64(1.20788204773309),
 'inf_Psi': np.False_,
 'cnd_Psi': np.float64(1411.6841898200016)}
```

18.14. EI: The Forrester Example

```
from spotpython.surrogate.kriging import Kriging
import numpy as np
import matplotlib.pyplot as plt
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot

# exact x locations are unknown:
X_train = np.array([0.0, 0.175, 0.225, 0.3, 0.35, 0.375, 0.5,1]).reshape(-1,1)

fun = Analytical().fun_forrester
fun_control = fun_control_init(
    PREFIX="07_EI_FORRESTER",
    sigma=1.0,
    seed=123,)
y_train = fun(X_train, fun_control=fun_control)

S = Kriging(name='kriging',  seed=123, log_level=50, isotropic=True, method="interpola
S.fit(X_train, y_train)

X = np.linspace(start=0, stop=1, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
if True:
    plt.fill_between(
        X.ravel(),
        mean_prediction - 2 * std_prediction,
        mean_prediction + 2 * std_prediction,
        alpha=0.5,
        label=r"95% confidence interval",
    )
plt.legend()
plt.xlabel("$x$")
```

```
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")
```

```
#plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
# plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, -ei, label="Expected Improvement")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")
```

18.15. Noise

```
import numpy as np
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.design.spacefilling import SpaceFilling
from spotpython.surrogate.kriging import Kriging
import matplotlib.pyplot as plt

gen = SpaceFilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_sphere
fun_control = fun_control_init(
    PREFIX="07_Y",
    sigma=2.0,
    seed=123,)
X = gen.scipy_lhd(10, lower=lower, upper = upper)
```

```
print(X)
y = fun(X, fun_control=fun_control)
print(y)
y.shape
X_train = X.reshape(-1,1)
y_train = y

S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            isotropic=True,
            method="interpolation")
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

#plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression on noisy dataset")
```

```
[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]
 [ 3.4468512 ]
 [ 6.36049088]
 [-7.77978539]]
[-1.57464135 16.13714981  2.77008442 93.14904827 71.59322218 14.28895359
 15.9770567  12.96468767 39.82265329 59.88028242]
```

```
S.get_model_params()
```

```
{'n': 10,
 'k': 1,
 'logtheta_loglambda_p_': array([-1.10547472]),
 'U': array([[ 1.00000001e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 1.71273395e-01,  9.85223548e-01,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 9.13185641e-01,  1.94770729e-01,  3.57989333e-01,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 1.75066871e-03, -3.03962963e-04, -3.32220593e-03,
          9.99992910e-01,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 1.77266503e-03,  2.46779726e-01, -1.18173360e-01,
```

```
         -3.20689955e-04,  9.61837613e-01,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 2.40962619e-01,  9.54670167e-01,  1.27460015e-01,
          2.92823173e-04, -4.96183480e-02,  1.08783189e-01,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 3.78787871e-01, -6.10436794e-02, -3.99469232e-01,
          9.30037927e-02, -3.40797737e-02,  2.28886550e-01,
          7.94366148e-01,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 5.37923899e-01, -8.19698170e-02, -4.73894981e-01,
          4.72464199e-02, -3.81494475e-02,  2.47600391e-01,
          6.30909853e-01,  1.27677676e-01,  0.00000000e+00,
          0.00000000e+00],
        [ 7.64573678e-02, -1.31037770e-02, -1.13704580e-01,
          4.31578050e-01, -1.06049024e-02,  7.65591464e-02,
          6.91377231e-01, -4.55944051e-01,  3.20831751e-01,
          0.00000000e+00],
        [ 3.87015246e-03,  3.51787171e-01, -1.60406586e-01,
         -4.32751818e-04,  9.03358193e-01, -1.23536924e-01,
          1.89427128e-02,  3.06145324e-02,  1.92052583e-02,
          1.32355763e-01]]),
 'X': array([[ 0.63529627],
        [-4.10764204],
        [-0.44071975],
        [ 9.63125638],
        [-8.3518118 ],
        [-3.62418901],
        [ 4.15331   ],
        [ 3.4468512 ],
        [ 6.36049088],
        [-7.77978539]]),
 'y': array([-1.57464135, 16.13714981,  2.77008442, 93.14904827, 71.59322218,
       14.28895359, 15.9770567 , 12.96468767, 39.82265329, 59.88028242]),
 'negLnLike': np.float64(26.185053861403645),
 'inf_Psi': np.False_,
 'cnd_Psi': np.float64(2435.982198259499)}
```

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            isotropic=True,
            method="regression")
```

```
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

#plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression with nugget on noisy dataset")
```

```
S.get_model_params()
```

```
{'n': 10,
 'k': 1,
 'logtheta_loglambda_p_': array([-2.96946871, -4.36752499]),
 'U': array([[ 1.00002145e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
```

```
        [ 9.76134124e-01,  2.17267286e-01,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 9.98737213e-01,  4.95999637e-02,  1.03307958e-02,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 9.16821246e-01, -3.60191645e-01, -8.88443611e-02,
          1.47818681e-01,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 9.16977835e-01,  3.94754755e-01, -3.27878537e-02,
          3.66312844e-02,  3.07626684e-02,  0.00000000e+00,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 9.80702573e-01,  1.95390826e-01,  2.97947792e-03,
         -1.95295765e-03,  2.02273105e-03,  8.42652902e-03,
          0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 9.86788787e-01, -1.55733877e-01, -1.99491898e-02,
          3.88580886e-02,  2.80728829e-03, -2.58948161e-03,
          1.07351401e-02,  0.00000000e+00,  0.00000000e+00,
          0.00000000e+00],
        [ 9.91534031e-01, -1.25469284e-01, -1.37299909e-02,
          2.92637185e-02,  2.04187526e-03, -2.17476893e-03,
          6.31851960e-03,  8.18075307e-03,  0.00000000e+00,
          0.00000000e+00],
        [ 9.65425304e-01, -2.45319330e-01, -4.38005419e-02,
          7.58546210e-02,  4.13698408e-03, -2.99235216e-03,
          6.62046280e-03,  2.90057093e-03,  8.03659457e-03,
          0.00000000e+00],
        [ 9.26823186e-01,  3.72507376e-01, -2.67177868e-02,
          3.00071969e-02,  2.43288149e-02,  1.24096720e-03,
         -3.28648755e-04, -2.24028838e-04,  1.35167355e-04,
          8.48878029e-03]]),
'X': array([[ 0.63529627],
        [-4.10764204],
        [-0.44071975],
        [ 9.63125638],
        [-8.3518118 ],
        [-3.62418901],
        [ 4.15331   ],
        [ 3.4468512 ],
        [ 6.36049088],
```

```
        [-7.77978539]]),
 'y': array([-1.57464135, 16.13714981,  2.77008442, 93.14904827, 71.59322218,
        14.28895359, 15.9770567 , 12.96468767, 39.82265329, 59.88028242]),
 'negLnLike': np.float64(21.820591738183587),
 'inf_Psi': np.False_,
 'cnd_Psi': np.float64(1.4611598243760502e+17)}
```

18.16. Cubic Function

```
import numpy as np
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.design.spacefilling import SpaceFilling
from spotpython.surrogate.kriging import Kriging
import matplotlib.pyplot as plt

gen = SpaceFilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_cubed
fun_control = fun_control_init(
    PREFIX="07_Y",
    sigma=10.0,
    seed=123,)

X = gen.scipy_lhd(10, lower=lower, upper = upper)
print(X)
y = fun(X, fun_control=fun_control)
print(y)
y.shape
X_train = X.reshape(-1,1)
y_train = y

S = Kriging(name='kriging',  seed=123, log_level=50, isotropic=True, method="interpola
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
```

```
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Cubed: Gaussian process regression on noisy dataset")
```

```
[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]
 [ 3.4468512 ]
 [ 6.36049088]
 [-7.77978539]]
[  -9.63480707  -72.98497325   12.7936499   895.34567477 -573.35961837
  -41.83176425   65.27989461   46.37081417  254.1530734  -474.09587355]
```

```
S = Kriging(name='kriging',  seed=123, log_level=0, isotropic=True, method="regressio
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Cubed: Gaussian process with nugget regression on noisy dataset")
```

```
import numpy as np
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.design.spacefilling import SpaceFilling
from spotpython.surrogate.kriging import Kriging
import matplotlib.pyplot as plt

gen = SpaceFilling(1)
rng = np.random.RandomState(1)
```

```
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_runge
fun_control = fun_control_init(
    PREFIX="07_Y",
    sigma=0.25,
    seed=123,)

X = gen.scipy_lhd(10, lower=lower, upper = upper)
print(X)
y = fun(X, fun_control=fun_control)
print(y)
y.shape
X_train = X.reshape(-1,1)
y_train = y

S = Kriging(name='kriging',  seed=123, log_level=50, isotropic=True, method="interpolation")
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noisy dataset")
```

```
[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]
 [ 3.4468512 ]
 [ 6.36049088]
 [-7.77978539]]
[ 0.46517267 -0.03599548  1.15933822  0.05915901  0.24419145  0.21502359
 -0.10432134  0.21312309 -0.05502681 -0.06434374]
```

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            isotropic=True,
            method="regression")
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression with nugget on noisy dataset")
```

18.17. Modifying Lambda Search Space

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            isotropic=True,
            method="regression",
            min_Lambda=0.1,
            max_Lambda=10)
S.fit(X_train, y_train)

print(f"Lambda: {S.Lambda}")
```

```
Lambda: [0.1]
```

```
X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
```

```
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression with nugget on noisy dataset. Modified Lamb
```

19. Handling Noise

This chapter demonstrates how noisy functions can be handled by `Spot` and how noise can be simulated, i.e., added to the objective function.

19.1. Example: `Spot` and the Noisy Sphere Function

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
import matplotlib.pyplot as plt
from spotpython.utils.init import fun_control_init, get_spot_tensorboard_path
from spotpython.utils.init import fun_control_init, design_control_init, surrogate_control_init

PREFIX = "08"
```

19.1.1. The Objective Function: Noisy Sphere

The `spotpython` package provides several classes of objective functions, which return a one-dimensional output $y = f(x)$ for a given input x (independent variable). Several objective functions allow one- or multidimensional input, some also combinations of real-valued and categorial input values.

An objective function is considered as "analytical" if it can be described by a closed mathematical formula, e.g.,

$$f(x, y) = x^2 + y^2.$$

To simulate measurement errors, adding artificial noise to the function value y is a common practice, e.g.,:

$$f(x, y) = x^2 + y^2 + \epsilon.$$

Usually, noise is assumed to be normally distributed with mean $\mu = 0$ and standard deviation σ. spotpython uses numpy's `scale` parameter, which specifies the standard

deviation (spread or "width") of the distribution is used. This must be a non-negative value, see https://numpy.org/doc/stable/reference/random/generated/numpy.random.normal.html.

Example: The sphere function without noise

The default setting does not use any noise.

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical().fun_sphere
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x,y, "k")
plt.show()
```

Example: The sphere function with noise

Noise can be added to the sphere function as follows:

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical(seed=123, sigma=0.02).fun_sphere
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x,y, "k")
plt.show()
```

19.1.2. Reproducibility: Noise Generation and Seed Handling

spotpython provides two mechanisms for generating random noise:

1. The seed is initialized once, i.e., when the objective function is instantiated. This can be done using the following call: `fun = Analytical(sigma=0.02, seed=123).fun_sphere`.
2. The seed is set every time the objective function is called. This can be done using the following call: `y = fun(x, sigma=0.02, seed=123)`.

These two different ways lead to different results as explained in the following tables:

Example: Noise added to the sphere function

Since `sigma` is set to `0.02`, noise is added to the function:

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical(sigma=0.02, seed=123).fun_sphere
x = np.array([1]).reshape(-1,1)
for i in range(3):
    print(f"{i}: {fun(x)}")
```

```
0: [0.98021757]
1: [0.98021757]
2: [0.98021757]
```

The seed is set once. Every call to `fun()` results in a different value. The whole experiment can be repeated, the initial seed is used to generate the same sequence as shown below:

Example: Noise added to the sphere function

Since `sigma` is set to `0.02`, noise is added to the function:

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical(sigma=0.02, seed=123).fun_sphere
x = np.array([1]).reshape(-1,1)
for i in range(3):
    print(f"{i}: {fun(x)}")
```

```
0: [0.98021757]
1: [0.98021757]
2: [0.98021757]
```

If `spotpython` is used as a hyperparameter tuner, it is important that only one realization of the noise function is optimized. This behaviour can be accomplished by passing the same seed via the dictionary `fun_control` to every call of the objective function `fun` as shown below:

Example: The same noise added to the sphere function

Since `sigma` is set to `0.02`, noise is added to the function:

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical().fun_sphere
fun_control = fun_control_init(
    PREFIX=PREFIX,
    sigma=0.02)
y = fun(x, fun_control=fun_control)
x = np.array([1]).reshape(-1,1)
for i in range(3):
    print(f"{i}: {fun(x)}")
```

```
0: [0.98021757]
1: [0.98021757]
2: [0.98021757]
```

19.2. spotpython's Noise Handling Approaches

The following setting will be used for the next steps:

```
fun = Analytical().fun_sphere
fun_control = fun_control_init(
    PREFIX=PREFIX,
    sigma=0.02,
)
```

`spotpython` is adopted as follows to cope with noisy functions:

1. `fun_repeats` is set to a value larger than 1 (here: 2)
2. `noise` is set to `true`. Therefore, a nugget (`Lambda`) term is added to the correlation matrix
3. `init size` (of the `design_control` dictionary) is set to a value larger than 1 (here: 3)

```
spot_1_noisy = Spot(fun=fun,
                    fun_control=fun_control_init(
                                    lower = np.array([-1]),
                                    upper = np.array([1]),
                                    fun_evals = 20,
                                    fun_repeats = 2,
                                    noise = True,
                                    show_models=True),
                    design_control=design_control_init(init_size=3, repeats=2),
                    surrogate_control=surrogate_control_init(method="regression"))
```

```
spot_1_noisy.run()
```

```
spotpython tuning: 0.03475418075554129 [####------] 40.00%. Success rate: 50.00%
```

```
spotpython tuning: 0.03337078483070389 [#####-----] 50.00%. Success rate: 50.00%
```

```
spotpython tuning: 0.015366462283161878 [######----] 60.00%. Success rate: 50.00%
```

```
spotpython tuning: 0.0005100105686610318 [#######---] 70.00%. Success rate: 50.00%
```

```
spotpython tuning: 4.512173241341755e-05 [########--] 80.00%. Success rate: 50.00%
```

```
spotpython tuning: 1.215181510824682e-07 [#########-] 90.00%. Success rate: 50.00%
```

```
spotpython tuning: 7.437561710315751e-08 [##########] 100.00%. Success rate: 50.00% Done...

Experiment saved to 000_res.pkl
```

19.3. Print the Results

```
spot_1_noisy.print_results()
```

```
min y: 7.437561710315751e-08
min mean y: 7.437561710315751e-08
x0: 0.00027271893425862004

[['x0', np.float64(0.00027271893425862004)]]
```

```
spot_1_noisy.plot_progress(log_y=False,
    filename="./figures/" + PREFIX + "_progress.png")
```

Figure 19.1.: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

19.4. Noise and Surrogates: The Nugget Effect

19.4.1. The Noisy Sphere

19.4.1.1. The Data

- We prepare some data first:

```
import numpy as np
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.design.spacefilling import SpaceFilling
from spotpython.surrogate.kriging import Kriging
import matplotlib.pyplot as plt

gen = SpaceFilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_sphere
fun_control = fun_control_init(
    PREFIX=PREFIX,
    sigma=4)
X = gen.scipy_lhd(10, lower=lower, upper = upper)
y = fun(X, fun_control=fun_control)
```

```
X_train = X.reshape(-1,1)
y_train = y
```

- A surrogate without nugget is fitted to these data:

```
S = Kriging(name='kriging',
            isotropic=True,
            method="interpolation")
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression on noisy dataset")
```

- In comparison to the surrogate without nugget, we fit a surrogate with nugget to the data:

```
S_nug = Kriging(name='kriging',
            isotropic=True,
            method="regression")
S_nug.fit(X_train, y_train)
X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S_nug.predict(X_axis, return_val="all")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression with nugget on noisy dataset")
```

- The value of the nugget term can be extracted from the model as follows:

```
S.Lambda
```

```
S_nug.Lambda
```

```
array([-3.25238809])
```

- We see:
  - the first model `S` has no nugget,
  - whereas the second model has a nugget value (`Lambda`) larger than zero.

19.5. Exercises

19.5.1. Noisy `fun_cubed`

- Analyse the effect of noise on the `fun_cubed` function with the following settings:

```
fun = Analytical().fun_cubed
fun_control = fun_control_init(
    sigma=10)
lower = np.array([-10])
upper = np.array([10])
```

19.5.2. `fun_runge`

- Analyse the effect of noise on the `fun_runge` function with the following settings:

```
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_runge
fun_control = fun_control_init(
    sigma=0.25)
```

19.5.3. `fun_forrester`

- Analyse the effect of noise on the `fun_forrester` function with the following settings:

```
lower = np.array([0])
upper = np.array([1])
fun = Analytical().fun_forrester
fun_control = fun_control_init(
    sigma=5)
```

19.5.4. `fun_xsin`

- Analyse the effect of noise on the `fun_xsin` function with the following settings:

```
lower = np.array([-1.])
upper = np.array([1.])
fun = Analytical().fun_xsin
fun_control = fun_control_init(
    sigma=0.5)
```

20. Optimal Computational Budget Allocation in spotpython

This chapter demonstrates how noisy functions can be handled `spotpython`:

- First, Optimal Computational Budget Allocation (OCBA) is introduced in Chapter 20.
- Then, the nugget effect is explained in Section 20.3.

Citation

If this document has been useful to you and you wish to cite it in a scientific publication, please refer to the following paper, which can be found on arXiv: https://arxiv.org/abs/2307.10262.

```
@ARTICLE{bart23iArXiv,
      author = {{Bartz-Beielstein}, Thomas},
      title = "{Hyperparameter Tuning Cookbook:
          A guide for scikit-learn, PyTorch, river, and spotpython}",
     journal = {arXiv e-prints},
    keywords = {Computer Science - Machine Learning,
      Computer Science - Artificial Intelligence, 90C26, I.2.6, G.1.6},
         year = 2023,
        month = jul,
          eid = {arXiv:2307.10262},
          doi = {10.48550/arXiv.2307.10262},
archivePrefix = {arXiv},
       eprint = {2307.10262},
 primaryClass = {cs.LG}
}
```

20.1. Example: spotpython, OCBA, and the Noisy Sphere Function

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
import matplotlib.pyplot as plt
from spotpython.utils.init import fun_control_init, get_spot_tensorboard_path
from spotpython.utils.init import fun_control_init, design_control_init, surrogate_co

PREFIX = "14"
```

20.1.1. The Objective Function: Noisy Sphere

The `spotpython` package provides several classes of objective functions. We will use an analytical objective function with noise, i.e., a function that can be described by a (closed) formula:

$$f(x) = x^2 + \epsilon$$

Since `sigma` is set to `0.1`, noise is added to the function:

```
fun = Analytical().fun_sphere
fun_control = fun_control_init(
    PREFIX=PREFIX,
    sigma=0.1)
```

A plot (Figure 20.1) illustrates the noise:

```
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x, fun_control=fun_control)
plt.figure()
plt.plot(x,y, "k")
plt.show()
```

Figure 20.1.: The noisy sphere function with noise.

Noise Handling in `spotpython`

`spotpython` has several options to cope with noisy functions:

1. `fun_repeats` is set to a value larger than 1, e.g., 2, which means every function evaluation during the search on the surrogate is repeated twice. The mean of the two evaluations is used as the function value.
2. `init size` (of the `design_control` dictionary) is set to a value larger than 1 (here: 2).
3. `ocba_delta` is set to a value larger than 1 (here: 2). This means that the OCBA algorithm is used to allocate the computational budget optimally.
4. Using a nugget term in the surrogate model. This is done by setting `method="regression"` in the `surrogate_control` dictionary. An example is given in Section 20.3.

20.2. Using Optimal Computational Budget Allocation (OCBA)

The Optimal Computational Budget Allocation (OCBA) algorithm is a powerful tool for efficiently distributing computational resources (C. H. Chen 2010). It is

specifically designed to maximize the Probability of Correct Selection (PCS) while minimizing computational costs. By strategically allocating more simulation effort to design alternatives that are either more challenging to evaluate or more likely to yield optimal results, OCBA ensures an efficient use of resources. This approach enables researchers and decision-makers to achieve accurate outcomes more quickly and with fewer computational demands, making it an invaluable method for simulation optimization.

The OCBA algorithm is implemented in `spotpython` and can be used by setting `ocba_delta` to a value larger than `0`. The source code is available in the `spotpython` package, see [DOC]. See also Bartz-Beielstein et al. (2011).

Example 20.1. To reproduce the example from p.49 in C. H. Chen (2010), the following `spotpython` code can be used:

```
import numpy as np
from spotpython.budget.ocba import get_ocba
mean_y = np.array([1,2,3,4,5])
var_y = np.array([1,1,9,9,4])
get_ocba(mean_y, var_y, 50)
```

```
array([11,  9, 19,  9,  2])
```

20.2.1. The Noisy Sphere

We will demonstrate the OCBA algorithm on the noisy sphere function defined in Section 20.1.1. The OCBA algorithm is used to allocate the computational budget optimally. This means that the function evaluations are repeated several times, and the best function value is used for the next iteration.

> **Visualizing the Search of the OCBA Algorithm**
>
> - The `show_models` parameter in the `fun_control` dictionary is set to `True`. This means that the surrogate model is shown during the search.
> - To keep the visualization simple, only the ground truth and the surrogate model are shown. The surrogate model is shown in blue, and the ground truth is shown in orange. The noisy function was shown in Figure 20.1.

```
spot_1_noisy = Spot(fun=fun,
                    fun_control=fun_control_init(
                    lower = np.array([-1]),
                    upper = np.array([1]),
                    fun_evals = 20,
```

```
                    fun_repeats = 1,
                    noise = True,
                    tolerance_x=0.0,
                    ocba_delta = 2,
                    show_models=True),
                    design_control=design_control_init(init_size=5, repeats=2),
                    surrogate_control=surrogate_control_init(method="regression"))
```

```
spot_1_noisy.run()
```

```
spotpython tuning: 0.005320352324811128 [######----] 55.00%. Success rate: 0.00%
```

```
spotpython tuning: 0.0004783786250623159 [######----] 60.00%. Success rate: 50.00%
```

```
spotpython tuning: 1.1021305166700617e-07 [######----] 65.00%. Success rate: 66.67%
```

```
spotpython tuning: 1.1021305166700617e-07 [#######---] 70.00%. Success rate: 50.00%
```

```
spotpython tuning: 1.2330075308902535e-11 [########--] 75.00%. Success rate: 60.00%
```

```
spotpython tuning: 1.1089814189632306e-11 [########--] 80.00%. Success rate: 66.67%
```

```
spotpython tuning: 7.071108734979218e-13 [########--] 85.00%. Success rate: 71.43%
```

```
spotpython tuning: 7.071108734979218e-13 [#########-] 90.00%. Success rate: 62.50%
```

```
spotpython tuning: 7.071108734979218e-13 [##########] 95.00%. Success rate: 55.56%
```

```
spotpython tuning: 7.071108734979218e-13 [##########] 100.00%. Success rate: 50.00% Do

Experiment saved to 000_res.pkl
```

20.2.2. Print the Results

```
spot_1_noisy.print_results()
```

```
min y: 7.071108734979218e-13
min mean y: 7.071108734979218e-13
x0: -8.408988485530955e-07
```

```
[['x0', np.float64(-8.408988485530955e-07)]]
```

```
spot_1_noisy.plot_progress(log_y=True)
```

20.3. Noise and Surrogates: The Nugget Effect

In the previous example, we have seen that the `fun_repeats` parameter can be used to repeat function evaluations. This is useful when the function is noisy. However, it is not always possible to repeat function evaluations, e.g., when the function is expensive to evaluate. In this case, we can use a surrogate model with a nugget term. The nugget term is a small value that is added to the diagonal of the covariance matrix. This allows the surrogate model to fit the data better, even if the data is noisy. The nugget term is added, if `method="regression"` is set in the `surrogate_control` dictionary.

20.3.1. The Noisy Sphere

20.3.1.1. The Data

We prepare some data first:

```
import numpy as np
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.design.spacefilling import SpaceFilling
from spotpython.surrogate.kriging import Kriging
import matplotlib.pyplot as plt

gen = SpaceFilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_sphere
```

```
fun_control = fun_control_init(
    sigma=2,
    seed=125)
X = gen.scipy_lhd(10, lower=lower, upper = upper)
y = fun(X, fun_control=fun_control)
X_train = X.reshape(-1,1)
y_train = y
```

A surrogate without nugget is fitted to these data:

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            isotropic=True,
            method="interpolation")
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression on noisy dataset")
```

In comparison to the surrogate without nugget, we fit a surrogate with nugget to the data:

```
S_nug = Kriging(name='kriging',
            seed=123,
            log_level=50,
            isotropic=True,
            method="regression")
S_nug.fit(X_train, y_train)
X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S_nug.predict(X_axis, return_val="all")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression with nugget on noisy dataset")
```

The value of the nugget term can be extracted from the model as follows:

```
S.Lambda
```

```
10**S_nug.Lambda
```

```
array([7.80803702e-05])
```

We see:

- the first model `S` has no nugget,
- whereas the second model has a nugget value (`Lambda`) larger than zero.

20.4. Exercises

20.4.1. Noisy `fun_cubed`

Analyse the effect of noise on the `fun_cubed` function with the following settings:

```
fun = Analytical().fun_cubed
fun_control = fun_control_init(
    sigma=10,
    seed=123)
lower = np.array([-10])
upper = np.array([10])
```

20.4.2. fun_runge

Analyse the effect of noise on the `fun_runge` function with the following settings:

```
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_runge
fun_control = fun_control_init(
    sigma=0.25,
    seed=123)
```

20.4.3. fun_forrester

Analyse the effect of noise on the `fun_forrester` function with the following settings:

```
lower = np.array([0])
upper = np.array([1])
fun = Analytical().fun_forrester
fun_control = {"sigma": 5,
               "seed": 123}
```

20.4.4. fun_xsin

Analyse the effect of noise on the `fun_xsin` function with the following settings:

```
lower = np.array([-1.])
upper = np.array([1.])
fun = Analytical().fun_xsin
fun_control = fun_control_init(
    sigma=0.5,
    seed=123)
```

20.5. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this chapter is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

21. Kriging with Varying Correlation-p

This chapter illustrates the difference between Kriging models with varying p. The difference is illustrated with the help of the `spotpython` package.

21.1. Example: `Spot` Surrogate and the 2-dim Sphere Function

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.utils.init import fun_control_init, surrogate_control_init
PREFIX="015"
```

21.1.1. The Objective Function: 2-dim Sphere

- The `spotpython` package provides several classes of objective functions.
- We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x, y) = x^2 + y^2$$

- The size of the `lower` bound vector determines the problem dimension.
- Here we will use `np.array([-1, -1])`, i.e., a two-dim function.

```
fun = Analytical().fun_sphere
fun_control = fun_control_init(PREFIX=PREFIX,
                               lower = np.array([-1, -1]),
                               upper = np.array([1, 1]))
```

- Although the default `spot` surrogate model is an isotropic Kriging model, we will explicitly set the `theta` parameter to a value of `1` for both dimensions. This is done to illustrate the difference between isotropic and anisotropic Kriging models.

```
surrogate_control=surrogate_control_init(n_p=1,
                                         p_val=2.0,)
```

```
spot_2 = Spot(fun=fun,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)

spot_2.run()
```

```
spotpython tuning: 6.611333241687761e-06 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 6.611333241687761e-06 [########--] 80.00%. Success rate: 50.00%
spotpython tuning: 6.611333241687761e-06 [#########-] 86.67%. Success rate: 33.33%
spotpython tuning: 6.611333241687761e-06 [#########-] 93.33%. Success rate: 25.00%
spotpython tuning: 3.1531067682958562e-06 [##########] 100.00%. Success rate: 40.00% I

Experiment saved to 015_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x106cac050>
```

21.1.2. Results

```
spot_2.print_results()
```

```
min y: 3.1531067682958562e-06
x0: -0.0012459503825024937
x1: -0.0012651934289418935
```

```
[['x0', np.float64(-0.0012459503825024937)],
 ['x1', np.float64(-0.0012651934289418935)]]
```

```
spot_2.plot_progress(log_y=True)
```

```
spot_2.surrogate.plot()
```

21.2. Example With Modified p

- We can use set `p_val` to a value other than 2 to obtain a different Kriging model.

```
surrogate_control = surrogate_control_init(n_p=1,
                                           p_val=1.0)
spot_2_p1= Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate_control=surrogate_control)
spot_2_p1.run()
```

```
spotpython tuning: 6.611333241687761e-06 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 6.611333241687761e-06 [########--] 80.00%. Success rate: 50.00%
spotpython tuning: 6.611333241687761e-06 [#########-] 86.67%. Success rate: 33.33%
spotpython tuning: 6.611333241687761e-06 [#########-] 93.33%. Success rate: 25.00%
spotpython tuning: 3.1531067682958562e-06 [##########] 100.00%. Success rate: 40.00% I

Experiment saved to 015_res.pkl

<spotpython.spot.spot.Spot at 0x138ab1450>
```

- The search progress of the optimization with the anisotropic model can be visualized:

```
spot_2_p1.plot_progress(log_y=True)
```

```
spot_2_p1.print_results()
```

```
min y: 3.1531067682958562e-06
x0: -0.0012459503825024937
x1: -0.0012651934289418935
```

```
[['x0', np.float64(-0.0012459503825024937)],
 ['x1', np.float64(-0.0012651934289418935)]]
```

```
spot_2_p1.surrogate.plot()
```

21.2.1. Taking a Look at the `p_val` Values

21.2.1.1. `p_val` Values from the `spot` Model

- We can check, which `p_val` values the `spot` model has used:
- The `p_val` values from the surrogate can be printed as follows:

```
spot_2_p1.surrogate.p_val
```

```
1.0
```

- Since the surrogate from the isotropic setting was stored as `spot_2`, we can also take a look at the `theta` value from this model:

```
spot_2.surrogate.p_val
```

```
2.0
```

21.3. Optimization of the `p_val` Values

```
surrogate_control = surrogate_control_init(n_p=1,
                                           optim_p=True)
spot_2_pm= Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate_control=surrogate_control)
spot_2_pm.run()
```

```
spotpython tuning: 6.5058325242694635e-06 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 6.5058325242694635e-06 [########--] 80.00%. Success rate: 50.00%
spotpython tuning: 6.5058325242694635e-06 [#########-] 86.67%. Success rate: 33.33%
spotpython tuning: 6.5058325242694635e-06 [#########-] 93.33%. Success rate: 25.00%
spotpython tuning: 4.326910928440869e-06 [##########] 100.00%. Success rate: 40.00% Do

Experiment saved to 015_res.pkl

<spotpython.spot.spot.Spot at 0x14ba016d0>
```

```
spot_2_pm.plot_progress(log_y=True)
```

```
spot_2_pm.print_results()
```

```
min y: 4.326910928440869e-06
x0: -0.0018878035376713096
x1: -0.0008735609489878526

[['x0', np.float64(-0.0018878035376713096)],
 ['x1', np.float64(-0.0008735609489878526)]]
```

```
spot_2_pm.surrogate.plot()
```

```
spot_2_pm.surrogate.p_val
```

```
array([1.44960552])
```

21.4. Optimization of Multiple p_val Values

```
surrogate_control = surrogate_control_init(n_p=2,
                                           optim_p=True)
spot_2_pmo= Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate_control=surrogate_control)
spot_2_pmo.run()
```

```
spotpython tuning: 8.652107435275436e-06 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 8.652107435275436e-06 [########--] 80.00%. Success rate: 50.00%
spotpython tuning: 8.652107435275436e-06 [#########-] 86.67%. Success rate: 33.33%
spotpython tuning: 8.652107435275436e-06 [#########-] 93.33%. Success rate: 25.00%
spotpython tuning: 8.652107435275436e-06 [##########] 100.00%. Success rate: 20.00% Do

Experiment saved to 015_res.pkl

<spotpython.spot.spot.Spot at 0x14bcd8a50>
```

```
spot_2_pmo.plot_progress(log_y=True)
```

```
spot_2_pmo.print_results()
```

```
min y: 8.652107435275436e-06
x0: 0.0006053414964068362
x1: 0.002878483821042489

[['x0', np.float64(0.0006053414964068362)],
 ['x1', np.float64(0.002878483821042489)]]
```

```
spot_2_pmo.surrogate.plot()
```

```
spot_2_pmo.surrogate.p_val
```

```
array([1.61192933, 1.76827516])
```

21.5. Exercises

21.5.1. fun_branin

- Describe the function.
  - The input dimension is 2. The search range is $-5 \leq x_1 \leq 10$ and $0 \leq x_2 \leq 15$.

- Compare the results from `spotpython` runs with different options for `p_val`.
- Modify the termination criterion: instead of the number of evaluations (which is specified via `fun_evals`), the time should be used as the termination criterion. This can be done as follows (`max_time=1` specifies a run time of one minute):

```
fun_evals=inf,
max_time=1,
```

21.5.2. `fun_sin_cos`

- Describe the function.
  - The input dimension is `2`. The search range is $-2\pi \leq x_1 \leq 2\pi$ and $-2\pi \leq x_2 \leq 2\pi$.
- Compare the results from `spotpython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

21.5.3. `fun_runge`

- Describe the function.
  - The input dimension is `2`. The search range is $-5 \leq x_1 \leq 5$ and $-5 \leq x_2 \leq 5$.
- Compare the results from `spotpython` runs with different options for `p_val`.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

21.5.4. `fun_wingwt`

- Describe the function.
  - The input dimension is `10`. The search ranges are between 0 and 1 (values are mapped internally to their natural bounds).
- Compare the results from `spotpython` runs with different options for `p_val`.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

21.6. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

22. Factorial Variables

Until now, we have considered continuous variables. However, in many applications, the variables are not continuous, but rather discrete or categorical. For example, the number of layers in a neural network, the number of trees in a random forest, or the type of kernel in a support vector machine are all discrete variables. In the following, we will consider a simple example with two numerical variables and one categorical variable.

```
from spotpython.design.spacefilling import SpaceFilling
from spotpython.surrogate.kriging import Kriging
from spotpython.fun.objectivefunctions import Analytical
import numpy as np
```

First, we generate the test data set for fitting the Kriging model. We use the `SpaceFilling` class to generate the first two diemnsion of $n = 30$ design points. The third dimension is a categorical variable, which can take the values 0, 1, or 2.

```
gen = SpaceFilling(2)
n = 30
rng = np.random.RandomState(1)
lower = np.array([-5,-0])
upper = np.array([10,15])
fun_orig = Analytical().fun_branin
fun = Analytical().fun_branin_factor

X0 = gen.scipy_lhd(n, lower=lower, upper = upper)
X1 = np.random.randint(low=0, high=3, size=(n,))
X = np.c_[X0, X1]
print(X[:5,:])
```

```
[[-2.84117593  5.97308949  0.        ]
 [-3.61017994  6.90781409  0.        ]
 [ 9.91204705  5.09395275  1.        ]
 [-4.4616725   1.3617128   1.        ]
 [-2.40987728  8.05505365  1.        ]]
```

The objective function is the `fun_branin_factor` in the `analytical` class [SOURCE]. It calculates the Branin function of (x_1, x_2) with an additional factor based on the value of x_3. If $x_3 = 1$, the value of the Branin function is increased by 10. If $x_3 = 2$, the value of the Branin function is decreased by 10. Otherwise, the value of the Branin function is not changed.

```
y = fun(X)
y_orig = fun_orig(X0)
data = np.c_[X, y_orig, y]
print(data[:5,:])
```

```
[[ -2.84117593   5.97308949   0.          32.09388125  32.09388125]
 [ -3.61017994   6.90781409   0.          43.965223    43.965223  ]
 [  9.91204705   5.09395275   1.           6.25588575  16.25588575]
 [ -4.4616725    1.3617128    1.         212.41884106 222.41884106]
 [ -2.40987728   8.05505365   1.           9.25981051  19.25981051]]
```

We fit two Kriging models, one with three numerical variables and one with two numerical variables and one categorical variable. We then compare the predictions of the two models.

```
S = Kriging(name='kriging',  seed=123, log_level=50, method="interpolation", var_type=
S.fit(X, y)
Sf = Kriging(name='kriging',  seed=123, log_level=50, method="interpolation", var_type
Sf.fit(X, y)
```

eps	np.float64(1....193847656e-08)
penalty	10000.0
method	'interpolation'
var_type	['num', 'num', ...]
name	'kriging'
seed	123
model_optimizer	<function dif...t 0x111cc6480>
model_fun_evals	100
n_theta	3
min_theta	-3.0
max_theta	2.0
theta_init_zero	False
p_val	2.0
n_p	1
optim_p	False
min_p	1.0

max_p	2.0
min_Lambda	-9.0
max_Lambda	0.0
log_level	50
spot_writer	None
counter	None
metric_factorial	'canberra'
isotropic	False
theta	array([-1.589... -2.88961467])
Lambda	None
use_nystrom	False
nystrom_m	None
nystrom_seed	1234
kernel	'gauss'
kernel_params	{}

We can now compare the predictions of the two models. We generate a new test data set and calculate the sum of the absolute differences between the predictions of the two models and the true values of the objective function. If the categorical variable is important, the sum of the absolute differences should be smaller than if the categorical variable is not important.

```
n = 100
k = 100
y_true = np.zeros(n*k)
y_pred= np.zeros(n*k)
y_factor_pred= np.zeros(n*k)
for i in range(k):
  X0 = gen.scipy_lhd(n, lower=lower, upper = upper)
  X1 = np.random.randint(low=0, high=3, size=(n,))
  X = np.c_[X0, X1]
  a = i*n
  b = (i+1)*n
  y_true[a:b] = fun(X)
  y_pred[a:b] = S.predict(X)
  y_factor_pred[a:b] = Sf.predict(X)
```

```
import pandas as pd
df = pd.DataFrame({"y":y_true, "Prediction":y_pred, "Prediction_factor":y_factor_pred})
df.head()
```

	y	Prediction	Prediction_factor
0	-3.315251	-4.125003	-2.529267
1	95.865258	97.375827	92.864932
2	49.811774	49.605423	49.633600
3	8.177150	3.930860	8.911852
4	-9.031623	-12.062624	-12.402044

```
df.tail()
```

	y	Prediction	Prediction_factor
9995	93.620503	93.518080	93.296759
9996	76.187178	75.589223	77.200934
9997	29.494401	48.785541	30.004073
9998	25.390268	13.996447	18.560661
9999	26.261264	26.283563	25.996336

```
s=np.sum(np.abs(y_pred - y_true))
sf=np.sum(np.abs(y_factor_pred - y_true))
res = (sf - s)
print(res)
```

```
-32194.597267247518
```

```
from spotpython.plot.validation import plot_actual_vs_predicted
plot_actual_vs_predicted(y_test=df["y"], y_pred=df["Prediction"], title="Default")
plot_actual_vs_predicted(y_test=df["y"], y_pred=df["Prediction_factor"], title="Facto
```

22.1. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

23. User-Specified Functions: Extending the Analytical Class

This chapter illustrates how user-specified functions can be optimized and analyzed. It covers singe-objective function in Section 23.2 and multi-objective functions in Section 23.6, and how to use the `spotpython` package to optimize them. It shows a simple approach to define a user-specified function, both for single- and multi-objective optimization, and how to extend the `Analytical` class to create a custom function.

Citation

- If this document has been useful to you and you wish to cite it in a scientific publication, please refer to the following paper, which can be found on arXiv: https://arxiv.org/abs/2307.10262.

```
@ARTICLE{bart23iArXiv,
      author = {{Bartz-Beielstein}, Thomas},
      title = "{Hyperparameter Tuning Cookbook:
          A guide for scikit-learn, PyTorch, river, and spotpython}",
     journal = {arXiv e-prints},
    keywords = {Computer Science - Machine Learning,
      Computer Science - Artificial Intelligence, 90C26, I.2.6, G.1.6},
         year = 2023,
        month = jul,
          eid = {arXiv:2307.10262},
          doi = {10.48550/arXiv.2307.10262},
archivePrefix = {arXiv},
       eprint = {2307.10262},
 primaryClass = {cs.LG}
}
```

23.1. Software Requirements

- The code examples in this chapter require the `spotpython` package, which can be installed via `pip`.

- Furthermore, the following Python packages are required:

```
import numpy as np
import matplotlib.pyplot as plt
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, surrogate_control_init
from spotpython.spot import Spot
```

23.2. The Single-Objective Function: User Specified

We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = \sum_{i}^{k} x_i^4.$$

This function is continuous, convex and unimodal. Since it returns one value for each input vector, it is a single-objective function. Multiple-objective functions can also be handled by `spotpython`. They are covered in Section 23.6.

The global minimum of the single-objective function is

$$f(x) = 0, \text{at } x = (0, 0, \dots, 0).$$

It can be implemented in Python as follows:

```
def user_fun(X):
    return(np.sum((X) **4, axis=1))
```

For example, if we have $X = (1, 2, 3)$, then

$$f(x) = 1^4 + 2^4 + 3^4 = 1 + 16 + 81 = 98,$$

and if we have $X = (4, 5, 6)$, then

$$f(x) = 4^4 + 5^4 + 6^4 = 256 + 625 + 1296 = 2177.$$

We can pass a 2D array to the function, and it will return a 1D array with the results for each row:

```
user_fun(np.array([[1, 2, 3], [4, 5, 6]]))
```

```
array([  98, 2177])
```

To make `user_fun` compatible with the `spotpython` package, we need to extend its argument list, so that it can handle the `fun_control` dictionary.

```
def user_fun(X, fun_control=None):
    return(np.sum((X) **4, axis=1))
```

Alternatively, you can add the `**kwargs` argument to the function, which will allow you to pass any additional keyword arguments:

```
def user_fun(X, **kwargs):
    return(np.sum((X) **4, axis=1))
```

```
fun_control = fun_control_init(
              lower = np.array( [-1, -1]),
              upper = np.array([1, 1]),
)
S = Spot(fun=user_fun,
                 fun_control=fun_control)
S.run()
S.plot_progress()
```

```
spotpython tuning: 3.715394917589437e-05 [#######---] 73.33%. Success rate: 0.00%
spotpython tuning: 3.715394917589437e-05 [########--] 80.00%. Success rate: 0.00%
spotpython tuning: 3.715394917589437e-05 [#########-] 86.67%. Success rate: 0.00%
spotpython tuning: 3.715394917589437e-05 [#########-] 93.33%. Success rate: 0.00%
spotpython tuning: 3.715394917589437e-05 [##########] 100.00%. Success rate: 0.00% Done...

Experiment saved to 000_res.pkl
```

> **Summary: Using `spotpython` with Single-Objective User-Specified Functions**
>
> - `spotpython` accepts user-specified functions that can be defined in Python.
> - The function should accept a 2D array as input and return a 1D array as output.
> - The function can be defined with an additional argument `fun_control` to handle control parameters.
> - The `fun_control` dictionary can be initialized with the `fun_control_init` function, which allows you to specify the bounds of the input variables.

23.3. The Objective Function: Extending the Analytical Class

- The `Analytical` class is a base class for analytical functions in the `spotpython` package.
- It provides a framework for defining and evaluating analytical functions, including the ability to add noise to the output.
- The `Analytical` class can be extended as follows:

```
from typing import Optional, Dict

class UserAnalytical(Analytical):
    def fun_user_function(self, X: np.ndarray, fun_control: Optional[Dict] = None) ->
        """
        Custom new function: f(x) = x^4

        Args:
            X (np.ndarray): Input data as a 2D array.
            fun_control (Optional[Dict]): Control parameters for the function.

        Returns:
            np.ndarray: Computed values with optional noise.

        Examples:
            >>> import numpy as np
            >>> X = np.array([[1, 2, 3], [4, 5, 6]])
            >>> fun = UserAnalytical()
            >>> fun.fun_user_function(X)
        """
        X = self._prepare_input_data(X, fun_control)
```

```
        offset = np.ones(X.shape[1]) * self.offset
        y = np.sum((X - offset) **4, axis=1)

        # Add noise if specified in fun_control
        return self._add_noise(y)
```

- In comparison to the `user_fun` function, the `UserAnalytical` class provides additional functionality, such as adding noise to the output and preparing the input data.
- First, we use the `user_fun` function as above.

```
user_fun = UserAnalytical()
X = np.array([[0, 0, 0], [1, 1, 1]])
results = user_fun.fun_user_function(X)
print(results)
```

```
[0. 3.]
```

- Then we can add an offset to the function, which will shift the function by a constant value. This is useful for testing the optimization algorithm's ability to find the global minimum.

```
user_fun = UserAnalytical(offset=1.0)
X = np.array([[0, 0, 0], [1, 1, 1]])
results = user_fun.fun_user_function(X)
print(results)
```

```
[3. 0.]
```

- And, we can add noise to the function, which will add a random value to the output. This is useful for testing the optimization algorithm's ability to find the global minimum in the presence of noise.

```
user_fun = UserAnalytical(sigma=1.0)
X = np.array([[0, 0, 0], [1, 1, 1]])
results = user_fun.fun_user_function(X)
print(results)
```

```
[0.06691138 3.11495313]
```

- Here is an example of how to use the `UserAnalytical` class with the `spotpython` package:

```
user_fun = UserAnalytical().fun_user_function
fun_control = fun_control_init(
              PREFIX="USER",
              lower = -1.0*np.ones(2),
              upper = np.ones(2),
              var_name=["User Pressure", "User Temp"],
              TENSORBOARD_CLEAN=True,
              tensorboard_log=True)
spot_user = Spot(fun=user_fun,
                 fun_control=fun_control)
spot_user.run()
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_11_04_20_41
Created spot_tensorboard_path: runs/spot_logs/USER_maans08_2025-11-
04_20-41-10 for SummaryWriter()
spotpython tuning: 3.715394917589437e-05 [#######---] 73.33%. Success rate: 0.00%
spotpython tuning: 3.715394917589437e-05 [########--] 80.00%. Success rate: 0.00%
spotpython tuning: 3.715394917589437e-05 [#########-] 86.67%. Success rate: 0.00%
spotpython tuning: 3.715394917589437e-05 [#########-] 93.33%. Success rate: 0.00%
spotpython tuning: 3.715394917589437e-05 [##########] 100.00%. Success rate: 0.00% Don

Experiment saved to USER_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x1592b7c50>
```

23.4. Results

```
_ = spot_user.print_results()
```

```
min y: 3.715394917589437e-05
User Pressure: 0.05170658955305796
User Temp: 0.07401195908206382
```

```
spot_user.plot_progress()
```

23.5. A Contour Plot

We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows.

> **Note:**
>
> We have specified bounds in the plot so that the contour plot can be plotted.

```
spot_user.plot_contour(i=0, j=1, min_z=0, max_z=2.25)
```

- The variable importance:

```
_ = spot_user.print_importance()
```

```
User Pressure:  59.78437370790242
User Temp:  100.0
```

```
spot_user.plot_importance()
```

23.6. Multi-Objective Functions

- The `spotpython` package can also handle multi-objective functions, which are functions that return multiple values for each input vector.
- As noted in Section 23.2, in the single-objective case, the function returns one value for each input vector and `spotpython` expects a 1D array as output.
- If the function returns a 2D array as output, `spotpython` will treat it as a multi-objective function result.

23.6.1. Response Surface Experiment

Myers, Montgomery, and Anderson-Cook (2016) describe a response surface experiment where three input variables (`reaction time`, `reaction temperature`, and `percent catalyst`) were used to model two characteristics of a chemical reaction: `percent conversion` and `thermal activity`. Their model is based on the following equations:

$$
\begin{aligned}
f_{\text{con}}(x) =& 81.09 + 1.0284 \cdot x_1 + 4.043 \cdot x_2 + 6.2037 \cdot x_3 + 1.8366 \cdot x_1^2 + 2.9382 \cdot x_2^2 \\
&+ 5.1915 \cdot x_3^2 + 2.2150 \cdot x_1 \cdot x_2 + 11.375 \cdot x_1 \cdot x_3 + 3.875 \cdot x_2 \cdot x_3
\end{aligned}
$$

and

$$
\begin{aligned}
f_{\text{act}}(x) =& 59.85 + 3.583 \cdot x_1 + 0.2546 \cdot x_2 + 2.2298 \cdot x_3 + 0.83479 \cdot x_1^2 + 0.07484 \cdot x_2^2 \\
&+ 0.05716 \cdot x_3^2 + 0.3875 \cdot x_1 \cdot x_2 + 0.375 \cdot x_1 \cdot x_3 + 0.3125 \cdot x_2 \cdot x_3.
\end{aligned}
$$

23.6.1.1. Defining the Multi-Objective Function `myer16a`

- The multi-objective function `myer16a` combines the results of two single-objective functions: conversion and activity.
- It is implemented in `spotpython` as follows:

```
import numpy as np

def conversion_pred(X):
    """
    Compute conversion predictions for each row in the input array.

    Args:
        X (np.ndarray): 2D array where each row is a configuration.

    Returns:
        np.ndarray: 1D array of conversion predictions.
    """
    return (
        81.09
        + 1.0284 * X[:, 0]
        + 4.043 * X[:, 1]
        + 6.2037 * X[:, 2]
        - 1.8366 * X[:, 0]**2
        + 2.9382 * X[:, 1]**2
        - 5.1915 * X[:, 2]**2
        + 2.2150 * X[:, 0] * X[:, 1]
        + 11.375 * X[:, 0] * X[:, 2]
        - 3.875 * X[:, 1] * X[:, 2]
    )

def activity_pred(X):
    """
    Compute activity predictions for each row in the input array.

    Args:
        X (np.ndarray): 2D array where each row is a configuration.

    Returns:
```

```
        np.ndarray: 1D array of activity predictions.
    """
    return (
        59.85
        + 3.583 * X[:, 0]
        + 0.2546 * X[:, 1]
        + 2.2298 * X[:, 2]
        + 0.83479 * X[:, 0]**2
        + 0.07484 * X[:, 1]**2
        + 0.05716 * X[:, 2]**2
        - 0.3875 * X[:, 0] * X[:, 1]
        - 0.375 * X[:, 0] * X[:, 2]
        + 0.3125 * X[:, 1] * X[:, 2]
    )

def fun_myer16a(X, fun_control=None):
    """
    Compute both conversion and activity predictions for each row in the input array.

    Args:
        X (np.ndarray): 2D array where each row is a configuration.
        fun_control (dict, optional): Additional control parameters (not used here).

    Returns:
        np.ndarray: 2D array where each row contains [conversion_pred, activity_pred].
    """
    return np.column_stack((conversion_pred(X), activity_pred(X)))
```

Now the function returns a 2D array with two columns, one for each objective function. The first column corresponds to the conversion prediction, and the second column corresponds to the activity prediction.

```
X = np.array([[1, 2, 3], [4, 5, 6]])
results = fun_myer16a(X)
print(results)
```

```
[[ 87.3132   72.25519]
 [200.8662   98.7442 ]]
```

23.6.1.2. Using a Weighted Sum

- The `spotpython` package can also handle multi-objective functions, which are functions that return multiple values for each input vector.

- In this case, we can use a weighted sum to combine the two objectives into a single objective function.
- The function `aggergate` takes the two objectives and combines them into a single objective function by applying weights to each objective.
- The weights can be adjusted to give more importance to one objective over the other.
- For example, if we want to give more importance to the conversion prediction, we can set the weight for the conversion prediction to 2 and the weight for the activity prediction to 0.1.

```
# Weight first objective with 2, second with 1/10
def aggregate(y):
    return np.sum(y*np.array([2, 0.1]), axis=1)
```

The `aggregate` function object is passed to the `fun_control` dictionary aas the `fun_mo2so` argument.

```
fun_control = fun_control_init(
              lower = np.array( [0, 0, 0]),
              upper = np.array([1, 1, 1]),
              fun_mo2so=aggregate)
S = Spot(fun=fun_myer16a,
        fun_control=fun_control)
S.run()
S.plot_progress()
```

```
spotpython tuning: 168.41794461566062 [#######---] 73.33%. Success rate: 100.00%
spotpython tuning: 168.35654210764483 [########--] 80.00%. Success rate: 100.00%
spotpython tuning: 168.16500000000002 [#########-] 86.67%. Success rate: 100.00%
Using spacefilling design as fallback.
spotpython tuning: 168.16500000000002 [#########-] 93.33%. Success rate: 75.00%
spotpython tuning: 166.99037900000002 [##########] 100.00%. Success rate: 80.00% Done

Experiment saved to 000_res.pkl
```

If no `fun_mo2so` function is specified, the `spotpython` package will use the first return value of the multi-objective function as the single objective function.

`spotpython` allows access to the complete history of multi-objective return values. They are stored in the `y_mo` attribute of the `Spot` object. The `y_mo` attribute is a 2D array where each row corresponds to a configuration and each column corresponds to an objective function. These values can be visualized as shown in Figure 23.1.

```
y_mo = S.y_mo
y = S.y
plt.xlim(0, len(y_mo))
plt.ylim(0.9 * np.min(y_mo), 1.1* np.max(y))
plt.scatter(range(len(y_mo)), y_mo[:, 0], label='Conversion', marker='o')
plt.scatter(range(len(y_mo)), y_mo[:, 1], label='Activity', marker='x')
plt.plot(np.minimum.accumulate(y_mo[:, 0]), label='Cum. Min Conversion')
plt.plot(np.minimum.accumulate(y_mo[:, 1]), label='Cum. Min Activity')
plt.scatter(range(len(y)), y, label='Agg. Result', marker='D', color='red')
plt.plot(np.minimum.accumulate(y), label='Cum. Min Agg. Res.', color='red')
plt.xlabel('Iteration')
plt.ylabel('Objective Function Value')
plt.grid()
plt.title('Single- and Multi-Obj. Function Values')
plt.legend(loc='upper left', bbox_to_anchor=(1, 1))
plt.tight_layout()
plt.show()
```

Figure 23.1.: Single- and Multi-Objective Function Values. The red line shows the optimization progress based on the aggregated objective function. The blue lines show the progress of the conversion objective, the orange line the progress of the activity objective. Points denote individual evaluations, lines the cumulative minimum of the respective objective function.

Since all values from the multi-objective functions can be accessed, more sophisticated multi-objective optimization methods can be implemented. For example, the `spotpython` package provides a `pareto_front` function that can be used to compute the Pareto front of the multi-objective function values, see pareto. The Pareto front is a set of solutions that are not dominated by any other solution in the objective space.

ℹ Summary: Using `spotpython` with Multi-Objective User-Specified Functions

- `spotpython` accepts user-specified multi-objective functions that can be defined in Python.
- The function should accept a 2D array as input and return a 2D array as output.
- An `aggregate` function can be used to combine multiple objectives into a single objective function.

23.7. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

24. Kriging with spotPython based on the Forrester et al. textbook

This chapter xplains how the spotpython Kriging class implements standard formulas from Forrester et al. (Engineering Design via Surrogate Modelling), focusing on Chapters 2, 3, and 6 (Gaussian process/Kriging modelling, likelihood-based hyperparameter estimation, prediction, and expected improvement).

24.1. High-level model and notation

- Training data: X R^{n×k}, y R^n.
- Correlation model: R(i,j) = exp(- _d _d · (x_i,d − x_j,d)^2) for ordered/numeric variables; factor variables use a categorical distance with the same exp(-D) envelope. are in log10 in this code, internally exponentiated as 10^ .
- Nugget: 0 (also in log10 here, exponentiated as 10^). Interpolation uses eps (tiny) instead of ; regression uses ; reinterpolation uses a → adjustment during prediction variance.
- Mean function: constant mean (ordinary Kriging).
- Concentrated likelihood (Forrester): given R (n×n) and residual variance ^2 = (r^T R^{-1} r)/n with r = y − 1 · and = (1^T R^{-1} y)/(1^T R^{-1} 1), the concentrated negative log-likelihood is n/2 · log(^2) + 1/2 · log|R| (constants dropped).

24.2. Correlation matrix build: build_Psi()

- Psi is the correlation matrix R (without the nugget) from pairwise distances:
  - Ordered/numeric variables use squared Euclidean dist weighted by 10^ .
  - Factor variables use a categorical metric (metric_factorial).
  - Final Psi = exp(−D).
- The method returns the upper triangle; the full R is formed later as R = Psi + Psi^T + I (and + I if applicable).

Code mapping:

```
def build_Psi(self) -> None:
    n, k = self.X_.shape
    theta10 = self._get_theta10_from_logtheta()  # 10**theta, isotropic expands to k
    Psi = np.zeros((n, n), dtype=np.float64)

    if self.ordered_mask.any():
        X_ordered = self.X_[:, self.ordered_mask]
        D_ordered = squareform(
            pdist(X_ordered,
            metric="sqeuclidean",
            w=theta10[self.ordered_mask]))
        Psi += D_ordered

    if self.factor_mask.any():
        X_factor = self.X_[:, self.factor_mask]
        D_factor = squareform(
            pdist(X_factor, metric=self.metric_factorial,
            w=theta10[self.factor_mask]))
        Psi += D_factor

    Psi = np.exp(-Psi)                  # R = exp(-D)
    return np.triu(Psi, k=1)            # upper triangle only
```

24.3. Likelihood and hyperparameters: likelihood()

- Input x packs [log10] plus, for regression/reinterpolation, log10 , and optionally p exponents (if enabled).
- The method:

  1) Forms R = (upper + transpose) + I + I (with = 10^log ; for interpolation := eps).
  2) Cholesky factorization R = U U^T.
  3) = (1^T R^{-1} y)/(1^T R^{-1} 1) via triangular solves with U.
  4) ^2 = (r^T R^{-1} r)/n with r = y − 1 · .
  5) −log L = n/2 · log(^2) + 1/2 · log|R|, with log|R| = 2 · log diag(U).

Code mapping (directly implements the standard concentrated likelihood):

```
def likelihood(self, x: np.ndarray) -> Tuple[float, np.ndarray, np.ndarray]:
    X = self.X_; y = self.y_.flatten()
    self.theta = x[: self.n_theta]
```

```
    if (self.method == "regression") or (self.method == "reinterpolation"):
        lambda_ = 10.0**x[self.n_theta : self.n_theta + 1]   # nugget on linear scale
        if self.optim_p:
            self.p_val = x[self.n_theta + 1 : self.n_theta + 1 + self.n_p]
    elif self.method == "interpolation":
        lambda_ = self.eps
        if self.optim_p:
            self.p_val = x[self.n_theta : self.n_theta + self.n_p]

    n = X.shape[0]; one = np.ones(n)
    Psi_up = self.build_Psi()
    Psi = Psi_up + Psi_up.T + np.eye(n) + np.eye(n) * lambda_  # R = corr + I +  I

    U = np.linalg.cholesky(Psi)
    LnDetPsi = 2.0 * np.sum(np.log(np.abs(np.diag(U))))        # log|R|

    # R^{-1}y and R^{-1}1 via triangular solves
    temp_y  = np.linalg.solve(U, y);   vy   = np.linalg.solve(U.T, temp_y)
    temp_1  = np.linalg.solve(U, one); vone = np.linalg.solve(U.T, temp_1)

    mu = (one @ vy) / (one @ vone)
    resid = y - one * mu
    tresid = np.linalg.solve(U, resid); tresid = np.linalg.solve(U.T, tresid)
    SigmaSqr = (resid @ tresid) / n

    negLnLike = (n / 2.0) * np.log(SigmaSqr) + 0.5 * LnDetPsi
    return negLnLike, Psi, U
```

24.4. Hyperparameter optimization: max_likelihood()

- Minimizes the concentrated negative log-likelihood over [log10 , log10 , p] using differential evolution.
- Objective = likelihood(x)[0].
- Bounds are assembled in fit() depending on method and options.

Code mapping:

```
def max_likelihood(self, bounds: List[Tuple[float, float]]) -> Tuple[np.ndarray, float]:
    def objective(logtheta_loglambda_p_: np.ndarray) -> float:
        neg_ln_like, _, _ = self.likelihood(logtheta_loglambda_p_)
        return neg_ln_like
```

```
    result = differential_evolution(func=objective, bounds=bounds, seed=self.seed)
    return result.x, result.fun
```

24.5. Model fitting workflow: fit()

- Sets k, n, variable type masks; selects n_theta (1 if isotropic else k).
- Builds bounds:
  - interpolation: k -bounds only.
  - regression/reinterpolation: k -bounds + 1 -bound.
  - If optim_p: adds p-bounds (n_p entries).
- Calls max_likelihood to obtain best x = [log10 , log10 , p].
- Stores:
  - theta = first n_theta entries of x.
  - Lambda = next 1 entry (for regression/reinterpolation) — kept on log10 scale in the model's state; transformed to linear only inside likelihood/_pred.
  - p_val if enabled.
- Computes final Psi, U and negLnLike at the found hyperparameters.

Code mapping:

```
def fit(self,
        X: np.ndarray,
        y: np.ndarray,
        bounds: Optional[List[Tuple[float, float]]] = None) -> "Kriging":
    X = np.asarray(X); y = np.asarray(y).flatten()
    self.X_, self.y_ = X, y
    self.n, self.k = self.X_.shape
    self._set_variable_types()
    if self.n_theta is None: self._set_theta()
    self.min_X = np.min(self.X_, axis=0); self.max_X = np.max(self.X_, axis=0)

    if bounds is None:
        if self.method == "interpolation":
            bounds = [(self.min_theta, self.max_theta)] * self.k
        else:
            bounds = [(self.min_theta, self.max_theta)] * self.k +
                      (self.min_Lambda, self.max_Lambda)]
    if self.optim_p:
        bounds += [(self.min_p, self.max_p)] * self.n_p

    self.logtheta_loglambda_p_, _ = self.max_likelihood(bounds)
```

```
    self.theta = self.logtheta_loglambda_p_[: self.n_theta]
    if (self.method == "regression") or (self.method == "reinterpolation"):
        self.Lambda = self.logtheta_loglambda_p_[self.n_theta : self.n_theta + 1]
        if self.optim_p:
            self.p_val = self.logtheta_loglambda_p_[self.n_theta + 1 : self.n_theta + 1 + self.n_p]
    elif self.method == "interpolation":
        self.Lambda = None
        if self.optim_p:
            self.p_val = self.logtheta_loglambda_p_[self.n_theta : self.n_theta + self.n_p]

    self.negLnLike, self.Psi_, self.U_ = self.likelihood(self.logtheta_loglambda_p_)
    self._update_log()
    return self
```

24.6. Correlation vector for a new point: build_psi_vec()

- For a new x R^k, compute (x) R^n with _i = exp(−D(x, x_i)), using same distance construction and weights 10^ as for R.

Code mapping:

```
def build_psi_vec(self, x: np.ndarray) -> None:
    n = self.X_.shape[0]
    theta10 = self._get_theta10_from_logtheta()
    D = np.zeros(n)

    if self.ordered_mask.any():
        X_ordered = self.X_[:, self.ordered_mask]
        x_ordered = x[self.ordered_mask]
        D += cdist(x_ordered.reshape(1, -1), X_ordered, metric="sqeuclidean", w=theta10[self.ordered

    if self.factor_mask.any():
        X_factor = self.X_[:, self.factor_mask]
        x_factor = x[self.factor_mask]
        D += cdist(x_factor.reshape(1, -1), X_factor, metric=self.metric_factorial, w=theta10[self.f

    psi = np.exp(-D)                  #  (x)
    return psi
```

24.7. Prediction at a new x: _pred() and predict()

- Ordinary Kriging predictor (Forrester):
  - Mean and R^{-1} reused (via stored U).
  - Predictor: f(x) = + (x)^T R^{-1} (y − 1 ·).
  - Predictive variance s^2(x) depends on method:
    - interpolation/regression: s^2(x) = ^2 [1 + − (x)^T R^{-1} (x)].
    - reinterpolation: uses R_adj = R − I + I for the variance term, consistent with a reinterpolation/nugget-adjusted scheme.
- Expected improvement (EI): computed when requested using standard normal CDF/PDF via erf; the code returns negative EI for minimization.

Code mapping:

```
def _pred(self, x: np.ndarray) -> float:
    y = self.y_.flatten()
    # load  ,  (  transformed to linear scale), p
    if (self.method == "regression") or (self.method == "reinterpolation"):
        self.theta = self.logtheta_loglambda_p_[: self.n_theta]
        lambda_ = 10.0**self.logtheta_loglambda_p_[self.n_theta : self.n_theta + 1]
        if self.optim_p:
            self.p_val = self.logtheta_loglambda_p_[self.n_theta + 1 : self.n_theta +
    elif self.method == "interpolation":
        self.theta = self.logtheta_loglambda_p_[: self.n_theta]
        lambda_ = self.eps

    U = self.U_; n = self.X_.shape[0]; one = np.ones(n)

    #  and R^{-1}r
    y_tilde = np.linalg.solve(U, y); y_tilde = np.linalg.solve(U.T, y_tilde)
    one_tilde = np.linalg.solve(U, one); one_tilde = np.linalg.solve(U.T, one_tilde)
    mu = (one @ y_tilde) / (one @ one_tilde)

    resid = y - one * mu
    resid_tilde = np.linalg.solve(U, resid); resid_tilde = np.linalg.solve(U.T, resid_

    psi = self.build_psi_vec(x)

    # ^2 and s^2(x)
    SigmaSqr = (resid @ resid_tilde) / n
    psi_tilde = np.linalg.solve(U, psi); psi_tilde = np.linalg.solve(U.T, psi_tilde)

    if (self.method == "interpolation") or (self.method == "regression"):
```

```
        SSqr = SigmaSqr * (1 + lambda_ - psi @ psi_tilde)   # s^2(x)
    else:
        Psi_adjusted = self.Psi_ - np.eye(n) * lambda_ + np.eye(n) * self.eps
        Uint = np.linalg.cholesky(Psi_adjusted)
        psi_tilde = np.linalg.solve(Uint, psi); psi_tilde = np.linalg.solve(Uint.T, psi_tilde)
        SSqr = SigmaSqr * (1 - psi @ psi_tilde)

    s = np.abs(SSqr) ** 0.5
    f = mu + psi @ resid_tilde                                  # predictor

    if self.return_ei:
        yBest = np.min(y)
        EITermOne = (yBest - f) * (0.5 + 0.5 * erf((1 / np.sqrt(2)) * ((yBest - f) / s)))
        EITermTwo = s * (1 / np.sqrt(2 * np.pi)) * np.exp(-0.5 * ((yBest - f) ** 2 / SSqr))
        ExpImp = np.log10(EITermOne + EITermTwo + self.eps)     # numerically stable
        return float(f), float(s), float(-ExpImp)               # negative EI
    else:
        return float(f), float(s)
```

24.8. Batch prediction: predict()

- Ensures X has shape (n_samples, k).
- Depending on return flags:
  - return_val="y": returns predictions only.
  - return_std or "s": returns standard deviation(s).
  - "ei" or "all": computes negative EI (and optionally y, s).

Key implementation detail: X reshaping to avoid 1D shape bugs.

```
def predict(self, X: np.ndarray, return_std=False, return_val: str = "y") -> np.ndarray:
    self.return_std = return_std
    X = self._reshape_X(X)                # ensures (n_samples, k)
    if return_std:
        predictions, stds = zip(*[self._pred(x_i)[:2] for x_i in X])
        return np.array(predictions), np.array(stds)
    if return_val == "s":
        predictions, stds = zip(*[self._pred(x_i)[:2] for x_i in X])
        return np.array(stds)
    elif return_val == "all":
        self.return_std = True; self.return_ei = True
        predictions, stds, eis = zip(*[self._pred(x_i) for x_i in X])
        return np.array(predictions), np.array(stds), np.array(eis)
```

```
    elif return_val == "ei":
        self.return_ei = True
        predictions, eis = zip(*[(self._pred(x_i)[0], self._pred(x_i)[2]) for x_i in X
        return np.array(eis)
    else:
        predictions = [self._pred(x_i)[0] for x_i in X]
        return np.array(predictions)
```

24.9. Method variants and options

- interpolation: Uses eps as tiny nugget; predictive variance s^2(x) = ^2[1 + eps − ^T R^{-1}].
- regression: Uses optimized ; same variance as above with .
- reinterpolation: For prediction variance only, adjusts R by removing and adding before the variance backsolve; predictor mean still uses R with .
- isotropic: n_theta = 1; a single controls all ordered dimensions; code expands 10^ to k when building distances.
- variable types: ordered_mask includes numeric and int; factor_mask uses a categorical metric and still lives inside exp(−D).
- parameters in log10: , are optimized and stored on log10 scale; transformed to linear (10^ ·) only at the points where R and s^2 require them.

24.10. Relation to Forrester (Ch. 2/3/6)

- Correlation form exp(− _d (Δx_d)^2) and the ordinary Kriging predictor f(x) = + r^T R^{-1}(y−1) are textbook.
- Concentrated likelihood n/2 log(^2) + 1/2 log|R| is used for hyperparameter estimation.
- Predictive variance s^2(x) = ^2(1 + − r^T R^{-1} r) reflects nugget ( ) for regression models; interpolation uses a tiny eps. The reinterpolation branch mirrors the "nugget-adjusted" variance computation.
- Expected Improvement (EI) follows the usual formula for minimization; the implementation uses erf for Φ and exp for , with a log10 stabilization.

24.11. Practical notes

- Bounds: fit() assembles bounds for [log] (k entries), [log] (1), and optionally p (n_p).

- Stability: Cholesky factorization is used; ill-conditioning penalized by returning a large negLnLike if factorization fails.
- Shape safety: predict() reshapes inputs; plot() uses a 1D grid reshaped to (n,1) accordingly.

24.12. Kriging 1D demo: 03bc, 03c3^2, s(x), and EI (Forrester et al., 2008, Ch. 2/3/6)

This notebook fits the provided Kriging implementation to a 1D function and visualizes:

- The predictor f^ (x)
- The estimated mean 03bc and variance 03c3^2 from the concentrated likelihood
- The predictive standard deviation s(x)
- The Expected Improvement (EI) curve (for minimization)

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from spotpython.surrogate.kriging import Kriging

# 1D training data (noisy-free quadratic)
rng = np.random.default_rng(0)
X_train = np.linspace(-1.0, 1.0, 9).reshape(-1, 1)
y_train = (X_train[:, 0]**2 + 0.1*X_train[:, 0]).astype(float)

# Fit Kriging (regression with nugget optimization)
model = Kriging(method="regression", var_type=["num"], seed=124)
model.fit(X_train, y_train)
print("theta(log10):", model.theta)
print("Lambda(log10):", model.Lambda)
```

```
theta(log10): [-1.14274728]
Lambda(log10): [-8.99954829]
```

```
# Compute \u03bc and \u03c3^2 as in likelihood() (Forrester Ch. 3)
y = model.y_.flatten()
n = model.X_.shape[0]
one = np.ones(n)
U = model.U_

temp_y = np.linalg.solve(U, y)
```

```
temp_1 = np.linalg.solve(U, one)
vy = np.linalg.solve(U.T, temp_y)
v1 = np.linalg.solve(U.T, temp_1)
mu = (one @ vy) / (one @ v1)
resid = y - one * mu
tresid = np.linalg.solve(U, resid)
tresid = np.linalg.solve(U.T, tresid)
sigma2 = (resid @ tresid) / n
print(f"mu: {mu:.6f}, sigma^2: {sigma2:.6e}")
```

```
mu: 9.838641, sigma^2: 2.836785e+01
```

```
# Prediction grid
X_grid = np.linspace(X_train.min(), X_train.max(), 200).reshape(-1, 1)

# Predict f^(x) and s(x)
y_pred, s_pred = model.predict(X_grid, return_std=True)

# EI: model.predict(..., return_val="all") returns (y, s, -log10(EI))
y_all, s_all, neg_log10_ei = model.predict(X_grid, return_val="all")
EI = 10.0 ** (-neg_log10_ei)

fig, axs = plt.subplots(3, 1, figsize=(7, 10), sharex=True)

# 1) Predictor with data
axs[0].plot(X_grid[:, 0], y_pred, 'k', label='Kriging predictor f^')
axs[0].scatter(X_train[:, 0], y_train, c='r', label='data')
axs[0].set_ylabel('f^ (x)')
axs[0].grid(True)
axs[0].legend(loc='best')

# 2) Predictive standard deviation
axs[1].plot(X_grid[:, 0], s_pred, 'b')
axs[1].set_ylabel('s(x)')
axs[1].grid(True)
axs[1].set_title(f"mu={mu:.4f}, sigma^2={sigma2:.2e}")

# 3) Expected Improvement (minimization)
axs[2].plot(X_grid[:, 0], EI, 'g')
axs[2].set_xlabel('x')
axs[2].set_ylabel('EI(x)')
axs[2].grid(True)
```

```
plt.tight_layout()
plt.show()
```

Kriging predictor f^
data
f^ (x)
1.0
0.8
0.6
0.4
0.2
0.0
mu=9.8386, sigma^2=2.84e+01
s(x)
0.000235
0.000230
0.000225
0.000220
0.000215
0.000210
0.000205
0.000200
EI(x)
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

Part IV.

Data-Driven Modeling and Optimization

25. Basic Statistics and Data Analysis

This chapter covers basic statistical concepts, namely descriptive statistics, probability distributions, and hypothesis testing. These concepts are fundamental to understanding data and making informed decisions based on data analysis. The chapter also introduces the concept of exploratory data analysis (EDA), data preprocessing (Principal Component Analysis), and data visualization techniques.

25.1. Exploratory Data Analysis

25.1.1. Histograms

Creating a histogram and calculating the probabilities from a dataset can be approached with scientific precision

1. Data Collection: Obtain the dataset you wish to analyze. This dataset could represent any quantitative measure, such to examine its distribution.
2. Decide on the Number of Bins: The number of bins influences the histogram's granularity. There are several statistical rules to determine an optimal number of bins:
   - Square-root rule: suggests using the square root of the number of data points as the number of bins.
   - Sturges' formula: $k = 1 + 3.322 \log_{10}(n)$, where n is the number of data points and k is the suggested number of bins.
   - Freedman-Diaconis rule: uses the interquartile range (IQR) and the cube root of the number of data points n to calculate bin width as $2\frac{IQR}{n^{1/3}}$.
3. Determine Range and Bin Width: Calculate the range of data by subtracting the minimum data point value from the maximum. Divide this range by the number of bins to determine the width of each bin.
4. Allocate Data Points to Bins: Iterate through the data, sorting each data point into the appropriate bin based on its value.
5. Draw the Histogram: Use a histogram to visualize the frequency or relative frequency (probability) of data points within each bin.

6. Calculate Probabilities: The relative frequency of data within each bin represents the probability of a randomly selected data point falling within that bin's range.

Below is a Python script that demonstrates how to generate a histogram and compute probabilities using the `matplotlib` library for visualization and `numpy` for data manipulation.

```
import numpy as np
import matplotlib.pyplot as plt

# Sample data: Randomly generated for demonstration
data = np.random.normal(0, 1, 1000)  # 1000 data points with a normal distribution

# Step 2: Decide on the number of bins
num_bins = int(np.ceil(1 + 3.322 * np.log10(len(data))))  # Sturges' formula

# Step 3: Determine range and bin width -- handled internally by matplotlib

# Steps 4 & 5: Sort data into bins and draw the histogram
fig, ax = plt.subplots()
n, bins, patches = ax.hist(data, bins=num_bins, density=True, alpha=0.75, edgecolor='b

# Calculate probabilities (relative frequencies) manually, if needed
bin_width = np.diff(bins)  # np.diff finds the difference between adjacent bin boundar
probabilities = n * bin_width  # n is already normalized to form a probability density

# Adding labels and title for clarity
ax.set_xlabel('Data Value')
ax.set_ylabel('Probability Density')
ax.set_title('Histogram with Probability Density')
```

```
Text(0.5, 1.0, 'Histogram with Probability Density')
```

(a) Histogram with Probability Density

(b)

Figure 25.1.

```python
for i, prob in enumerate(probabilities):
    print(f"Bin {i+1} Probability: {prob:.4f}")

# Ensure probabilities sum to 1 (or very close, due to floating-point arithmetic)
print(f"Sum of probabilities: {np.sum(probabilities)}")
```

```
Bin 1 Probability: 0.0040
Bin 2 Probability: 0.0210
Bin 3 Probability: 0.0670
Bin 4 Probability: 0.1620
Bin 5 Probability: 0.2470
Bin 6 Probability: 0.2440
Bin 7 Probability: 0.1700
Bin 8 Probability: 0.0650
Bin 9 Probability: 0.0140
```

```
Bin 10 Probability: 0.0050
Bin 11 Probability: 0.0010
Sum of probabilities: 1.0
```

This code segment goes through the necessary steps to generate a histogram and calculate probabilities for a synthetic dataset. It demonstrates important scientific and computational practices including binning, visualization, and probability calculation in Python.

Key Points:

- The histogram represents the distribution of data, with the histogram's bins outlining the data's spread and density.
- The option `density=True` in `ax.hist()` normalizes the histogram so that the total area under the histogram sums to 1, thereby converting frequencies to probability densities.
- The choice of bin number and width has a significant influence on the histogram's shape and the insights that can be drawn from it, highlighting the importance of selecting appropriate binning strategies based on the dataset's characteristics and the analysis objectives.
- Video: Histograms, Clearly Explained

25.1.2. Boxplots

- Video: Boxplots are Awesome

25.2. Probability Distributions

What happens when we use smaller bins in a histogram? The histogram becomes more detailed, revealing the distribution of data points with greater precision. However, as the bin size decreases, the number of data points within each bin may decrease, leading to sparse or empty bins. This sparsity can make it challenging to estimate probabilities accurately, especially for data points that fall within these empty bins.

Advantages, when using a probability distribution, include:

- Blanks can be filled
- Probabilities can be calculated
- Parameters are sufficient to describe the distribution, e.g., mean and variance for the normal distribution

Probability distributions offer a powerful solution to the challenges posed by limited data in estimating probabilities. When data is scarce, constructing a histogram to determine the probability of certain outcomes can lead to inaccurate or unreliable results due to the lack of detail in the dataset. However, collecting vast amounts of data to populate a histogram for more precise estimates can often be impractical, time-consuming, and expensive.

A probability distribution is a mathematical function that provides the probabilities of occurrence of different possible outcomes for an experiment. It is a more efficient approach to understanding the likelihood of various outcomes than relying solely on extensive data collection. For continuous data, this is often represented graphically by a smooth curve.

- Video: The Main Ideas behind Probability Distributions

25.2.1. Sampling from a Distribution

- Video: Sampling from a Distribution, Clearly Explained!!!

25.3. Discrete Distributions

Discrete probability distributions are essential tools in statistics, providing a mathematical foundation to model and analyze situations with discrete outcomes. Histograms, which can be seen as discrete distributions with data organized into bins, offer a way to visualize and estimate probabilities based on the collected data. However, they come with limitations, especially when data is scarce or when we encounter gaps in the data (blank spaces in histograms). These gaps can make it challenging to accurately estimate probabilities.

A more efficient approach, especially for discrete data, is to use mathematical equations—particularly those defining discrete probability distributions—to calculate probabilities directly, thus bypassing the intricacies of data collection and histogram interpretation.

25.3.1. Bernoulli Distribution

The Bernoulli distribution, named after Swiss scientist Jacob Bernoulli, is a discrete probability distribution, which takes value 1 with success probability p and value 0 with failure probability $q = 1 - p$. So if X is a random variable with this distribution, we have:

$$P(X = 1) = 1 - P(X = 0) = p = 1 - q.$$

25.3.2. Binomial Distribution

The Binomial Distribution is a prime example of a discrete probability distribution that is particularly useful for binary outcomes (e.g., success/failure, yes/no, pumpkin pie/blueberry pie). It leverages simple mathematical principles to calculate the probability of observing a specific number of successes (preferred outcomes) in a fixed number of trials, given the probability of success in each trial.

Example 25.1 (Pie Preference)**.** Consider a scenario from "StatLand" where 70% of people prefer pumpkin pie over blueberry pie. The question is: What is the probability that, out of three people asked, the first two prefer pumpkin pie and the third prefers blueberry pie?

Using the concept of the Binomial Distribution, the probability of such an outcome can be calculated without the need to layout every possible combination by hand. This process not only simplifies calculations but also provides a clear and precise method to determine probabilities in scenarios involving discrete choices. We will use Python to calculate the probability of observing exactly two out of three people prefer pumpkin pie, given the 70% preference rate:

```
from scipy.stats import binom
n = 3  # Number of trials (people asked)
p = 0.7  # Probability of success (preferring pumpkin pie)
x = 2  # Number of successes (people preferring pumpkin pie)
# Probability calculation using Binomial Distribution
prob = binom.pmf(x, n, p)
print(f"The probability that exactly 2 out of 3 people prefer pumpkin pie is: {prob:.3
```

```
The probability that exactly 2 out of 3 people prefer pumpkin pie is: 0.441
```

This code uses the `binom.pmf()` function from `scipy.stats` to calculate the probability mass function (PMF) of observing exactly `x` successes in `n` trials, where each trial has a success probability of `p`.

A Binomial random variable is the sum of n independent, identically distributed Bernoulli random variables, each with probability p of success. We may indicate a random variable X with Bernoulli distribution using the notation $X \sim \mathrm{Bi}(1, \theta)$. Then, the notation for the Binomial is $X \sim \mathrm{Bi}(n, \theta)$. Its probability and distribution functions are, respectively,

$$p_X(x) = \binom{n}{x} \theta^x (1-\theta)^{n-x}, \qquad F_X(x) = \Pr\{X \leq x\} = \sum_{i=0}^{x} \binom{n}{i} \theta^i (1-\theta)^{n-i}.$$

The mean of the binomial distribution is $\mathrm{E}[X] = n\theta$. The variance of the distribution is $\mathrm{Var}[X] = n\theta(1-\theta)$ (see next section).

A process consists of a sequence of n independent trials, i.e., the outcome of each trial does not depend on the outcome of previous trials. The outcome of each trial is either a success or a failure. The probability of success is denoted as p, and p is constant for each trial. Coin tossing is a classical example for this setting.

The binomial distribution is a statistical distribution giving the probability of obtaining a specified number of successes in a binomial experiment; written Binomial(n, p), where n is the number of trials, and p the probability of success in each.

Definition 25.1 (Binomial Distribution)**.** The binomial distribution with parameters n and p, where n is the number of trials, and p the probability of success in each, is

$$p(x) = \binom{n}{k} p^x (1-p)^{n-x} \qquad x = 0, 1, \dots, n. \tag{25.1}$$

The mean μ and the variance σ^2 of the binomial distribution are

$$\mu = np \tag{25.2}$$

and

$$\sigma^2 = np(1-p). \tag{25.3}$$

Note, the Bernoulli distribution is simply Binomial(1,p).

25.4. Continuous Distributions

Our considerations regarding probability distributions, expectations, and standard deviations will be extended from discrete distributions to continuous distributions. One simple example of a continuous distribution is the uniform distribution. Continuous distributions are defined by probability density functions.

25.4.1. Distribution functions: PDFs and CDFs

The density for a continuous distribution is a measure of the relative probability of "getting a value close to x." Probability density functions f and cumulative distribution function F are related as follows.

$$f(x) = \frac{d}{dx} F(x) \tag{25.4}$$

25.5. Background: Expectation, Mean, Standard Deviation

The distribution of a random vector is characterized by some indexes. These are the expectation, the mean, and the standard deviation. The expectation is a measure of the central tendency of a random variable, while the standard deviation quantifies the spread of the distribution. These indexes are essential for understanding the behavior of random variables and making predictions based on them.

Definition 25.2 (Random Variable)**.** A random variable X is a mapping from the sample space of a random experiment to the real numbers. It assigns a numerical value to each outcome of the experiment. Random variables can be either:

- Discrete: If X takes on a countable number of distinct values.
- Continuous: If X takes on an uncountable number of values.

Mathematically, a random variable is a function $X : \Omega \to \mathbb{R}$, where Ω is the sample space.

Definition 25.3 (Probability Distribution)**.** A probability distribution describes how the values of a random variable are distributed. It is characterized for a discrete random variable X by the probability mass function (PMF) $p_X(x)$ and for a continuous random variable X by the probability density function (PDF) $f_X(x)$.

Definition 25.4 (Probability Mass Function (PMF))**.** $p_X(x) = P(X = x)$ gives the probability that X takes the value x.

Definition 25.5 (Probability Density Function (PDF):)**.** $f_X(x)$ is a function such that for any interval $[a, b]$, the probability that X falls within this interval is given by the integral $\int_a^b f_X(x)\mathrm{d}x$.

The distribution function must satisfy:

$$\sum_{x \in D_X} p_X(x) = 1$$

for discrete random variables, where D_X is the domain of X and

$$\int_{-\infty}^{\infty} f_X(x)\mathrm{d}x = 1$$

for continuous random variables.

With these definitions in place, we can now introduce the definition of the expectation, which is a fundamental measure of the central tendency of a random variable.

Definition 25.6 (Expectation)**.** The expectation or expected value of a random variable X, denoted $E[X]$, is defined as follows:

For a discrete random variable X:

$$E[X] = \sum_{x \in D_X} x p_X(x) \quad \text{if } X \text{ is discrete.}$$

For a continuous random variable X:

$$E[X] = \int_{x \in D_X} x f_X(x) \mathrm{d}x \quad \text{if } X \text{ is continuous.}$$

The mean, μ, of a probability distribution is a measure of its central tendency or location. That is, $E(X)$ is defined as the average of all possible values of X, weighted by their probabilities.

Example 25.2 (Expectation)**.** Let X denote the number produced by rolling a fair die. Then

$$E(X) = 1 \times 1/6 + 2 \times 1/6 + 3 \times 1/6 + 4 \times 1/6 + 5 \times 1/6 + 6 \times 1/6 = 3.5.$$

Definition 25.7 (Sample Mean)**.** The sample mean is an important estimate of the population mean. The sample mean of a sample $\{x_i\}$ $(i = 1, 2, \ldots, n)$ is defined as

$$\overline{x} = \frac{1}{n} \sum_i x_i.$$

While both the expectation of a random variable and the sample mean provide measures of central tendency, they differ in their context, calculation, and interpretation.

- The expectation is a theoretical measure that characterizes the average value of a random variable over an infinite number of repetitions of an experiment. The expectation is calculated using a probability distribution and provides a parameter of the entire population or distribution. It reflects the long-term average or central value of the outcomes generated by the random process.
- The sample mean is a statistic. It provides an estimate of the population mean based on a finite sample of data. It is computed directly from the data sample, and its value can vary between different samples from the same population. It serves as an approximation or estimate of the population mean. It is used in statistical inference to make conclusions about the population mean based on sample data.

If we are trying to predict the value of a random variable X by its mean $\mu = E(X)$, the error will be $X - \mu$. In many situations it is useful to have an idea how large this deviation or error is. Since $E(X - \mu) = E(X) - \mu = 0$, it is necessary to use the absolute value or the square of $(X - \mu)$. The squared error is the first choice, because the derivatives are easier to calculate. These considerations motivate the definition of the variance:

Definition 25.8 (Variance)**.** The variance of a random variable X is the mean squared deviation of X from its expected value $\mu = E(X)$.

$$Var(X) = E[(X - \mu)^2]. \tag{25.5}$$

The variance is a measure of the spread of a distribution. It quantifies how much the values of a random variable differ from the mean. A high variance indicates that the values are spread out over a wide range, while a low variance indicates that the values are clustered closely around the mean.

Definition 25.9 (Standard Deviation)**.** Taking the square root of the variance to get back to the same scale of units as X gives the standard deviation. The standard deviation of X is the square root of the variance of X.

$$sd(X) = \sqrt{Var(X)}. \tag{25.6}$$

25.6. Distributions and Random Numbers in Python

Results from computers are deterministic, so it sounds like a contradiction in terms to generate random numbers on a computer. Standard computers generate pseudo-randomnumbers, i.e., numbers that behave as if they were drawn randomly.

> **Deterministic Random Numbers**
>
> - Idea: Generate deterministically numbers that **look** (behave) as if they were drawn randomly.

25.6.1. Calculation of the Standard Deviation with Python

The function `numpy.std` returns the standard deviation, a measure of the spread of a distribution, of the array elements. The argument `ddof` specifies the Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. By default `ddof` is zero, i.e., `std` uses the formula

$$\sqrt{\frac{1}{N}\sum_i (x_i - \bar{x})^2} \qquad \text{with} \qquad \bar{x} = \sum_{i=1}^{N} x_i/N.$$

Example 25.3 (Standard Deviation with Python)**.** Consider the array $[1, 2, 3]$: Since $\bar{x} = 2$, the following value is computed:

$$\sqrt{1/3 \times ((1-2)^2 + (2-2)^2 + (3-2)^2)} = \sqrt{2/3}.$$

```
import numpy as np
a = np.array([[1, 2, 3]])
np.std(a)
```

```
np.float64(0.816496580927726)
```

The empirical standard deviation (which uses $N-1$), $\sqrt{1/2 \times ((1-2)^2 + (2-2)^2 + (3-2)^2)} = \sqrt{2/2}$, can be calculated in Python as follows:

```
np.std(a, ddof=1)
```

```
np.float64(1.0)
```

25.6.2. The Argument "axis"

When you compute `np.std` with `axis=0`, it calculates the standard deviation along the vertical axis, meaning it computes the standard deviation for each column of the array. On the other hand, when you compute `np.std` with `axis=1`, it calculates the standard deviation along the horizontal axis, meaning it computes the standard deviation for each row of the array. If the axis parameter is not specified, `np.std` computes the standard deviation of the flattened array, i.e., it calculates the standard deviation of all the elements in the array.

Example 25.4 (Axes along which the standard deviation is computed)**.**

```
A = np.array([[1, 2], [3, 4]])
A
```

```
array([[1, 2],
       [3, 4]])
```

First, we calculate the standard deviation of all elements in the array:

```
np.std(A)
```

```
np.float64(1.118033988749895)
```

Setting `axis=0` calculates the standard deviation along the vertical axis (column-wise):

```
np.std(A, axis=0)
```

```
array([1., 1.])
```

Finally, setting `axis=1` calculates the standard deviation along the horizontal axis (row-wise):

```
np.std(A, axis=1)
```

```
array([0.5, 0.5])
```

25.7. Expectation (Continuous)

Definition 25.10 (Expectation (Continuous))**.**

$$\mathrm{E}(X) = \int_{-\infty}^{\infty} x f(x)\, dx \tag{25.7}$$

25.8. Variance and Standard Deviation (Continuous)

Definition 25.11 (Variance (Continuous))**.** Variance can be calculated with $\mathrm{E}(X)$ and

$$\mathrm{E}(X^2) = \int_{-\infty}^{\infty} x^2 f(x)\, dx \tag{25.8}$$

as

$$\mathrm{Var}(X) = \mathrm{E}(X^2) - [E(X)]^2.$$

□

Definition 25.12 (Standard Deviation (Continuous))**.** Standard deviation can be calculated as

$$\mathrm{sd}(X) = \sqrt{\mathrm{Var}(X)}.$$

□

- Video: Population and Estimated Parameters, Clearly Explained
- Video: Calculating the Mean, Variance, and Standard Deviation

25.8.1. The Uniform Distribution

Definition 25.13 (The Uniform Distribution)**.** The probability density function of the uniform distribution is defined as:

$$f_X(x) = \frac{1}{b-a} \qquad \text{for } x \in [a, b].$$

Generate 10 random numbers from a uniform distribution between $a = 0$ and $b = 1$:

```
import numpy as np
# Initialize the random number generator
rng = np.random.default_rng(seed=123456789)
n = 10
x = rng.uniform(low=0.0, high=1.0, size=n)
x
```

```
array([0.02771274, 0.90670006, 0.88139355, 0.62489728, 0.79071481,
       0.82590801, 0.84170584, 0.47172795, 0.95722878, 0.94659153])
```

Generate 10,000 random numbers from a uniform distribution between 0 and 10 and plot a histogram of the numbers:

```
import numpy as np
import matplotlib.pyplot as plt

# Initialize the random number generator
rng = np.random.default_rng(seed=123456789)

# Generate random numbers from a uniform distribution
x = rng.uniform(low=0, high=10, size=10000)

# Plot a histogram of the numbers
plt.hist(x, bins=50, density=True, edgecolor='black')
plt.title('Uniform Distribution [0,10]')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```

25.9. The Uniform Distribution

This variable is defined in the interval $[a, b]$. We write it as $X \sim U[a, b]$. Its density and cumulative distribution functions are, respectively,

$$f_X(x) = \frac{I_{[a,b]}(x)}{b-a}, \qquad F_X(x) = \frac{1}{b-a} \int_{-\infty}^{x} I_{[a,b]}(t)\mathrm{d}t = \frac{x-a}{b-a},$$

where $I_{[a,b]}(\cdot)$ is the indicator function of the interval $[a, b]$. Note that, if we set $a = 0$ and $b = 1$, we obtain $F_X(x) = x$, $x \in [0, 1]$.

A typical example is the following: the cdf of a continuous r.v. is uniformly distributed in $[0, 1]$. The proof of this statement is as follows: For $u \in [0, 1]$, we have

$$\begin{aligned} \Pr\{F_X(X) \leq u\} &= \Pr\{F_X^{-1}(F_X(X)) \leq F_X^{-1}(u)\} = \Pr\{X \leq F_X^{-1}(u)\} \\ &= F_X(F_X^{-1}(u)) = u. \end{aligned}$$

This means that, when X is continuous, there is a one-to-one relationship (given by the cdf) between $x \in D_X$ and $u \in [0, 1]$.

The *uniform distribution* has a constant density over a specified interval, say $[a, b]$. The uniform $U(a, b)$ distribution has density

$$f(x) = \begin{cases} 1/(b-a) & \text{if } a < x < b, \\ 0 & \text{otherwise} \end{cases} \tag{25.9}$$

25.9.1. The Normal Distribution

A normally distributed random variable is a random variable whose associated probability distribution is the normal (or Gaussian) distribution. The normal distribution is a continuous probability distribution characterized by a symmetric bell-shaped curve.

The distribution is defined by two parameters: the mean μ and the standard deviation σ. The mean indicates the center of the distribution, while the standard deviation measures the spread or dispersion of the distribution.

This distribution is widely used in statistics and the natural and social sciences as a simple model for random variables with unknown distributions.

Definition 25.14 (The Normal Distribution)**.** The probability density function of the normal distribution is defined as:

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right), \tag{25.10}$$

where: μ is the mean; σ is the standard deviation.

To generate ten random numbers from a normal distribution, the following command can be used.

```
import numpy as np
rng = np.random.default_rng()
n = 10
mu, sigma = 2, 0.1
x = rng.normal(mu, sigma, n)
x
```

```
array([1.80851257, 2.08596823, 1.93693038, 2.14683389, 1.94082515,
       1.9603525 , 2.04689473, 1.98464834, 2.05929595, 1.88618405])
```

Verify the mean:

```
abs(mu - np.mean(x))
```

```
np.float64(0.014355421181810657)
```

Note: To verify the standard deviation, we use `ddof = 1` (empirical standard deviation):

```
abs(sigma - np.std(x, ddof=1))
```

```
np.float64(0.0008373897053816359)
```

```
plot_normal_distribution(mu=0, sigma=1, num_samples=10000)
```

25.9.2. Visualization of the Standard Deviation

The standard deviation of normal distributed can be visualized in terms of the histogram of X:

- about 68% of the values will lie in the interval within one standard deviation of the mean
- 95% lie within two standard deviation of the mean
- and 99.9% lie within 3 standard deviations of the mean.

25.9.3. Realizations of a Normal Distribution

Realizations of a normal distribution refers to the actual values that you get when you draw samples from a normal distribution. Each sample drawn from the distribution is a realization of that distribution.

Example 25.5 (Realizations of a Normal Distribution)**.** If you have a normal distribution with a mean of 0 and a standard deviation of 1, each number you draw from that distribution is a realization. Here is a Python example that generates 10 realizations of a normal distribution with a mean of 0 and a standard deviation of 1:

```
import numpy as np
mu = 0
sigma = 1
realizations = np.random.normal(mu, sigma, 10)
print(realizations)
```

```
[ 0.48951662  0.23879586 -0.44811181 -0.610795   -2.02994507  0.60794659
 -0.35410888  0.15258149  0.50127485 -0.78640277]
```

In this code, `np.random.normal` generates ten realizations of a normal distribution with a mean of 0 and a standard deviation of 1. The realizations array contains the actual values drawn from the distribution.

25.10. The Normal Distribution

A commonly encountered probability distribution is the normal distribution, known for its characteristic bell-shaped curve. This curve represents how the values of a variable are distributed: most of the observations cluster around the mean (or center) of the distribution, with frequencies gradually decreasing as values move away from the mean.

The normal distribution is particularly useful because of its defined mathematical properties. It is determined entirely by its mean (mu, μ) and its standard deviation (sigma, σ). The area under the curve represents probability, making it possible to calculate the likelihood of a random variable falling within a specific range.

- Video: The Normal Distribution, Clearly Explained!!!

Example 25.6 (Normal Distribution: Estimating Probabilities)**.** Consider we are interested in the heights of adults in a population. Instead of measuring the height of every adult (which would be impractical), we can use the normal distribution to estimate the probability of adults' heights falling within certain intervals, assuming we know the mean and standard deviation of the heights.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
mu = 170  # e.g., mu height of adults in cm
sd = 10  # e.g., standard deviation of heights in cm
heights = np.linspace(mu - 3*sd, mu + 3*sd, 1000)
# Calculate the probability density function for the normal distribution
pdf = norm.pdf(heights, mu, sd)
# Plot the normal distribution curve
plt.plot(heights, pdf, color='blue', linewidth=2)
plt.fill_between(heights, pdf, where=(heights >= mu - 2 * sd) & (heights <= mu + 2*sd)
plt.xlabel('Height (cm)')
plt.ylabel('Probability Density')
plt.show()
```

Figure 25.2.: Normal Distribution Curve with Highlighted Probability Area. 95 percent of the data falls within two standard deviations of the mean.

This Python code snippet generates a plot of the normal distribution for adult heights, with a mean of 170 cm and a standard deviation of 10 cm. It visually approximates a histogram with a blue bell-shaped curve, and highlights (in grey) the area under the curve between $\mu \pm 2 \times \sigma$. This area corresponds to the probability of randomly selecting an individual whose height falls within this range.

By using the area under the curve, we can efficiently estimate probabilities without needing to collect and analyze a vast amount of data. This method not only saves time and resources but also provides a clear and intuitive way to understand and communicate statistical probabilities.

Definition 25.15 (Normal Distribution)**.** This variable is defined on the support $D_X = \mathbb{R}$ and its density function is given by

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\}.$$

The density function is identified by the pair of parameters (μ, σ^2), where $\mu \in \mathbb{R}$ is the mean (or location parameter) and $\sigma^2 > 0$ is the variance (or dispersion parameter) of X. □

The density function is symmetric around μ. The normal distribution belongs to the location-scale family distributions. This means that, if $Z \sim N(0,1)$ (read, Z has a standard normal distribution; i.e., with $\mu = 0$ and $\sigma^2 = 1$), and we consider the linear transformation $X = \mu + \sigma Z$, then $X \sim N(\mu, \sigma^2)$ (read, X has a normal distribution with mean μ and variance σ^2). This means that one can obtain the probability of any interval $(-\infty, x]$, $x \in R$ for any normal distribution (i.e., for any pair of the parameters μ and σ) once the quantiles of the standard normal distribution are known. Indeed

$$\begin{aligned} F_X(x) &= \Pr\{X \leq x\} = \Pr\left\{\frac{X-\mu}{\sigma} \leq \frac{x-\mu}{\sigma}\right\} \\ &= \Pr\left\{Z \leq \frac{x-\mu}{\sigma}\right\} = F_Z\left(\frac{x-\mu}{\sigma}\right) \qquad x \in \mathbb{R}. \end{aligned}$$

The quantiles of the standard normal distribution are available in any statistical program. The density and cumulative distribution function of the standard normal r.v.~at point x are usually denoted by the symbols $\phi(x)$ and $\Phi(x)$.

The standard normal distribution is based on the *standard normal density function*

$$\varphi(z) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z^2}{2}\right). \tag{25.11}$$

An important application of the standardization introduced in Equation 25.11 reads as follows. In case the distribution of X is approximately normal, the distribution of X^{*} is approximately standard normal. That is

$$P(X \leq b) = P(\frac{X-\mu}{\sigma} \leq \frac{b-\mu}{\sigma}) = P(X^* \leq \frac{b-\mu}{\sigma})$$

The probability $P(X \leq b)$ can be approximated by $\Phi(\frac{b-\mu}{\sigma})$, where Φ is the standard normal cumulative distribution function.

If X is a normal random variable with mean μ and variance σ^2, i.e., $X \sim N(\mu, \sigma^2)$, then

$$X = \mu + \sigma Z \text{ where } Z \sim N(0,1). \tag{25.12}$$

If $Z \sim N(0,1)$ and $X \sim N(\mu, \sigma^2)$, then

$$X = \mu + \sigma Z.$$

The probability of getting a value in a particular interval is the area under the corresponding part of the curve. Consider the density function of the normal distribution. It can be plotted using the following commands. The result is shown in Figure 25.3.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
x = np.arange(-4, 4, 0.1)
# Calculating the normal distribution's density function values for each point in x
y = norm.pdf(x, 0, 1)
plt.plot(x, y, linestyle='-', linewidth=2)
plt.title('Normal Distribution')
plt.xlabel('X')
plt.ylabel('Density')
plt.grid(True)
plt.show()
```

Figure 25.3.: Normal Distribution Density Function

The *cumulative distribution function* (CDF) describes the probability of "hitting" x or less in a given distribution. We consider the CDF function of the normal distribution. It can be plotted using the following commands. The result is shown in Figure 25.4.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Generating a sequence of numbers from -4 to 4 with 0.1 intervals
x = np.arange(-4, 4, 0.1)

# Calculating the cumulative distribution function value of the normal distribution fo
y = norm.cdf(x, 0, 1)  # mean=0, stddev=1

# Plotting the results. The equivalent of 'type="l"' in R (line plot) becomes the defa
plt.plot(x, y, linestyle='-', linewidth=2)
plt.title('Normal Distribution CDF')
plt.xlabel('X')
plt.ylabel('Cumulative Probability')
plt.grid(True)
plt.show()
```

Figure 25.4.: Normal Distribution Cumulative Distribution Function

25.11. The Exponential Distribution

The exponential distribution is a continuous probability distribution that describes the time between events in a Poisson process, where events occur continuously and independently at a constant average rate. It is characterized by a single parameter, the rate parameter λ, which represents the average number of events per unit time.

25.11.1. Standardization of Random Variables

To compare statistical properties of random variables which use different units, it is a common practice to transform these random variables into standardized variables.

Definition 25.16 (Standard Units)**.** If a random variable X has expectation $E(X) = \mu$ and standard deviation $sd(X) = \sigma > 0$, the random variable

$$X^* = (X - \mu)/\sigma$$

is called X in standard units. It has $E(X^*) = 0$ and $sd(X^*) = 1$.

25.12. Covariance and Correlation

25.12.1. The Multivariate Normal Distribution

The multivariate normal, multinormal, or Gaussian distribution serves as a generalization of the one-dimensional normal distribution to higher dimensions. We will consider n-dimensional random vectors $X = (X_1, X_2, \dots, X_n)$. When drawing samples from this distribution, it results in a set of values represented as $\{x_1, x_2, \dots, x_n\}$. To fully define this distribution, it is necessary to specify its mean μ and covariance matrix Σ. These parameters are analogous to the mean, which represents the central location, and the variance (squared standard deviation) of the one-dimensional normal distribution introduced in Equation 25.10.

Definition 25.17 (The Multivariate Normal Distribution)**.** The probability density function (PDF) of the multivariate normal distribution is defined as:

$$f_X(x) = \frac{1}{\sqrt{(2\pi)^n \det(\Sigma)}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right),$$

where: μ is the $n \times 1$ mean vector; Σ is the $n \times n$ covariance matrix. The covariance matrix Σ is assumed to be positive definite, so that its determinant is strictly positive.

In the context of the multivariate normal distribution, the mean takes the form of a coordinate within an k-dimensional space. This coordinate represents the location where samples are most likely to be generated, akin to the peak of the bell curve in a one-dimensional or univariate normal distribution.

Definition 25.18 (Covariance of two random variables)**.** For two random variables X and Y, the covariance is defined as the expected value (or mean) of the product of their deviations from their individual expected values:

$$\operatorname{cov}(X, Y) = \mathrm{E}\left[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])\right]$$

For discrete random variables, covariance can be written as:

$$\operatorname{cov}(X, Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - E(X))(y_i - E(Y)).$$

The covariance within the multivariate normal distribution denotes the extent to which two variables vary together. The elements of the covariance matrix, such as Σ_{ij}, represent the covariances between the variables x_i and x_j. These covariances describe how the different variables in the distribution are related to each other in terms of their variability.

Example 25.7 (The Bivariate Normal Distribution with Positive Covariances)**.** Figure 25.5 shows draws from a bivariate normal distribution with $\mu = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and $\Sigma = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$.

The covariance matrix of a bivariate normal distribution determines the shape, orientation, and spread of the distribution in the two-dimensional space.

The diagonal elements of the covariance matrix (σ_1^2, σ_2^2) are the variances of the individual variables. They determine the spread of the distribution along each axis. A larger variance corresponds to a greater spread along that axis.

The off-diagonal elements of the covariance matrix (σ_{12}, σ_{21}) are the covariances between the variables. They determine the orientation and shape of the distribution. If the covariance is positive, the distribution is stretched along the line $y = x$, indicating that the variables tend to increase together. If the covariance is negative, the distribution is stretched along the line $y = -x$, indicating that one variable tends to decrease as the other increases. If the covariance is zero, the variables are uncorrelated and the distribution is axis-aligned.

In Figure 25.5, the variances are identical and the variables are correlated (covariance is 4), so the distribution is stretched along the line $y = x$.

Figure 25.5.: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$

Figure 25.6.: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$.

Example 25.8 (The Bivariate Normal Distribution with Mean Zero and Zero Covariances)**.** The Bivariate Normal Distribution with Mean Zero and Zero Covariances $\sigma_{12} = \sigma_{21} = 0$.

$\Sigma = \begin{pmatrix} 9 & 0 \\ 0 & 9 \end{pmatrix}$

Figure 25.7.: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & 0 \\ 0 & 9 \end{pmatrix}$

Example 25.9 (The Bivariate Normal Distribution with Mean Zero and Negative Covariances)**.** The Bivariate Normal Distribution with Mean Zero and Negative Covariances $\sigma_{12} = \sigma_{21} = -4$.

$\Sigma = \begin{pmatrix} 9 & -4 \\ -4 & 9 \end{pmatrix}$

25.13. Covariance

In statistics, understanding the relationship between random variables is crucial for making inferences and predictions. Two common measures of such relationships are covariance and correlation. Covariance is a measure of how much two random variables change together. If the variables tend to show similar behavior (i.e., when one increases,

Figure 25.8.: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & -4 \\ -4 & 9 \end{pmatrix}$

the other tends to increase), the covariance is positive. Conversely, if they tend to move in opposite directions, the covariance is negative.

Definition 25.19 (Covariance)**.** Covariance is calculated as:

$$\mathrm{Cov}(X,Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y]$$

Here, $E[X]$ and $E[Y]$ are the expected values (means) of X and Y, respectively. Covariance has units that are the product of the units of X and Y.

For a vector of random variables $\mathbf{Y} = \left(Y^{(1)}, \dots, Y^{(n)}\right)^T$, the covariance matrix Σ encapsulates the covariances between each pair of variables:

$$\Sigma = \mathrm{Cov}(\mathbf{Y}, \mathbf{Y}) = \begin{pmatrix} \mathrm{Var}(Y^{(1)}) & \mathrm{Cov}(Y^{(1)}, Y^{(2)}) & \dots \\ \mathrm{Cov}(Y^{(2)}, Y^{(1)}) & \mathrm{Var}(Y^{(2)}) & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

The diagonal elements represent the variances, while the off-diagonal elements are the covariances.

25.14. Correlation

25.14.1. Definitions

Definition 25.20 ((Pearson) Correlation Coefficient)**.** The Pearson correlation coefficient, often denoted by ρ for the population or r for a sample, is calculated by dividing the covariance of two variables by the product of their standard deviations.

$$\rho_{XY} = \frac{\mathrm{Cov}(X,Y)}{\sigma_X \sigma_Y}, \tag{25.13}$$

where $\mathrm{Cov}(X,Y)$ is the covariance between variables X and Y, and σ_X and σ_Y are the standard deviations of X and Y, respectively.

Correlation, specifically the correlation coefficient, is a normalized measure of the linear relationship between two variables. It provides a value ranging from -1 to 1, which is scale-free, making it easier to interpret:

- -1: Perfect negative correlation, indicating that as one variable increases, the other decreases.
- 0: No correlation, indicating no linear relationship between the variables.
- 1: Perfect positive correlation, indicating that both variables increase together.

The correlation matrix Ψ provides a way to quantify the linear relationship between multiple variables, extending the notion of the correlation coefficient beyond just pairs of variables. It is derived from the covariance matrix Σ by normalizing each element with respect to the variances of the relevant variables.

Definition 25.21 (The Correlation Matrix Ψ)**.** Given a set of random variables $X_1, X_2, \dots, X_n$, the covariance matrix Σ is:

$$\Sigma = \begin{pmatrix} \sigma_{11} & \sigma_{12} & \cdots & \sigma_{1n} \\ \sigma_{21} & \sigma_{22} & \cdots & \sigma_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{n1} & \sigma_{n2} & \cdots & \sigma_{nn} \end{pmatrix},$$

where $\sigma_{ij} = \mathrm{cov}(X_i, X_j)$ is the covariance between the ith and j^{th} variables. The correlation matrix Ψ is then defined as:

$$\Psi = \left(\rho_{ij}\right) = \left(\tfrac{\sigma_{ij}}{\sqrt{\sigma_{ii}\sigma_{jj}}}\right),$$

where:

- ρ_{ij} is the correlation coefficient between the ith and j^{th} variables.
- σ_{ii} is the variance of the ith variable, i.e., σ_i^2.
- $\sqrt{\sigma_{ii}}$ is the standard deviation of the ith variable, denoted as σ_i.

Thus, Ψ can also be expressed as:

$$\Psi = \begin{pmatrix} 1 & \frac{\sigma_{12}}{\sigma_1\sigma_2} & \cdots & \frac{\sigma_{1n}}{\sigma_1\sigma_n} \\ \frac{\sigma_{21}}{\sigma_2\sigma_1} & 1 & \cdots & \frac{\sigma_{2n}}{\sigma_2\sigma_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\sigma_{n1}}{\sigma_n\sigma_1} & \frac{\sigma_{n2}}{\sigma_n\sigma_2} & \cdots & 1 \end{pmatrix}$$

The correlation matrix Ψ has the following properties:

- The matrix Ψ is symmetric, meaning $\rho_{ij} = \rho_{ji}$.
- The diagonal elements are all 1, as $\rho_{ii} = \frac{\sigma_{ii}}{\sigma_i\sigma_i} = 1$.
- Each off-diagonal element is constrained between -1 and 1, indicating the strength and direction of the linear relationship between pairs of variables.

25.14.2. Computations

Example 25.10 (Computing a Correlation Matrix)**.** Suppose you have a dataset consisting of three variables: X, Y, and Z. You can compute the correlation matrix as follows:

1. Calculate the covariance matrix Σ, which contains covariances between all pairs of variables.
2. Extract the standard deviations for each variable from the diagonal elements of Σ.
3. Use the standard deviations to compute the correlation matrix Ψ.

Suppose we have two sets of data points:

- $X = [1, 2, 3]$
- $Y = [4, 5, 6]$

We want to compute the correlation matrix Ψ for these variables. First, calculate the mean of each variable.

$$\bar{X} = \frac{1+2+3}{3} = 2$$

$$\bar{Y} = \frac{4+5+6}{3} = 5$$

Second, compute the covariance between X and Y. The covariance is calculated as:

$$\text{Cov}(X, Y) = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y})$$

For our data:

$$\text{Cov}(X, Y) = \frac{1}{3-1} \left[(1-2)(4-5) + (2-2)(5-5) + (3-2)(6-5)\right]$$

$$= \frac{1}{2} \left[1 + 0 + 1\right] = 1$$

Third, calculate the variances of X and Y. Variance is calculated as:

$$\text{Var}(X) = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{X})^2$$

$$= \frac{1}{2}\left[(1-2)^2 + (2-2)^2 + (3-2)^2\right] = \frac{1}{2}(1+0+1) = 1$$

Similarly,

$$\mathrm{Var}(Y) = \frac{1}{2}\left[(4-5)^2 + (5-5)^2 + (6-5)^2\right] = \frac{1}{2}(1+0+1) = 1$$

Then, compute the correlation coefficient. The correlation coefficient ρ_{XY} is:

$$\rho_{XY} = \frac{\mathrm{Cov}(X,Y)}{\sqrt{\mathrm{Var}(X)} \cdot \sqrt{\mathrm{Var}(Y)}}$$

$$= \frac{1}{\sqrt{1} \cdot \sqrt{1}} = 1$$

Finally, construct the correlation matrix. The correlation matrix Ψ is given as:

$$\Psi = \begin{pmatrix} 1 & \rho_{XY} \\ \rho_{XY} & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

Thus, for these two variables, the correlation matrix indicates a perfect positive linear relationship (correlation coefficient of 1) between X and Y.

25.14.3. The Outer-product and the `np.outer` Function

The function `np.outer` from the NumPy library computes the outer product of two vectors. The outer product of two vectors results in a matrix, where each element is the product of an element from the first vector and an element from the second vector.

Definition 25.22 (Outer Product)**.** For two vectors $\mathbf{a}$ and $\mathbf{b}$, the outer product is defined in terms of their elements as:

$$\mathrm{outer}(\mathbf{a}, \mathbf{b}) = \begin{pmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & \cdots & a_1 \cdot b_n \\ a_2 \cdot b_1 & a_2 \cdot b_2 & \cdots & a_2 \cdot b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_m \cdot b_1 & a_m \cdot b_2 & \cdots & a_m \cdot b_n \end{pmatrix},$$

where $\mathbf{a}$ is a vector of length m and $\mathbf{b}$ is a vector of length n.

Example 25.11 (Computing the Outer Product)**.** We will consider two vectors, **a** and **b**:

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5])
outer_product = np.outer(a, b)
print("Vector a:", a)
print("Vector b:", b)
print("Outer Product:\n", outer_product)
```

```
Vector a: [1 2 3]
Vector b: [4 5]
Outer Product:
 [[ 4  5]
 [ 8 10]
 [12 15]]
```

For the vectors defined:

$$\mathbf{a} = [1, 2, 3], \quad \mathbf{b} = [4, 5]$$

The outer product will be:

$$\begin{pmatrix} 1 \cdot 4 & 1 \cdot 5 \\ 2 \cdot 4 & 2 \cdot 5 \\ 3 \cdot 4 & 3 \cdot 5 \end{pmatrix} = \begin{pmatrix} 4 & 5 \\ 8 & 10 \\ 12 & 15 \end{pmatrix}.$$

Thus, `np.outer` creates a matrix with dimensions $m \times n$, where m is the length of the first vector and n is the length of the second vector. The function is particularly useful in various mathematical and scientific computations where matrix representations of vector relationships are needed.

Example 25.12 (Computing the Covariance and the Correlation Matrix)**.** The following Python code computes the covariance and correlation matrices using the NumPy library.

```
import numpy as np

def calculate_cov_corr_matrices(data, rowvar=False)->(np.array, np.array):
    """
    Calculate the covariance and correlation matrices of the input data.
```

```
    Args:
        data (np.array):
            Input data array.
        rowvar (bool):
            Whether the data is row-wise or column-wise.
            Default is False (column-wise).

    Returns:
        np.array: Covariance matrix.
        np.array: Correlation matrix.

    Examples:
        >>> data = np.array([[1, 2, 3],
                             [4, 5, 6],
                             [7, 8, 9]])
        >>> calculate_cov_corr_matrices(data)
    """
    cov_matrix = np.cov(data, rowvar=rowvar)
    std_devs = np.sqrt(np.diag(cov_matrix))
    # check whether the standard deviations are zero
    # and throw an error if they are
    if np.any(std_devs == 0):
        raise ValueError("Correlation matrix cannot be computed,"+
                         " because one or more variables have zero variance.")
    corr_matrix = cov_matrix / np.outer(std_devs, std_devs)
    return cov_matrix, corr_matrix
```

```
A = np.array([[0,1],
                 [1,0]])
print(f"Input matrix:\n {A}")
Sigma, Psi = calculate_cov_corr_matrices(A)
print(f"Covariance matrix:\n {Sigma}")
print(f"Correlation matrix:\n {Psi}")
```

```
Input matrix:
 [[0 1]
 [1 0]]
Covariance matrix:
 [[ 0.5 -0.5]
 [-0.5  0.5]]
Correlation matrix:
 [[ 1. -1.]
 [-1.  1.]]
```

25.14.4. Correlation and Independence

Definition 25.23 (Statistical Independence (Independence of Random Vectors))**.** Two random vectors are statistically independent if the joint distribution of the vectors is equal to the product of their marginal distributions.

This means that knowing the realization of one vector gives you no information about the realization of the other vector. This independence is a probabilistic concept used in statistics and probability theory to denote that two sets of random variables do not affect each other. Independence implies that all pairwise covariances between the components of the two vectors are zero, but zero covariance does not imply independence unless certain conditions are met (e.g., normality). Statistical independence is a stronger condition than zero covariance. Statistical independence is not related to the linear independence of vectors in linear algebra.

Example 25.13 (Covariance of Independent Variables)**.** Consider a covariance matrix where variables are independent:

```
A = np.array([[1,-1],
[2,0],
[3,1],
[4,0],
[5,-1]])
print(f"Input matrix:\n {A}")
Sigma, Psi = calculate_cov_corr_matrices(A)
print(f"Covariance matrix:\n {Sigma}")
print(f"Correlation matrix:\n {Psi}")
```

```
Input matrix:
 [[ 1 -1]
 [ 2  0]
 [ 3  1]
 [ 4  0]
 [ 5 -1]]
Covariance matrix:
 [[2.5 0. ]
 [0.  0.7]]
Correlation matrix:
 [[1. 0.]
 [0. 1.]]
```

Here, since the off-diagonal elements are 0, the variables are uncorrelated. X increases linearly, while Y alternates in a simple pattern with no trend that is linearly related to Y.

Example 25.14 (Strong Correlation)**.** For a covariance matrix with strong positive correlation:

```
A = np.array([[10,-1],
[20,0],
[30,1],
[40,2],
[50,3]])
print(f"Input matrix:\n {A}")
Sigma, Psi = calculate_cov_corr_matrices(A)
print(f"Covariance matrix:\n {Sigma}")
print(f"Correlation matrix:\n {Psi}")
```

```
Input matrix:
 [[10 -1]
 [20  0]
 [30  1]
 [40  2]
 [50  3]]
Covariance matrix:
 [[250.   25. ]
 [ 25.    2.5]]
Correlation matrix:
 [[1. 1.]
 [1. 1.]]
```

A value close to 1 suggests a strong positive relationship between the variables.

Example 25.15 (Strong Negative Correlation)**.**

```
A = np.array([[10,1],
[20,0],
[30,-1],
[40,-2],
[50,-3]])
print(f"Input matrix:\n {A}")
Sigma, Psi = calculate_cov_corr_matrices(A)
print(f"Covariance matrix:\n {Sigma}")
print(f"Correlation matrix:\n {Psi}")
```

```
Input matrix:
 [[10  1]
 [20  0]
```

```
 [30 -1]
 [40 -2]
 [50 -3]]
Covariance matrix:
 [[250.  -25. ]
 [-25.    2.5]]
Correlation matrix:
 [[ 1. -1.]
 [-1.  1.]]
```

This matrix indicates a perfect negative correlation where one variable increases as the other decreases.

25.14.5. Pearson's Correlation

- Video: [Pearson's Correlation, Clearly Explained]

25.14.6. Interpreting the Correlation: Correlation Squared

Rummel (1976) describes how to interpret correlations as follows:

Seldom, indeed, will a correlation be zero or perfect. Usually, the covariation between things will be something like .56 or $-.16$. Clearly .56 is positive, indicating positive covariation; $-.16$ is negative, indicating some negative covariation. Moreover, we can say that the positive correlation is greater than the negative. But, we require more than. If we have a correlation of .56 between two variables, for example, what precisely can we say other than the correlation is positive and .56? The squared correlation describes the proportion of variance in common between the two variables. If we multiply this by 100 we then get the percent of variance in common between two variables. That is:

$$r_{XY}^2 \times 100 = \text{percent of variance in common between} X \text{ and } Y.$$

For example, we found that the correlation between a nation's power and its defense budget was .66. This correlation squared is .45, which means that across the fourteen nations constituting the sample 45 percent of their variance on the two variables is in common (or 55 percent is not in common). In thus squaring correlations and transforming covariance to percentage terms we have an easy to understand meaning of correlation. And we are then in a position to evaluate a particular correlation. As a matter of routine it is the squared correlations that should be interpreted. This is because the correlation coefficient is misleading in suggesting the existence of more covariation than exists, and this problem gets worse as the correlation approaches zero.

Example 25.16 (The relationship between study time and test scores)**.** Imagine we are examining the relationship between the number of hours students study for a subject (Variable A) and their scores on a test (Variable B). After analyzing the data, we calculate a correlation of 0.8 between study time and test scores. When we square this correlation coefficient ($0.8^2 = 0.64$), we get 0.64 or 64%. This means that 64% of the variability in test scores can be accounted for by the variability in study hours. This indicates that a substantial part of why students score differently on the test can be explained by how much they studied. However, there remains 36% of the variability in test scores that needs to be explained by other factors, such as individual abilities, the difficulty of the test, or other external influences.

25.14.7. Partial Correlation

Often, a correlation between two variables X and Y can be found only because both variables are correlated with a third variable Z. The correlation between X and Y is then a spurious correlation. Therefore, it is often of interest to determine the correlation between X and Y while partializing a variable Z, i.e., the correlation between X and Y that exists without the influence of Z. Such a correlation $\rho_{(X,Y)/Z}$ is called the partial correlation of X and Y while holding Z constant. It is given by

$$\rho_{(X,Y)/Z} = \frac{\rho_{XY} - \rho_{XZ}\rho_{YZ}}{\sqrt{(1-\rho_{XZ}^2)(1-\rho_{YZ}^2)}}, \tag{25.14}$$

where ρ_{XY} is the correlation between X and Y, ρ_{XZ} is the correlation between X and Z, and ρ_{YZ} is the correlation between Y and Z (Hartung, Elpert, and Klösener 1995).

If the variables X, Y and Z are jointly normally distributed in the population of interest, one can estimate $\rho_{(X,Y)/Z}$ based on n realizations $x_1, \dots, x_n$, $y_1, \dots, y_n$ and $z_1, \dots, z_n$ of the random variables X, Y and Z by replacing the simple correlations ρ_{XY}, ρ_{XZ} and ρ_{YZ} with the empirical correlations $\hat{\rho}_{XY}$, $\hat{\rho}_{XZ}$ and $\hat{\rho}_{YZ}$. The partial correlation coefficient $\hat{\rho}_{(X,Y)/Z}$ is then estimated using

$$r_{(X,Y)/Z} = \frac{r_{XY} - r_{XZ}r_{YZ}}{\sqrt{(1-r_{XZ}^2)(1-r_{YZ}^2)}}. \tag{25.15}$$

Based on this estimated value for the partial correlation, a test at the α level for partial uncorrelatedness or independence of X and Y under Z can also be carried out. The hypothesis

$$H_0 : \rho_{(X,Y)/Z} = 0 \tag{25.16}$$

is tested against the alternative

$$H_1 : \rho_{(X,Y)/Z} \neq 0 \tag{25.17}$$

at the level α is discarded if

$$\left| \frac{r_{(X,Y)/Z}\sqrt{n-3}}{\sqrt{1-r^2_{(X,Y)/Z}}} \right| > t_{n-3,1-\alpha/2}$$

applies. Here $t_{n-3,1-\alpha/2}$ is the $(1-\alpha/2)$-quantile of the t-distribution with $n-3$ degrees of freedom.

Example 25.17. For example, given economic data on the consumption X, income Y, and wealth Z of various individuals, consider the relationship between consumption and income. Failing to control for wealth when computing a correlation coefficient between consumption and income would give a misleading result, since income might be numerically related to wealth which in turn might be numerically related to consumption; a measured correlation between consumption and income might actually be contaminated by these other correlations. The use of a partial correlation avoids this problem (Wikipedia contributors 2024).

Example 25.18 (Partial Correlation. Numerical Example)**.** Given the following data, calculate the partial correlation between A and B, controlling for C.

$$A = \begin{pmatrix} 2 \\ 4 \\ 15 \\ 20 \end{pmatrix}, \quad B = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}, \quad C = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

```
from spotpython.utils.stats import partial_correlation
import numpy as np
import pandas as pd
data = pd.DataFrame({
    'A': [2, 4, 15, 20],
    'B': [1, 2, 3, 4],
    'C': [0, 0, 1, 1]
})
print(f"Correlation between A and B: {data['A'].corr(data['B'])}")
pc = partial_correlation(data, method='pearson')
print(f"Partial Correlation between A and B: {pc["estimate"][0, 1]}")
```

```
Correlation between A and B: 0.9695015519208121
Partial Correlation between A and B: 0.9191450300180576
```

Instead of considering only one variable Z, multiple variables Z_i can be considered. The formal definiton of partial correlation reads as follows:

Definition 25.24 (Partial Correlation)**.** Formally, the partial correlation between X and Y given a set of n controlling variables $\mathbf{Z} = \{Z_1, Z_2, \dots, Z_n\}$, written $\rho_{XY \cdot \mathbf{Z}}$, is the correlation between the residuals e_X and e_Y resulting from the linear regression of X with $\mathbf{Z}$ and of Y with $\mathbf{Z}$, respectively. The first-order partial correlation (i.e., when $n = 1$) is the difference between a correlation and the product of the removable correlations divided by the product of the coefficients of alienation of the removable correlations (Wikipedia contributors 2024).

Like the correlation coefficient, the partial correlation coefficient takes on a value in the range from -1 to 1. The value -1 conveys a perfect negative correlation controlling for some variables (that is, an exact linear relationship in which higher values of one variable are associated with lower values of the other); the value 1 conveys a perfect positive linear relationship, and the value 0 conveys that there is no linear relationship (Wikipedia contributors 2024).

Lemma 25.1 (Matrix Representation of the Partial Correlation)**.** *The partial correlation can also be written in terms of the joint precision matrix (Wikipedia contributors 2024). Consider a set of random variables,* $\mathbf{V} = \{X_1, \dots, X_n\}$ *of cardinality* n*. We want the partial correlation between two variables* X_i *and* X_j *given all others, i.e.,* $\mathbf{V} \setminus \{X_i, X_j\}$*. Suppose the (joint/full) covariance matrix* $\Sigma = (\sigma_{ij})$ *is positive definite and therefore invertible. If the precision matrix is defined as* $\Omega = (p_{ij}) = \Sigma^{-1}$*, then*

$$\rho_{X_i X_j \cdot \mathbf{V} \setminus \{X_i, X_j\}} = -\frac{p_{ij}}{\sqrt{p_{ii} p_{jj}}} \tag{25.18}$$

The semipartial correlation statistic is similar to the partial correlation statistic; both compare variations of two variables after certain factors are controlled for. However, to calculate the semipartial correlation, one holds the third variable constant for either X or Y but not both; whereas for the partial correlation, one holds the third variable constant for both (Wikipedia contributors 2024).

25.15. Hypothesis Testing and the Null-Hypothesis

25.15.1. Alternative Hypotheses, Main Ideas

25.15.2. p-values: What they are and how to interpret them

25.15.2.1. How to calculate p-values

25.15.2.2. p-hacking: What it is and how to avoid it

25.16. Statistical Power

- Video: Statistical Power, Clearly Explained

25.16.0.1. Power Analysis

- Video: Power Analysis, Clearly Explained!!!

25.17. The Central Limit Theorem

- Video: The Central Limit Theorem, Clearly Explained!!!

25.18. Maximum Likelihood

Maximum likelihood estimation is a method used to estimate the parameters of a statistical model. It is based on the principle of choosing the parameter values that maximize the likelihood of the observed data. The likelihood function represents the probability of observing the data given the model parameters. By maximizing this likelihood, we can find the parameter values that best explain the observed data.

Example 25.19 (Maximum Likelihood Estimation: Bernoulli Experiment)**.** Bernoulli experiment for the event A, repeated n times, with the probability of success p. Result given as n tuple with entries A and $\overline{A}$. A appears k times. The probability of this event is given by

$$L(p) = p^k(1-p)^{n-k} \tag{25.19}$$

Applying maximum likelihood estimation, we find the maximum of the likelihood function $L(p)$, i.e., we are trying to find the value of p that maximizes the probability of observing the data. This value will be denoted as $\hat{p}$.

Differentiating the likelihood function with respect to p and setting the derivative to zero, we find the maximum likelihood estimate $\hat{p}$. We get

$$\frac{d}{dp}L(p) = kp^{k-1}(1-p)^{n-k} - p^k(n-k)(1-p)^{n-k-1} \tag{25.20}$$

$$= p^{k-1}(1-p)^{n-k-1}\left(k(1-p) - p(n-k)\right) = 0 \tag{25.21}$$

Because

$$p \neq 0 \text{ and } (1-p)p \neq 0,$$

we can divide by $p^{k-1}(1-p)^{n-k-1}$ and get

$$k(1-p) - p(n-k) = 0. \tag{25.22}$$

Solving for p gives

$$\hat{p} = \frac{k}{n} \tag{25.23}$$

Therefore, the maximum likelihood estimate for the probability of success in a Bernoulli experiment is the ratio of the number of successes to the total number of trials.

□

Example 25.20 (Maximum Likelihood Estimation: Normal Distribution)**.** Random variable $X \sim \mathcal{N}(\mu, \sigma^2)$ with n observations $x_1, x_2, \dots, x_n$. The likelihood function is given by

$$L(x_1, x_2, \dots, x_n, \mu, \sigma^2) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_i-\mu)^2}{2\sigma^2}\right) \tag{25.24}$$

Taking the logarithm of the likelihood function, we get

$$\log L(x_1, x_2, \dots, x_n, \mu, \sigma^2) = -\frac{n}{2}\log(2\pi) - \frac{n}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2 \tag{25.25}$$

Partial derivative with respect to μ is

$$\frac{\partial}{\partial\mu}\log L(x_1, x_2, \dots, x_n, \mu, \sigma^2) = \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i-\mu) = 0 \tag{25.26}$$

We obtain the maximum likelihood estimate for μ as

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i, \tag{25.27}$$

which is the sample mean $\overline{x}$.

The partial derivative with respect to σ^2 is

$$\frac{\partial}{\partial \sigma^2} \log L(x_1, x_2, \dots, x_n, \mu, \sigma^2) = -\frac{n}{2\sigma^2} + \frac{1}{2(\sigma^2)^2} \sum_{i=1}^{n} (x_i - \mu)^2 = 0 \tag{25.28}$$

This can be simplified to

$$-n + \frac{1}{\sigma^2} \sum_{i=1}^{n} (x_i - \mu)^2 = 0 \tag{25.29}$$

$$\Rightarrow n\sigma^2 = \sum_{i=1}^{n} (x_i - \mu)^2 \tag{25.30}$$

Using the maximum likelihood estimate for μ from Equation 25.27, we get

$$\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \hat{\mu})^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \overline{x})^2 \tag{25.31}$$

$$= \frac{n-1}{n} \frac{\sum_{i=1}^{n} (x_i - \hat{\mu})^2}{n-1} = \frac{n-1}{n} s^2, \tag{25.32}$$

where

$$s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})}{n-1}} \tag{25.33}$$

is the sample standard deviation. We obtain the maximum likelihood estimate for σ^2 as

$$\hat{\sigma}^2 = \frac{n-1}{n} s^2 \tag{25.34}$$

- Video: Maximum Likelihood, clearly explained!!!
- Video: Probability is not Likelihood. Find out why!!!

25.19. Maximum Likelihood Estimation: Multivariate Normal Distribution

25.19.1. The Joint Probability Density Function of the Multivariate Normal Distribution

Consider the first n terms of an identically and independently distributed (i.i..d.) sequence $X^{(j)}$ of k-dimensional multivariate normal random vectors, i.e.,

$$X^{(j)} \sim N(\mu, \Sigma), j = 1, 2, \dots . \tag{25.35}$$

The joint probability density function of the j-th term of the sequence is

$$f_X(x_j) = \frac{1}{\sqrt{(2\pi)^k \det(\Sigma)}} \exp\left(-\frac{1}{2}(x_j - \mu)^T \Sigma^{-1} (x_j - \mu)\right),$$

where: μ is the $k \times 1$ mean vector; Σ is the $k \times k$ covariance matrix. The covariance matrix Σ is assumed to be positive definite, so that its determinant is strictly positive. We use $x_1, \dots x_n$, i.e., the realizations of the first n random vectors in the sequence, to estimate the two unknown parameters μ and Σ.

25.19.2. The Log-Likelihood Function

Definition 25.25 (Likelihood Function)**.** The likelihood function is defined as the joint probability density function of the observed data, viewed as a function of the unknown parameters.

Since the terms in the sequence Equation 25.35 are independent, their joint density is equal to the product of their marginal densities. As a consequence, the likelihood function can be written as the product of the individual densities:

$$\begin{aligned} L(\mu, \Sigma) &= \prod_{j=1}^{n} f_X(x_j) = \prod_{j=1}^{n} \frac{1}{\sqrt{(2\pi)^k \det(\Sigma)}} \exp\left(-\frac{1}{2}(x_j - \mu)^T \Sigma^{-1} (x_j - \mu)\right) \\ &= \frac{1}{(2\pi)^{nk/2} \det(\Sigma)^{n/2}} \exp\left(-\frac{1}{2} \sum_{j=1}^{n} (x_j - \mu)^T \Sigma^{-1} (x_j - \mu)\right). \end{aligned} \tag{25.36}$$

Taking the natural logarithm of the likelihood function, we obtain the log-likelihood function:

Example 25.21 (Log-Likelihood Function of the Multivariate Normal Distribution)**.** The log-likelihood function of the multivariate normal distribution is given by

$$\ell(\mu, \Sigma) = -\frac{nk}{2} \ln(2\pi) - \frac{n}{2} \ln(\det(\Sigma)) - \frac{1}{2} \sum_{j=1}^{n} (x_j - \mu)^T \Sigma^{-1} (x_j - \mu).$$

The likelihood function is well-defined only if $\det(\Sigma) > 0$.

25.20. Cross-Validation

- Video: Machine Learning Fundamentals: Cross Validation

25.20.0.1. Bias and Variance

- Video: Machine Learning Fundamentals: Bias and Variance

25.21. Mutual Information

R^2 works only for numerical data. Mutual information can also be used, when the dependent variable is boolean or categorical. Mutual information provides a way to quantify the relationship of a mixture of continuous and discrete variables to a target variable. Mutual information explains how closely related two variables are. It is a measure of the amount of information that one variable provides about another variable.

Definition 25.26 (Mutual Information)**.**

$$\mathrm{MI} = \sum_{x \in X} \sum_{y \in Y} p(x, y) \log \left(\frac{p(x, y)}{p(x)p(y)} \right) \tag{25.37}$$

The terms in the nominator and denominator in Equation 25.37 are the joint probability, $p(x, y)$, the marginal probability of X, $p(x)$ and the marginal probability of Y, $p(y)$ respectively. Joint probabilites are the probabilities of two events happening together. Marginal probabilities are the probabilities of individual events happening. If $p(x, y) = p(x)p(y)$, then X and Y are independent. In this case, the mutual information is one.

The mutual information of two variables X and Y is

$$\mathrm{MI}(X, Y) = p(X, Y) \log \left(\frac{p(X, Y)}{p(X)p(Y)} \right) + p(X, \overline{Y}) \log \left(\frac{p(X, \overline{Y})}{p(X)p(\overline{Y})} \right) + \tag{25.38}$$

$$p(\overline{X}, Y) \log \left(\frac{p(\overline{X}, Y)}{p(\overline{X})p(Y)} \right) + p(\overline{X}, \overline{Y}) \log \left(\frac{p(\overline{X}, \overline{Y})}{p(\overline{X})p(\overline{Y})} \right) \tag{25.39}$$

In general, when at least one of the two features has no variance (i.e., it is constant), the mutual information is zero, because something that never changes cannot tell us about something that does. When two features change, but change in exact the same way, then MI is 1/2. When two factors change, but in exact the opposite way, then MI is 1/2. When both features change, it does not matter if they change in the exact same or exact opposite ways; both result in the same MI value

Mutual Information

- Video: Mutual Information, Clearly Explained

25.22. t-SNE

- Video: t-SNE, Clearly Explained

26. Hypothesis Testing

26.1. Paired-Sample Analysis

The paired-sample experiment is characterized by a structure where each observation in one sample is uniquely and non-randomly coupled with an observation in another sample. This coupling is not a matter of convenience but a deliberate methodological choice intended to control for extraneous variability. Common applications include "before-and-after" studies, where the same subjects are measured prior to and following an intervention; studies involving naturally matched pairs, such as twins or spouses; or scenarios where two different measurement techniques are applied to the same set of specimens.

The analytical elegance of the paired design lies in its transformation of a two-sample comparison into a more straightforward one-sample problem. Rather than analyzing the distributions of the two raw measurements, x_1 and x_2, the procedure begins by calculating a single vector of difference scores for each pair, $d_i = x_{1i} - x_{2i}$.

The entire statistical inference is then conducted on this sample of differences, focusing on a single population parameter: the mean of the paired differences, denoted as μ_d.

The research question is thus reframed from "Is the mean of x_1 different from the mean of x_2?" to "Is the mean of the differences, μ_d, different from a specified value?" In the vast majority of cases, this specified value is zero, representing the null hypothesis of no difference between the paired measurements.

The primary statistical advantage of this transformation is a significant enhancement in statistical power, which is the ability of a test to detect a true effect. This increase in power is a direct mathematical consequence of the design's structure, because the variance of the difference between two random variables, x_1 and x_2, is defined by the formula:

$$\mathrm{Var}(x_1 - x_2) = \mathrm{Var}(x_1) + \mathrm{Var}(x_2) - 2 \cdot \mathrm{Cov}(x_1, x_2).$$

In a paired-sample design, the measurements x_1 and x_2 are derived from the same or closely matched subjects, which typically induces a positive correlation between them. This positive correlation results in a positive covariance term, $\mathrm{Cov}(x_1, x_2)$. As this term is subtracted in the variance formula, the resulting variance of the difference scores, $\mathrm{Var}(d)$, is smaller than the sum of the individual variances that would be relevant in an independent-samples design. A smaller variance leads to a smaller standard error of the mean difference, which in turn produces a larger test statistic for a given observed

difference. This amplified "signal-to-noise" ratio makes it more likely that a true effect will be deemed statistically significant, thereby increasing the power of the test.

The paired design should therefore be understood not merely as a data structure but as an active experimental control strategy. Much of the "noise" or unexplained variance in an experiment stems from stable, pre-existing differences between individual subjects (e.g., genetic predispositions, baseline health, cognitive ability). By using each subject as its own control, the paired design effectively isolates the effect of the intervention from this inter-subject variability.

Paired designs are more powerful, they often require a smaller sample size to detect an effect of a given magnitude compared to an independent-groups design.

26.1.1. Directional Hypotheses for a Paired Experiment

The formulation of statistical hypotheses is the formal process of translating a research question into a testable claim about a population parameter. This process is governed by a strict logical framework involving two competing statements: the null hypothesis (H_0) and the alternative hypothesis (H_1). The null hypothesis represents the claim of "no effect" or the status quo. It is the default position that is assumed to be true unless sufficient evidence is presented to refute it. In many experimental contexts, H_0 is the proposition that the researcher aims to reject. The alternative hypothesis, conversely, embodies the research hypothesis—it is the claim that an effect exists, for which the researcher is seeking evidence.

A critical requirement of this framework is that H_0 and H_1 must be mutually exclusive and exhaustive. This means that they cannot both be true simultaneously, and together they must cover all possible values of the population parameter under investigation.

26.1.1.1. Defining the Parameter of Interest and the Alternative Hypothesis

For a paired-sample experiment, the statistical inference does not focus on the individual population means of the two measurements (μ_{x_1} and μ_{x_2}). Instead, the central parameter of interest is the population mean of the paired differences, $\mu_d = \mu_{x_1} - \mu_{x_2}$. The user's specific research question—that x_1 is better than x_2, where "better" is defined as a smaller value—must be translated into a precise mathematical statement about this parameter. The claim x_1 is smaller than x_2 corresponds to the population-level statement $\mu_{x_1} < \mu_{x_2}$. By rearranging this inequality, we can express it in terms of the parameter μ_d: $\mu_{x_1} - \mu_{x_2} < 0$, which is equivalent to $\mu_d < 0$.

This statement, $\mu_d < 0$, becomes the alternative hypothesis (H_1). It is the specific, directional claim that the experiment is designed to support. The formulation is as follows:

Alternative Hypothesis,

$$H_1 : \mu_d < 0$$

This hypothesis dictates that the statistical test will be a one-tailed test, specifically a lower-tailed or left-tailed test. The term "one-tailed" signifies that the researcher is interested only in evidence indicating a difference in a particular direction (in this case, a negative difference). The entire region of rejection for the null hypothesis will be located in the extreme left tail of the test statistic's sampling distribution.

26.1.1.2. Establishing the Null Hypothesis

Once the alternative hypothesis is defined, the null hypothesis is constructed as its logical complement.

The composite null hypothesis includes the boundary case of zero difference as well as any possible difference in the direction opposite to the alternative hypothesis:

Composite Null: $H_0 : \mu_d \geq 0$

This statement posits that the mean difference is either zero or positive (i.e., x_1 is not smaller than x_2).

26.1.1.3. The Critical Justification for a One-Tailed Test

The choice to conduct a one-tailed test is a significant methodological decision that must be justified a priori—that is, before the data are collected or analyzed. A one-tailed test offers a distinct advantage in statistical power. By concentrating the entire significance level (alpha, α) in one tail of the distribution, the test is more sensitive to detecting an effect in the specified direction. However, this increased power comes at a substantial cost: the test is rendered completely incapable of detecting a statistically significant effect in the opposite direction, no matter how large that effect might be.

Therefore, a one-tailed test is only appropriate when there is a strong theoretical foundation or overwhelming prior evidence suggesting that an effect in the opposite direction is either impossible or of no substantive interest.

26.1.1.4. One-tailed Test Justification Example

If a new manufacturing process is being tested that can only possibly improve efficiency or have no effect, but cannot logically make it worse, a one-tailed test might be justified. However, consider a scenario where a new drug is developed with the belief that it will be an improvement over an existing drug. Opting for a one-tailed test to maximize the chances of detecting this improvement would mean failing to test for the possibility that the new drug is actually less effective or even harmful. Such an oversight could have serious ethical and practical consequences.

26.1.2. The Paired Samples t-Test: A Technical Examination

The paired samples t-test is the statistical procedure used to evaluate the hypotheses formulated for a paired experimental design. It is a parametric test, meaning its validity relies on certain assumptions about the underlying data. Crucially, these assumptions apply not to the raw scores of the two measurements, but to the calculated difference scores (d) between the pairs.

26.1.2.1. 3.1 Core Assumptions of the Paired t-Test

The validity of the conclusions drawn from a paired t-test is contingent upon the satisfaction of four primary assumptions.

- Continuous Data: The dependent variable, and therefore the calculated differences, must be measured on a continuous scale, which includes interval and ratio levels of measurement. Examples include variables like temperature, weight, blood pressure, or test scores.
- Independence of Pairs: The observations between pairs must be independent of one another. For instance, the outcome for one subject in a before-after study should not influence the outcome for any other subject. This assumption is typically met through the use of random sampling from the population of interest.
- Absence of Significant Outliers: The set of difference scores should not contain extreme outliers. Outliers are data points that deviate markedly from the overall pattern of the data. They can exert a disproportionate influence on the sample mean and standard deviation, potentially distorting the test statistic and leading to invalid results. The presence of outliers can be effectively diagnosed by examining a boxplot of the difference scores.
- Normality of Differences: The population of difference scores from which the sample is drawn is assumed to be approximately normally distributed. The t-test is known to be fairly "robust" to violations of this assumption, especially as the sample size increases. This means that even if the data deviate moderately from a perfect normal distribution, the test can still provide valid results. This assumption can be checked visually using tools like histograms or Q-Q plots of the differences, or more formally with statistical tests such as the Shapiro-Wilk test.

The assumptions of the paired t-test are not arbitrary rules but are direct consequences of the test's mathematical derivation. The test statistic,

$$t = \bar{d}/(s_d/\sqrt{n}),$$

is constructed under the premise that the sampling distribution of the mean difference ($\bar{d}$) is normally distributed. The Central Limit Theorem ensures this will be approximately true for large samples, but for smaller samples, the normality of the underlying population of differences is a more critical condition. Furthermore, the formula relies on

the sample standard deviation of the differences (s_d) as an estimate of the population standard deviation (σ_d). The presence of outliers can drastically inflate s_d, which in turn increases the denominator of the t-statistic. This artificially reduces the magnitude of the test statistic, thereby decreasing the power of the test to detect a true effect. Therefore, satisfying these assumptions is essential for ensuring that the calculated t-statistic is a reliable measure of the evidence against the null hypothesis.

26.1.2.2. A Nuanced Approach to Assumption Checking

While formal statistical tests for assumptions like normality exist, a sophisticated and practical approach is warranted. Over-reliance on tests like the Shapiro-Wilk test can be misleading. The central issue is that no real-world dataset is perfectly normal. Consequently, with a large sample size, a normality test will have high power and will likely detect even trivial and inconsequential deviations from normality, leading to a rejection of the null hypothesis of normality. Conversely, with a small sample size, the test will have low power and may fail to detect substantial and problematic deviations from normality that could invalidate the t-test.

A more informative approach focuses on assessing the impact of any non-normality. Visual inspection of the distribution of differences using histograms and Q-Q plots is often more valuable than a p-value from a formal test. The primary concerns are severe skewness or the presence of very heavy tails (i.e., a high propensity for outliers), as these can most seriously affect the validity of the t-test. The robustness of the t-test, particularly for sample sizes greater than 30, means that minor to moderate deviations from normality are generally not a cause for alarm.

26.1.2.3. Calculating the Test Statistic

The calculation of the paired t-test statistic is mathematically equivalent to performing a one-sample t-test on the vector of difference scores, testing whether their population mean is equal to zero. The procedure involves four steps:

- Calculate the mean of the differences ($\bar{d}$): This is the sample average of all the calculated difference scores.

$$\bar{d} = \frac{\sum_{i=1}^{n} d_i}{n}$$

  where d_i is the difference for the i-th pair and n is the number of pairs.

- Calculate the standard deviation of the differences (s_d): This measures the variability of the difference scores around their mean.

$$s_d = \sqrt{\frac{\sum_{i=1}^{n}(d_i - \bar{d})^2}{n-1}}$$

- Calculate the standard error of the mean difference ($SE_{\bar{d}}$): This is an estimate of the standard deviation of the sampling distribution of the mean difference.

$$SE_{\bar{d}} = \frac{s_d}{\sqrt{n}}$$

- Calculate the t-statistic (t): This is the ratio of the observed mean difference to its standard error. It quantifies how many standard errors the observed mean difference is away from the null-hypothesized value of zero.

$$t = \frac{\bar{d} - \mu_0}{SE_{\bar{d}}} = \frac{\bar{d}}{s_d/\sqrt{n}}$$

  where μ_0 is the value from the null hypothesis, which is 0 in this case.

26.1.2.4. The Decision Framework

The calculated t-statistic is then compared to the theoretical Student's t-distribution with $n-1$ degrees of freedom (df) to determine the statistical significance of the result. This comparison can be made using two complementary approaches.

- **The Critical Value Approach:** A critical value is determined from a t-distribution table or calculator based on the chosen significance level (α) and the degrees of freedom (df). For the lower-tailed test ($H_1 : \mu_d < 0$), this will be a negative value in the left tail of the distribution (e.g., t_{critical} for $\alpha = 0.05$ and df=29 is -1.699). The decision rule is: if the calculated t-statistic is less than or equal to the negative critical value ($t_{\text{calculated}} \leq t_{\text{critical}}$), the null hypothesis is rejected.
- **The p-value Approach:** The p-value represents the probability of observing a sample mean difference as extreme as, or more extreme than, the one obtained in the study, assuming that the null hypothesis is true. For a lower-tailed test, the p-value is the area under the t-distribution curve to the left of the calculated t-statistic: $p = P(T_{df} \leq t_{\text{calculated}})$. The decision rule is: if the p-value is less than or equal to the significance level ($p \leq \alpha$), the null hypothesis is rejected. This approach is generally preferred in modern statistical practice as it provides a more nuanced measure of the strength of evidence against the null hypothesis.

26.1.3. A Priori Sample Size Determination for Desired Power

One of the most critical steps in the design of a rigorous quantitative study is the a priori determination of the required sample size. Conducting a study with too few subjects runs a high risk of a Type II error—failing to detect a real effect that exists in the population. Conversely, a study with too many subjects wastes resources and may needlessly expose participants to experimental conditions. A formal power

analysis allows researchers to calculate the minimum sample size needed to have a high probability of detecting an effect of a specified magnitude, thereby balancing statistical rigor with practical and ethical constraints.

26.1.3.1. The Four Pillars of Power Analysis

The calculation of sample size for a paired t-test is a function of four interrelated parameters. A researcher must specify values for each of these to proceed.

Definition 26.1 (Significance Level (α))**.** The significant level is the probability of committing a Type I error, which is the error of rejecting the null hypothesis when it is actually true (a "false positive"). This value represents the threshold for statistical significance and is conventionally set at 0.05, corresponding to a 5% risk of a false positive conclusion.

Definition 26.2 (Statistical Power ($1 - \beta$))**.** Power is the probability of correctly rejecting the null hypothesis when it is false. It is the probability that the test will detect a true effect of a specified magnitude. The desired power is typically set at 0.80 or 0.90, which means the researcher desires an 80% or 90% chance of finding a statistically significant result if the hypothesized effect truly exists in the population. The term β represents the probability of a Type II error (a "false negative").

26.1.4. Effect Size (Δ)

The effect size (Δ) is the magnitude of the mean difference (μ_d) that the study is designed to detect. It should represent the smallest difference that is considered scientifically, clinically, or practically meaningful within the research domain. This is often the most challenging parameter to specify, as it requires substantive expertise rather than purely statistical knowledge.

Definition 26.3 (Standard Deviation of Differences (σ_d))**.** The Standard Deviation of Differences (σ_d) is an estimate of the population standard deviation of the paired differences. It quantifies the expected variability, or "noise," in the difference scores.

Example 26.1 (Sample Size Calculation)**.** For instructional purposes, the sample size formula can be presented using an approximation based on the standard normal (Z) distribution. This formula is generally accurate for larger sample sizes (e.g., $n > 30$). For a one-tailed test, the formula is:

$$n = \frac{(Z_\alpha + Z_\beta)^2 \sigma_d^2}{\Delta^2}$$

Where:

- n is the required number of pairs.
- Z_α is the critical Z-value corresponding to the one-tailed significance level α. For $\alpha = 0.05$, Z_α is 1.645.
- Z_β is the critical Z-value corresponding to the desired power. For a power of 0.80 ($\beta = 0.20$), Z_β is 0.84.
- σ_d is the estimated standard deviation of the differences.
- Δ is the target effect size or the mean difference to be detected.

While this formula provides a good estimate, more precise calculations, particularly for smaller sample sizes, are performed using the non-central t-distribution.

26.1.4.1. Practical Challenges: Estimating σ_d and Δ

The primary practical hurdles in any sample size calculation are obtaining reasonable estimates for the effect size (Δ) and the variability (σ_d).

- **Estimating Δ:** The choice of Δ should be driven by subject-matter expertise. The question to ask is not "What effect do I expect to find?" but rather "What is the smallest effect that would be considered important and meaningful in my field?" This ensures the study is powered to detect a difference that matters.
- **Estimating σ_d:** Several strategies can be employed to estimate the standard deviation of the differences:
  - **Conduct a Pilot Study:** A small-scale preliminary study can provide a direct sample estimate (s_d) of the variability.
  - **Use Data from Previous Research:** Similar studies published in the literature may report the standard deviation of the differences, which can be used as an estimate.
  - **Estimate from Individual Standard Deviations and Correlation:** This is a particularly powerful technique. If previous research provides estimates for the standard deviations of the two individual measurements (σ_1 and σ_2) and the correlation between them (ρ), the standard deviation of the differences can be calculated using the formula:

$$\sigma_d = \sqrt{\sigma_1^2 + \sigma_2^2 - 2\rho\sigma_1\sigma_2}$$

This last formula reveals a crucial design principle. The most effective way to design a more efficient experiment (i.e., one that requires a smaller sample size) is to reduce σ_d. The formula shows that σ_d decreases as the correlation ρ increases. This provides a direct, actionable insight for experimental design: researchers should strive to maximize the correlation between the paired measurements. In a before-after study, this might mean minimizing the time between measurements to reduce the influence of other factors. In a matched-pairs study, it means matching subjects on variables that are known to be strongly related to the outcome measure. This transforms the abstract

statistical concept of correlation into a practical tool for creating more powerful and efficient research.

The following table serves as a practical checklist for researchers preparing to conduct a power analysis, ensuring all necessary components are considered.

Parameter	Symbol	Description	Common Values / Source
Significance Level	α	Probability of a Type I error (false positive).	0.05, 0.01
Statistical Power	$1-\beta$	Probability of detecting a true effect.	0.80, 0.90
Directionality	Tails	Number of tails in the hypothesis test.	One-tailed (for directional H_1)
Effect Size	Δ	Minimum meaningful mean difference (μ_d) to detect.	Based on domain knowledge, clinical relevance.
Variability	σ_d	Standard deviation of the paired differences.	Estimated from pilot data, literature, or correlation.

Example 26.2 (Testing a New Fuel Additive)**.** A team of automotive engineers has developed a new fuel additive (c_1) that is hypothesized to reduce fuel consumption in passenger vehicles compared to the standard fuel (c_2). In this context, a "better" outcome is a smaller value for fuel consumption, typically measured in liters per 100 kilometers (L/100km). To test this hypothesis, the team plans a paired-sample experiment. They will use a fleet of cars, and each car will be driven over a standardized test course twice: once with the standard fuel and once with the new additive. This design ensures that factors specific to each car (e.g., engine wear, tire type) are controlled for, as each car serves as its own control.

26.1.4.2. Formulating the Hypotheses

- **Research Question:** Does the new fuel additive reduce fuel consumption?
- **Parameter of Interest:** The population mean of the paired differences in fuel consumption, $\mu_d = \mu_{\text{additive}} - \mu_{\text{standard}}$.
- **Hypotheses:** The research question implies a directional prediction. A reduction in consumption means the value for the additive (μ_{additive}) should be less than the value for the standard fuel (μ_{standard}), resulting in a negative mean difference.

  - **Null Hypothesis (H_0):** $\mu_d \geq 0$. (The new additive does not reduce fuel consumption or increases it.)
  - **Alternative Hypothesis (H_1):** $\mu_d < 0$. (The new additive reduces fuel consumption.)

- **Type of Test:** This setup requires a one-tailed (lower-tailed) paired t-test.

26.1.4.3. A Priori Sample Size Calculation

Before commencing the experiment, the engineers perform a power analysis to determine the number of cars required for the study.

- **Specification of Parameters:**
  - **Significance Level (α):** They choose a conventional α of 0.05. For a one-tailed test, the corresponding Z_α value is 1.645.
  - **Statistical Power ($1-\beta$):** They desire a power of 0.80, which is a standard choice in many fields. This corresponds to $\beta = 0.20$, and the Z_β value is 0.84.
  - **Effect Size (Δ):** After deliberation, the team decides that a reduction of 0.5 L/100km is the minimum effect size that would be commercially viable and practically meaningful. Thus, $\Delta = -0.5$.
  - **Standard Deviation of Differences (σ_d):** Based on a small pilot study with a few vehicles, they estimate the standard deviation of the differences to be $\sigma_d = 1.2$ L/100km.

- **Calculation (using the Z-approximation):**

$$n = \frac{(Z_\alpha + Z_\beta)^2 \sigma_d^2}{\Delta^2} = \frac{(1.645 + 0.84)^2 (1.2)^2}{(-0.5)^2} = \frac{(2.485)^2 (1.44)}{0.25} = \frac{(6.175)(1.44)}{0.25} \approx 35.58$$

- **Conclusion:** Since the number of subjects must be an integer, the result is rounded up. The team concludes that a sample size of 36 cars is required to achieve 80% power to detect a mean reduction of 0.5 L/100km at a 0.05 significance level.

26.1.4.4. Data Collection and Presentation (Hypothetical)

The experiment is conducted with 36 cars. For each car, fuel consumption is measured with the standard fuel (c_2) and the new additive (c_1). The difference ($d = c_1 - c_2$) is calculated for each car.

26.1.4.5. Checking the Assumptions

The engineers analyze the 36 difference scores. A boxplot reveals no extreme outliers. A histogram and a Q-Q plot of the differences show a distribution that is roughly symmetric and unimodal, suggesting that the assumption of normality is reasonably met, especially given the sample size of 36, which is large enough for the Central Limit Theorem to lend robustness to the t-test.

26.1.4.6. Calculating the Test Statistic

The analysis of the hypothetical data from the 36 cars yields the following summary statistics for the difference scores:

- Sample Mean of the Differences ($\bar{d}$): -0.65 L/100km
- Sample Standard Deviation of the Differences (s_d): 1.15 L/100km

Using these values, the test statistic is calculated:

- **Standard Error of the Mean Difference ($SE_{\bar{d}}$):**

$$SE_{\bar{d}} = \frac{s_d}{\sqrt{n}} = \frac{1.15}{\sqrt{36}} = \frac{1.15}{6} \approx 0.1917$$

- **t-statistic (t):**

$$t = \frac{\bar{d}}{SE_{\bar{d}}} = \frac{-0.65}{0.1917} \approx -3.39$$

- **Degrees of Freedom (df):** df $= n - 1 = 36 - 1 = 35$.

26.1.4.7. Determining the p-value

The calculated t-statistic ($t = -3.39$) with 35 degrees of freedom is used to find the p-value. Using statistical software or a t-distribution calculator for this lower-tailed test, the p-value is found to be approximately 0.0008.

26.1.4.8. Drawing a Conclusion

This final step involves making a statistical decision and interpreting it in the context of the original research question.

- **Statistical Decision:** The calculated p-value (0.0008) is substantially smaller than the pre-specified significance level ($\alpha = 0.05$). Therefore, the decision is to reject the null hypothesis (H_0).

- **Interpretation in Context:** There is strong, statistically significant evidence to conclude that the new fuel additive is effective in reducing fuel consumption. The observed average reduction in the sample was 0.65 L/100km.
- **Confidence Interval for the Mean Difference:** To provide a more complete picture, a confidence interval for the true mean difference is calculated. Because a one-tailed test with $\alpha = 0.05$ was performed, a corresponding 90% two-sided confidence interval is appropriate (as it leaves 5% in each tail). The formula is $\bar{d} \pm t^*_{\text{critical}} \cdot SE_{\bar{d}}$, where t^*_{critical} for a 90% CI with 35 df is approximately 1.69. Interval $= -0.65 \pm 1.69 \cdot 0.1917$ Interval $= -0.65 \pm 0.324$ 90% Confidence Interval: [-0.974, -0.326]
- **Final Report Conclusion:** The study provides statistically significant evidence ($t(35) = -3.39, p = 0.0008$) that the new fuel additive reduces fuel consumption. The average reduction observed in the sample of 36 cars was 0.65 L/100km. We are 90% confident that the true mean reduction in fuel consumption for the entire population of similar vehicles lies between 0.326 and 0.974 L/100km. Since this interval does not contain zero and consists entirely of negative values (indicating a reduction), the result is consistent with the rejection of the null hypothesis and provides a range of plausible magnitudes for the additive's true effect.

26.2. Non-inferiority Test

- **The Goal:** To show that the performance of a model `c1` is not substantially worse than the performance of a model `c2`. "Worse" in this context means that the loss `c1` being larger than the loss `c2`. A specific, acceptable margin for this difference, which is `delta`, is defined. The goal is to prove that the true mean difference (`_d = _c1 - _c2`) is less than or equal to this margin.
- **Alternative Hypothesis (H_1):** This is the research claim. It can be stated as:
  - $H_1 : \mu_d \leq \text{delta}$
- **Null Hypothesis (H_0):** This is the opposite of the claim, representing the scenario to find evidence against. It can be stated as:
  - $H_0 : \mu_d > \text{delta}$
- **Type of Test:** This is a **one-tailed (left-tailed) test**. The null will be rejected only if the sample data provides strong evidence that the mean difference is significantly *less than* delta.

The key difference from a standard superiority test (where we test against zero) is that the comparison point is a non-zero value, `delta`. This is handled by modifying the t-statistic formula. The analysis still uses a paired t-test framework, but it compares the sample mean difference ($\bar{d}$) to `delta` instead of 0. The formula for the test statistic becomes:

$$t = \frac{\bar{d} - \text{delta}}{s_d/\sqrt{n}},$$

where:

- $\bar{d}$ is the sample mean of the differences $(c_1 - c_2)$.
- **delta** is the specified margin of non-inferiority.
- s_d is the sample standard deviation of the differences.
- n is the number of pairs.

This calculated t-statistic is compared to a critical value from the t-distribution with $n - 1$ degrees of freedom.

Example 26.3 (Non-Inferiority of a New Manufacturing Process)**.** A factory wants to switch to a new, cheaper manufacturing process (`c1`) to produce a specific component. The old process is `c2`. A key quality metric is the number of micro-fractures per component; a lower number is better.

The new process is only acceptable if it's not substantially worse than the old one. Management defines "not substantially worse" as producing, on average, no more than 5 extra micro-fractures per component.

1. State the Goal and Hypotheses

- **Goal:** Show that the new process is non-inferior to the old one, within a margin of `delta = 5` micro-fractures.
- **Analysis Variable:** $d = c_1 - c_2$ (Difference in micro-fractures).
- **Hypotheses:**
  - $H_0 : \mu_d > 5$ (The new process is unacceptably worse; it adds more than 5 micro-fractures on average).
  - $H_1 : \mu_d \leq 5$ (The new process is acceptable; it is non-inferior).
- **Significance Level:** We'll use $\alpha = 0.05$.

2. Collect Sample Data (Hypothetical)

The factory runs a test on $n = 30$ components, using both processes on each component and counting the micro-fractures.

- **Sample Mean Difference ($\bar{d}$):** 3.8 micro-fractures. (On average, the new process produced 3.8 more fractures than the old one).
- **Sample Standard Deviation of Differences (s_d):** 4.5 micro-fractures.

3. Calculate the Test Statistic

Using the modified formula with `delta = 5`:

$$t = \frac{\bar{d} - \text{delta}}{s_d/\sqrt{n}}$$

$$t = \frac{3.8 - 5}{4.5/\sqrt{30}} \approx -1.46$$

4. Determine the Critical Value and Make a Decision

- **Degrees of Freedom (df):** $n - 1 = 30 - 1 = 29$.
- **Critical Value:** We need the critical t-value for a **one-tailed (left-tailed)** test with $\alpha = 0.05$ and $df = 29$. Looking this up in a t-distribution table, we find:
  - $t_{critical} \approx -1.699$
- **Rejection Rule:** We reject H_0 if our calculated t-statistic is *less than* (more negative than) the critical value.
- **Comparison:** Is $-1.46 < -1.699$? No, it is not.

5. State the Conclusion

Since the calculated t-statistic (-1.46) is not less than the critical value (-1.699), it does not fall into the rejection region. Therefore, we fail to reject the null hypothesis (H_0).

In the context of the problem, this means: **There is not enough statistical evidence to conclude that the new manufacturing process is non-inferior.** Even though the sample average of 3.8 was within the acceptable margin of 5, the result was not statistically significant enough to rule out the possibility that the true average difference could be greater than 5. The factory should not adopt the new process based on these results.

27. Addressing Multicollinearity: Principle Component Analysis (PCA) and Factor Analysis (FA)

27.1. Introduction

The concepts of Principal Component Analysis (PCA) and Factor Analysis (FA) are both dimensionality reduction techniques. They operate on different assumptions and serve distinct purposes. PCA aims to transform correlated variables into a smaller set of uncorrelated principal components that capture maximum variance, whereas Factor Analysis seeks to explain the correlations between observed variables in terms of a smaller number of unobserved, underlying factors.

After loading and preprocessing the data in Section 27.2, we will explore these methods to reduce dimensions and address multicollinearity. In Section 27.3 we will conduct linear regression on the extracted components or factors. Section 27.4 provides diagnostics for multicollinearity, including the coefficient table, eigenvalues, condition indices, and the KMO measure. Section 27.5 explains how PCA is applied to the data, while Section 27.6 discusses Factor Analysis. Both methods are used to mitigate multicollinearity issues in regression models. Section 27.8 shows how the reduced dimensions can be used in other machine learning models, such as Random Forests.

The following packages are used in this chapter:

```
import pandas as pd
import numpy as np
import statsmodels.api as sm
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from factor_analyzer import FactorAnalyzer
from factor_analyzer.factor_analyzer import calculate_kmo
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
import copy
```

```
from spotpython.utils.stats import condition_index
from spotpython.utils.pca import (get_pca, plot_pca_scree, plot_pca1vs2, get_pca_topk
```

27.2. The Car-Sales Data Set

First, the data is preprocessed to ensure that it does not contain any NaN or infinite values. We load the data set, which contains information about car sales, including various features such as price, engine size, horsepower, and more. The initial shape of the DataFrame is `(157, 27)`.

```
df = pd.read_csv("data/car_sales.csv", encoding="utf-8", index_col=None)
print(df.shape)
df.head()
```

```
(157, 27)
```

	Unnamed: 0	manufact	model	sales	resale	type	price	engine_s	horsepow	whee
0	1	Acura	Integra	16.919	16.360	0	21.50	1.8	140.0	101.2
1	2	Acura	TL	39.384	19.875	0	28.40	3.2	225.0	108.1
2	3	Acura	CL	14.114	18.225	0	NaN	3.2	225.0	106.9
3	4	Acura	RL	8.588	29.725	0	42.00	3.5	210.0	114.6
4	5	Audi	A4	20.397	22.255	0	23.99	1.8	150.0	102.6

The first column is removed as it's an index or non-informative column.

```
df = df.drop(df.columns[0], axis=1)
df.head()
```

	manufact	model	sales	resale	type	price	engine_s	horsepow	wheelbas	width
0	Acura	Integra	16.919	16.360	0	21.50	1.8	140.0	101.2	67.3
1	Acura	TL	39.384	19.875	0	28.40	3.2	225.0	108.1	70.3
2	Acura	CL	14.114	18.225	0	NaN	3.2	225.0	106.9	70.6
3	Acura	RL	8.588	29.725	0	42.00	3.5	210.0	114.6	71.4
4	Audi	A4	20.397	22.255	0	23.99	1.8	150.0	102.6	68.2

27.2.1. The Target Variable

The `sales` variable, which is our target, is transformed to a log scale. Missing or zero values are handled by replacing them with the median.

```
df['ln_sales'] = np.log(df['sales'].replace(0, np.nan))
if df['ln_sales'].isnull().any() or np.isinf(df['ln_sales']).any():
    df['ln_sales'] = df['ln_sales'].fillna(df['ln_sales'].median()) # Or any other strategy
y = df['ln_sales']
```

27.2.2. The Features

27.2.2.1. Numerical Features

The following steps are performed during data preprocessing for numerical features:

1. Check for NaN or infinite values in X.
2. Replace NaN and infinite values with the median of the respective column.
3. Remove constant or nearly constant columns (not explicitly shown in code but stated in preprocessing steps).
4. Standardize the numerical predictors in X using StandardScaler.
5. Verify that X_scaled does not contain any NaN or infinite values.

```
# Use columns from 'price' to 'mpg' as predictors
independent_var_columns = ['price', 'engine_s', 'horsepow', 'wheelbas',
                           'width', 'length', 'curb_wgt', 'fuel_cap', 'mpg']
# Select those columns, ensuring they are numeric
X = df[independent_var_columns].apply(pd.to_numeric, errors='coerce')
# Handle missing/nans in features by using an appropriate imputation strategy
X = X.fillna(X.median()) # Impute with median or any other appropriate strategy
# Display the first few rows of the features
X.head()
```

	price	engine_s	horsepow	wheelbas	width	length	curb_wgt	fuel_cap	mpg
0	21.500	1.8	140.0	101.2	67.3	172.4	2.639	13.2	28.0
1	28.400	3.2	225.0	108.1	70.3	192.9	3.517	17.2	25.0
2	22.799	3.2	225.0	106.9	70.6	192.0	3.470	17.2	26.0
3	42.000	3.5	210.0	114.6	71.4	196.6	3.850	18.0	22.0
4	23.990	1.8	150.0	102.6	68.2	178.0	2.998	16.4	27.0

```
if X.isnull().any().any():
    print("NaNs detected in X. Filling with column medians.")
    X = X.fillna(X.median())

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
if np.isnan(X_scaled).any() or np.isinf(X_scaled).any():
    raise ValueError("X_scaled contains NaN or infinite values after preprocessing.")
# Convert the scaled data back to a DataFrame
X_scaled = pd.DataFrame(X_scaled, columns=X.columns)
# Display the first few rows of the scaled features
X_scaled.head()
```

	price	engine_s	horsepow	wheelbas	width	length	curb_wgt	fuel_cap	n
0	-0.410053	-1.214376	-0.814577	-0.827661	-1.121287	-1.119971	-1.182726	-1.228700	0
1	0.075070	0.134385	0.694066	0.081122	-0.246689	0.416070	0.223285	-0.193381	0
2	-0.318723	0.134385	0.694066	-0.076927	-0.159229	0.348634	0.148020	-0.193381	0
3	1.031255	0.423406	0.427835	0.937221	0.073997	0.693306	0.756545	0.013683	-0
4	-0.234987	-1.214376	-0.637089	-0.643270	-0.858907	-0.700370	-0.607830	-0.400444	0

27.2.2.2. Categorical Features

Categorical features (like 'type') are one-hot encoded and then combined with the scaled numerical features.

```
categorical_cols = ['type'] # Replace if more categorical variables exist
encoder = OneHotEncoder(drop='first', sparse_output=False)
X_categorical_encoded = encoder.fit_transform(df[categorical_cols])
# Convert encoded data into a DataFrame
X_categorical_encoded_df = pd.DataFrame(X_categorical_encoded,
                                        columns=encoder.get_feature_names_out(categori
X_categorical_encoded_df.describe(include='all')
```

	type_1
count	157.000000
mean	0.261146
std	0.440665
min	0.000000
25%	0.000000
50%	0.000000

	type_1
75%	1.000000
max	1.000000

27.2.3. Combining Non-categorical and Categorical (encoded) Data

The final feature set `X_encoded` is created by concatenating the scaled numerical features and the one-hot encoded categorical features. This combined DataFrame will be used for regression analysis.

```
X_encoded = pd.concat([X_scaled, X_categorical_encoded_df], axis=1)
print(f"Dimension: {X_encoded.shape}")
print(list(X_encoded.columns))
```

```
Dimension: (157, 10)
['price', 'engine_s', 'horsepow', 'wheelbas', 'width', 'length', 'curb_wgt', 'fuel_cap', 'mpg', 'typ
```

```
X_encoded.describe(include='all')
```

	price	engine_s	horsepow	wheelbas	width	length	curb_w
count	1.570000e+02	1.570000e+02	1.570000e+02	1.570000e+02	1.570000e+02	1.570000e+02	1.570000
mean	-5.091469e-16	1.018294e-16	1.584012e-16	3.892145e-15	-2.489162e-16	-1.584012e-16	-2.94173
std	1.003200e+00	1.003200e+00	1.003200e+00	1.003200e+00	1.003200e+00	1.003200e+00	1.003200
min	-1.272376e+00	-1.985097e+00	-2.323220e+00	-1.960346e+00	-2.491490e+00	-2.843334e+00	-2.37415
25%	-6.459349e-01	-7.326758e-01	-6.370894e-01	-5.905870e-01	-8.006008e-01	-7.303412e-01	-6.44662
50%	-3.187230e-01	-5.829498e-02	-1.489990e-01	-6.375655e-02	-1.738055e-01	4.142561e-02	-5.69560
75%	3.232563e-01	4.234056e-01	5.165788e-01	6.211231e-01	6.570627e-01	6.558419e-01	6.412452
max	4.089638e+00	4.758711e+00	4.687533e+00	4.111375e+00	2.552025e+00	2.783820e+00	3.514119

27.3. Fit the Linear Regression Model

An Ordinary Least Squares (OLS) regression model is fitted using the preprocessed and combined features (`X_encoded`).

```
X_encoded_with_const = sm.add_constant(X_encoded) # Adds a constant term (intercept) to the model
model = sm.OLS(df['ln_sales'], X_encoded_with_const).fit()
```

27.3.1. Model Summary and Interpretation

27.3.1.1. Model Summary (ANOVA Table)

The ANOVA table shows a significant F-value (Prob (F-statistic) close to zero), indicating that the model is statistically significant and better than simply estimating the mean. The Adj. R-squared value, close to 0.40, suggests that nearly 40% of the variation in `ln_sales` is explained by the model.

```
print(model.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               ln_sales   R-squared:                       0.485
Model:                            OLS   Adj. R-squared:                  0.449
Method:                 Least Squares   F-statistic:                     13.73
Date:                Tue, 04 Nov 2025   Prob (F-statistic):           7.69e-
17
Time:                        20:41:51   Log-Likelihood:                -
213.62
No. Observations:                 157   AIC:                             449.2
Df Residuals:                     146   BIC:                             482.9
Df Model:                          10
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          3.0678      0.114     26.962      0.000       2.843       3.293
price         -0.6451      0.177     -3.655      0.000      -0.994      -
0.296
engine_s       0.3557      0.192      1.854      0.066      -0.023       0.735
horsepow      -0.1364      0.229     -0.596      0.552      -0.589       0.316
wheelbas       0.3166      0.174      1.816      0.071      -0.028       0.661
width         -0.0763      0.140     -0.547      0.586      -0.352       0.200
length         0.2029      0.185      1.099      0.273      -0.162       0.568
curb_wgt       0.0842      0.211      0.399      0.691      -0.333       0.501
fuel_cap      -0.2284      0.179     -1.276      0.204      -0.582       0.125
mpg            0.3232      0.167      1.941      0.054      -0.006       0.652
type_1         0.8735      0.317      2.756      0.007       0.247       1.500
==============================================================================
Omnibus:                       41.296   Durbin-Watson:                   1.423
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              107.145
Skew:                          -1.064   Prob(JB):                     5.42e-
24
```

```
Kurtosis:                       6.442   Cond. No.                         11.4
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Despite the positive model fit, many predictors show non-significant coefficients (P>|t| much larger than 0.05), suggesting they contribute little to the model.

27.4. Collinearity Diagnostics

27.4.1. The Coefficient Table

The coefficient table provides further evidence of multicollinearity. The function `compute_coefficients_table()` from the `spotpython` package is used here for comprehensive diagnostics.

```
from spotpython.utils.stats import compute_coefficients_table
coeffs_table = compute_coefficients_table(
    model=model, X_encoded=X_encoded_with_const, y=y, vif_table=None
)
print("\nCoefficients Table:")
print(coeffs_table)
```

```
Coefficients Table:
   Variable  Zero-Order r  Partial r  Semipartial r  Tolerance       VIF
0     price     -0.551325  -0.289521      -0.217155   0.195662  5.110865
1  engine_s     -0.139066   0.151682       0.110172   0.165615  6.038084
2  horsepow     -0.386896  -0.049229      -0.035386   0.116244  8.602562
3  wheelbas      0.292461   0.148618       0.107895   0.200566  4.985881
4     width      0.040572  -0.045185      -0.032473   0.312675  3.198207
5    length      0.216882   0.090597       0.065310   0.178835  5.591740
6  curb_wgt     -0.040042   0.032981       0.023691   0.136742  7.313045
7  fuel_cap     -0.017278  -0.105041      -0.075831   0.190355  5.253349
8       mpg      0.119998   0.158587       0.115313   0.219810  4.549388
9    type_1      0.273500   0.222382       0.163754   0.314477  3.179880
```

For most predictors, the partial correlations (Partial r) decrease significantly compared to the zero-order correlations (Zero-Order r), which suggests multicollinearity. Tolerance values (1 minus the proportion of variance explained by other predictors) are low, indicating that approximately 70%-90% of a given predictor's variance can be explained

by other predictors. Tolerances close to 0 signify high multicollinearity. A Variance Inflation Factor (VIF) greater than 2 is typically considered problematic, and in this table, the smallest VIF is already greater than 2, confirming serious multicollinearity.

27.4.2. Eigenvalues and Condition Indices

Eigenvalues indicate how many factors or components can be meaningfully extracted. An eigenvalue greater than 1 suggests that the factor/component explains more variance than a single variable.

27.4.2.1. Eigenvalues

We use the `FactorAnalyzer` function from the `factor_analyzer` package to compute eigenvalues.

```
fa_temp = FactorAnalyzer(n_factors=X_encoded.shape[1], method="principal", rotation=No
try:
    fa_temp.fit(X_encoded)
    ev, _ = fa_temp.get_eigenvalues()
    ev = np.sort(ev) # The source prints in ascending order
    print("Eigenvalues for each component:\n", ev)
except Exception as e:
    print(f"Error during factor analysis fitting: {e}")
    print("Consider reducing multicollinearity or removing problematic features.")
```

```
Eigenvalues for each component:
 [0.06453844 0.09238346 0.13143422 0.15658714 0.20129034 0.25457714
 0.33712112 1.14556836 1.64880929 5.96769049]
```

The eigenvalue-based diagnostics confirm severe multicollinearity. Several eigenvalues are close to 0, indicating strong correlations among predictors.

27.4.2.2. Condition Indices

From `spotpython.utils.stats`, we can compute the condition index, which is a measure of multicollinearity. A condition index greater than 15 suggests potential multicollinearity issues, and values above 30 indicate severe problems.

Condition indices, calculated as the square roots of the ratios of the largest eigenvalue to each subsequent eigenvalue, also highlight the issue.

Definition 27.1 (Condition Index)**.** The Condition Index (CI_i) for the i-th eigenvalue is defined as:

$$\mathrm{CI}_i = \sqrt{\frac{\lambda\mathrm{max}}{\lambda_i}},$$

where $\lambda_{\max}$ is the largest eigenvalue of the scaled predictor correlation matrix, and λ_i is the i-th eigenvalue of the same matrix.

CI_i-values greater than 15 suggest a potential problem, and values over 30 indicate a severe problem.

```
X_cond = copy.deepcopy(X_encoded)
condition_index_df = condition_index(X_cond)
print("\nCondition Index:")
print(condition_index_df)
```

```
Condition Index:
   Index  Eigenvalue  Condition Index
0      0    0.047116        11.150268
1      1    0.067199         9.336595
2      2    0.121066         6.955955
3      3    0.146634         6.320499
4      4    0.157663         6.095428
5      5    0.248119         4.858905
6      6    0.338187         4.161884
7      7    0.736900         2.819449
8      8    1.531162         1.955951
9      9    5.857833         1.000000
```

27.4.3. Kayser-Meyer-Olkin (KMO) Measure

The KMO (Kaiser-Meyer-Olkin) measure is a metric for assessing the suitability of data for Factor Analysis. A KMO value of 0.6 or higher is generally considered acceptable, while a value below 0.5 indicates that the data is not suitable for Factor Analysis.

The KMO measure is based on the correlation and partial correlation between variables. It is calculated as the ratio of the squared sums of correlations to the squared sums of correlations plus the squared sums of partial correlations. KMO values range between 0 and 1, where values close to 1 suggest strong correlations and suitability for Factor Analysis, and values close to 0 indicate weak correlations and unsuitability.

```
kmo_all, kmo_model = calculate_kmo(X_encoded)
print(f"\nKMO measure: {kmo_model:.3f} (0.6+ is often considered acceptable)")
```

```
KMO measure: 0.835 (0.6+ is often considered acceptable)
```

A KMO measure of 0.835 indicates that the data is well-suited for Factor Analysis.

27.5. Addressing Multicollinearity with Principal Component Analysis (PCA)

Definition 27.2 (Multicollinearity and Multicorrelation)**.** Multicorrelation is a general term that describes correlation between multiple variables. Multicollinearity is a specific problem in regression models caused by strong correlations between independent variables, making model interpretation difficult.

27.5.1. Introduction to PCA

Principal Component Analysis (PCA) is a popular unsupervised dimensionality reduction technique. It transforms a set of possibly correlated variables into a set of linearly uncorrelated variables called principal components. The first principal component accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible. PCA is primarily used for data compression and simplifying complex datasets.

27.5.2. Application of PCA in Regression Problems:

- **Dimensionality Reduction:** PCA reduces the number of explanatory variables by transforming original variables into a smaller set of uncorrelated principal components, making regression algorithms less prone to overfitting, especially with many features.
- **Reducing Multicollinearity:** PCA effectively eliminates multicollinearity in linear regression models because the resulting principal components are orthogonal (uncorrelated) to each other, leading to more stable coefficient estimates.
- **Handling High-Dimensional Data:** It can reduce the dimensions of datasets with many variables to a manageable level before regression.
- **Reduced Overfitting Tendencies:** By removing redundant and highly correlated variables, PCA helps reduce the risk of overfitting by focusing the model on the most influential features.

- **Improved Model Performance:** Performing regression on the most important principal components often leads to better generalization and improved model performance on new data.
- **Interpretation of Feature Importance:** PCA provides insights into the importance of original features through the variance explained by each principal component, which can identify combinations of variables best representing the data.

27.5.3. Scree Plot

Definition 27.3 (Scree Plot)**.** A scree plot is a graphical representation of the eigenvalues of a covariance or correlation matrix in descending order. It is used to determine the number of significant components or factors in dimensionality reduction techniques.

Mathematically, the eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_p$ are plotted against their corresponding component or factor indices $i = 1, 2, \dots, p$, where p is the total number of components or factors.

The eigenvalues are defined as:

$$\lambda_i = \mathrm{Var}(\mathbf{z}_i),$$

where $\mathbf{z}_i$ is the i-th principal component or factor, and $\mathrm{Var}(\mathbf{z}_i)$ is its variance.

The scree plot is constructed by plotting the points (i, λ_i) for $i = 1, 2, \dots, p$. The "elbow" in the plot, where the eigenvalues start to level off, indicates the optimal number of components or factors to retain.

27.5.4. Loading Scores (for PCA)

Loading scores in the context of Principal Component Analysis (PCA) represent the correlation or relationship between the original variables and the principal components.

Definition 27.4 (Loading Scores)**.** The loading score for the j-th variable on the i-th principal component is defined as:

$$L_{ij} = \mathbf{a}_i^\top \mathbf{x}_j,$$

where:

$\mathbf{a}_i$ is the eigenvector corresponding to the i-th principal component, $\mathbf{x}_j$ is the standardized value of the j-th variable.

In PCA, the loading scores indicate how much each original variable contributes to a given principal component. High absolute values of L_{ij} suggest that the j-th variable strongly influences the i-th principal component. In PCA, loading scores can be viewed as directional vectors in the feature space. The magnitude of the score indicates how dominant the variable is in a component, while the sign represents the direction of the relationship. A high positive loading means a positive influence and correlation with the component, and a high negative loading indicates a negative correlation. Loading score values also show how much each original variable contributes to the explained variance in its respective principal component.

> **Summary of Loading Scores**
>
> Loading scores are used in Principal Component Analysis (PCA).
>
> - Definition: Loading scores represent the correlation or relationship between the original variables and the principal components.
> - Purpose: They indicate how much each original variable contributes to a given principal component.
> - Mathematical Representation: In PCA, the loading scores are the elements of the eigenvectors of the covariance (or correlation) matrix, scaled by the square root of the corresponding eigenvalues.
> - Interpretation: High absolute values of loading scores suggest that the variable strongly influences the corresponding principal component.

Section 27.7.3 explains the difference between loading scores in PCA and factor loadings in FA.

27.5.5. PCA for Car Sales Example

27.5.5.1. Computing the Principal Components

The Principal Component Analysis (PCA) is applied only to the features (`X_encoded`), not to the target variable. We will use functions from `spotpython.utils.pca`, which are based on `sklearn.decomposition.PCA` to perform PCA.

Step 1: Perform PCA and scale the data

```
pca, scaled_data, feature_names, sample_names, df_pca_components = get_pca(df=X_encode
```

Step 2: Plot the scree plot

```
plot_pca_scree(pca, df_name="Car Sales Data", max_scree=10)
```

Figure 27.1.: Scree plot for PCA showing the explained variance ratio for each principal component.

Step 3: Plot the first two principal components

```
plot_pca1vs2(pca, df_pca_components, df_name="Car Sales Data")
```

Figure 27.2.: Scatter plot of the first two principal components (PC1 vs PC2) for the Car Sales Data.

Step 4: Get the top k features influencing PC1 and PC2

```
top_k_features_pc1, top_k_features_pc2 = get_pca_topk(pca, feature_names, k=10)
print("Top 10 features influencing PC1:", top_k_features_pc1)
print("Top 10 features influencing PC2:", top_k_features_pc2)
```

```
Top 10 features influencing PC1: ['curb_wgt', 'engine_s', 'fuel_cap', 'mpg', 'width',
Top 10 features influencing PC2: ['price', 'wheelbas', 'horsepow', 'length', 'engine_s
```

27.5.5.2. Loading Scores for PCA (10 Components)

```
# Get and print loading scores
loading_scores_df = get_loading_scores(pca, X_encoded.columns)
print("PCA Loading Scores (10 Components):\n", loading_scores_df)
```

```
PCA Loading Scores (10 Components):
                PC1       PC2       PC3       PC4       PC5       PC6  \
price      0.251214  0.568904  0.145341 -0.484049  0.335697 -0.164197
engine_s   0.364662  0.204801  0.120816  0.392355 -0.419180  0.432765
horsepow   0.317527  0.438619  0.305988  0.022026 -0.199571 -0.072824
```

```
wheelbas  0.300637 -0.470803  0.154529 -0.380962 -0.080738 -0.283304
width     0.343110 -0.186317  0.239367  0.582904  0.634863 -0.193732
length    0.302587 -0.398562  0.440896 -0.197738 -0.274463  0.077374
curb_wgt  0.382220 -0.036672 -0.252691 -0.118033  0.162548  0.314687
fuel_cap  0.358821 -0.102589 -0.424633 -0.203499  0.220563  0.384133
mpg      -0.351143 -0.076695  0.468213 -0.176015  0.326376  0.640074
type_1    0.072646 -0.095107 -0.362901  0.030564 -0.072634  0.019721

               PC7       PC8       PC9      PC10
price    -0.009121 -0.098316 -0.395916  0.227288
engine_s  0.279081 -0.161115 -0.433737 -0.039611
horsepow  0.132544  0.184063  0.718133 -0.019567
wheelbas  0.568127 -0.236665 -0.001192 -0.231065
width     0.030011  0.022386 -0.009774  0.091688
length   -0.517766  0.214795 -0.090943  0.335102
curb_wgt -0.410187 -0.591957  0.262427 -0.248584
fuel_cap  0.163440  0.640656 -0.004986 -0.074928
mpg       0.243304 -0.105455  0.148714  0.109367
type_1    0.241967 -0.236377  0.200080  0.832423
```

Figure 27.3 shows the loading scores heatmap for the first 10 principal components. The heatmap visualizes how much each original feature contributes to each principal component, with darker colors indicating stronger contributions.

```
plot_loading_scores(loading_scores_df)
```

Figure 27.3.: PCA Loading Scores Heatmap showing the influence of original features on the principal components.

27.5.6. Creating the Regression Model with Principal Components

Now, a linear regression model is fitted using the principal components derived from PCA. These components are uncorrelated, which should eliminate multicollinearity issues.

```
X_pca_model_with_const = sm.add_constant(df_pca_components)
model_pca = sm.OLS(y, X_pca_model_with_const).fit()
print("\nRegression on PCA Components:")
print(model_pca.summary())
```

```
Regression on PCA Components:
                            OLS Regression Results
==============================================================================
Dep. Variable:               ln_sales   R-squared:                       0.485
Model:                            OLS   Adj. R-squared:                  0.449
```

```
Method:                 Least Squares   F-statistic:                     13.73
Date:                Tue, 04 Nov 2025   Prob (F-statistic):           7.69e-
17
Time:                        20:41:52   Log-Likelihood:                -
213.62
No. Observations:                 157   AIC:                             449.2
Df Residuals:                     146   BIC:                             482.9
Df Model:                          10
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          3.2959      0.078     42.215      0.000       3.142       3.450
PC1           -0.0450      0.032     -1.392      0.166      -0.109       0.019
PC2           -0.6572      0.063    -10.383      0.000      -0.782      -
0.532
PC3           -0.0624      0.091     -0.683      0.495      -0.243       0.118
PC4            0.2500      0.135      1.856      0.065      -0.016       0.516
PC5           -0.4628      0.157     -2.943      0.004      -0.774      -
0.152
PC6            0.3734      0.197      1.893      0.060      -0.016       0.763
PC7            0.3777      0.205      1.847      0.067      -0.027       0.782
PC8           -0.4887      0.225     -2.171      0.032      -0.934      -
0.044
PC9            0.2311      0.302      0.765      0.445      -0.366       0.828
PC10           0.5885      0.361      1.631      0.105      -0.125       1.302
==============================================================================
Omnibus:                       41.296   Durbin-Watson:                   1.423
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              107.145
Skew:                          -1.064   Prob(JB):                     5.42e-
24
Kurtosis:                       6.442   Cond. No.                         11.2
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

When all principal components are retained, the PCA regression model performs identically to the original OLS model in terms of R-squared, Adjusted R-squared, MSE, and RMSE. This is because PCA merely rotates the data, preserving all variance if all components are used. Its benefit lies in handling multicollinearity and enabling dimensionality reduction if fewer components are chosen without significant loss of information.

27.5.7. Collinearity Diagnostics for PCA Regression Model

Consider the eigenvalues of the PCA components to verify that they are uncorrelated. The eigenvalues should be close to 1, indicating that the components are orthogonal and do not exhibit multicollinearity.

```
fa_temp = FactorAnalyzer(n_factors=df_pca_components.shape[1], method="principal", rot
try:
    fa_temp.fit(df_pca_components)
    ev, _ = fa_temp.get_eigenvalues()
    ev = np.sort(ev) # The source prints in ascending order
    print("Eigenvalues for each component:\n", ev)
except Exception as e:
    print(f"Error during factor analysis fitting: {e}")
    print("Consider reducing multicollinearity or removing problematic features.")
```

```
Eigenvalues for each component:
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Next, we compute the condition indices for the PCA components to confirm that they are uncorrelated.

```
coeffs_table = compute_coefficients_table(
    model=model_pca, X_encoded=X_pca_model_with_const, y=y, vif_table=None
)
print("\nCoefficients Table:")
print(coeffs_table)
```

```
Coefficients Table:
  Variable  Zero-Order r  Partial r  Semipartial r  Tolerance  VIF
0      PC1     -0.082694  -0.114428      -0.082694        1.0  1.0
1      PC2     -0.616895  -0.651723      -0.616895        1.0  1.0
2      PC3     -0.040608  -0.056473      -0.040608        1.0  1.0
3      PC4      0.110272   0.151818       0.110272        1.0  1.0
4      PC5     -0.174882  -0.236673      -0.174882        1.0  1.0
5      PC6      0.112489   0.154797       0.112489        1.0  1.0
6      PC7      0.109722   0.151078       0.109722        1.0  1.0
7      PC8     -0.129005  -0.176859      -0.129005        1.0  1.0
8      PC9      0.045456   0.063189       0.045456        1.0  1.0
9     PC10      0.096903   0.133763       0.096903        1.0  1.0
```

As expected, results indicate that there is no multicollinearity among the principal components. This confirms that PCA successfully addresses the multicollinearity problem. The R-squared and Adjusted R-squared values remain the same as the original OLS model since PCA preserves the total variance when all components are retained.

27.5.8. PCA: Creating the Regression Model with three Principle Components only

```
# Create a regression model using only the first three principal components
df_pc_reduced = df_pca_components.iloc[:, :3] # select the first three factors
X_model_pc_reduced = sm.add_constant(df_pc_reduced)
model_pc_reduced = sm.OLS(y, X_model_pc_reduced).fit()
print("\nRegression on PCs (three PCs only):")
print(model_pc_reduced.summary())

# Verify collinearity statistics for reduced PCs scores
coeffs_table_pc_reduced = compute_coefficients_table(
    model=model_pc_reduced, X_encoded=X_model_pc_reduced, y=y, vif_table=None
)
print("\nCoefficients Table (Reduced PCs Analysis Model):")
print(coeffs_table_pc_reduced)

# Verify condition indices for reduced FA scores
X_cond_pc_reduced = copy.deepcopy(df_pc_reduced)
condition_index_df_pc_reduced = condition_index(X_cond_pc_reduced)
print("\nCondition Index (Reduced PC Analysis Model):")
print(condition_index_df_pc_reduced)
```

```
Regression on PCs (three PCs only):
                            OLS Regression Results
==============================================================================
Dep. Variable:               ln_sales   R-squared:                       0.389
Model:                            OLS   Adj. R-squared:                  0.377
Method:                 Least Squares   F-statistic:                     32.48
Date:                Tue, 04 Nov 2025   Prob (F-statistic):           2.66e-
16
Time:                        20:41:52   Log-Likelihood:                -
226.97
No. Observations:                 157   AIC:                             461.9
Df Residuals:                     153   BIC:                             474.2
```

```
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          3.2959      0.083     39.693      0.000       3.132       3.460
PC1           -0.0450      0.034     -1.309      0.193      -0.113       0.023
PC2           -0.6572      0.067     -9.762      0.000      -0.790      -
0.524
PC3           -0.0624      0.097     -0.643      0.521      -0.254       0.129
==============================================================================
Omnibus:                       43.520   Durbin-Watson:                   1.413
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              125.210
Skew:                          -1.081   Prob(JB):                     6.47e-
28
Kurtosis:                       6.804   Cond. No.                         2.82
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly spec

Coefficients Table (Reduced PCs Analysis Model):
  Variable  Zero-Order r  Partial r  Semipartial r  Tolerance  VIF
0      PC1     -0.082694  -0.105209      -0.082694        1.0  1.0
1      PC2     -0.616895  -0.619530      -0.616895        1.0  1.0
2      PC3     -0.040608  -0.051883      -0.040608        1.0  1.0

Condition Index (Reduced PC Analysis Model):
   Index  Eigenvalue  Condition Index
0      0    0.736900         2.819449
1      1    1.531162         1.955951
2      2    5.857833         1.000000
```

27.6. Addressing Multicollinearity and Latent Structure with Factor Analysis (FA)

27.6.1. Introduction to Factor Analysis

Factor Analysis (FA) is a statistical method used to describe variability among observed, correlated variables in terms of a potentially lower number of unobserved variables called factors or latent variables. Unlike PCA, which is primarily a data reduction technique focused on maximizing variance explained, FA assumes that the observed

variables are linear combinations of these underlying factors plus an error term. FA's main goal is to uncover the underlying structure that explains the correlations among observed variables.

27.6.2. Determining the Number of Factors for Factor Analysis

For Factor Analysis, the number of factors to extract is a crucial decision. A common approach, consistent with the KMO measure, is to consider factors with eigenvalues greater than 1 (Kaiser's criterion). Factor analysis is then performed, often with a rotation method like Varimax to improve factor interpretability.

```
anz_fak = 10 # Number of factors to extract, similar to the components in PCA
n_factors = min(anz_fak, X_encoded.shape[1])
fa = FactorAnalyzer(n_factors=n_factors, method="principal", rotation="varimax")
fa.fit(X_encoded) # Fit the Factor Analyzer
actual_factors = fa.loadings_.shape[1] # Number of factors actually extracted
print(f"actual_factors: {actual_factors}")
if actual_factors < n_factors:
    print(
        f"\nWarning: Only {actual_factors} factors could be extracted "
        f"(requested {n_factors})."
    )
factor_columns = [f"Factor{i+1}" for i in range(actual_factors)]
```

```
actual_factors: 10
```

27.6.3. Scree Plot for Factor Analysis

Figure 27.4 shows the eigenvalues for each factor extracted from Factor Analysis. The scree plot helps in determining the number of factors to retain by identifying the "elbow" point where the eigenvalues start to level off, indicating diminishing returns in explained variance.

```
plt.figure(figsize=(10, 6))
ev_fa, _ = fa.get_eigenvalues()
plt.plot(range(1, len(ev_fa) + 1), ev_fa, marker='o', linestyle='--')
plt.title('Scree Plot for Factor Analysis')
plt.xlabel('Number of Factors')
plt.ylabel('Eigenvalue')
plt.grid(True)
plt.xticks(range(1, len(ev_fa) + 1))
plt.show()
```

Figure 27.4.: Scree plot for Factor Analysis showing the eigenvalues for each factor.

27.6.4. Factor Loadings

Factor Loadings indicate how strongly each original variable is correlated with the extracted factors. High absolute values suggest that the variable has a significant influence on, or is strongly associated with, that factor. Loadings help in interpreting the meaning of each underlying factor.

> **Summary of Factor Loadings**
>
> Factor loadings are used in Factor Analysis (FA). * Definition: Factor loadings represent the correlation or relationship between the observed variables and the latent factors. * Purpose: They indicate how much each observed variable is explained by a given factor. * Mathematical Representation: In FA, factor loadings are derived from the factor model, where observed variables are expressed as linear combinations of latent factors plus error terms. * Interpretation: High absolute values of factor loadings suggest that the variable is strongly associated with the corresponding factor.

Section 27.7.3 explains the difference between loading scores in PCA and factor loadings in FA.

```python
# Print factor loadings with 2 decimals
factor_loadings = fa.loadings_
print("Factor Loadings (rounded to 2 decimals):\n", np.round(factor_loadings, 2))

# Create a DataFrame for the factor loadings for better visualization
factor_loadings_df = pd.DataFrame(
    factor_loadings, index=X_encoded.columns, # Original feature names
    columns=factor_columns # Factor names
)

# Plot the heatmap for factor loadings
plt.figure(figsize=(10, 8))
sns.heatmap(
    factor_loadings_df, annot=True, # Annotate with values
    fmt=".2f", # Format values to 2 decimals
    cmap="coolwarm", # Color map
    cbar=True # Show color bar
)
plt.title("Factor Loadings Heatmap")
plt.xlabel("Factors")
plt.ylabel("Features")
plt.tight_layout()
plt.show()
```

```
Factor Loadings (rounded to 2 decimals):
 [[ 0.12  0.97 -0.01  0.06 -0.01  0.08  0.01  0.09  0.01 -0.15]
 [ 0.12  0.6   0.18  0.28  0.61  0.2   0.31  0.09 -0.01  0.02]
 [ 0.06  0.87 -0.01  0.19  0.26  0.16  0.19  0.01 -0.02  0.26]
 [ 0.13  0.07  0.26  0.18  0.04  0.09  0.9   0.03  0.26 -0.01]
 [ 0.14  0.25  0.12  0.77  0.15  0.14  0.51  0.08  0.    0.01]
 [ 0.09  0.12  0.    0.2   0.14  0.11  0.92  0.09 -0.23  0.03]
 [ 0.28  0.43  0.41  0.26  0.15  0.25  0.44  0.48 -0.01 -0.  ]
 [ 0.63  0.32  0.46  0.22  0.11  0.24  0.4   0.12  0.01  0.  ]
 [-0.2  -0.41 -0.46 -0.2  -0.17 -0.66 -0.25 -0.1  -0.   -0.01]
 [ 0.1  -0.06  0.98  0.06  0.05  0.12  0.1   0.05  0.01 -0.  ]]
```

27.6.5. Factor Scores

The factor scores are the transformed values of the original variables based on the extracted factors. These scores represent the values of the latent factors for each observation and can be used as new features in regression models, similar to principal components in PCA.

Definition 27.5 (Factor Scores)**.** A **factor score** represents the value of a latent factor for a given observation, calculated as a linear combination of the observed variables weighted by the factor score coefficients.

Mathematically, the factor score for the i-th factor and the j-th observation is defined as:

$$F_{ji} = w_{i1}x_{j1} + w_{i2}x_{j2} + \cdots + w_{ip}x_{jp} = \sum_{k=1}^{p} w_{ik}x_{jk},$$

where

- F_{ji} is the factor score for factor i and observation j,
- w_{ik} is the factor score coefficient for variable k on factor i,
- x_{jk} is the standardized value of variable k for observation j, and
- p is the number of observed variables.

```
# Factor scores for each row (shape: [n_samples, actual_factors])
X_factor_scores = fa.transform(X_encoded)
print(f"X_factor_scores shape: {X_factor_scores.shape}")

# Adapt the factor column names to the actual factor count
df_factors = pd.DataFrame(X_factor_scores, columns=factor_columns)
print(f"df_factors shape: {df_factors.shape}")
print(f"df_factors head:\n{df_factors.head()}")
```

```
X_factor_scores shape: (157, 10)
df_factors shape: (157, 10)
df_factors head:
    Factor1   Factor2   Factor3   Factor4   Factor5   Factor6   Factor7  \
0 -0.647996 -0.310986 -0.395620 -0.514476 -0.753763 -0.171572 -0.691765
1 -0.171241  0.352069 -0.579629 -0.677204  0.113380 -0.329903  0.434305
2  0.077192  0.050156 -0.595317 -0.396626  0.412052 -0.688322  0.246025
3 -0.683708  0.820534 -0.676114 -0.796906 -0.241928  0.602161  1.058645
4  0.615152 -0.262258 -0.541357 -0.489288 -1.207964 -0.186946 -0.485740

    Factor8   Factor9  Factor10
0 -0.233725  0.567292 -0.139248
1  0.852994 -0.099874  1.690789
2  0.941176 -0.209195  2.468886
3  1.063771  1.022527 -1.245557
4  0.259073  0.073952  0.308099
```

27.6.6. Creating the Regression Model with Extracted Factors (from FA)

A linear regression model is built using all ten extracted factors from Factor Analysis. The expectation is that these factors are uncorrelated, addressing multicollinearity.

```
X_model_fa = sm.add_constant(df_factors)
model_factors = sm.OLS(y, X_model_fa).fit()
print("\nRegression on Factor Scores (all 10 factors):")
print(model_factors.summary())

# Verify collinearity statistics for Factor Analysis scores (VIF and Tolerance)
coeffs_table_fa = compute_coefficients_table(
    model=model_factors, X_encoded=X_model_fa, y=y, vif_table=None
)
print("\nCoefficients Table (Factor Analysis Model):")
print(coeffs_table_fa)

# Verify condition indices
X_cond_fa = copy.deepcopy(df_factors)
condition_index_df_fa = condition_index(X_cond_fa)
print("\nCondition Index (Factor Analysis Model):")
print(condition_index_df_fa)
```

```
Regression on Factor Scores (all 10 factors):
                            OLS Regression Results
==============================================================================
Dep. Variable:               ln_sales   R-squared:                       0.485
Model:                            OLS   Adj. R-squared:                  0.449
Method:                 Least Squares   F-statistic:                     13.73
Date:                Tue, 04 Nov 2025   Prob (F-statistic):           7.69e-
17
Time:                        20:41:52   Log-Likelihood:                 -
213.62
No. Observations:                 157   AIC:                             449.2
Df Residuals:                     146   BIC:                             482.9
Df Model:                          10
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          3.2959      0.078     42.215      0.000       3.142       3.450
Factor1       -0.1366      0.078     -1.749      0.082      -0.291       0.018
Factor2       -0.7022      0.078     -8.994      0.000      -0.856      -
0.548
Factor3        0.3035      0.078      3.888      0.000       0.149       0.458
Factor4     9.177e-06      0.078      0.000      1.000      -0.154       0.154
Factor5        0.1719      0.078      2.201      0.029       0.018       0.326
```

```
Factor6       -0.1653      0.078     -2.117      0.036      -0.320      -
0.011
Factor7        0.4130      0.078      5.290      0.000       0.259       0.567
Factor8       -0.0072      0.078     -0.092      0.927      -0.161       0.147
Factor9        0.0317      0.078      0.407      0.685      -0.123       0.186
Factor10       0.0665      0.078      0.852      0.396      -0.088       0.221
==============================================================================
Omnibus:                       41.296   Durbin-Watson:                   1.423
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              107.145
Skew:                          -1.064   Prob(JB):                     5.42e-
24
Kurtosis:                       6.442   Cond. No.                         1.00
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Coefficients Table (Factor Analysis Model):
   Variable  Zero-Order r  Partial r  Semipartial r  Tolerance  VIF
0   Factor1     -0.103920  -0.143257      -0.103920        1.0  1.0
1   Factor2     -0.534367  -0.597080      -0.534367        1.0  1.0
2   Factor3      0.231004   0.306300       0.231004        1.0  1.0
3   Factor4      0.000007   0.000010       0.000007        1.0  1.0
4   Factor5      0.130790   0.179228       0.130790        1.0  1.0
5   Factor6     -0.125772  -0.172560      -0.125772        1.0  1.0
6   Factor7      0.314284   0.401023       0.314284        1.0  1.0
7   Factor8     -0.005478  -0.007630      -0.005478        1.0  1.0
8   Factor9      0.024158   0.033630       0.024158        1.0  1.0
9  Factor10      0.050594   0.070298       0.050594        1.0  1.0

Condition Index (Factor Analysis Model):
   Index  Eigenvalue  Condition Index
0      0     1.00641              1.0
1      1     1.00641              1.0
2      2     1.00641              1.0
3      3     1.00641              1.0
4      4     1.00641              1.0
5      5     1.00641              1.0
6      6     1.00641              1.0
7      7     1.00641              1.0
8      8     1.00641              1.0
9      9     1.00641              1.0
```

As expected, the collinearity statistics (VIF and Tolerance) for the factor values show that they are uncorrelated (VIF=1, Tolerance=1). The condition indices are also all

close to 1, confirming that Factor Analysis successfully mitigates multicollinearity. The coefficient estimates are larger relative to their standard errors compared to the original model, which can lead to more factors being identified as statistically significant.

If the R-squared and Adjusted R-squared values for `model_factors` are close to those of the original `model`, it indicates that the regression model based on Factor Analysis performs similarly well, while successfully reducing multicollinearity. When all factors are used, the predictive performance metrics are identical to the original OLS model.

27.6.7. Factor Analysis: Creating the Regression Model with three Extracted Factors only

27.6.7.1. Setting Up the Regression Model with Reduced Factors

To demonstrate the effect of dimensionality reduction, a regression model is created using only the first three extracted factors from Factor Analysis.

```
# Create a regression model using only the first three factors
df_factors_reduced = df_factors.iloc[:, :3] # select the first three factors
X_model_fa_reduced = sm.add_constant(df_factors_reduced)
model_factors_reduced = sm.OLS(y, X_model_fa_reduced).fit()
print("\nRegression on Factor Scores (three factors only):")
print(model_factors_reduced.summary())

# Verify collinearity statistics for reduced FA scores
coeffs_table_fa_reduced = compute_coefficients_table(
    model=model_factors_reduced, X_encoded=X_model_fa_reduced, y=y, vif_table=None
)
print("\nCoefficients Table (Reduced Factor Analysis Model):")
print(coeffs_table_fa_reduced)

# Verify condition indices for reduced FA scores
X_cond_fa_reduced = copy.deepcopy(df_factors_reduced)
condition_index_df_fa_reduced = condition_index(X_cond_fa_reduced)
print("\nCondition Index (Reduced Factor Analysis Model):")
print(condition_index_df_fa_reduced)
```

```
Regression on Factor Scores (three factors only):
                            OLS Regression Results
==============================================================================
Dep. Variable:               ln_sales   R-squared:                       0.350
Model:                            OLS   Adj. R-squared:                  0.337
```

```
Method:                 Least Squares   F-statistic:                     27.43
Date:                Tue, 04 Nov 2025   Prob (F-statistic):           2.99e-
14
Time:                        20:41:52   Log-Likelihood:                -
231.87
No. Observations:                 157   AIC:                             471.7
Df Residuals:                     153   BIC:                             484.0
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          3.2959      0.086     38.474      0.000       3.127       3.465
Factor1       -0.1366      0.086     -1.594      0.113      -0.306       0.033
Factor2       -0.7022      0.086     -8.197      0.000      -0.871      -
0.533
Factor3        0.3035      0.086      3.543      0.001       0.134       0.473
==============================================================================
Omnibus:                       43.992   Durbin-Watson:                   1.418
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              134.618
Skew:                          -1.068   Prob(JB):                     5.86e-
30
Kurtosis:                       7.002   Cond. No.                         1.00
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Coefficients Table (Reduced Factor Analysis Model):
  Variable  Zero-Order r  Partial r  Semipartial r  Tolerance  VIF
0  Factor1     -0.103920  -0.127811      -0.103920        1.0  1.0
1  Factor2     -0.534367  -0.552381      -0.534367        1.0  1.0
2  Factor3      0.231004   0.275385       0.231004        1.0  1.0

Condition Index (Reduced Factor Analysis Model):
   Index  Eigenvalue  Condition Index
0      0     1.00641              1.0
1      1     1.00641              1.0
2      2     1.00641              1.0
```

The collinearity statistics for the reduced factor set continue to show that they are uncorrelated, with VIFs of 1.0 and condition indices close to 1.

27.6.7.2. Comparison of Model Performance of the Reduced FA Model and the Full OLS Model

When reducing the number of factors from 10 to 3, the R-squared and Adjusted R-squared values for the Factor Analysis model decrease significantly (from ~0.48 to ~0.35). This indicates a trade-off: while reducing dimensionality successfully addresses multicollinearity, retaining too few factors can lead to information loss and reduced predictive accuracy. Lower MSE and RMSE values still suggest better predictive performance for the full OLS model in this specific comparison, as it retains more information.

27.7. Summary: Comparing OLS, PCA, and Factor Analysis Models

Multicollinearity is a common issue in regression models that can lead to unstable and difficult-to-interpret coefficients. Both Principal Component Analysis (PCA) and Factor Analysis (FA) are powerful techniques for addressing multicollinearity and reducing dimensionality.

- **PCA** is a standard method for addressing multicollinearity by transforming correlated variables into uncorrelated principal components. These components can be effectively used in linear regression and other models like Random Forest. While PCA components are not always easy to interpret directly in terms of original variables, they excel at data compression and reducing model complexity.
- **Factor Analysis** provides a way to simplify data by identifying underlying latent structures (factors) that explain correlations among variables. It also results in uncorrelated factors, making it suitable for regression problems affected by multicollinearity. Interpretation of factors relies on factor loadings.

The choice between PCA and Factor Analysis depends on the specific goals: PCA for dimensionality reduction and variance explanation, FA for discovering latent constructs. Both are valuable tools in the data scientist's toolkit for handling complex, highly correlated datasets.

27.7.1. Interpretation of the Regression Models

- **OLS Model (`model`):** This model uses the original variables directly. Coefficients indicate the direct relationship between each original variable and the target variable.

- **PCA Regression Model (`model_pca`):** This model uses principal components, which are linear combinations of the original variables, as predictors. The coefficients show the relationship between the target variable and these abstract components.
- **Factor Analysis Model (`model_factors`):** This model uses extracted factors, which are also linear combinations of original variables, designed to capture underlying latent structures. Coefficients indicate the relationship between the target variable and these latent factors.

27.7.2. Differences Compared to the Standard OLS Model

Feature	OLS Model (Standard)	PCA Regression Model	Factor Analysis Model
Input Variables	Uses original variables (e.g., `X_encoded`) as predictors.	Uses principal components (e.g., `df_pca_components`) as predictors.	Uses extracted factors (e.g., `df_factors`) as predictors.
Multicollinearity	Can suffer from multicollinearity if predictors are highly correlated, leading to unstable coefficients and inflated standard errors.	Reduces multicollinearity because principal components are orthogonal (uncorrelated).	Reduces multicollinearity by using uncorrelated factors as predictors.
Interpretability	Coefficients correspond directly to original variables, making interpretation straightforward.	Coefficients relate to abstract principal components, making direct interpretation of original variable influence more challenging.	Coefficients relate to abstract factors, making interpretation more challenging. Factor loadings must be analyzed for meaning.
Dimensionality	Uses all original variables, potentially including redundant or irrelevant features.	Reduces the number of predictors by combining original variables into fewer principal components.	Reduces the number of predictors by combining original variables into fewer factors.
Purpose	Direct relationship modeling, inference.	Dimensionality reduction, variance maximization, multicollinearity mitigation.	Discovering latent structures, explaining correlations.
Assumptions	None on underlying structure beyond linear relationship.	Does not assume an underlying causal model.	Assumes observed variables are caused by underlying factors.

Feature	OLS Model (Standard)	PCA Regression Model	Factor Analysis Model
Error Variance	Does not explicitly separate unique variance.	Does not separate unique variance from common variance.	Explicitly models unique variance for each variable.

27.7.3. Key Differences Between Loading Scores (PCA) and Factor Loadings (FA)

Aspect	Loading Scores (PCA)	Factor Loadings (FA)
Context	Principal Component Analysis (PCA)	Factor Analysis (FA)
Purpose	Describe the contribution of variables to principal components.	Describe the relationship between variables and latent factors.
Underlying Model	No assumption of latent structure; purely variance-based.	Assumes a latent structure explaining observed variables.
Error Term	PCA does not explicitly model error variance.	FA explicitly models unique (error) variance for each variable.
Interpretability	Components are orthogonal (uncorrelated).	Factors may not be orthogonal, depending on rotation.

While both loading scores and factor loadings describe relationships between variables and derived components or factors, **loading scores** are specific to PCA and focus on maximizing variance, while **factor loadings** are specific to FA and aim to uncover latent structures.

27.7.4. Advantages of Using PCA and FA

Principal Component Analysis (PCA):

- **Reduced Multicollinearity:** By using uncorrelated principal components, the model avoids instability caused by multicollinearity.
- **Dimensionality Reduction:** The model uses fewer predictors if desired, improving computational efficiency and potentially generalization by removing noise.
- **Variance Maximization:** Components are constructed to capture the maximum possible variance from the original data.

Factor Analysis (FA):

- **Reduced Multicollinearity:** Similar to PCA, using uncorrelated factors prevents instability from multicollinearity.
- **Dimensionality Reduction:** Reduces the number of predictors, improving computational efficiency and generalization.
- **Focus on Underlying Structure:** Factor analysis aims to capture the latent structure of the data, potentially providing better insights into the fundamental relationships between variables.

27.7.5. Disadvantages of Using PCA and FA

Principal Component Analysis (PCA):

- **Loss of Interpretability:** Principal components are abstract combinations of the original variables, making it harder to directly interpret the coefficients. Understanding individual variable influence requires examining loading scores.
- **Potential Information Loss:** If too few components are retained, information from the original variables may be lost, potentially reducing predictive accuracy.

Factor Analysis (FA):

- **Loss of Interpretability:** Factors are abstract combinations of the original variables, making it harder to directly interpret the coefficients. Factor loadings must be analyzed to understand the influence of individual variables.
- **Potential Information Loss:** If too few factors are retained, information from the original variables may be lost, reducing predictive accuracy.
- **Complexity:** The process of extracting factors and interpreting their meaning adds complexity to the modeling process.
- **Dependence on Factor Selection:** The number of factors to retain is subjective and can affect model performance. Too few factors may oversimplify, while too many may reintroduce multicollinearity.
- **Assumption of Latent Structure:** Relies on the assumption that an underlying latent structure exists, which may not always be true for all datasets.

27.7.6. When to Use Which Method

- Use PCA when the primary goal is **dimensionality reduction**, **data compression**, and **multicollinearity resolution**, especially when interpretability of the new components is secondary to predictive performance.
- Use Factor Analysis when the goal is to **uncover underlying latent constructs or factors** that explain the relationships among variables, and when you seek to understand the conceptual meaning of these latent variables, even if it adds complexity.

- The original OLS model is preferable when **interpretability of original variables is crucial** and multicollinearity is not a significant issue.

27.8. Using Principal Components / Factors in Other Models

The principal components from PCA or factors from Factor Analysis can also be effectively used as predictors in other machine learning models, not just linear regression.

27.8.1. Random Forest Regressor with the Full Dataset

First, a Random Forest Regressor is trained using the original, full dataset (`X_encoded`).

```
# 1. Prepare Data #
# Use the original input features (X_encoded) as predictors
X_original = X_encoded
# Split the data into training and testing sets
X_train_orig, X_test_orig, y_train_orig, y_test_orig = train_test_split(X_original, y

# 2. Fit Random Forest Model
from sklearn.ensemble import RandomForestRegressor
rf_model_orig = RandomForestRegressor(n_estimators=100, random_state=42)
# Train the model on the training data
rf_model_orig.fit(X_train_orig, y_train_orig)

# 3. Evaluate the Model
# Make predictions on the test set
y_pred_orig = rf_model_orig.predict(X_test_orig)
# Calculate evaluation metrics
r2_rf_orig = r2_score(y_test_orig, y_pred_orig)
mse_rf_orig = mean_squared_error(y_test_orig, y_pred_orig)
rmse_rf_orig = np.sqrt(mse_rf_orig)

# Print the results
print("\nRandom Forest Model (using original data):")
print(f"R-squared: {r2_rf_orig:.4f}")
print(f"MSE: {mse_rf_orig:.4f}")
print(f"RMSE: {rmse_rf_orig:.4f}")
```

```
Random Forest Model (using original data):
R-squared: 0.4032
MSE: 1.3118
RMSE: 1.1453
```

27.8.2. Random Forest Regressor with PCA Components

Next, a Random Forest Regressor is trained using the principal components derived from PCA. This tests if the dimensionality reduction and multicollinearity resolution of PCA benefit non-linear models.

```
# 1. Prepare Data
# Use the extracted PCA components as predictors (using the 10 components)
X_pca_rf = df_pca_components

# Split the data into training and testing sets
X_train_pca_rf, X_test_pca_rf, y_train_pca_rf, y_test_pca_rf = train_test_split(X_pca_rf, y, test_si

# 2. Fit Random Forest Model
# Initialize the Random Forest Regressor
rf_model_pca = RandomForestRegressor(n_estimators=100, random_state=42)
# Train the model on the training data
rf_model_pca.fit(X_train_pca_rf, y_train_pca_rf)

#  3. Evaluate the Model
# Make predictions on the test set
y_pred_pca_rf = rf_model_pca.predict(X_test_pca_rf)
# Calculate evaluation metrics
r2_rf_pca = r2_score(y_test_pca_rf, y_pred_pca_rf)
mse_rf_pca = mean_squared_error(y_test_pca_rf, y_pred_pca_rf)
rmse_rf_pca = np.sqrt(mse_rf_pca)

# Print the results
print("\nRandom Forest Model (using PCA components):")
print(f"R-squared: {r2_rf_pca:.4f}")
print(f"MSE: {mse_rf_pca:.4f}")
print(f"RMSE: {rmse_rf_pca:.4f}")
```

```
Random Forest Model (using PCA components):
R-squared: 0.2871
MSE: 1.5670
RMSE: 1.2518
```

27.8.3. Random Forest Regressor with Extracted Factors (from FA)

Finally, a Random Forest Regressor is trained using the extracted factors from Factor Analysis (using the 3 factors from the reduced model for this example to illustrate potential impact of reduction).

```
# 1. Prepare Data
# Use the extracted factors as predictors (using the 3 factors from the reduced FA mod
X_factors_rf = df_factors_reduced

# Split the data into training and testing sets
X_train_fa_rf, X_test_fa_rf, y_train_fa_rf, y_test_fa_rf = train_test_split(X_factors_

# 2. Fit Random Forest Model
# Initialize the Random Forest Regressor
rf_model_fa = RandomForestRegressor(n_estimators=100, random_state=42)
# Train the model on the training data
rf_model_fa.fit(X_train_fa_rf, y_train_fa_rf)

# 3. Evaluate the Model
# Make predictions on the test set
y_pred_fa_rf = rf_model_fa.predict(X_test_fa_rf)
# Calculate evaluation metrics
r2_rf_fa = r2_score(y_test_fa_rf, y_pred_fa_rf)
mse_rf_fa = mean_squared_error(y_test_fa_rf, y_pred_fa_rf)
rmse_rf_fa = np.sqrt(mse_rf_fa)

# Print the results
print("\nRandom Forest Model (using extracted factors):")
print(f"R-squared: {r2_rf_fa:.4f}")
print(f"MSE: {mse_rf_fa:.4f}")
print(f"RMSE: {rmse_rf_fa:.4f}")
```

```
Random Forest Model (using extracted factors):
R-squared: 0.2901
MSE: 1.5605
RMSE: 1.2492
```

27.8.4. Comparison of the Random Forest Models

```python
# Print comparison of Random Forest models
print("\nComparison of Random Forest Models:")
print("\nUsing Original Data:")
print(f"R-squared: {r2_rf_orig:.4f}")
print(f"MSE: {mse_rf_orig:.4f}")
print(f"RMSE: {rmse_rf_orig:.4f}")

print("\nUsing PCA Components:")
print(f"R-squared: {r2_rf_pca:.4f}")
print(f"MSE: {mse_rf_pca:.4f}")
print(f"RMSE: {rmse_rf_pca:.4f}")

print("\nUsing Extracted Factors (from FA):")
print(f"R-squared: {r2_rf_fa:.4f}")
print(f"MSE: {mse_rf_fa:.4f}")
print(f"RMSE: {rmse_rf_fa:.4f}")
```

```
Comparison of Random Forest Models:

Using Original Data:
R-squared: 0.4032
MSE: 1.3118
RMSE: 1.1453

Using PCA Components:
R-squared: 0.2871
MSE: 1.5670
RMSE: 1.2518

Using Extracted Factors (from FA):
R-squared: 0.2901
MSE: 1.5605
RMSE: 1.2492
```

In this example, for Random Forest, using the reduced set of 3 factors from PCA and FA led to a decrease in R-squared and an increase in MSE/RMSE compared to using the original variables. This highlights that while dimensionality reduction can be beneficial, choosing too few components or factors can lead to information loss, negatively impacting predictive performance.

27.9. Videos: Principal Component Analysis (PCA)

- Video: Principal Component Analysis (PCA), Step-by-Step
- Video: PCA - Practical Tips
- Video: PCA in Python

27.10. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

28. Regression

28.1. Supervised and Unsupervised Learning

Two important types: supervised and unsupervised learning. There is even more, e.g., semi-supervised learning.

28.1.1. Starting point

- Outcome measurement Y (dependent variable, response, target).
- Vector of p predictor measurements X (inputs, regressors, covariates, features, independent variables).
- Training data $(x_1, y1), \ldots, (x_N, y_N)$. These are observations (examples, instances) of these measurements.

In the *regression* problem, Y is quantitative (e.g., price, blood pressure). In the *classification* problem, Y takes values in a finite, unordered set (e.g., survived/died, digit 0-9, cancer class of tissue sample).

28.1.2. Philosophy

It is important to understand the ideas behind the various techniques, in order to know how and when to use them. One has to understand the simpler methods first, in order to grasp the more sophisticated ones. It is important to accurately assess the performance of a method, to know how well or how badly it is working (simpler methods often perform as well as fancier ones!) This is an exciting research area, having important applications in science, industry and finance. Statistical learning is a fundamental ingredient in the training of a modern data scientist.

28.1.3. Supervised Learning

Objectives of supervised learning: On the basis of the training data we would like to:

- Accurately predict unseen test cases.
- Understand which inputs affect the outcome, and how.
- Assess the quality of our predictions and inferences.

Note: Supervised means Y is known.

Exercise 28.1.

- Do children learn supervised?
- When do you learn supervised?
- Can learning be unsupervised?

28.1.4. Unsupervised Learning

No outcome variable, just a set of predictors (features) measured on a set of samples. The objective is more fuzzy—find groups of samples that behave similarly, find features that behave similarly, find linear combinations of features with the most variation. It is difficult to know how well your are doing. Unsupervised learning different from supervised learning, but can be useful as a pre-processing step for supervised learning. Clustering and principle component analysis are important techniques.

Unsupervised: Y is unknown, there is no Y, no trainer, no teacher, but: distances between the inputs values (features). A distance (or similarity) measure is necessary.

28.1.4.0.1. Statistical Learning

We consider supervised learning first.

Figure 28.1.: Sales as a function of TV, radio and newspaper. Taken from James et al. (2014)

Sales figures from a marketing campaign, see Figure 28.1. Trend shown using regression. First seems to be stronger than the third.

Can we predict $Y =$ Sales using these three? Perhaps we can do better using a model

$$Y = Sales \approx f(X_1 = TV, X_2 = Radio, X_3 = Newspaper)$$

modeling the joint relationsship.

Here Sales is a response or target that we wish to predict. We generically refer to the response as Y. TV is a feature, or input, or predictor; we name it X_1. Likewise name Radio as X_2, and so on. We can refer to the input vector collectively as

$$X = \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix}$$

Now we write our model as

$$Y = f(X) + \epsilon$$

where ϵ captures measurement errors and other discrepancies.

What is f good for? With a good f we can make predictions of Y at new points $X = x$. We can understand which components of $X = (X_1, X_2, \dots X_p)$ are important in explaining Y, and which are irrelevant.

For example, Seniority and Years of Education have a big impact on Income, but Marital Status typically does not. Depending on the complexity of f, we may be able to understand how each component X_j of X affects Y.

28.2. Linear Regression

28.2.1. The main ideas of fitting a line to data (The main ideas of least squares and linear regression.)

- Video: The main ideas of fitting a line to data (The main ideas of least squares and linear regression.)

28.2.1.1. Linear Regression

- Video: Linear Regression, Clearly Explained

28.2.1.2. Multiple Regression

- Video: Multiple Regression, Clearly Explained

28.2.1.3. A Gentle Introduction to Machine Learning

- Video: A Gentle Introduction to Machine Learning

28.2.1.4. Regression Function

Figure 28.2.: Scatter plot of 2000 points (population). What is a good function f? There are many function values at $X = 4$. A function can return only one value. We can take the mean from these values as a return value. Taken from James et al. (2014)

Consider Figure 28.2. Is there an ideal $f(X)$? In particular, what is a good value for $f(X)$ at any selected value of X, say $X = 4$? There can be many Y values at $X = 4$. A good value is

$$f(4) = E(Y|X = 4).$$

$E(Y|X = 4)$ means **expected value** (average) of Y given $X = 4$.

The ideal $f(x) = E(Y|X = x)$ is called the **regression function**. Read: The regression function gives the conditional expectation of Y given X.

The regression function $f(x)$ is also defined for the vector X; e.g., $f(x) = f(x_1, x_2, x_3) = E(Y|X_1 = x_1, X_2 = x_2, X_3 = x_3)$.

28.2.2. Optimal Predictor

The regression function is the **ideal** or **optimal predictor** of Y with regard to mean-squared prediction error: It means that $f(x) = E(Y|X = x)$ is the function that

minimizes

$$E[(Y - g(X))^2 | X = x]$$

over all functions g at all points $X = x$.

28.2.2.1. Residuals, Reducible and Irreducible Error

At each point X we make mistakes:

$$\epsilon = Y - f(x)$$

is the **residual**. Even if we knew $f(x)$, we would still make errors in prediction, since at each $X = x$ there is typically a distribution of possible Y values as is illustrated in Figure 28.2.

For any estimate $\hat{f}(x)$ of $f(x)$, we have

$$E\left[(Y - \hat{f}(X))^2 | X = x\right] = \left[f(x) - \hat{f}(x)\right]^2 + \text{var}(\epsilon),$$

and $\left[f(x) - \hat{f}(x)\right]^2$ is the **reducible** error, because it depends on the model (changing the model f might reduce this error), and $\text{var}(\epsilon)$ is the **irreducible** error.

28.2.2.2. Local Regression (Smoothing)

Typically we have few if any data points with $X = 4$ exactly. So we cannot compute $E(Y|X = x)$! Idea: Relax the definition and let

$$\hat{f}(x) = Ave(Y|X \in N(x)),$$

where $N(x)$ is some neighborhood of x, see Figure 28.3.

Figure 28.3.: Relaxing the definition. There is no Y value at $X = 4$. Taken from James et al. (2014)

Nearest neighbor averaging can be pretty good for small p, i.e., $p \leq 4$ and large-ish N. We will discuss smoother versions, such as kernel and spline smoothing later in the course.

28.2.3. Curse of Dimensionality and Parametric Models

Figure 28.4.: A 10% neighborhood in high dimensions need no longer be local. Left: Values of two variables x_1 and x_2, uniformly distributed. Form two 10% neighborhoods: (a) the first is just involving x_1 ignoring x_2. (b) is the neighborhood in two dimension. Notice that the radius of the circle is much larger than the lenght of the interval in one dimension. Right: radius plotted against fraction of the volume. In 10 dim, you have to break out the interval $[-1; +1]$ to get 10% of the data. Taken from James et al. (2014)

Local, e.g., nearest neighbor, methods can be lousy when p is large. Reason: **the curse of dimensionality**, i.e., nearest neighbors tend to be far away in high dimensions. We need to get a reasonable fraction of the N values of y_i to average to bring the variance down—e.g., 10%. A 10% neighborhood in high dimensions need no longer be local, so we lose the spirit of estimating $E(Y|X = x)$ by local averaging, see Figure 28.4. If the curse of dimensionality does not exist, nearest neighbor models would be perfect prediction models.

We will use structured (parametric) models to deal with the curse of dimensionality. The linear model is an important example of a parametric model:

$$f_L(X) = \beta_0 + \beta_1 X_1 + ... + \beta_p X_p.$$

A linear model is specified in terms of $p + 1$ parameters $\beta_1, \beta_2, ..., \beta_p$. We estimate the parameters by fitting the model to *training data*. Although it is almost never correct, a linear model often serves as a good and interpretable approximation to the unknown true function $f(X)$.

The linear model is avoiding the curse of dimensionality, because it is not relying on any local properties. Linear models belong to the class of *model-based* approaches: they replace the problem of estimating f with estimating a fixed set of coefficients β_i, with $i = 1, 2, \ldots, p$.

Figure 28.5.: A linear model $\hat{f}_L$ gives a reasonable fit. Taken from James et al. (2014)

Figure 28.6.: A quadratic model $\hat{f}_Q$ fits slightly better. Taken from James et al. (2014)

A linear model

$$\hat{f}_L(X) = \hat{\beta}_0 + \hat{\beta}_1 X$$

gives a reasonable fit, see Figure 28.5. A quadratic model

$$\hat{f}_Q(X) = \hat{\beta}_0 + \hat{\beta}_1 X + \hat{\beta}_2 X^2$$

gives a slightly improved fit, see Figure 28.6.

Figure 28.7 shows a simulated example. Red points are simulated values for income from the model

$$income = f(education, seniority) + \epsilon$$

f is the blue surface.

Figure 28.7.: The true model. Red points are simulated values for income from the model, f is the blue surface. Taken from James et al. (2014)

Figure 28.8.: Linear regression fit to the simulated data (red points). Taken from James et al. (2014)

The linear regression model

$$\hat{f}(education, seniority) = \hat{\beta}_0 + \hat{\beta}_1 \times education + \hat{\beta}_2 \times seniority$$

captures the important information. But it does not capture everything. More flexible regression model

$$\hat{f}_S(education, seniority)$$

fit to the simulated data. Here we use a technique called a **thin-plate spline** to fit a flexible surface. Even more flexible spline regression model

$$\hat{f}_S(education, seniority)$$

fit to the simulated data. Here the fitted model makes no errors on the training data! Also known as **overfitting**.

Figure 28.9.: Thin-plate spline models $\hat{f}_S(education, seniority)$ fitted to the model from Figure 28.7. Taken from James et al. (2014)

28.2.3.1. Trade-offs

- Prediction accuracy versus interpretability: Linear models are easy to interpret; thin-plate splines are not.
- Good fit versus over-fit or under-fit: How do we know when the fit is just right?
- Parsimony (Occam's razor) versus black-box: We often prefer a simpler model involving fewer variables over a black-box predictor involving them all.

The trad-offs are visualized in Figure 28.11.

Figure 28.10.: Thin-plate spline models $\hat{f}_S(education, seniority)$ fitted to the model from Figure 28.7. The model makes no errors on the training data (overfitting). Taken from James et al. (2014)

Figure 28.11.: Interpretability versus flexibility. Flexibility corresponds with the number of model parameters. Taken from James et al. (2014)

28.2.4. Assessing Model Accuracy and Bias-Variance Trade-off

Figure 28.12.: Black curve is truth. Red curve on right is $MSETe$, grey curve is $MSETr$. Orange, blue and green curves/squares correspond to fits of different flexibility. The dotted line represents the irreducible error, i.e., $var(\epsilon)$. Taken from James et al. (2014)

Suppose we fit a model $f(x)$ to some training data $Tr = \{x_i, y_i\}_1^N$, and we wish to see how well it performs. We could compute the average squared prediction error over Tr:

$$MSE_{Tr} = Ave_{i \in Tr}[y_i - \hat{f}(x_i)]^2.$$

This may be biased toward more overfit models. Instead we should, if possible, compute it using fresh **test data** $Te == \{x_i, y_i\}_1^N$:

$$MSE_{Te} = Ave_{i \in Te}[y_i - \hat{f}(x_i)]^2.$$

The red curve, which illustrated the test error, can be estimated by holding out some data to get the test-data set.

28.2.4.1. Bias-Variance Trade-off

Suppose we have fit a model $f(x)$ to some training data Tr, and let (x_0, y_0) be a test observation drawn from the population. If the true model is

$$Y = f(X) + \epsilon \qquad \text{with } f(x) = E(Y|X = x),$$

Figure 28.13.: Here, the truth is smoother. Black curve is truth. Red curve on right is $MSETe$, grey curve is $MSETr$. Orange, blue and green curves/squares correspond to fits of different flexibility. The dotted line represents the irreducible error, i.e., $var(\epsilon)$. Taken from James et al. (2014)

Figure 28.14.: Here the truth is wiggly and the noise is low, so the more flexible fits do the best. Black curve is truth. Red curve on right is $MSETe$, grey curve is $MSETr$. Orange, blue and green curves/squares correspond to fits of different flexibility. The dotted line represents the irreducible error, i.e., $var(\epsilon)$. Taken from James et al. (2014)

then

$$E\left(y_0 - \hat{f}(x_0)\right)^2 = \text{var}(\hat{f}(x_0)) + [Bias(\hat{f}(x_0))]^2 + \text{var}(\epsilon). \tag{28.1}$$

Here, var(ϵ) is the irreducible error. The reducible error consists of two components:

- var($\hat{f}(x_0)$) is the variance that comes from different training sets. Different training sets result in different functions $\hat{f}$.
- $Bias(\hat{f}(x_0)) = E[\hat{f}(x_0)] - f(x_0)$.

The expectation averages over the variability of y_0 as well as the variability in Tr. Note that

$$Bias(\hat{f}(x_0)) = E[\hat{f}(x_0)] - f(x_0).$$

Typically as the flexibility of $\hat{f}$ increases, its variance increases (because the fits differ from training set to trainig set), and its bias decreases. So choosing the flexibility based on average test error amounts to a bias-variance trade-off, see Figure 28.15.

Figure 28.15.: Bias-variance trade-off for the three examples. Taken from James et al. (2014)

If we add the two components (reducible and irreducible error), we get the MSE in Figure 28.15 as can be seen in Equation 28.1.

- Video: Machine Learning Fundamentals: Bias and Variance

28.3. Multiple Regression

28.4. R-squared

28.4.1. R-Squared in Simple Linear Regression

In simple linear regression, the relationship between the independent variable X and the dependent variable Y is modeled using the equation:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

Here, β_0 is the intercept, β_1 is the slope or regression coefficient, and ϵ is the error term.

Definition 28.1 (R-Squared (R^2))**.** R^2 is a measure of how well the regression model explains the variance in the dependent variable. It is calculated as the square of the correlation coefficient (r) between the actual values Y and the predicted values $\hat{Y}$ from the regression model. It ranges from 0 to 1, where:

- 1 indicates that the regression predictions perfectly fit the data.
- 0 indicates that the model does not explain any of the variability in the target data around its mean.

In simple linear regression, where there is one independent variable X and one dependent variable Y, the R-squared (R^2) is the square of the Pearson correlation coefficient (r) between the observed values of the dependent variable and the values predicted by the regression model. That is, in simple linear regression, we have

$$R^2 = r^2.$$

This equivalence holds specifically for simple linear regression due to the direct relationship between the linear fit and the correlation of two variables. In multiple linear regression, while R^2 still represents the proportion of variance explained by the model, it is not simply the square of a single correlation coefficient as it involves multiple predictors.

- Video: R-squared, Clearly Explained

28.5. Assessing Confounding Effects in Multiple Regression

Confounding is a bias introduced by the imbalanced distribution of extraneous risk factors among comparison groups (Wang 2007). `spotpython` provides tools for assessing confounding effects in multiple regression models.

Example 28.1 (Assessing Confounding Effects in Multiple Regression with `spotpython`)**.** Consider the following data generation function `generate_data` and the `fit_ols_model` function to fit an ordinary least squares (OLS) regression model.

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

def generate_data(n_samples=100, b0=0, b1=-1, b2=0, b3=10, b12=0, b13=0, b23=0, b123=0
    """
    Generate data for the linear formula y ~ b0 + b1*x1 + b2*x2 + b3*x3 + b12*x1*x2 +

    Args:
        n_samples (int): Number of samples to generate.
        b0 (float): Coefficient for the intercept.
        b1 (float): Coefficient for x1.
        b2 (float): Coefficient for x2.
        b3 (float): Coefficient for x3.
        b12 (float): Coefficient for the interaction term x1*x2.
        b13 (float): Coefficient for the interaction term x1*x3.
        b23 (float): Coefficient for the interaction term x2*x3.
        b123 (float): Coefficient for the interaction term x1*x2*x3.
        noise_std (float): Standard deviation of the Gaussian noise added to y.

    Returns:
        pd.DataFrame: A DataFrame containing the generated data with columns ['x1', 'x
    """
    np.random.seed(42)  # For reproducibility
    x1 = np.random.uniform(0, 1, n_samples)
    x2 = np.random.uniform(0, 1, n_samples)
    x3 = np.random.uniform(0, 1, n_samples)

    y = (b0 + b1*x1 + b2*x2 + b3*x3 + b12*x1*x2 + b13*x1*x3 + b23*x2*x3 + b123*x1*x2*x
         np.random.normal(0, noise_std, n_samples))

    data = pd.DataFrame({'y': y, 'x1': x1, 'x2': x2, 'x3': x3})
    return data
```

```python
def fit_ols_model(formula, data) -> dict:
    """
    Fit an OLS model using the given formula and data, and print the results.

    Args:
        formula (str): The formula for the OLS model.
        data (pd.DataFrame): The data frame containing the variables.

    Returns:
        dict: A dictionary containing the p-values, estimates, confidence intervals, and AIC value.
    """
    mod_0 = smf.ols(formula=formula, data=data).fit()
    p = mod_0.pvalues.iloc[1]
    estimate = mod_0.params.iloc[1]
    conf_int = mod_0.conf_int().iloc[1]
    aic_value = mod_0.aic

    print(f"p-values: {p}")
    print(f"estimate: {estimate}")
    print(f"conf_int: {conf_int}")
    print(f"aic: {aic_value}")
```

These functions can be used to generate data and fit an OLS model. Here we use the model

$$y = f(x_1, x_2, x_3) + \epsilon = x_1 + 10x_3 + \epsilon.$$

We set up the basic model $y_0 = f_0(x_1)$ and analyze how the model fit changes when adding x_2 and x_3 to the model. If the p-values are decreasing by adding a variable, this indicates that the variable is relevant for the model. Similiarly, if the p-values are increasing by removing a variable, this indicates that the variable is not relevant for the model.

```python
data = generate_data(b0=0, b1=1, b2=0, b3=10, b12=0, b13=0, b23=0, b123=0, noise_std=1)
fit_ols_model("y ~ x1", data)
fit_ols_model("y ~ x1 + x2", data)
fit_ols_model("y ~ x1 + x3", data)
fit_ols_model("y ~ x1 + x2 + x3", data)
```

```
p-values: 0.34343741859526267
estimate: 1.025306391110114
conf_int: 0   -1.111963
1    3.162575
Name: x1, dtype: float64
```

```
aic: 517.6397392012537
p-values: 0.3637511850778461
estimate: 0.9810502049698089
conf_int: 0   -1.152698
1    3.114798
Name: x1, dtype: float64
aic: 518.1426513151566
p-values: 4.9467606744218404e-05
estimate: 1.4077923469421165
conf_int: 0    0.750106
1    2.065479
Name: x1, dtype: float64
aic: 282.73524524532
p-values: 4.849840959643538e-05
estimate: 1.4159292625696247
conf_int: 0    0.755494
1    2.076364
Name: x1, dtype: float64
aic: 284.34665447613634
```

The function `fit_all_lm()` simplifies this procedure. It can be used to fit all possible linear models with the given data and print the results in a systematic way for various combinations of variables.

```
from spotpython.utils.stats import fit_all_lm, plot_coeff_vs_pvals, plot_coeff_vs_pval
res = fit_all_lm("y ~ x1", ["x2", "x3"], data)
print(res["estimate"])
```

```
The basic model is: y ~ x1
The following features will be used for fitting the basic model: Index(['x1', 'y', 'x3
p-values: 0.34343741859526267
estimate: 1.025306391110114
conf_int: 0   -1.111963
1    3.162575
Name: x1, dtype: float64
aic: 517.6397392012537
Combinations: [('x2',), ('x3',), ('x2', 'x3')]
  variables  estimate  conf_low  conf_high         p         aic    n
0     basic  1.025306 -1.111963   3.162575  0.343437  517.639739  100
1        x2  0.981050 -1.152698   3.114798  0.363751  518.142651  100
2        x3  1.407792  0.750106   2.065479  0.000049  282.735245  100
3    x2, x3  1.415929  0.755494   2.076364  0.000048  284.346654  100
```

Interpreting the results, we can see that the p-values decrease when adding x_3 (as well as both x_2 and $x_3) to the model, indicating that x_3 is relevant for the model. Adding only x_2 does not significantly improve the model fit.

In addition to the textural output, the function `plot_coeff_vs_pvals_by_included()` can be used to visualize the coefficients and p-values of the fitted models.

```
plot_coeff_vs_pvals_by_included(res)
```

Figure 28.16.: Coefficients vs. p-values for different models. The right plot indicates that x3 should be included in the model, whereas the left plot shows that x2 is not relevant.

Figure 28.16 shows the coefficients and p-values for different models. Because y depends on $x1$ and $x3$, the p-value much smaller if $x3$ is included in the model as can be seen in the right plot in Figure 28.16. The left plot shows that including $x2$ in the model does not significantly improve the model fit.

28.6. Cross-Validation

- Video: Machine Learning Fundamentals: Cross Validation

29. Classification

In classification we have a qualitative response variable.

Figure 29.1.: Classification. Taken from James et al. (2014)

Here the response variable Y is qualitative, e.g., email is one of $C = (spam, ham)$, where ham is good email, digit class is one of $C = \{0, 1, \dots, 9\}$. Our goals are to:

- Build a classifier $C(X)$ that assigns a class label from C to a future unlabeled observation X.
- Assess the uncertainty in each classification
- Understand the roles of the different predictors among $X = (X_1, X_2, \dots, X_p)$.

Simulation example depicted in@fig-0218a. Y takes two values, zero and one, and X has only one value. Big sample: each single vertical bar indicates an occurrance of a zero (orange) or one (blue) as a function of the Xs. Black curve generated the data: it is the probability of generating a one. For high values of X, the probability of ones is increasing. What is an ideal classifier $C(X)$?

Suppose the K elements in C are numbered $1, 2, \dots, K$. Let

$$p_k(x) = Pr(Y = k|X = x), k = 1, 2, \dots, K.$$

These are the **conditional class probabilities** at x; e.g. see little barplot at $x = 5$. Then the **Bayes optimal classifier** at x is

$$C(x) = j \qquad \text{if } p_j(x) = \max\{p_1(x), p_2(x), \dots, p_K(x)\}.$$

At $x = 5$ there is an 80% probability of one, and an 20% probability of a zero. So, we classify this point to the class with the highest probability, the majority class.

Nearest-neighbor averaging can be used as before. This is illustrated in Fig.~**??**. Here, we consider 100 points only. Nearest-neighbor averaging also breaks down as dimension grows. However, the impact on $\hat{C}(x)$ is less than on $\hat{p}_k(x)$, $k = 1, \dots, K$.

Figure 29.2.: Classification. Taken from James et al. (2014)

29.1. Classification: Some Details

Average number of errors made to measure the performance. Typically we measure the performance of $\hat{C}(x)$ using the **misclassification error rate**:

$$Err_{Te} = Ave_{i \in Te} I[y_i \neq \hat{C}(x_i)].$$

The Bayes classifier (using the true $p_k(x)$) has smallest error (in the population).

29.2. k-Nearest Neighbor Classification

Consider k-nearest neighbors in two dimensions. Orange and blue dots label the true class memberships of the underlying points in the 2-dim plane. Dotted line is the decision boundary, that is the contour with equal probability for both classes.

Nearest-neighbor averaging in 2-dim. At any given point we want to classify, we spread out a little neighborhood, say $K = 10$ points from the neighborhood and calulated the percentage of blue and orange. We assign the color with the highest probability to this point. If this is done for every point in the plane, we obtain the solid black curve as the esitmated decsion boundary.

We can use $K = 1$. This is the **nearest-neighbor classifier**. The decision boundary is piecewise linear. Islands occur. Approximation is rather noisy.

$K = 100$ leads to a smooth decision boundary. But gets uninteresting.

Figure 29.3.: K-nearest neighbors in two dimensions. Taken from James et al. (2014)

K large means higher bias, so $1/K$ is chosen, because we go from low to high complexity on the x-error, see Figure 29.6. Horizontal dotted line is the base error.

29.2.1. Minkowski Distance

The Minkowski distance of order p (where p is an integer) between two points $X = (x_1, x_2, \dots, x_n)$ and $Y = (y_1, y_2, \dots, y_n) \in \mathbb{R}^n$ is defined as:

$$D(X, Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}.$$

- Video: StatQuest: K-nearest neighbors, Clearly Explained

Figure 29.4.: K-nearest neighbors in two dimensions. Taken from James et al. (2014)

Figure 29.5.: K-nearest neighbors in two dimensions. Taken from James et al. (2014)

Figure 29.6.: K-nearest neighbors classification error. Taken from James et al. (2014)

29.3. Decision and Classification Trees

29.3.1. Decision Trees

29.3.1.1. Decision and Classification Trees, Clearly Explained

29.3.1.2. StatQuest: Decision Trees, Part 2 - Feature Selection and Missing Data

29.3.2. Regression Trees

29.3.2.1. Regression Trees, Clearly Explained!!!

29.3.2.2. How to Prune Regression Trees, Clearly Explained!!!

29.4. The Confusion Matrix

- Video: Machine Learning Fundamentals: The Confusion Matrix

29.4.1. Sensitivity and Specificity

- Video: Machine Learning Fundamentals: Sensitivity and Specificity

29.5. Naive Bayes

- Video: Naive Bayes, Clearly Explained!!!

29.6. Gaussian Naive Bayes

- Video: Gaussian Naive Bayes, Clearly Explained!!!

30. Clustering

30.1. DBSCAN

- Video: Clustering with DBSCAN, Clearly Explained!!!

30.2. k-Means Clustering

The k-means algorithm is an unsupervised learning algorithm that has a loose relationship to the k-nearest neighbor classifier. The k-means algorithm works as follows:

- Step 1: Randomly choose k centers. Assign points to cluster.
- Step 2: Determine the distances of each data point to the centroids and re-assign each point to the closest cluster centroid based upon minimum distance
- Step 3: Calculate cluster centroids again
- Step 4: Repeat steps 2 and 3 until we reach global optima where no improvements are possible and no switching of data points from one cluster to other.

The basic principle of the k-means algorithm is illustrated in Figure 30.1, Figure 30.2, Figure 30.3, and Figure 30.4.

- Video: K-means clustering

30.3. DDMO-Additional Videos

- Odds and Log(Odds), Clearly Explained!!!
- One-Hot, Label, Target and K-Fold Target Encoding, Clearly Explained!!!
- Maximum Likelihood for the Exponential Distribution, Clearly Explained!!!
- ROC and AUC, Clearly Explained!
- Entropy (for data science) Clearly Explained!!!
- Classification Trees in Python from Start to Finish: Long live video!

Figure 30.1.: k-means algorithm. Step 1. Randomly choose k centers. Assign points to cluster. k initial 'means'(in this case $k = 3$) are randomly generated within the data domain (shown in color). Attribution: I, Weston.pace, CC BY-SA 3.0 http://creativecommons.org/licenses/by-sa/3.0/, via Wikimedia Commons

Figure 30.2.: k-means algorithm. Step 2. k clusters are created by associating every observation with the nearest mean. The partitions here represent the Voronoi diagram generated by the means. Attribution: I, Weston.pace, CC BY-SA 3.0 http://creativecommons.org/licenses/by-sa/3.0/, via Wikimedia Commons

Figure 30.3.: k-means algorithm. Step 3. The centroid of each of the k clusters becomes the new mean. Attribution: I, Weston.pace, CC BY-SA 3.0 http://creativecommons.org/licenses/by-sa/3.0/, via Wikimedia Commons

Figure 30.4.: k-means algorithm. Step 4. Steps 2 and 3 are repeated until convergence has been reached. Attribution: I, Weston.pace, CC BY-SA 3.0 http://creativecommons.org/licenses/by-sa/3.0/, via Wikimedia Commons

30.4. DDMO-Exercises

Exercise 30.1 (Smaller Bins)**.** What happens when we use smaller bins in a histogram?

Exercise 30.2 (Density Curve)**.** Why plot a curve to approximate a histogram?

Exercise 30.3 (TwoSDQuestion)**.** How many samples are plus/minus two SD around the mean?

Exercise 30.4 (OneSDQuestion)**.** How many samples are plus/minus one SD around the mean?

Exercise 30.5 (ThreeSDQuestion)**.** How many samples are plus/minus three SD around the mean?

Exercise 30.6 (DataRangeQuestion)**.** You have a mean at 100 and a SD of 10. Where are 95% of the data?

Exercise 30.7 (PeakHeightQuestion)**.** If the peak is very high, is the SD low or high?

Exercise 30.8 (ProbabilityQuestion)**.** If we have a certain curve and want to calculate the probability of values equal to 20 if the mean is 20.

Exercise 30.9 (MeanDifferenceQuestion)**.** The difference between μ and x-bar?

Exercise 30.10 (EstimateMeanQuestion)**.** How do you calculate the sample mean?

Exercise 30.11 (SigmaSquaredQuestion)**.** What is sigma squared?

Exercise 30.12 (EstimatedSDQuestion)**.** What is the formula for the estimated standard deviation?

Exercise 30.13 (VarianceDifferenceQuestion)**.** Difference between the variance and the estimated variance?

Exercise 30.14 (ModelBenefitsQuestion)**.** What are the benefits of using models?

Exercise 30.15 (SampleDefinitionQuestion)**.** What is a sample in statistics?

Exercise 30.16 (RejectHypothesisQuestion)**.** What does it mean to reject a hypothesis?

Exercise 30.17 (NullHypothesisQuestion)**.** What is a null hypothesis?

Exercise 30.18 (BetterDrugQuestion)**.** How can you show that you have found a better drug?

Exercise 30.19 (PValueIntroductionQuestion)**.** What is the reason for introducing the p-value?

Exercise 30.20 (PValueRangeQuestion)**.** Is there any range for p-values? Can it be negative?

Exercise 30.21 (PValueRangeQuestion)**.** Is there any range for p-values? Can it be negative?

Exercise 30.22 (TypicalPValueQuestion)**.** What are typical values of the p-value and what does it mean? 5%?

Exercise 30.23 (FalsePositiveQuestion)**.** What is a false-positive?

Exercise 30.24 (CalculatePValueQuestion)**.** How to calculate p-value?

Exercise 30.25 (SDCalculationQuestion)**.** What is the SD if the mean is 155 and in the range from 142 - 169 there are 95% of the data?

Exercise 30.26 (SidedPValueQuestion)**.** When do we need the two-sided p-value and when the one-sided?

Exercise 30.27 (CoinTestQuestion)**.** Test a coin with Tail-Head-Head. What is the p-value?

Exercise 30.28 (BorderPValueQuestion)**.** If you get exactly the 0.05 border value, can you reject?

Exercise 30.29 (OneSidedPValueCautionQuestion)**.** Why should you be careful with a one-sided p-test?

Exercise 30.30 (BinomialDistributionQuestion)**.** What is the binomial distribution?

Exercise 30.31 (PHackingWaysQuestion)**.** Name two typical ways of p-hacking.

Exercise 30.32 (AvoidPHackingQuestion)**.** How can p-hacking be avoided?

Exercise 30.33 (MultipleTestingProblemQuestion)**.** What is the multiple testing problem?

30.4.0.1. Covariance

Exercise 30.34 (CovarianceDefinitionQuestion)**.** What is covariance?

Exercise 30.35 (CovarianceMeaningQuestion)**.** What is the meaning of covariance?

Exercise 30.36 (CovarianceVarianceRelationshipQuestion)**.** What is the relationship between covariance and variance?

Exercise 30.37 (HighCovarianceQuestion)**.** If covariance is high, is there a strong relationship?

Exercise 30.38 (ZeroCovarianceQuestion)**.** What if the covariance is zero?

Exercise 30.39 (NegativeCovarianceQuestion)**.** Can covariance be negative?

Exercise 30.40 (NegativeVarianceQuestion)**.** Can variance be negative?

Exercise 30.41 (CorrelationValueQuestion)**.** What do you do if the correlation value is 10?

Exercise 30.42 (CorrelationRangeQuestion)**.** What is the possible range of correlation values?

Exercise 30.43 (CorrelationFormulaQuestion)**.** What is the formula for correlation?

Exercise 30.44 (UnderstandingStatisticalPower)**.** What is the definition of power in a statistical test?

Exercise 30.45 (DistributionEffectOnPower)**.** What is the implication for power analysis if the samples come from the same distribution?

Exercise 30.46 (IncreasingPower)**.** How can you increase the power if the distributions are very similar?

Exercise 30.47 (PreventingPHacking)**.** What should be done to avoid p-hacking when the distributions are close to each other?

Exercise 30.48 (SampleSizeAndPower)**.** If there is overlap and the sample size is small, will the power be high or low?

Exercise 30.49 (FactorsAffectingPower)**.** Which are the two main factors that affect power?

Exercise 30.50 (PurposeOfPowerAnalysis)**.** What does power analysis tell us?

Exercise 30.51 (ExperimentRisks)**.** What are the two risks faced when performing an experiment?

Exercise 30.52 (PerformingPowerAnalysis)**.** How do you perform a power analysis?

Exercise 30.53 (CentralLimitTheoremExplanation)**.** What does the Central Limit Theorem state?

Exercise 30.54 (MedianInBoxplot)**.** What is represented by the middle line in a boxplot?

Exercise 30.55 (BoxContentInBoxplot)**.** What does the box in a boxplot represent?

Exercise 30.56 (RSquaredDefinition)**.** What is R-squared? Show the formula.

Exercise 30.57 (NegativeRSquared)**.** Can the R-squared value be negative?

Exercise 30.58 (RSquaredCalculation)**.** Perform a calculation involving R-squared.

Exercise 30.59 (LeastSquaresMeaning)**.** What is the meaning of the least squares method?

Exercise 30.60 (RegressionVsClassification)**.** What is the difference between regression and classification?

Exercise 30.61 (LikelihoodConcept)**.** What is the idea of likelihood?

Exercise 30.62 (ProbabilityVsLikelihood)**.** What is the difference between probability and likelihood?

Exercise 30.63 (TrainVsTestData)**.** What is the difference between training and testing data?

Exercise 30.64 (SingleValidationIssue)**.** What is the problem if you validate the model only once?

Exercise 30.65 (FoldDefinition)**.** What is a fold in cross-validation?

Exercise 30.66 (LeaveOneOutValidation)**.** What is leave-one-out cross-validation?

Exercise 30.67 (DrawingConfusionMatrix)**.** Draw the confusion matrix.

Exercise 30.68 (SensitivitySpecificityCalculation1)**.** Calculate the sensitivity and specificity for a given confusion matrix.

Exercise 30.69 (SensitivitySpecificityCalculation2)**.** Calculate the sensitivity and specificity for a given confusion matrix.

Exercise 30.70 (BiasAndVariance)**.** What are bias and variance?

Exercise 30.71 (MutualInformationExample)**.** Provide an example and calculate if mutual information is high or low.

Exercise 30.72 (WhatIsPCA)**.** What is PCA?

Exercise 30.73 (ScreePlotExplanation)**.** What is a scree plot?

Exercise 30.74 (LeastSquaresInPCA)**.** Does PCA use least squares?

Exercise 30.75 (PCASteps)**.** Which steps are performed by PCA?

Exercise 30.76 (EigenvaluePC1)**.** What is the eigenvalue of the first principal component?

Exercise 30.77 (DifferencesBetweenPoints)**.** Are the differences between red and yellow the same as the differences between red and blue points?

Exercise 30.78 (ScalingInPCA)**.** How to scale data in PCA?

Exercise 30.79 (DetermineNumberOfComponents)**.** How to determine the number of principal components?

Exercise 30.80 (LimitingNumberOfComponents)**.** How is the number of principal components limited?

Exercise 30.81 (WhyUseTSNE)**.** Why use t-SNE?

Exercise 30.82 (MainIdeaOfTSNE)**.** What is the main idea of t-SNE?

Exercise 30.83 (BasicConceptOfTSNE)**.** What is the basic concept of t-SNE?

Exercise 30.84 (TSNESteps)**.** What are the steps in t-SNE?

Exercise 30.85 (HowKMeansWorks)**.** How does K-means clustering work?

Exercise 30.86 (QualityOfClusters)**.** How can the quality of the resulting clusters be calculated?

Exercise 30.87 (IncreasingK)**.** Why is it not a good idea to increase k too much?

Exercise 30.88 (CorePointInDBSCAN)**.** What is a core point in DBSCAN?

Exercise 30.89 (AddingVsExtending)**.** What is the difference between adding and extending in DBSCAN?

Exercise 30.90 (OutliersInDBSCAN)**.** What are outliers in DBSCAN?

Exercise 30.91 (AdvantagesAndDisadvantagesOfK)**.** What are the advantages and disadvantages of k = 1 and k = 100 in K-nearest neighbors?

Exercise 30.92 (NaiveBayesFormula)**.** What is the formula for Naive Bayes?

Exercise 30.93 (CalculateProbabilities)**.** Calculate the probabilities for a given example using Naive Bayes.

Exercise 30.94 (UnderflowProblem)**.** Why is underflow a problem in Gaussian Naive Bayes?

Exercise 30.95 (Tree Usage)**.** For what can we use trees?

Exercise 30.96 (Tree Usage)**.** Based on a shown tree graph:

- How can you use this tree?
- What is the root node?
- What are branches and internal nodes?
- What are the leafs?
- Are the leafs pure or impure?
- Which of the leafs is more impure?

Exercise 30.97 (Tree Feature Importance)**.** Is the most or least important feature on top?

Exercise 30.98 (Tree Feature Imputation)**.** How can you fill a gap/missing data?

Solution 30.1 (Tree Feature Imputation)*.*

- Mean
- Median
- Comparing to column with high correlation

Exercise 30.99 (Regression Tree Limitations)**.** What are limitations?

Exercise 30.100 (Regression Tree Score)**.** How is the tree score calculated?

Exercise 30.101 (Regression Tree Alpha Value Small)**.** What can we say about the tree if the alpha value is small?

Exercise 30.102 (Regression Tree Increase Alpha Value)**.** What happens if you increase alpha?

Exercise 30.103 (Regression Tree Pruning)**.** What is the meaning of pruning?

Part V.

Machine Learning and AI

31. Machine Learning and Artificial Intelligence

31.1. Jupyter Notebooks

- The Jupyter-Notebook version of this file can be found here: malai.ipynb

31.2. Videos

31.2.1. June, 11th 2024

- Happy Halloween (Neural Networks Are Not Scary)
- The Essential Main Ideas of Neural Networks

31.2.2. June, 18th 2024

- The Chain Rule
- Gradient Descent, Step-by-Step
- Neural Networks Pt. 2: Backpropagation Main Ideas

31.2.2.1. Gradient Descent

Exercise 31.1 (GradDescStepSize)**.** How is the step size calculated?

Exercise 31.2 (GradDescIntercept)**.** How to calculate the new intercept?

Exercise 31.3 (GradDescIntercept)**.** When does the gradient descend stop?

31.2.2.2. Backpropagation

Exercise 31.4 (ChainRuleAndGradientDescent)**.** What are the key components involved in backpropagation?

Exercise 31.5 (BackpropagationNaming)**.** Why is it called backpropagation?

31.2.2.3. ReLU

Exercise 31.6 (Graph ReLU)**.** Draw the graph of a ReLU function.

- Backpropagation Details Pt. 1: Optimizing 3 parameters simultaneously.
- Backpropagation Details Pt. 2: Going bonkers with The Chain Rule
- Neural Networks Pt. 3: ReLU In Action!!!
- Neural Networks Pt. 4: Multiple Inputs and Outputs
- Neural Networks Part 5: ArgMax and SoftMax
- Tensors for Neural Networks, Clearly Explained!!!
- Essential Matrix Algebra for Neural Networks, Clearly Explained!!!
- The StatQuest Introduction to PyTorch

31.2.2.4. PyTorch Links

- StatQuest: Introduction to Coding Neural Networks with PyTorch
- ML-AI Pytorch Introduction

31.2.3. June, 25th 2024

31.2.4. CNNs

31.2.4.1. Neural Networks Part 8: Image Classification with Convolutional Neural Networks (CNNs)

Exercise 31.7 (CNNImageRecognition)**.** Why are classical neural networks poor at image recognition?

Exercise 31.8 (CNNFiltersInitialization)**.** How are the filter values in CNNs initialized and optimized?

Exercise 31.9 (CNNFilterInitialization)**.** How are the filter values determined in Convolutional Neural Networks (CNNs)?

Exercise 31.10 (GenNNStockPrediction)**.** What is a limitation of using classical neural networks for stock market prediction?

31.2.5. RNN

31.2.5.1. Recurrent Neural Networks (RNNs), Clearly Explained!!!

Exercise 31.11 (RNNUnrolling)**.** How does the unrolling process work in Recurrent Neural Networks (RNNs)?

Exercise 31.12 (RNNReliability)**.** Why do Recurrent Neural Networks (RNNs) sometimes fail to work reliably?

31.2.6. LSTM

31.2.6.1. Long Short-Term Memory (LSTM), Clearly Explained

Exercise 31.13 (LSTMSigmoidTanh)**.** What are the differences between the sigmoid and tanh activation functions?

Exercise 31.14 (LSTMSigmoidTanh)**.** What is the ?

Exercise 31.15 (LSTMGates)**.** What are the gates in an LSTM network and their functions?

Exercise 31.16 (LSTMLongTermInfo)**.** In which gate is long-term information used in an LSTM network?

Exercise 31.17 (LSTMUpdateGates)**.** In which Gates is it updated in an LSTM?

31.2.7. Pytorch/Lightning

31.2.7.1. Introduction to Coding Neural Networks with PyTorch and Lightning

Exercise 31.18 (PyTorchRequiresGrad)**.** What does `requires_grad` mean in PyTorch?

31.2.8. July, 2nd 2024

- Word Embedding and Word2Vec, Clearly Explained!!!
- Sequence-to-Sequence (seq2seq) Encoder-Decoder Neural Networks, Clearly Explained!!!
- Attention for Neural Networks, Clearly Explained!!!

31.2.8.1. Embeddings

Exercise 31.19 (NN Strings)**.** Can neural networks process strings?

Exercise 31.20 (Embedding Definition)**.** What is the meaning of word embedding?

Exercise 31.21 (Embedding Dimensions)**.** Why do we need high dimension in word embedding?

31.2.8.2. Sequence to Sequence

Exercise 31.22 (LSTM)**.** Why are LSTMs used?

Exercise 31.23 (Teacher Forcing)**.** Why is teacher forcing used?

Exercise 31.24 (Attention)**.** What is the idea of attention?

31.2.9. Additional Lecture (July, 9th 2024)?

- Transformer Neural Networks, ChatGPT's foundation, Clearly Explained!!!
- Decoder-Only Transformers, ChatGPTs specific Transformer, Clearly Explained!!!
- The matrix math behind transformer neural networks, one step at a time!!!
- Word Embedding in PyTorch + Lightning

31.2.9.1. Transformers

Exercise 31.25 (ChatGPT)**.** What kind of transformer does ChatGPT use?

Exercise 31.26 (Translation)**.** What kind of NN are used for translation?

Exercise 31.27 (Difference Encoder-Decoder and Decoder Only.)**.** What is the encoder-decoder transformer and the decoder only transformer?

Exercise 31.28 (Weights)**.** How are the weights initialized (a) and trained (b)?

Exercise 31.29 (Order of Words)**.** How is the word order preserved?

Exercise 31.30 (Relationship Between Words)**.** How is the relationship between words modeled?

Exercise 31.31 (Masked Self Attention)**.** What is masked self-attention?

Exercise 31.32 (Softmax)**.** Why is Softmax used to calculate percentage of similarities?

Exercise 31.33 (Softmax Output)**.** How is the percentage output of softmax in Transformers used?

Exercise 31.34 (V´s)**.** What is done with the scaled V´s that we get for each token so far (example: "is","what")?

Exercise 31.35 (Residual Connections)**.** What are residual connections?

Exercise 31.36 (Generate Known Word in Sequence)**.** Why do we want to generate the word in the sequence that comes after "what" that we already know? (Example from video)

Exercise 31.37 (Masked-Self-Attention Values and Bypass)**.** How do we use the two values ("masked-self-attention values + bypass") which we have for each input? (Example from video: ("What", "is", "StatQuest"))

31.2.10. Additional Videos

- The SoftMax Derivative, Step-by-Step!!!
- Neural Networks Part 6: Cross Entropy
- Neural Networks Part 7: Cross Entropy Derivatives and Backpropagation

31.2.11. All Videos in a Playlist

- Full Playlist ML-AI

31.3. The StatQuest Introduction to PyTorch

The following code is taken from The StatQuest Introduction to PyTorch. Attribution goes to Josh Starmer, the creator of StatQuest, see Josh Starmer.

```
import torch # torch provides basic functions, from setting a random seed (for reprodu
import torch.nn as nn # torch.nn allows us to create a neural network.
import torch.nn.functional as F # nn.functional give us access to the activation and l
from torch.optim import SGD # optim contains many optimizers. Here, we're using SGD, s

import matplotlib.pyplot as plt ## matplotlib allows us to draw graphs.
import seaborn as sns ## seaborn makes it easier to draw nice-looking graphs.

%matplotlib inline
```

Building a neural network in PyTorch means creating a new class with two methods: init() and forward(). The init() method defines and initializes all of the parameters that we want to use, and the forward() method tells PyTorch what should happen during a forward pass through the neural network.

31.3.1. Build a Simple Neural Network in PyTorch

`__init__()` is the class constructor function, and we use it to initialize the weights and biases.

```
## create a neural network class by creating a class that inherits from nn.Module.
class BasicNN(nn.Module):

    def __init__(self): # __init__() is the class constructor function, and we use it

        super().__init__() # initialize an instance of the parent class, nn.Model.

        ## Now create the weights and biases that we need for our neural network.
        ## Each weight or bias is an nn.Parameter, which gives us the option to optimi
        ## requires_grad, which is short for "requires gradient", to True. Since we do
        ## parameters now, we set requires_grad=False.
        ##
        ## NOTE: Because our neural network is already fit to the data, we will input
        ## for each weight and bias. In contrast, if we had not already fit the neural
        ## we might start with a random initalization of the weights and biases.
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)

        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)
```

```
        self.final_bias = nn.Parameter(torch.tensor(-16.), requires_grad=False)


    def forward(self, input): ## forward() takes an input value and runs it though the neural networ
                              ## illustrated at the top of this notebook.

        ## the next three lines implement the top of the neural network (using the top node in the h
        input_to_top_relu = input * self.w00 + self.b00
        top_relu_output = F.relu(input_to_top_relu)
        scaled_top_relu_output = top_relu_output * self.w01

        ## the next three lines implement the bottom of the neural network (using the bottom node in
        input_to_bottom_relu = input * self.w10 + self.b10
        bottom_relu_output = F.relu(input_to_bottom_relu)
        scaled_bottom_relu_output = bottom_relu_output * self.w11

        ## here, we combine both the top and bottom nodes from the hidden layer with the final bias.
        input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + self.final_bias

        output = F.relu(input_to_final_relu)

        return output # output is the predicted effectiveness for a drug dose.
```

Once we have created the class that defines the neural network, we can create an actual neural network and print out its parameters, just to make sure things are what we expect.

```
## create the neural network.
model = BasicNN()

## print out the name and value for each parameter
for name, param in model.named_parameters():
    print(name, param.data)
```

```
w00 tensor(1.7000)
b00 tensor(-0.8500)
w01 tensor(-40.8000)
w10 tensor(12.6000)
b10 tensor(0.)
w11 tensor(2.7000)
final_bias tensor(-16.)
```

31.3.2. Use the Neural Network and Graph the Output

Now that we have a neural network, we can use it on a variety of doses to determine which will be effective. Then we can make a graph of these data, and this graph should match the green bent shape fit to the training data that's shown at the top of this document. So, let's start by making a sequence of input doses…

```
## now create the different doses we want to run through the neural network.
## torch.linspace() creates the sequence of numbers between, and including, 0 and 1.
input_doses = torch.linspace(start=0, end=1, steps=11)

# now print out the doses to make sure they are what we expect...
input_doses
```

```
tensor([0.0000, 0.1000, 0.2000, 0.3000, 0.4000, 0.5000, 0.6000, 0.7000, 0.8000,
        0.9000, 1.0000])
```

Now that we have input_doses, let's run them through the neural network and graph the output…

```
## create the neural network.
model = BasicNN()

## now run the different doses through the neural network.
output_values = model(input_doses)

## Now draw a graph that shows the effectiveness for each dose.
##
## First, set the style for seaborn so that the graph looks cool.
sns.set(style="whitegrid")

## create the graph (you might not see it at this point, but you will after we save it
sns.lineplot(x=input_doses,
     y=output_values,
     color='green',
     linewidth=2.5)

## now label the y- and x-axes.
plt.ylabel('Effectiveness')
plt.xlabel('Dose')

## optionally, save the graph as a PDF.
# plt.savefig('BasicNN.pdf')
```

```
Text(0.5, 0, 'Dose')
```

The graph shows that the neural network fits the training data. In other words, so far, we don't have any bugs in our code.

31.3.3. Optimize (Train) a Parameter in the Neural Network and Graph the Output

Now that we know how to create and use a simple neural network, and we can graph the output relative to the input, let's see how to train a neural network. The first thing we need to do is tell PyTorch which parameter (or parameters) we want to train, and we do that by setting requiresgrad=True. In this example, we'll train finalbias.

Now we create a neural network by creating a class that inherits from nn.Module.

NOTE: This code is the same as before, except we changed the class name to BasicNN_train and we modified final_bias in two ways:

```
1) we set the value of the tensor to 0, and
2) we set "requires_grad=True".
```

Now let's graph the output of BasicNN_train, which is currently not optimized, and compare it to the graph we drew earlier of the optimized neural network.

```
class BasicNN_train(nn.Module):

    def __init__(self): # __init__ is the class constructor function, and we use it to

        super().__init__() # initialize an instance of the parent class, nn.Module.

        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)

        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)

        ## we want to modify final_bias to demonstrate how to optimize it with backpro
        ## The optimal value for final_bias is -16...
#         self.final_bias = nn.Parameter(torch.tensor(-16.), requires_grad=False)
        ## ...so we set it to 0 and tell Pytorch that it now needs to calculate the gr
        self.final_bias = nn.Parameter(torch.tensor(0.), requires_grad=True)

    def forward(self, input):

        input_to_top_relu = input * self.w00 + self.b00
        top_relu_output = F.relu(input_to_top_relu)
        scaled_top_relu_output = top_relu_output * self.w01

        input_to_bottom_relu = input * self.w10 + self.b10
        bottom_relu_output = F.relu(input_to_bottom_relu)
        scaled_bottom_relu_output = bottom_relu_output * self.w11

        input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + sel

        output = F.relu(input_to_final_relu)

        return output
```

```
## create the neural network.
model = BasicNN_train()

## now run the different doses through the neural network.
output_values = model(input_doses)

## Now draw a graph that shows the effectiveness for each dose.
##
```

```
## set the style for seaborn so that the graph looks cool.
sns.set(style="whitegrid")

## create the graph (you might not see it at this point, but you will after we save it as a PDF).
sns.lineplot(x=input_doses,
             y=output_values.detach(), ## NOTE: because final_bias has a gradident, we call detach()
                                       ## to return a new tensor that only has the value and not the
             color='green',
             linewidth=2.5)

## now label the y- and x-axes.
plt.ylabel('Effectiveness')
plt.xlabel('Dose')

## lastly, save the graph as a PDF.
# plt.savefig('BasicNN_train.pdf')
```

```
Text(0.5, 0, 'Dose')
```

The graph shows that when the dose is 0.5, the output from the unoptimized neural network is 17, which is wrong, since the output value should be 1. So, now that we have a parameter we can optimize, let's create some training data that we can use to optimize it.

```
## create the training data for the neural network.
inputs = torch.tensor([0., 0.5, 1.])
labels = torch.tensor([0., 1., 0.])
```

..and now let's use that training data to train (or optimize) final_bias.

```
## create the neural network we want to train.
model = BasicNN_train()

optimizer = SGD(model.parameters(), lr=0.1) ## here we're creating an optimizer to tra
                                            ## NOTE: There are a bunch of different wa
                                            ## In this example, we'll use Stochastic (
                                            ## another popular algortihm is Adam (whic

print("Final bias, before optimization: " + str(model.final_bias.data) + "\n")

## this is the optimization loop. Each time the optimizer sees all of the training dat
for epoch in range(100):

    ## we create and initialize total_loss for each epoch so that we can evaluate how
    ## training data. At first, when the model doesn't fit the training data very well
    ## will be large. However, as gradient descent improves the fit, total_loss will g
    ## If total_loss gets really small, we can decide that the model fits the data wel
    ## optimizing the fit. Otherwise, we can just keep optimizing until we reach the m
    total_loss = 0

    ## this internal loop is where the optimizer sees all of the training data and whe
    ## calculate the total_loss for all of the training data.
    for iteration in range(len(inputs)):

        input_i = inputs[iteration] ## extract a single input value (a single dose)..
        label_i = labels[iteration] ## ...and its corresponding label (the effectivene

        output_i = model(input_i) ## calculate the neural network output for the input

        loss = (output_i - label_i)**2 ## calculate the loss for the single value.
                                       ## NOTE: Because output_i = model(input_i), "lo
                                       ## and the derivative (calculated in the next s
                                       ## in "model".

        loss.backward() # backward() calculates the derivative for that single value a

        total_loss += float(loss) # accumulate the total loss for this epoch.
```

```
    if (total_loss < 0.0001):
        print("Num steps: " + str(epoch))
        break

    optimizer.step() ## take a step toward the optimal value.
    optimizer.zero_grad() ## This zeroes out the gradient stored in "model".
                          ## Remember, by default, gradients are added to the previous step (the gra
                          ## and we took advantage of this process to calculate the derivative one d
                          ## NOTE: "optimizer" has access to "model" because of how it was created w
                          ## (made earlier): optimizer = SGD(model.parameters(), lr=0.1).
                          ## ALSO NOTE: Alternatively, we can zero out the gradient with model.zero_
    if epoch % 10 == 0:
        print("Step: " + str(epoch) + " Final Bias: " + str(model.final_bias.data) + "\n")
    ## now go back to the start of the loop and go through another epoch.

print("Total loss: " + str(total_loss))
print("Final bias, after optimization: " + str(model.final_bias.data))
```

```
Final bias, before optimization: tensor(0.)

Step: 0 Final Bias: tensor(-3.2020)

Step: 10 Final Bias: tensor(-14.6348)

Step: 20 Final Bias: tensor(-15.8623)

Step: 30 Final Bias: tensor(-15.9941)

Num steps: 34
Total loss: 6.58966600894928e-05
Final bias, after optimization: tensor(-16.0019)
```

So, if everything worked correctly, the optimizer should have converged on final_bias = 16.0019 after 34 steps, or epochs. BAM!

Lastly, let's graph the output from the optimized neural network and see if it's the same as what we started with. If so, then the optimization worked.

```
## run the different doses through the neural network
output_values = model(input_doses)

## set the style for seaborn so that the graph looks cool.
sns.set(style="whitegrid")
```

```
## create the graph (you might not see it at this point, but you will after we save it
sns.lineplot(x=input_doses,
     y=output_values.detach(), ## NOTE: we call detach() because final_bias has a grad
     color='green',
     linewidth=2.5)

## now label the y- and x-axes.
plt.ylabel('Effectiveness')
plt.xlabel('Dose')

## lastly, save the graph as a PDF.
# plt.savefig('BascNN_optimized.pdf')
```

```
Text(0.5, 0, 'Dose')
```

And we see that the optimized model results in the same graph that we started with, so the optimization worked as expected.

31.4. Build a Long Short-Term Memory unit by hand using PyTorch + Lightning

The following code is based on Long Short-Term Memory with PyTorch + Lightning and StatQuest: Long Short-Term Memory (LSTM) with PyTorch + Lightning!!!. Attribution goes to Josh Starmer, the creator of StatQuest, see Josh Starmer.

```
import torch # torch will allow us to create tensors.
import torch.nn as nn # torch.nn allows us to create a neural network.
import torch.nn.functional as F # nn.functional give us access to the activation and loss functions.
from torch.optim import Adam # optim contains many optimizers. This time we're using Adam

import lightning as L # lightning has tons of cool tools that make neural networks easier
from torch.utils.data import TensorDataset, DataLoader # these are needed for the training data
```

A Long Short-Term Memory (LSTM) unit is a type of neural network, and that means we need to create a new class. To make it easy to train the LSTM, this class will inherit from LightningModule and we'll create the following methods:

- `init()` to initialize the Weights and Biases and keep track of a few other house keeping things.
- `lstm_unit()` to do the LSTM math. For example, to calculate the percentage of the long-term memory to remember.
- `forward()` to make a forward pass through the unrolled LSTM. In other words forward() calls `lstm_unit()` for each data point.
- `configure_optimizers()` to configure the opimimizer. In the past, we have use SGD (Stochastic Gradient Descent), however, in this tutorial we'll change things up and use Adam, another popular algorithm for optimizing the Weights and Biases.
- `training_step()` to pass the training data to forward(), calculate the loss and to keep track of the loss values in a log file.

```
class LSTMbyHand(L.LightningModule):

    def __init__(self):
        super().__init__()
        L.seed_everything(seed=42)

        ## NOTE: nn.LSTM() uses random values from a uniform distribution to initialize the tensors
        ## Here we can do it 2 different ways 1) Normal Distribution and 2) Uniform Distribution
        ## We'll start with the Normal distribution.
        mean = torch.tensor(0.0)
        std = torch.tensor(1.0)
```

```
        ## NOTE: In this case, I'm only using the normal distribution for the Weights
        ## All Biases are initialized to 0.
        ##
        ## These are the Weights and Biases in the first stage, which determines what
        ## of the long-term memory the LSTM unit will remember.
        self.wlr1 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.wlr2 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.blr1 = nn.Parameter(torch.tensor(0.), requires_grad=True)

        ## These are the Weights and Biases in the second stage, which determines the
        ## potential long-term memory and what percentage will be remembered.
        self.wpr1 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.wpr2 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.bpr1 = nn.Parameter(torch.tensor(0.), requires_grad=True)

        self.wp1 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.wp2 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.bp1 = nn.Parameter(torch.tensor(0.), requires_grad=True)

        ## These are the Weights and Biases in the third stage, which determines the
        ## new short-term memory and what percentage will be sent to the output.
        self.wo1 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.wo2 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.bo1 = nn.Parameter(torch.tensor(0.), requires_grad=True)

        ## We can also initialize all Weights and Biases using a uniform distribution
        ## how nn.LSTM() does it.
#         self.wlr1 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.wlr2 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.blr1 = nn.Parameter(torch.rand(1), requires_grad=True)

#         self.wpr1 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.wpr2 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.bpr1 = nn.Parameter(torch.rand(1), requires_grad=True)

#         self.wp1 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.wp2 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.bp1 = nn.Parameter(torch.rand(1), requires_grad=True)

#         self.wo1 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.wo2 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.bo1 = nn.Parameter(torch.rand(1), requires_grad=True)
```

```
    def lstm_unit(self, input_value, long_memory, short_memory):
        ## lstm_unit does the math for a single LSTM unit.

        ## NOTES:
        ## long term memory is also called "cell state"
        ## short term memory is also called "hidden state"

        ## 1) The first stage determines what percent of the current long-term memory
        ##    should be remembered
        long_remember_percent = torch.sigmoid((short_memory * self.wlr1) +
                                              (input_value * self.wlr2) +
                                              self.blr1)

        ## 2) The second stage creates a new, potential long-term memory and determines what
        ##    percentage of that to add to the current long-term memory
        potential_remember_percent = torch.sigmoid((short_memory * self.wpr1) +
                                                   (input_value * self.wpr2) +
                                                   self.bpr1)
        potential_memory = torch.tanh((short_memory * self.wp1) +
                                      (input_value * self.wp2) +
                                      self.bp1)

        ## Once we have gone through the first two stages, we can update the long-term memory
        updated_long_memory = ((long_memory * long_remember_percent) +
                               (potential_remember_percent * potential_memory))

        ## 3) The third stage creates a new, potential short-term memory and determines what
        ##    percentage of that should be remembered and used as output.
        output_percent = torch.sigmoid((short_memory * self.wo1) +
                                       (input_value * self.wo2) +
                                       self.bo1)
        updated_short_memory = torch.tanh(updated_long_memory) * output_percent

        ## Finally, we return the updated long and short-term memories
        return([updated_long_memory, updated_short_memory])

    def forward(self, input):
        ## forward() unrolls the LSTM for the training data by calling lstm_unit() for each day of t
        ## that we have. forward() also keeps track of the long and short-term memories after each d
        ## the final short-term memory, which is the 'output' of the LSTM.

        long_memory = 0 # long term memory is also called "cell state" and indexed with c0, c1, ...,
        short_memory = 0 # short term memory is also called "hidden state" and indexed with h0, h1,
        day1 = input[0]
```

```
        day2 = input[1]
        day3 = input[2]
        day4 = input[3]

        ## Day 1
        long_memory, short_memory = self.lstm_unit(day1, long_memory, short_memory)

        ## Day 2
        long_memory, short_memory = self.lstm_unit(day2, long_memory, short_memory)

        ## Day 3
        long_memory, short_memory = self.lstm_unit(day3, long_memory, short_memory)

        ## Day 4
        long_memory, short_memory = self.lstm_unit(day4, long_memory, short_memory)

        ##### Now return short_memory, which is the 'output' of the LSTM.
        return short_memory

    def configure_optimizers(self): # this configures the optimizer we want to use for
        # return Adam(self.parameters(), lr=0.1) # NOTE: Setting the learning rate to
                                                 # using the default learning rate, lr
                                                 # training. However, if we use the de
                                                 # the exact same Weights and Biases t
                                                 # the LSTM Clearly Explained StatQues
                                                 # default value.
        return Adam(self.parameters())


    def training_step(self, batch, batch_idx): # take a step during gradient descent.
        input_i, label_i = batch # collect input
        output_i = self.forward(input_i[0]) # run input through the neural network
        loss = (output_i - label_i)**2 ## loss = sum of squared residual
        # Logging the loss and the predicted values so we can evaluate the training:
        self.log("train_loss", loss)
        ## NOTE: Our dataset consists of two sequences of values representing Company
        ## For Company A, the goal is to predict that the value on Day 5 = 0, and for
        ## the goal is to predict that the value on Day 5 = 1. We use label_i, the val
        ## predict, to keep track of which company we just made a prediction for and
        ## log that output value in a company specific file
        if (label_i == 0):
            self.log("out_0", output_i)
        else:
            self.log("out_1", output_i)
```

```
        return loss
```

Once we have created the class that defines an LSTM, we can use it to create a model and print out the randomly initialized Weights and Biases. Then, just for fun, we'll see what those random Weights and Biases predict for Company A and Company B. If they are good predictions, then we're done! However, the chances of getting good predictions from random values is very small.

```
## Create the model object, print out parameters and see how well
## the untrained LSTM can make predictions...
model = LSTMbyHand()

print("Before optimization, the parameters are...")
for name, param in model.named_parameters():
    print(name, param.data)

print("\nNow let's compare the observed and predicted values...")
## NOTE: To make predictions, we pass in the first 4 days worth of stock values
## in an array for each company. In this case, the only difference between the
## input values for Company A and B occurs on the first day. Company A has 0 and
## Company B has 1.
print("Company A: Observed = 0, Predicted =",
      model(torch.tensor([0., 0.5, 0.25, 1.])).detach())
print("Company B: Observed = 1, Predicted =",
      model(torch.tensor([1., 0.5, 0.25, 1.])).detach())
```

```
Before optimization, the parameters are...
wlr1 tensor(0.3367)
wlr2 tensor(0.1288)
blr1 tensor(0.)
wpr1 tensor(0.2345)
wpr2 tensor(0.2303)
bpr1 tensor(0.)
wp1 tensor(-1.1229)
wp2 tensor(-0.1863)
bp1 tensor(0.)
wo1 tensor(2.2082)
wo2 tensor(-0.6380)
bo1 tensor(0.)

Now let's compare the observed and predicted values...
Company A: Observed = 0, Predicted = tensor(-0.0377)
Company B: Observed = 1, Predicted = tensor(-0.0383)
```

With the unoptimized paramters, the predicted value for Company A, -0.0377, isn't terrible, since it is relatively close to the observed value, 0. However, the predicted value for Company B, -0.0383, is terrible, because it is relatively far from the observed value, 1. So, that means we need to train the LSTM.

31.4.1. Train the LSTM unit and use Lightning and TensorBoard to evaluate: Part 1 - Getting Started

Since we are using Lightning training, training the LSTM we created by hand is pretty easy. All we have to do is create the training data and put it into a DataLoader…

```
## create the training data for the neural network.
inputs = torch.tensor([[0., 0.5, 0.25, 1.], [1., 0.5, 0.25, 1.]])
labels = torch.tensor([0., 1.])

dataset = TensorDataset(inputs, labels)
dataloader = DataLoader(dataset)

# show the training data
for i, (input_i, label_i) in enumerate(dataloader):
    print("Training data: ", input_i, label_i)
```

```
Training data:  tensor([[0.0000, 0.5000, 0.2500, 1.0000]]) tensor([0.])
Training data:  tensor([[1.0000, 0.5000, 0.2500, 1.0000]]) tensor([1.])
```

…and then create a Lightning Trainer, L.Trainer, and fit it to the training data. NOTE: We are starting with 2000 epochs. This may be enough to successfully optimize all of the parameters, but it might not. We'll find out after we compare the predictions to the observed values.

```
trainer = L.Trainer(max_epochs=2000) # with default learning rate, 0.001 (this tiny le
trainer.fit(model, train_dataloaders=dataloader)
```

```
     Name           Type   Params   Mode   FLOPs

     other params   n/a        12   n/a      n/a


Trainable params: 12
Non-trainable params: 0
Total params: 12
```

```
Total estimated model params size (MB): 0
Modules in train mode: 0
Modules in eval mode: 0
Total FLOPs: 0
```

```
Output()
```

Now that we've trained the model with 2000 epochs, we can see how good the predictions are…

```
print("\nNow let's compare the observed and predicted values...")
print("Company A: Observed = 0, Predicted =", model(torch.tensor([0., 0.5, 0.25, 1.])).detach())
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])).detach())
```

```
Now let's compare the observed and predicted values...
Company A: Observed = 0, Predicted = tensor(0.4342)
Company B: Observed = 1, Predicted = tensor(0.6171)
```

Unfortunately, these predictions are terrible. So it seems like we'll have to do more training. However, it would be awesome if we could be confident that more training will actually improve the predictions. If not, we can spare ourselves a lot of time, and potentially money, and just give up. So, before we dive into more training, let's look at the loss values and predictions that we saved in log files with TensorBoard. TensorBoard will graph everything that we logged during training, making it super easy to see if things are headed in the right direction or not.

To get TensorBoard working:

- First, check to see if the TensorBoard plugin is installed. If it's not, install it with the following command: pip install tensorboard
- Next, run the following command: `tensorboard --logdir lightning_logs`

NOTE: If your graphs look messed up and you see a bunch of different lines, instead of just one red line per graph, then check where this notebook is saved for a directory called `lightning_logs`. Delete `lightning_logs` and the re-run everything in this notebook. One source of problems with the graphs is that every time we train a model, a new batch of log files is created and stored in lightning_logs and TensorBoard, by default, will plot all of them. You can turn off unwanted log files in TensorBoard, and we'll do this later on in this notebook, but for now, the easiest thing to do is to start with a clean slate.

Anyway, if we look at the loss (trainloss), we see that it is going down, which is good, but it still has further to go. When we look at the predictions for Company A (out0), we see that they started out pretty good, close to 0, but then got really bad early on in

training, shooting all the way up to 0.5, but are starting to get smaller. In contrast, when we look at the predictions for Company B (out_1), we see that they started out really bad, close to 0, but have been getting better ever since and look like they could continue to get better if we kept training.

In summary, the graphs seem to suggest that if we continued training our model, the predictions would improve. So let's add more epochs to the training.

31.4.2. Optimizing (Training) the Weights and Biases in the LSTM that we made by hand: Part 2 - Adding More Epochs without Starting Over

The good news is that because we're using Lightning, we can pick up where we left off training without having to start over from scratch. This is because when we train with Lightning, it creates checkpoint files that keep track of the Weights and Biases as they change. As a result, all we have to do to pick up where we left off is tell the Trainer where the checkpoint files are located. This is awesome and will save us a lot of time since we don't have to retrain the first 2000 epochs. So let's add an additional 1000 epochs to the training.

```
## First, find where the most recent checkpoint files are stored
path_to_checkpoint = trainer.checkpoint_callback.best_model_path ## By default, "best"
print("The new trainer will start where the last left off, and the check point data is
      path_to_checkpoint + "\n")

## Then create a new Lightning Trainer
trainer = L.Trainer(max_epochs=3000) # Before, max_epochs=2000, so, by setting it to 3
## And then call fit() using the path to the most recent checkpoint files
## so that we can pick up where we left off.
trainer.fit(model, train_dataloaders=dataloader, ckpt_path=path_to_checkpoint)
```

```
The new trainer will start where the last left off, and the check point data is here:
Tuning-Cookbook/lightning_logs/version_2870/checkpoints/epoch=1999-
step=4000.ckpt


     Name           Type   Params   Mode   FLOPs

     other params   n/a        12   n/a      n/a


Trainable params: 12
Non-trainable params: 0
```

```
Total params: 12
Total estimated model params size (MB): 0
Modules in train mode: 0
Modules in eval mode: 0
Total FLOPs: 0

Output()
```

Now that we have added 1000 epochs to the training, let's check the predictions…

```
print("\nNow let's compare the observed and predicted values...")
print("Company A: Observed = 0, Predicted =", model(torch.tensor([0., 0.5, 0.25, 1.])).detach())
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])).detach())
```

```
Now let's compare the observed and predicted values...
Company A: Observed = 0, Predicted = tensor(0.2708)
Company B: Observed = 1, Predicted = tensor(0.7534)
```

The blue lines in each graph represents the values we logged during the extra 1000 epochs. The loss is getting smaller and the predictions for both companies are improving! Hooray!!! However, because it looks like there is even more room for improvement, let's add 2000 more epochs to the training.

```
## First, find where the most recent checkpoint files are stored
path_to_checkpoint = trainer.checkpoint_callback.best_model_path ## By default, "best" = "most recen
print("The new trainer will start where the last left off, and the check point data is here: " +
      path_to_checkpoint + "\n")

## Then create a new Lightning Trainer
trainer = L.Trainer(max_epochs=5000) # Before, max_epochs=3000, so, by setting it to 5000, we're add
## And then call fit() using the path to the most recent checkpoint files
## so that we can pick up where we left off.
trainer.fit(model, train_dataloaders=dataloader, ckpt_path=path_to_checkpoint)
```

```
The new trainer will start where the last left off, and the check point data is here: /Users/bartz/w
Tuning-Cookbook/lightning_logs/version_2871/checkpoints/epoch=2999-
step=6000.ckpt


     Name           Type   Params   Mode   FLOPs

     other params   n/a        12   n/a      n/a
```

```
Trainable params: 12
Non-trainable params: 0
Total params: 12
Total estimated model params size (MB): 0
Modules in train mode: 0
Modules in eval mode: 0
Total FLOPs: 0
```

```
Output()
```

Now that we have added 2000 more epochs to the training (for a total of 5000 epochs), let's check the predictions.

```
print("\nNow let's compare the observed and predicted values...")
print("Company A: Observed = 0, Predicted =", model(torch.tensor([0., 0.5, 0.25, 1.])
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])
```

```
Now let's compare the observed and predicted values...
Company A: Observed = 0, Predicted = tensor(0.0022)
Company B: Observed = 1, Predicted = tensor(0.9693)
```

The prediction for Company A is super close to 0, which is exactly what we want, and the prediction for Company B is close to 1, which is also what we want.

The dark red lines show how things changed when we added an additional 2000 epochs to the training, for a total of 5000 epochs. Now we see that the loss (train_loss) and the predictions for each company appear to be tapering off, suggesting that adding more epochs may not improve the predictions much, so we're done!

Lastly, let's print out the final estimates for the Weights and Biases. In theory, they should be the same (within rounding error) as what we used in the StatQuest on Long Short-Term Memory and seen in the diagram of the LSTM unit at the top of this Jupyter notebook.

```
print("After optimization, the parameters are...")
for name, param in model.named_parameters():
    print(name, param.data)
```

```
After optimization, the parameters are...
wlr1 tensor(2.7043)
wlr2 tensor(1.6307)
blr1 tensor(1.6234)
```

```
wpr1 tensor(1.9983)
wpr2 tensor(1.6525)
bpr1 tensor(0.6204)
wp1 tensor(1.4122)
wp2 tensor(0.9393)
bp1 tensor(-0.3217)
wo1 tensor(4.3848)
wo2 tensor(-0.1943)
bo1 tensor(0.5935)
```

31.5. Using and optimzing the PyTorch LSTM, nn.LSTM()

Now that we know how to create an LSTM unit by hand, train it, and then use it to make good predictions, let's learn how to take advantage of PyTorch's `nn.LSTM()` function. For the most part, using nn.LSTM() allows us to simplify the init() function and the forward() function. The other big difference is that this time, we're not going to try and recreate the parameter values we used in the StatQuest on Long Short-Term Memory, and that means we can set the learning rate for the Adam to 0.1. This will speed up training a lot. Everything else stays the same.

```
## Instead of coding an LSTM by hand, let's see what we can do with PyTorch's nn.LSTM()
class LightningLSTM(L.LightningModule):

    def __init__(self): # __init__() is the class constructor function, and we use it to initialize

        super().__init__() # initialize an instance of the parent class, LightningModule.

        L.seed_everything(seed=42)

        ## input_size = number of features (or variables) in the data. In our example
        ##              we only have a single feature (value)
        ## hidden_size = this determines the dimension of the output
        ##               in other words, if we set hidden_size=1, then we have 1 output node
        ##               if we set hidden_size=50, then we hve 50 output nodes (that can then be 50
        ##               nodes to a subsequent fully connected neural network.
        self.lstm = nn.LSTM(input_size=1, hidden_size=1)


    def forward(self, input):
        ## transpose the input vector
        input_trans = input.view(len(input), 1)
```

```
        lstm_out, temp = self.lstm(input_trans)

        ## lstm_out has the short-term memories for all inputs. We make our predictio
        prediction = lstm_out[-1]
        return prediction


    def configure_optimizers(self): # this configures the optimizer we want to use fo
        return Adam(self.parameters(), lr=0.1) ## we'll just go ahead and set the lea


    def training_step(self, batch, batch_idx): # take a step during gradient descent.
        input_i, label_i = batch # collect input
        output_i = self.forward(input_i[0]) # run input through the neural network
        loss = (output_i - label_i)**2 ## loss = squared residual
        self.log("train_loss", loss)

        if (label_i == 0):
            self.log("out_0", output_i)
        else:
            self.log("out_1", output_i)

        return loss
```

Now let's create the model and print out the initial Weights and Biases and predictions.

```
model = LightningLSTM() # First, make model from the class

## print out the name and value for each parameter
print("Before optimization, the parameters are...")
for name, param in model.named_parameters():
    print(name, param.data)

print("\nNow let's compare the observed and predicted values...")
print("Company A: Observed = 0, Predicted =", model(torch.tensor([0., 0.5, 0.25, 1.])
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])
```

```
Before optimization, the parameters are...
lstm.weight_ih_l0 tensor([[ 0.7645],
        [ 0.8300],
        [-0.2343],
        [ 0.9186]])
```

```
lstm.weight_hh_l0 tensor([[-0.2191],
        [ 0.2018],
        [-0.4869],
        [ 0.5873]])
lstm.bias_ih_l0 tensor([ 0.8815, -0.7336,  0.8692,  0.1872])
lstm.bias_hh_l0 tensor([ 0.7388,  0.1354,  0.4822, -0.1412])

Now let's compare the observed and predicted values...
Company A: Observed = 0, Predicted = tensor([0.6675])
Company B: Observed = 1, Predicted = tensor([0.6665])
```

As expected, the predictions are bad, so we will train the model. However, because we've increased the learning rate to 0.1, we only need to train for 300 epochs.

```
## NOTE: Because we have set Adam's learning rate to 0.1, we will train much, much faster.
## Before, with the hand made LSTM and the default learning rate, 0.001, it took about 5000 epochs t
## the model. Now, with the learning rate set to 0.1, we only need 300 epochs. Now, because we are d
## we have to tell the trainer add stuff to the log files every 2 steps (or epoch, since we have to
## because the default, updating the log files every 50 steps, will result in a terrible looking gra
trainer = L.Trainer(max_epochs=300, log_every_n_steps=2)

trainer.fit(model, train_dataloaders=dataloader)

print("After optimization, the parameters are...")
for name, param in model.named_parameters():
    print(name, param.data)
```

```
     Name   Type   Params   Mode    FLOPs

  0  lstm   LSTM       16   train       0


Trainable params: 16
Non-trainable params: 0
Total params: 16
Total estimated model params size (MB): 0
Modules in train mode: 1
Modules in eval mode: 0
Total FLOPs: 0


Output()
```

```
After optimization, the parameters are...
lstm.weight_ih_l0 tensor([[3.5364],
        [1.3869],
        [1.5390],
        [1.2488]])
lstm.weight_hh_l0 tensor([[5.2070],
        [2.9577],
        [3.2652],
        [2.0678]])
lstm.bias_ih_l0 tensor([-0.9143,  0.3724, -0.1815,  0.6376])
lstm.bias_hh_l0 tensor([-1.0570,  1.2414, -0.5685,  0.3092])
```

Now that training is done, let's print out the new predictions…

```
print("\nNow let's compare the observed and predicted values...")
print("Company A: Observed = 0, Predicted =", model(torch.tensor([0., 0.5, 0.25, 1.])
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])
```

```
Now let's compare the observed and predicted values...
Company A: Observed = 0, Predicted = tensor([6.8527e-05])
Company B: Observed = 1, Predicted = tensor([0.9809])
```

…and, as we can see, after just 300 epochs, the LSTM is making great predictions. The prediction for Company A is close to the observed value 0 and the prediction for Company B is close to the observed value 1.

Lastly, let's go back to TensorBoard to see the latest graphs. NOTE: To make it easier to see what we just did, deselect version0, version1 and version2 and make sure version3 is checked on the left-hand side of the page, under where it says Runs. This allows us to just look at the log files from the most recent training, which only went for 300 epochs.

In all three graphs, the loss (trainloss) and the predictions for Company A (out0) and Company B (out_1) started to taper off after 500 steps, or just 250 epochs, suggesting that adding more epochs may not improve the predictions much, so we're done!

Part VI.

Introduction to Hyperparameter Tuning

32. Hyperparameter Tuning

32.1. Structure of the Hyperparameter Tuning Chapters

The first part is structured as follows:

The concept of the hyperparameter tuning is described in Section 32.2.

Hyperparameter tuning with sklearn in Python is described in Chapter 33.

Hyperparameter tuning with river in Python is described in Chapter 36.

This part of the book is concluded with a description of the most recent `PyTorch` hyperparameter tuning approach, which is the integration of `spotpython` into the `PyTorch Lightning` training workflow. Hyperparameter tuning with PyTorch Lightning in Python is described in Chapter 44. This is considered as the most effective, efficient, and flexible way to integrate `spotpython` into the `PyTorch` training workflow.

Figure 32.1 shows the graphical user interface of `spotpython` that is used in this book.

Figure 32.1.: spot GUI

32.2. Goals of Hyperparameter Tuning

The goal of hyperparameter tuning is to optimize the hyperparameters in a way that improves the performance of the machine learning or deep learning model. Hyperparameters are parameters that are not learned during the training process, but are set before the training process begins. Hyperparameter tuning is an important, but often difficult

and computationally intensive task. Changing the architecture of a neural network or the learning rate of an optimizer can have a significant impact on the performance.

Hyperparameter tuning is referred to as "hyperparameter optimization" (HPO) in the literature. However, since we do not consider the optimization, but also the understanding of the hyperparameters, we use the term "hyperparameter tuning" in this book. See also the discussion in Chapter 2 of Bartz et al. (2022), which lays the groundwork and presents an introduction to the process of tuning Machine Learning and Deep Learning hyperparameters and the respective methodology. Since the key elements such as the hyperparameter tuning process and measures of tunability and performance are presented in Bartz et al. (2022), we refer to this chapter for details.

The simplest, but also most computationally expensive, hyperparameter tuning approach uses manual search (or trial-and-error (Meignan et al. 2015)). Commonly encountered is simple random search, i.e., random and repeated selection of hyperparameters for evaluation, and lattice search ("grid search"). In addition, methods that perform directed search and other model-free algorithms, i.e., algorithms that do not explicitly rely on a model, e.g., evolution strategies (Bartz-Beielstein et al. 2014) or pattern search (Lewis, Torczon, and Trosset 2000) play an important role. Also, "hyperband", i.e., a multi-armed bandit strategy that dynamically allocates resources to a set of random configurations and uses successive bisections to stop configurations with poor performance (Li et al. 2016), is very common in hyperparameter tuning. The most sophisticated and efficient approaches are the Bayesian optimization and surrogate model based optimization methods, which are based on the optimization of cost functions determined by simulations or experiments.

We consider a surrogate optimization based hyperparameter tuning approach that uses the Python version of the SPOT ("Sequential Parameter Optimization Toolbox") (Bartz-Beielstein, Lasarczyk, and Preuss 2005), which is suitable for situations where only limited resources are available. This may be due to limited availability and cost of hardware, or due to the fact that confidential data may only be processed locally, e.g., due to legal requirements. Furthermore, in our approach, the understanding of algorithms is seen as a key tool for enabling transparency and explainability. This can be enabled, for example, by quantifying the contribution of machine learning and deep learning components (nodes, layers, split decisions, activation functions, etc.). Understanding the importance of hyperparameters and the interactions between multiple hyperparameters plays a major role in the interpretability and explainability of machine learning models. SPOT provides statistical tools for understanding hyperparameters and their interactions. Last but not least, it should be noted that the SPOT software code is available in the open source `spotpython` package on github[1], allowing replicability of the results. This tutorial describes the Python variant of SPOT, which is called `spotpython`. The R implementation is described in Bartz et al. (2022). SPOT is an established open source software that has been maintained for more than 15 years (Bartz-Beielstein, Lasarczyk, and Preuss 2005) (Bartz et al. 2022).

[1] https://github.com/sequential-parameter-optimization

Part VII.

Hyperparameter Tuning with Sklearn

33. HPT: sklearn

33.1. Introduction to sklearn

34. HPT: sklearn SVC on Moons Data

This chapter is a tutorial for the Hyperparameter Tuning (HPT) of a `sklearn` SVC model on the Moons dataset.

34.1. Packages used in this Chapter

```
import os
from math import inf
import numpy as np
import warnings
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import RidgeCV, ElasticNet, LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import GradientBoostingClassifier, GradientBoostingRegressor, RandomForestClas
from sklearn.metrics import mean_absolute_error, accuracy_score, roc_curve, roc_auc_score, log_loss,

from spotpython.hyperparameters.values import set_control_key_value
from spotpython.plot.validation import plot_confusion_matrix, plot_roc
from spotpython.spot import Spot
from spotpython.hyperparameters.values import (get_one_core_model_from_X, add_core_model_to_fun_cont
from spotpython.hyperdict.sklearn_hyper_dict import SklearnHyperDict
from spotpython.hyperparameters.values import (modify_hyper_parameter_bounds, modify_hyper_parameter
from spotpython.fun.hypersklearn import HyperSklearn
from spotpython.utils.eda import print_res_table
from spotpython.utils.init import (fun_control_init, design_control_init, surrogate_control_init, ge
from spotpython.utils.file import load_result
```

```
if not os.path.exists('./figures'):
    os.makedirs('./figures')
warnings.filterwarnings("ignore")
```

34.2. Step 1: Setup

Before we consider the detailed experimental setup, we select the parameters that affect run time, initial design size and the device that is used.

> Caution: Run time (and initial design size) should be increased for real experiments
>
> - MAX_TIME is set to one minute for demonstration purposes. For real experiments, this should be increased to at least 1 hour.

```
MAX_TIME = 1
PREFIX = "401_sklearn_classification"
```

34.3. Step 2: Initialization of the Empty `fun_control` Dictionary

`spotpython` supports the visualization of the hyperparameter tuning process with TensorBoard. The following example shows how to use TensorBoard with `spotpython`. The `fun_control` dictionary is the central data structure that is used to control the optimization process. It is initialized as follows:

```
fun_control = fun_control_init(
    PREFIX=PREFIX,
    TENSORBOARD_CLEAN=True,
    max_time=MAX_TIME,
    fun_evals=inf,
    tolerance_x = np.sqrt(np.spacing(1)))
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_11_04_20_47
```

Tip: TensorBoard

- Since the `spot_tensorboard_path` argument is not `None`, which is the default, `spotpython` will log the optimization process in the TensorBoard folder.
- The `TENSORBOARD_CLEAN` argument is set to `True` to archive the TensorBoard folder if it already exists. This is useful if you want to start a hyperparameter tuning process from scratch. If you want to continue a hyperparameter tuning process, set `TENSORBOARD_CLEAN` to `False`. Then the TensorBoard folder will not be archived and the old and new TensorBoard files will shown in the TensorBoard dashboard.

34.4. Step 3: SKlearn Load Data (Classification)

Randomly generate classification data.

```
n_features = 2
n_samples = 500
target_column = "y"
ds =  make_moons(n_samples, noise=0.5, random_state=0)
X, y = ds
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
train = pd.DataFrame(np.hstack((X_train, y_train.reshape(-1, 1))))
test = pd.DataFrame(np.hstack((X_test, y_test.reshape(-1, 1))))
train.columns = [f"x{i}" for i in range(1, n_features+1)] + [target_column]
test.columns = [f"x{i}" for i in range(1, n_features+1)] + [target_column]
train.head()
```

	x1	x2	y
0	1.960101	0.383172	0.0
1	2.354420	-0.536942	1.0
2	1.682186	-0.332108	0.0
3	1.856507	0.687220	1.0
4	1.925524	0.427413	1.0

Figure 34.1 shows the moons data used for the hyperparameter tuning example.

```
x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
cm = plt.cm.RdBu
cm_bright = ListedColormap(["#FF0000", "#0000FF"])
ax = plt.subplot(1, 1, 1)
ax.set_title("Input data")
# Plot the training points
ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright, edgecolors="k")
# Plot the testing points
ax.scatter(
    X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6, edgecolors="k"
)
ax.set_xlim(x_min, x_max)
ax.set_ylim(y_min, y_max)
ax.set_xticks(())
ax.set_yticks(())
plt.tight_layout()
plt.show()
```

Figure 34.1.: Moons data used for the hyperparameter tuning example.

```
n_samples = len(train)
# add the dataset to the fun_control
fun_control.update({"data": None, # dataset,
                "train": train,
                "test": test,
                "n_samples": n_samples,
                "target_column": target_column})
```

34.5. Step 4: Specification of the Preprocessing Model

Data preprocessing can be very simple, e.g., you can ignore it. Then you would choose the `prep_model` "None":

```
prep_model = None
fun_control.update({"prep_model": prep_model})
```

A default approach for numerical data is the `StandardScaler` (mean 0, variance 1). This can be selected as follows:

```
prep_model = StandardScaler
fun_control.update({"prep_model": prep_model})
```

Even more complicated pre-processing steps are possible, e.g., the following pipeline:

```
categorical_columns = []
one_hot_encoder = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
prep_model = ColumnTransformer(
         transformers=[
             ("categorical", one_hot_encoder, categorical_columns),
         ],
         remainder=StandardScaler,
     )
```

34.6. Step 5: Select Model (`algorithm`) and `core_model_hyper_dict`

The selection of the algorithm (ML model) that should be tuned is done by specifying the its name from the `sklearn` implementation. For example, the `SVC` support vector machine classifier is selected as follows:

```
add_core_model_to_fun_control(core_model=SVC,
                              fun_control=fun_control,
                              hyper_dict=SklearnHyperDict,
                              filename=None)
```

Now `fun_control` has the information from the JSON file. The corresponding entries for the `core_model` class are shown below.

```
fun_control['core_model_hyper_dict']
```

```
{'C': {'type': 'float',
  'default': 1.0,
  'transform': 'None',
  'lower': 0.1,
  'upper': 10.0},
 'kernel': {'levels': ['linear', 'poly', 'rbf', 'sigmoid'],
  'type': 'factor',
  'default': 'rbf',
  'transform': 'None',
  'core_model_parameter_type': 'str',
  'lower': 0,
  'upper': 3},
 'degree': {'type': 'int',
  'default': 3,
  'transform': 'None',
  'lower': 3,
  'upper': 3},
 'gamma': {'levels': ['scale', 'auto'],
  'type': 'factor',
  'default': 'scale',
  'transform': 'None',
  'core_model_parameter_type': 'str',
  'lower': 0,
  'upper': 1},
 'coef0': {'type': 'float',
  'default': 0.0,
  'transform': 'None',
  'lower': 0.0,
  'upper': 0.0},
 'shrinking': {'levels': [0, 1],
  'type': 'factor',
  'default': 0,
  'transform': 'None',
```

```
 'core_model_parameter_type': 'bool',
 'lower': 0,
 'upper': 1},
'probability': {'levels': [0, 1],
 'type': 'factor',
 'default': 0,
 'transform': 'None',
 'core_model_parameter_type': 'bool',
 'lower': 0,
 'upper': 1},
'tol': {'type': 'float',
 'default': 0.001,
 'transform': 'None',
 'lower': 0.0001,
 'upper': 0.01},
'cache_size': {'type': 'float',
 'default': 200,
 'transform': 'None',
 'lower': 100,
 'upper': 400},
'break_ties': {'levels': [0, 1],
 'type': 'factor',
 'default': 0,
 'transform': 'None',
 'core_model_parameter_type': 'bool',
 'lower': 0,
 'upper': 1}}
```

sklearn Model Selection

The following sklearn models are supported by default:

- RidgeCV
- RandomForestClassifier
- SVC
- LogisticRegression
- KNeighborsClassifier
- GradientBoostingClassifier
- GradientBoostingRegressor
- ElasticNet

They can be imported as follows:

```
from sklearn.linear_model import RidgeCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import ElasticNet
```

34.7. Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model`

`spotpython` provides functions for modifying the hyperparameters, their bounds and factors as well as for activating and de-activating hyperparameters without re-compilation of the Python source code. These functions were described in Section 12.19.1.

34.7.1. Modify hyperparameter of type numeric and integer (boolean)

Numeric and boolean values can be modified using the `modify_hyper_parameter_bounds` method.

> **`sklearn Model` Hyperparameters**
>
> The hyperparameters of the `sklearn SVC` model are described in the sklearn documentation.

- For example, to change the `tol` hyperparameter of the `SVC` model to the interval [1e-5, 1e-3], the following code can be used:

```
modify_hyper_parameter_bounds(fun_control, "tol", bounds=[1e-5, 1e-3])
modify_hyper_parameter_bounds(fun_control, "probability", bounds=[0, 0])
fun_control["core_model_hyper_dict"]["tol"]
```

```
{'type': 'float',
 'default': 0.001,
 'transform': 'None',
 'lower': 1e-05,
 'upper': 0.001}
```

34.7.2. Modify hyperparameter of type factor

Factors can be modified with the `modify_hyper_parameter_levels` function. For example, to exclude the `sigmoid` kernel from the tuning, the `kernel` hyperparameter of the `SVC` model can be modified as follows:

```
modify_hyper_parameter_levels(fun_control, "kernel", ["poly", "rbf"])
fun_control["core_model_hyper_dict"]["kernel"]
```

```
{'levels': ['poly', 'rbf'],
 'type': 'factor',
 'default': 'rbf',
 'transform': 'None',
 'core_model_parameter_type': 'str',
 'lower': 0,
 'upper': 1}
```

34.7.3. Optimizers

Optimizers are described in Section 15.2.

34.8. Step 7: Selection of the Objective (Loss) Function

There are two metrics:

1. `metric_river` is used for the river based evaluation via `eval_oml_iter_progressive`.
2. `metric_sklearn` is used for the sklearn based evaluation.

```
fun_control.update({
                "metric_sklearn": log_loss,
                "weights": 1.0,
                })
```

`metric_sklearn`: Minimization and Maximization

- Because the `metric_sklearn` is used for the sklearn based evaluation, it is important to know whether the metric should be minimized or maximized.
- The `weights` parameter is used to indicate whether the metric should be minimized or maximized.
- If `weights` is set to `-1.0`, the metric is maximized.
- If `weights` is set to `1.0`, the metric is minimized, e.g., `weights = 1.0` for

> `mean_absolute_error`, or `weights = -1.0` for `roc_auc_score`.

34.8.1. Predict Classes or Class Probabilities

If the key `"predict_proba"` is set to `True`, the class probabilities are predicted. `False` is the default, i.e., the classes are predicted.

```
fun_control.update({
                "predict_proba": False,
                })
```

34.9. Step 8: Calling the SPOT Function

34.9.1. The Objective Function

The objective function is selected next. It implements an interface from `sklearn`'s training, validation, and testing methods to `spotpython`.

```
fun = HyperSklearn().fun_sklearn
```

The following code snippet shows how to get the default hyperparameters as an array, so that they can be passed to the `Spot` function.

```
X_start = get_default_hyperparameters_as_array(fun_control)
```

34.9.2. Run the `Spot` Optimizer

The class `Spot` [SOURCE] is the hyperparameter tuning workhorse. It is initialized with the following parameters:

- `fun`: the objective function
- `fun_control`: the dictionary with the control parameters for the objective function
- `design`: the experimental design
- `design_control`: the dictionary with the control parameters for the experimental design
- `surrogate`: the surrogate model
- `surrogate_control`: the dictionary with the control parameters for the surrogate model

- `optimizer`: the optimizer
- `optimizer_control`: the dictionary with the control parameters for the optimizer

> **Note: Total run time**
>
> The total run time may exceed the specified `max_time`, because the initial design (here: `init_size` = INIT_SIZE as specified above) is always evaluated, even if this takes longer than `max_time`.

```
design_control = design_control_init()

surrogate_control = surrogate_control_init(method="regression")

spot_tuner = Spot(fun=fun,
                  fun_control=fun_control,
                  design_control=design_control,
                  surrogate_control=surrogate_control)
spot_tuner.run(X_start=X_start)
```

```
spotpython tuning: 6.436366676628063 [----------] 2.62%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [----------] 4.56%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#---------] 6.26%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#---------] 7.77%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#---------] 9.61%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#---------] 11.45%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#---------] 13.14%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#---------] 14.71%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [##--------] 16.39%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [##--------] 18.00%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [##--------] 19.48%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [##--------] 21.05%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [##--------] 22.90%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [##--------] 24.24%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [###-------] 26.23%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [###-------] 28.16%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [###-------] 30.17%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [###-------] 32.01%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [###-------] 34.03%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [####------] 36.20%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [####------] 38.46%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [####------] 40.61%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [####------] 42.63%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [####------] 44.90%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#####-----] 47.05%. Success rate: 0.00%
```

```
spotpython tuning: 6.436366676628063 [#####-----] 48.82%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#####-----] 51.03%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#####-----] 53.04%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [######----] 55.01%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [######----] 57.23%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [######----] 59.30%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [######----] 61.51%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [######----] 63.84%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#######---] 66.00%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#######---] 68.28%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#######---] 70.43%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#######---] 72.42%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#######---] 74.78%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [########--] 77.23%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [########--] 79.16%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [########--] 81.16%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [########--] 83.15%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#########-] 85.53%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#########-] 87.37%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#########-] 89.28%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#########-] 91.47%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [#########-] 93.43%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [##########] 95.36%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [##########] 97.21%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [##########] 99.23%. Success rate: 0.00%
spotpython tuning: 6.436366676628063 [##########] 100.00%. Success rate: 0.00% Done..

Experiment saved to 401_sklearn_classification_res.pkl


<spotpython.spot.spot.Spot at 0x15e933a10>
```

34.9.3. TensorBoard

Now we can start TensorBoard in the background with the following command, where `./runs` is the default directory for the TensorBoard log files:

```
tensorboard --logdir="./runs"
```

> Tip: TENSORBOARD_PATH
>
> The TensorBoard path can be printed with the following command:

```
get_tensorboard_path(fun_control)
```

```
'runs/'
```

We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The TensorBoard plot illustrates how `spotpython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate [SOURCE] is plotted against the number of optimization steps.

Figure 34.2.: TensorBoard visualization of the spotpython optimization process and the surrogate model.

34.10. Step 9: Results

After the hyperparameter tuning run is finished, the results can be saved and reloaded with the following commands:

```
spot_tuner = load_result(PREFIX=PREFIX)
```

```
Loaded experiment from 401_sklearn_classification_res.pkl
```

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized. The black points represent the performance values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress(log_y=True)
```

Results can also be printed in tabular form.

```
print_res_table(spot_tuner)
```

```
| name         | type    | default    |    lower |    upper | tuned                  | trans
|-------------|--------|-----------|---------|---------|----------------------
|-------------|-------------|---------|
| C            | float   | 1.0        |      0.1 |     10.0 | 1.3459476182876375     | None
| kernel       | factor  | rbf        |      0.0 |      1.0 | rbf                    | None
| degree       | int     | 3          |      3.0 |      3.0 | 3.0                    | None
| gamma        | factor  | scale      |      0.0 |      1.0 | scale                  | None
| coef0        | float   | 0.0        |      0.0 |      0.0 | 0.0                    | None
| shrinking    | factor  | 0          |      0.0 |      1.0 | 1                      | None
| probability  | factor  | 0          |      0.0 |      0.0 | 0                      | None
| tol          | float   | 0.001      |    1e-05 |    0.001 | 2.988661226661179e-
05 | None         |          0.04 |          |
| cache_size   | float   | 200.0      |    100.0 |    400.0 | 174.45504889441855     | None
| break_ties   | factor  | 0          |      0.0 |      1.0 | 0                      | None
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=0.0025)
```

34.11. Get Default Hyperparameters

The default hyperparameters, whihc will be used for a comparion with the tuned hyperparameters, can be obtained with the following commands:

```
X_start = get_default_hyperparameters_as_array(fun_control)
model_default = get_one_core_model_from_X(X_start, fun_control, default=True)
model_default
```

C	1.0
kernel	'rbf'
degree	3
gamma	'scale'
coef0	0.0
shrinking	False

probability	False
tol	0.001
cache_size	200.0
class_weight	None
verbose	False
max_iter	-1
decision_function_shape	'ovr'
break_ties	False
random_state	None

34.12. Get SPOT Results

In a similar way, we can obtain the hyperparameters found by `spotpython`.

```
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
model_spot = get_one_core_model_from_X(X, fun_control)
```

34.12.1. Plot: Compare Predictions

```
plot_roc(model_list=[model_default, model_spot], fun_control= fun_control, model_names
```

```
plot_confusion_matrix(model=model_default, fun_control=fun_control, title = "Default")
```

```
# label: 401_plot_confusion_matrix
plot_confusion_matrix(model=model_spot, fun_control=fun_control, title="SPOT")
```

```
min(spot_tuner.y), max(spot_tuner.y)
```

```
(np.float64(6.436366676628063), np.float64(10.813096016735146))
```

34.12.2. Detailed Hyperparameter Plots

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
C:  0.595225337410174
kernel:  100.0
gamma:  0.005577957313750548
shrinking:  0.005577957313750548
```

```
tol:  0.036209781852549854
cache_size:  0.005577957313750548
break_ties:  0.15685728682655498
```

Prediction Surface
Prediction
C
kernel
Prediction Error Surface
Error
C
kernel
Prediction Contour
kernel
C
Error Contour
kernel
C

34.12.3. Parallel Coordinates Plot

```
spot_tuner.parallel_plot()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

34.12.4. Plot all Combinations of Hyperparameters

- Warning: this may take a while.

```
PLOT_ALL = False
if PLOT_ALL:
    n = spot_tuner.k
    for i in range(n-1):
        for j in range(i+1, n):
            spot_tuner.plot_contour(i=i, j=j, min_z=min_z, max_z = max_z)
```

35. HPT: sklearn SVR on Regression Data

This chapter is a tutorial for the Hyperparameter Tuning (HPT) of a `sklearn` SVR model on a regression dataset.

35.1. Step 1: Setup

Before we consider the detailed experimental setup, we select the parameters that affect run time, initial design size and the device that is used.

> Caution: Run time and initial design size should be increased for real experiments
>
> - MAX_TIME is set to one minute for demonstration purposes. For real experiments, this should be increased to at least 1 hour.
> - INIT_SIZE is set to 5 for demonstration purposes. For real experiments, this should be increased to at least 10.

```
MAX_TIME = 1
INIT_SIZE = 20
PREFIX = "18"
```

35.2. Step 2: Initialization of the Empty `fun_control` Dictionary

`spotpython` supports the visualization of the hyperparameter tuning process with TensorBoard. The following example shows how to use TensorBoard with `spotpython`. The `fun_control` dictionary is the central data structure that is used to control the optimization process. It is initialized as follows:

```
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import set_control_key_value
from spotpython.utils.eda import print_res_table
fun_control = fun_control_init(
    PREFIX=PREFIX,
    TENSORBOARD_CLEAN=True,
    max_time=MAX_TIME,
    fun_evals=inf,
    tolerance_x = np.sqrt(np.spacing(1)))
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_11_04_20_50
```

Tip: TensorBoard

- Since the `spot_tensorboard_path` argument is not `None`, which is the default, `spotpython` will log the optimization process in the TensorBoard folder.
- The `TENSORBOARD_CLEAN` argument is set to `True` to archive the TensorBoard folder if it already exists. This is useful if you want to start a hyperparameter tuning process from scratch. If you want to continue a hyperparameter tuning process, set `TENSORBOARD_CLEAN` to `False`. Then the TensorBoard folder will not be archived and the old and new TensorBoard files will shown in the TensorBoard dashboard.

35.3. Step 3: SKlearn Load Data (Classification)

Randomly generate classification data. Here, we use similar data as in Comparison of kernel ridge regression and SVR.

```
import numpy as np

rng = np.random.RandomState(42)

X = 5 * rng.rand(10, 1)
y = np.sin(1/X).ravel()*np.cos(X).ravel()

# Add noise to targets
y[::5] += 3 * (0.5 - rng.rand(X.shape[0] // 5))

X_plot = np.linspace(0, 5, 100000)[:, None]
```

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

n_features = 1
target_column = "y"
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
train = pd.DataFrame(np.hstack((X_train, y_train.reshape(-1, 1))))
test = pd.DataFrame(np.hstack((X_test, y_test.reshape(-1, 1))))
train.columns = [f"x{i}" for i in range(1, n_features+1)] + [target_column]
test.columns = [f"x{i}" for i in range(1, n_features+1)] + [target_column]
train.head()
```

	x1	y
0	1.872701	1.286910
1	4.330881	-0.085207
2	3.659970	-0.234389
3	3.540363	-0.256848
4	0.780093	0.681389

```
n_samples = len(train)
# add the dataset to the fun_control
fun_control.update({"data": None, # dataset,
                "train": train,
                "test": test,
                "n_samples": n_samples,
                "target_column": target_column})
```

35.4. Step 4: Specification of the Preprocessing Model

Data preprocesssing can be very simple, e.g., you can ignore it. Then you would choose the `prep_model` "None":

```
prep_model = None
fun_control.update({"prep_model": prep_model})
```

A default approach for numerical data is the `StandardScaler` (mean 0, variance 1). This can be selected as follows:

```
from sklearn.preprocessing import StandardScaler
prep_model = StandardScaler
fun_control.update({"prep_model": prep_model})
```

Even more complicated pre-processing steps are possible, e.g., the follwing pipeline:

```
categorical_columns = []
one_hot_encoder = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
prep_model = ColumnTransformer(
         transformers=[
             ("categorical", one_hot_encoder, categorical_columns),
         ],
         remainder=StandardScaler,
     )
```

35.5. Step 5: Select Model (`algorithm`) and `core_model_hyper_dict`

The selection of the algorithm (ML model) that should be tuned is done by specifying the its name from the `sklearn` implementation. For example, the `SVC` support vector machine classifier is selected as follows:

```
from spotpython.hyperparameters.values import add_core_model_to_fun_control
from spotpython.hyperdict.sklearn_hyper_dict import SklearnHyperDict
from sklearn.svm import SVR
add_core_model_to_fun_control(core_model=SVR,
                              fun_control=fun_control,
                              hyper_dict=SklearnHyperDict,
                              filename=None)
```

Now `fun_control` has the information from the JSON file. The corresponding entries for the `core_model` class are shown below.

```
fun_control['core_model_hyper_dict']
```

```
{'C': {'type': 'float',
  'default': 1.0,
  'transform': 'None',
  'lower': 0.1,
  'upper': 10.0},
```

```
'kernel': {'levels': ['linear', 'poly', 'rbf', 'sigmoid'],
 'type': 'factor',
 'default': 'rbf',
 'transform': 'None',
 'core_model_parameter_type': 'str',
 'lower': 0,
 'upper': 3},
'degree': {'type': 'int',
 'default': 3,
 'transform': 'None',
 'lower': 3,
 'upper': 3},
'gamma': {'levels': ['scale', 'auto'],
 'type': 'factor',
 'default': 'scale',
 'transform': 'None',
 'core_model_parameter_type': 'str',
 'lower': 0,
 'upper': 1},
'coef0': {'type': 'float',
 'default': 0.0,
 'transform': 'None',
 'lower': 0.0,
 'upper': 0.0},
'epsilon': {'type': 'float',
 'default': 0.1,
 'transform': 'None',
 'lower': 0.01,
 'upper': 1.0},
'shrinking': {'levels': [0, 1],
 'type': 'factor',
 'default': 0,
 'transform': 'None',
 'core_model_parameter_type': 'bool',
 'lower': 0,
 'upper': 1},
'tol': {'type': 'float',
 'default': 0.001,
 'transform': 'None',
 'lower': 0.0001,
 'upper': 0.01},
'cache_size': {'type': 'float',
 'default': 200,
 'transform': 'None',
 'lower': 100,
```

```
'upper': 400}}
```

sklearn Model Selection

The following `sklearn` models are supported by default:

- RidgeCV
- RandomForestClassifier
- SVC
- SVR
- LogisticRegression
- KNeighborsClassifier
- GradientBoostingClassifier
- GradientBoostingRegressor
- ElasticNet

They can be imported as follows:

```
from sklearn.linear_model import RidgeCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.svm import SVR
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import ElasticNet
```

35.6. Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model`

`spotpython` provides functions for modifying the hyperparameters, their bounds and factors as well as for activating and de-activating hyperparameters without re-compilation of the Python source code. These functions were described in Section 12.19.1.

35.6.1. Modify hyperparameter of type numeric and integer (boolean)

Numeric and boolean values can be modified using the `modify_hyper_parameter_bounds` method.

> **sklearn Model Hyperparameters**
>
> The hyperparameters of the sklearn SVC model are described in the sklearn documentation.

- For example, to change the tol hyperparameter of the SVC model to the interval [1e-5, 1e-3], the following code can be used:

```
from spotpython.hyperparameters.values import modify_hyper_parameter_bounds
modify_hyper_parameter_bounds(fun_control, "tol", bounds=[1e-5, 1e-3])
modify_hyper_parameter_bounds(fun_control, "epsilon", bounds=[0.1, 1.0])
# modify_hyper_parameter_bounds(fun_control, "degree", bounds=[2, 5])
fun_control["core_model_hyper_dict"]["tol"]
```

```
{'type': 'float',
 'default': 0.001,
 'transform': 'None',
 'lower': 1e-05,
 'upper': 0.001}
```

35.6.2. Modify hyperparameter of type factor

Factors can be modified with the modify_hyper_parameter_levels function. For example, to exclude the sigmoid kernel from the tuning, the kernel hyperparameter of the SVR model can be modified as follows:

```
from spotpython.hyperparameters.values import modify_hyper_parameter_levels
# modify_hyper_parameter_levels(fun_control, "kernel", ["poly", "rbf"])
modify_hyper_parameter_levels(fun_control, "kernel", ["rbf"])
fun_control["core_model_hyper_dict"]["kernel"]
```

```
{'levels': ['rbf'],
 'type': 'factor',
 'default': 'rbf',
 'transform': 'None',
 'core_model_parameter_type': 'str',
 'lower': 0,
 'upper': 0}
```

35.6.3. Optimizers

Optimizers are described in Section 15.2.

35.7. Step 7: Selection of the Objective (Loss) Function

There are two metrics:

1. `metric_river` is used for the river based evaluation via `eval_oml_iter_progressive`.
2. `metric_sklearn` is used for the sklearn based evaluation.

```
from sklearn.metrics import mean_absolute_error, accuracy_score, roc_curve, roc_auc_sc
fun_control.update({
                "metric_sklearn": mean_squared_error,
                "weights": 1.0,
                })
```

`metric_sklearn`: Minimization and Maximization

- Because the `metric_sklearn` is used for the sklearn based evaluation, it is important to know whether the metric should be minimized or maximized.
- The `weights` parameter is used to indicate whether the metric should be minimized or maximized.
- If `weights` is set to `-1.0`, the metric is maximized.
- If `weights` is set to `1.0`, the metric is minimized, e.g., `weights = 1.0` for `mean_absolute_error`, or `weights = -1.0` for `roc_auc_score`.

35.7.1. Predict Classes or Class Probabilities

If the key `"predict_proba"` is set to `True`, the class probabilities are predicted. `False` is the default, i.e., the classes are predicted.

```
fun_control.update({
                "predict_proba": False,
                })
```

35.8. Step 8: Calling the SPOT Function

35.8.1. The Objective Function

The objective function is selected next. It implements an interface from `sklearn`'s training, validation, and testing methods to `spotpython`.

```
from spotpython.fun.hypersklearn import HyperSklearn
fun = HyperSklearn().fun_sklearn
```

The following code snippet shows how to get the default hyperparameters as an array, so that they can be passed to the `Spot` function.

```
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
```

35.8.2. Run the `Spot` Optimizer

The class `Spot` [SOURCE] is the hyperparameter tuning workhorse. It is initialized with the following parameters:

- `fun`: the objective function
- `fun_control`: the dictionary with the control parameters for the objective function
- `design`: the experimental design
- `design_control`: the dictionary with the control parameters for the experimental design
- `surrogate`: the surrogate model
- `surrogate_control`: the dictionary with the control parameters for the surrogate model
- `optimizer`: the optimizer
- `optimizer_control`: the dictionary with the control parameters for the optimizer

> **Note: Total run time**
>
> The total run time may exceed the specified `max_time`, because the initial design (here: `init_size` = INIT_SIZE as specified above) is always evaluated, even if this takes longer than `max_time`.

```
from spotpython.utils.init import design_control_init, surrogate_control_init
design_control = design_control_init()
set_control_key_value(control_dict=design_control,
                      key="init_size",
                      value=INIT_SIZE,
                      replace=True)

surrogate_control = surrogate_control_init(method="regression")
from spotpython.spot import Spot
spot_tuner = Spot(fun=fun,
```

```
                    fun_control=fun_control,
                    design_control=design_control,
                    surrogate_control=surrogate_control)
spot_tuner.run(X_start=X_start)
```

```
spotpython tuning: 0.11831336203046443 [----------] 1.03%. Success rate: 0.00%
spotpython tuning: 0.11831336203046443 [----------] 1.57%. Success rate: 0.00%
spotpython tuning: 0.11831336203046443 [----------] 1.94%. Success rate: 0.00%
spotpython tuning: 0.11831336203046443 [----------] 2.82%. Success rate: 0.00%
spotpython tuning: 0.11458038664624888 [----------] 3.57%. Success rate: 20.00%
spotpython tuning: 0.11236079316031995 [----------] 4.44%. Success rate: 33.33%
spotpython tuning: 0.10404978403850111 [#---------] 5.39%. Success rate: 42.86%
spotpython tuning: 0.10404978403850111 [#---------] 6.27%. Success rate: 37.50%
spotpython tuning: 0.10404978403850111 [#---------] 7.27%. Success rate: 33.33%
spotpython tuning: 0.1024613375231395 [#---------] 8.39%. Success rate: 40.00%
spotpython tuning: 0.1024613375231395 [#---------] 9.68%. Success rate: 36.36%
spotpython tuning: 0.1024613375231395 [#---------] 10.90%. Success rate: 33.33%
spotpython tuning: 0.1024613375231395 [#---------] 12.15%. Success rate: 30.77%
spotpython tuning: 0.10191006508156582 [#---------] 13.41%. Success rate: 35.71%
spotpython tuning: 0.10191006508156582 [#---------] 14.66%. Success rate: 33.33%
spotpython tuning: 0.10191006508156582 [##--------] 15.98%. Success rate: 31.25%
spotpython tuning: 0.10191006508156582 [##--------] 17.36%. Success rate: 29.41%
spotpython tuning: 0.09826627364288443 [##--------] 18.53%. Success rate: 33.33%
spotpython tuning: 0.08786246635029203 [##--------] 19.81%. Success rate: 36.84%
spotpython tuning: 0.08786246635029203 [##--------] 20.94%. Success rate: 35.00%
spotpython tuning: 0.08424763287416259 [##--------] 22.44%. Success rate: 38.10%
spotpython tuning: 0.08423427702103133 [##--------] 23.92%. Success rate: 40.91%
spotpython tuning: 0.08423427702103133 [###-------] 25.42%. Success rate: 39.13%
spotpython tuning: 0.08423427702103133 [###-------] 26.71%. Success rate: 37.50%
spotpython tuning: 0.08423427702103133 [###-------] 28.00%. Success rate: 36.00%
spotpython tuning: 0.08423427702103133 [###-------] 29.50%. Success rate: 34.62%
spotpython tuning: 0.08423427702103133 [###-------] 30.83%. Success rate: 33.33%
spotpython tuning: 0.08423427702103133 [###-------] 32.17%. Success rate: 32.14%
spotpython tuning: 0.08423427702103133 [###-------] 33.50%. Success rate: 31.03%
spotpython tuning: 0.08423427702103133 [###-------] 34.87%. Success rate: 30.00%
spotpython tuning: 0.08423427702103133 [####------] 36.11%. Success rate: 29.03%
spotpython tuning: 0.08423427702103133 [####------] 37.49%. Success rate: 28.12%
spotpython tuning: 0.08423427702103133 [####------] 38.99%. Success rate: 27.27%
spotpython tuning: 0.08423427702103133 [####------] 40.41%. Success rate: 26.47%
spotpython tuning: 0.08423427702103133 [####------] 41.86%. Success rate: 25.71%
spotpython tuning: 0.08423427702103133 [####------] 43.11%. Success rate: 25.00%
spotpython tuning: 0.08423427702103133 [####------] 44.59%. Success rate: 24.32%
spotpython tuning: 0.08423427702103133 [#####-----] 45.79%. Success rate: 23.68%
spotpython tuning: 0.08423427702103133 [#####-----] 46.99%. Success rate: 23.08%
```

```
Using spacefilling design as fallback.
spotpython tuning: 0.08423427702103133 [#####-----] 48.34%. Success rate: 22.50%
spotpython tuning: 0.08423427702103133 [#####-----] 49.56%. Success rate: 21.95%
spotpython tuning: 0.08423427702103133 [#####-----] 51.27%. Success rate: 21.43%
spotpython tuning: 0.08423427702103133 [#####-----] 52.56%. Success rate: 20.93%
spotpython tuning: 0.08423427702103133 [#####-----] 53.85%. Success rate: 20.45%
spotpython tuning: 0.08423427702103133 [######----] 55.03%. Success rate: 20.00%
spotpython tuning: 0.08423427702103133 [######----] 56.26%. Success rate: 19.57%
spotpython tuning: 0.08423427702103133 [######----] 57.63%. Success rate: 19.15%
spotpython tuning: 0.08423427702103133 [######----] 59.16%. Success rate: 18.75%
spotpython tuning: 0.08423427702103133 [######----] 60.58%. Success rate: 18.37%
spotpython tuning: 0.08423427702103133 [######----] 61.95%. Success rate: 18.00%
spotpython tuning: 0.08423427702103133 [######----] 63.29%. Success rate: 17.65%
spotpython tuning: 0.08423427702103133 [######----] 64.89%. Success rate: 17.31%
spotpython tuning: 0.08423427702103133 [#######---] 66.27%. Success rate: 16.98%
spotpython tuning: 0.08423427702103133 [#######---] 67.80%. Success rate: 16.67%
spotpython tuning: 0.08423427702103133 [#######---] 69.10%. Success rate: 16.36%
spotpython tuning: 0.08423427702103133 [#######---] 70.29%. Success rate: 16.07%
Using spacefilling design as fallback.
spotpython tuning: 0.08423427702103133 [#######---] 71.54%. Success rate: 15.79%
spotpython tuning: 0.08423427702103133 [#######---] 72.92%. Success rate: 15.52%
spotpython tuning: 0.08423427702103133 [#######---] 74.12%. Success rate: 15.25%
spotpython tuning: 0.08423427702103133 [########--] 75.30%. Success rate: 15.00%
spotpython tuning: 0.08423427702103133 [########--] 76.63%. Success rate: 14.75%
spotpython tuning: 0.08423427702103133 [########--] 77.94%. Success rate: 14.52%
spotpython tuning: 0.08423427702103133 [########--] 79.33%. Success rate: 14.29%
spotpython tuning: 0.08423427702103133 [########--] 80.52%. Success rate: 14.06%
spotpython tuning: 0.08423427702103133 [########--] 81.82%. Success rate: 13.85%
spotpython tuning: 0.08423427702103133 [########--] 83.13%. Success rate: 13.64%
spotpython tuning: 0.08423427702103133 [########--] 84.54%. Success rate: 13.43%
spotpython tuning: 0.08423427702103133 [#########-] 85.83%. Success rate: 13.24%
spotpython tuning: 0.08423427702103133 [#########-] 87.14%. Success rate: 13.04%
spotpython tuning: 0.08423427702103133 [#########-] 88.48%. Success rate: 12.86%
spotpython tuning: 0.08423427702103133 [#########-] 90.02%. Success rate: 12.68%
spotpython tuning: 0.08423427702103133 [#########-] 91.66%. Success rate: 12.50%
spotpython tuning: 0.08423427702103133 [#########-] 93.27%. Success rate: 12.33%
spotpython tuning: 0.08423427702103133 [#########-] 94.69%. Success rate: 12.16%
spotpython tuning: 0.08423427702103133 [##########] 96.27%. Success rate: 12.00%
spotpython tuning: 0.08423427702103133 [##########] 97.95%. Success rate: 11.84%
spotpython tuning: 0.08423427702103133 [##########] 99.36%. Success rate: 11.69%
spotpython tuning: 0.08423427702103133 [##########] 100.00%. Success rate: 11.54% Done...

Experiment saved to 18_res.pkl

<spotpython.spot.spot.Spot at 0x142dcacf0>
```

35.8.3. TensorBoard

Now we can start TensorBoard in the background with the following command, where `./runs` is the default directory for the TensorBoard log files:

```
tensorboard --logdir="./runs"
```

```
from spotpython.utils.init import get_tensorboard_path
get_tensorboard_path(fun_control)
```

```
'runs/'
```

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress(log_y=True)
```

Results can also be printed in tabular form.

```
print_res_table(spot_tuner)
```

```
| name        | type    | default    |   lower |    upper | tuned              | transform
|------------|--------|-----------|---------|---------|-------------------
|-------------|--------------|---------|
| C           | float   | 1.0        |     0.1 |     10.0 | 4.592477888931823 | None
| kernel      | factor | rbf        |     0.0 |      0.0 | rbf                | None
| degree      | int     | 3          |     3.0 |      3.0 | 3.0                | None
```

```
| gamma      | factor | scale     |     0.0 |     1.0 | scale             | None        |         0.
| coef0      | float  | 0.0       |     0.0 |     0.0 | 0.0               | None        |         0.
| epsilon    | float  | 0.1       |     0.1 |     1.0 | 0.1               | None        |       100.
| shrinking  | factor | 0         |     0.0 |     1.0 | 1                 | None        |         0.
| tol        | float  | 0.001     |   1e-05 |   0.001 | 0.001             | None        |         0.
| cache_size | float  | 200.0     |   100.0 |   400.0 | 333.4056601001116 | None        |         0.
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=0.0025)
```

35.9. Get Default Hyperparameters

The default hyperparameters, which will be used for a comparion with the tuned hyperparameters, can be obtained with the following commands:

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
model_default = get_one_core_model_from_X(X_start, fun_control, default=True)
model_default
```

kernel	'rbf'
degree	3
gamma	'scale'
coef0	0.0
tol	0.001
C	1.0
epsilon	0.1
shrinking	False
cache_size	200.0
verbose	False
max_iter	-1

35.10. Get SPOT Results

In a similar way, we can obtain the hyperparameters found by `spotpython`.

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
X_tuned = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
model_spot = get_one_core_model_from_X(X_tuned, fun_control)
```

35.10.1. Plot: Compare Predictions

```
model_default.fit(X_train, y_train)
y_default = model_default.predict(X_plot)
```

```
model_spot.fit(X_train, y_train)
y_spot = model_spot.predict(X_plot)
```

```
import matplotlib.pyplot as plt
plt.scatter(X[:100], y[:100], c="orange", label="data", zorder=1, edgecolors=(0, 0, 0
plt.plot(
    X_plot,
    y_default,
    c="red",
    label="Default SVR")

plt.plot(
    X_plot, y_spot, c="blue", label="SPOT SVR")
```

```
plt.xlabel("data")
plt.ylabel("target")
plt.title("SVR")
_ = plt.legend()
```

35.10.2. Detailed Hyperparameter Plots

```
spot_tuner.plot_important_hyperparameter_contour(filename=None)
```

```
C:  0.015527814127685221
gamma:  0.015527814127685221
epsilon:  100.0
shrinking:  0.015527814127685221
tol:  0.13562795430830607
cache_size:  0.015527814127685221
```

Prediction Surface
Prediction
epsilon
tol
Prediction Error Surface
Error
Prediction Contour
Error Contour

Prediction Surface
Prediction
epsilon
shrinking
Prediction Error Surface
Error
epsilon
shrinking
Prediction Contour
shrinking
epsilon
Error Contour
shrinking
epsilon

Prediction Surface
Prediction
epsilon
cache_size
Prediction Error Surface
Error
epsilon
cache_size
Prediction Contour
cache_size
epsilon
Error Contour
cache_size
epsilon

Prediction Surface
Prediction
tol
cache_size
Prediction Error Surface
Error
tol
cache_size
Prediction Contour
cache_size
tol
Error Contour
cache_size
tol

Prediction Surface
Prediction
C
epsilon
Prediction Error Surface
Error
C
epsilon
Prediction Contour
epsilon
C
Error Contour
epsilon
C

Prediction Surface
Prediction
C
tol
Prediction Error Surface
Error
C
tol
Prediction Contour
tol
C
Error Contour
tol
C

Prediction Surface
Prediction
C
gamma
Prediction Error Surface
Error
C
gamma
Prediction Contour
gamma
C
Error Contour
gamma
C

Prediction Surface
Prediction
C
shrinking
Prediction Error Surface
Error
C
shrinking
Prediction Contour
shrinking
C
Error Contour
shrinking
C

Prediction Surface
Prediction
C
cache_size
Prediction Error Surface
Error
C
cache_size
Prediction Contour
cache_size
C
Error Contour
cache_size
C

Prediction Surface
Prediction
gamma
epsilon
Prediction Error Surface
Error
gamma
epsilon
Prediction Contour
epsilon
gamma
Error Contour
epsilon
gamma

Prediction Surface
Prediction
gamma
tol
Prediction Error Surface
Error
gamma
tol
Prediction Contour
tol
gamma
Error Contour
tol
gamma

Prediction Surface
Prediction
0.2019
0.2018
0.2017
0.2016
gamma
shrinking
Prediction Error Surface
Error
0.059800
0.059775
0.059750
0.059725
0.059700
0.059675
0.059650
0.059625
Prediction Contour
0.20198
0.20192
0.20186
0.20180
0.20174
0.20168
0.20162
0.20156
0.20150
Error Contour
0.059808
0.059784
0.059760
0.059736
0.059712
0.059688
0.059664
0.059640
0.059616

Prediction Surface
Prediction
gamma
cache_size
Prediction Error Surface
Error
gamma
cache_size
Prediction Contour
cache_size
gamma
Error Contour
cache_size
gamma

Prediction Surface
Prediction
shrinking
tol
Prediction Error Surface
Error
shrinking
tol
Prediction Contour
tol
shrinking
Error Contour
tol
shrinking

35.10.3. Parallel Coordinates Plot

```
spot_tuner.parallel_plot()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

Part VIII.

Hyperparameter Tuning with River

36. HPT: River

36.1. Introduction to River

37. Simplifying Hyperparameter Tuning in Online Machine Learning—The spotRiverGUI

37.1. Introduction

Batch Machine Learning (BML) often encounters limitations when processing substantial volumes of streaming data (Keller-McNulty 2004; Gaber, Zaslavsky, and Krishnaswamy 2005; Aggarwal 2007). These limitations become particularly evident in terms of available memory, managing drift in data streams (Bifet and Gavaldà 2007, 2009; Gama et al. 2004; Bartz-Beielstein 2024c), and processing novel, unclassified data (Bifet 2010), (Dredze, Oates, and Piatko 2010). As a solution, Online Machine Learning (OML) serves as an effective alternative to BML, adeptly addressing these constraints. OML's ability to sequentially process data proves especially beneficial for handling data streams (Bifet et al. 2010a; Masud et al. 2011; Gama, Sebastião, and Rodrigues 2013; Putatunda 2021; Bartz-Beielstein and Hans 2024).

The Online Machine Learning (OML) methods provided by software packages such as `river` (Montiel et al. 2021) or `MOA` (Bifet et al. 2010b) require the specification of many hyperparameters. To give an example, Hoeffding trees (Hoeglinger and Pears 2007), which are very popular in OML, offer a variety of "splitters" to generate subtrees. There are also several methods to limit the tree size, ensuring time and memory requirements remain manageable. Given the multitude of parameters, manually searching for the optimal hyperparameter setting can be a daunting and often futile task due to the complexity of possible combinations. This article elucidates how automatic hyperparameter optimization, or "tuning", can be achieved. Beyond optimizing the OML process, Hyperparameter Tuning (HPT) executed with the Sequential Parameter Optimization Toolbox (SPOT) enhances the explainability and interpretability of OML procedures. This can result in a more efficient, resource-conserving algorithm, contributing to the concept of "Green AI".

> **Note**
>
> Note: This document refers to `spotRiverGUI` version 0.0.26 which was released on Feb 18, 2024 on GitHub, see: https://github.com/sequential-parameter-

> optimization/spotGUI/tree/main. The GUI is under active development and new features will be added soon.

This article describes the `spotRiverGUI`, which is a graphical user interface for the `spotriver` package. The GUI allows the user to select the task, the data set, the preprocessing model, the metric, and the online machine learning model. The user can specify the experiment duration, the initial design, and the evaluation options. The GUI provides information about the data set and allows the user to save and load experiments. It also starts and stops a tensorboard process to observe the tuning online and provides an analysis of the hyperparameter tuning process. The `spotRiverGUI` releases the user from the burden of manually searching for the optimal hyperparameter setting. After providing the data, users can compare different OML algorithms from the powerful `river` package in a convenient way and tune the selected algorithm very efficiently.

This article is structured as follows:

Section 37.2 describes how to install the software. It also explains how the `spotRiverGUI` can be started. Section 37.3 describes the binary classification task and the options available in the `spotRiverGUI`. Section 37.4 provides information about the planned regression task. Section 37.5 describes how the data can be visualized in the `spotRiverGUI`. Section 37.6 provides information about saving and loading experiments. Section 37.7 describes how to start an experiment and how the associated tensorboard process can be started and stopped. Section 37.8 provides information about the analysis of the results from the hyperparameter tuning process. Section 37.9 concludes the article and provides an outlook.

37.2. Installation and Starting

37.2.1. Installation

We strongly recommend using a virtual environment for the installation of the `river`, `spotriver`, `build` and `spotRiverGUI` packages.

Miniforge, which holds the minimal installers for Conda, is a good starting point. Please follow the instructions on https://github.com/conda-forge/miniforge. Using Conda, the following commands can be used to create a virtual environment (Python 3.11 is recommended):

```
>> conda create -n myenv python=3.11
>> conda activate myenv
```

Now the `river` and `spotriver` packages can be installed:

```
>> (myenv) pip install river spotriver build
```

Although the `spotGUI` package is available on PyPI, we recommend an installation from the GitHub repository https://github.com/sequential-parameter-optimization/spotGUI, because the `spotGUI` package is under active development and new features will be added soon. The installation from the GitHub repository is done by executing the following command:

```
>> (myenv) git clone git@github.com:sequential-parameter-optimization/spotGUI.git
```

Building the `spotGUI` package is done by executing the following command:

```
>> (myenv) cd spotGUI
>> (myenv) python -m build
```

Now the `spotRiverGUI` package can be installed:

```
>> (myenv) pip install dist/spotGUI-0.0.26.tar.gz
```

37.2.2. Starting the GUI

The GUI can be started by executing the `spotRiverGUI.py` file in the `spotGUI/spotRiverGUI` directory. Change to the `spotRiverGUI` directory and start the GUI:

```
>> (myenv) cd spotGUI/spotRiverGUI
>> (myenv) python spotRiverGUI.py
```

The GUI window will open, as shown in Figure 37.1.

After the GUI window has opened, the user can select the task. Currently, `Binary Classification` is available. Further tasks like `Regression` will be available soon.

Depending on the task, the user can select the data set, the preprocessing model, the metric, and the online machine learning model.

37.3. Binary Classification

37.3.1. Binary Classification Options

If the `Binary Classification` task is selected, the user can select pre-specified data sets from the `Data` drop-down menu.

Figure 37.1.: spotriver GUI

37.3.1.1. River Data Sets

The following data sets from the `river` package are available (the descriptions are taken from the `river` package):

- `Bananas`: An artificial dataset where instances belongs to several clusters with a banana shape.There are two attributes that correspond to the x and y axis, respectively. More: https://riverml.xyz/dev/api/datasets/Bananas/.
- `CreditCard`: Credit card frauds. The datasets contains transactions made by credit cards in September 2013 by European cardholders. Feature '`Class`' is the response variable and it takes value 1 in case of fraud and 0 otherwise. More: https://riverml.xyz/dev/api/datasets/CreditCard/.
- `Elec2`: Electricity prices in New South Wales. This is a binary classification task, where the goal is to predict if the price of electricity will go up or down. This data was collected from the Australian New South Wales Electricity Market. In this market, prices are not fixed and are affected by demand and supply of the market. They are set every five minutes. Electricity transfers to/from the neighboring state of Victoria were done to alleviate fluctuations. More: https://riverml.xyz/dev/api/datasets/Elec2/.
- `Higgs`: The data has been produced using Monte Carlo simulations. The first 21 features (columns 2-22) are kinematic properties measured by the particle detectors in the accelerator. The last seven features are functions of the first 21 features; these are high-level features derived by physicists to help discriminate between the two classes. More: https://riverml.xyz/dev/api/datasets/Higgs/.
- `HTTP`: HTTP dataset of the KDD 1999 cup. The goal is to predict whether or not an HTTP connection is anomalous or not. The dataset only contains 2,211 (0.4%) positive labels. More: https://riverml.xyz/dev/api/datasets/HTTP/.

- `Phishing`: Phishing websites. This dataset contains features from web pages that are classified as phishing or not.https://riverml.xyz/dev/api/datasets/Phishing/

37.3.1.2. User Data Sets

Besides the `river` data sets described in Section 37.3.1.1, the user can also select a user-defined data set. Currently, comma-separated values (CSV) files are supported. Further formats will be supported soon. The user-defined CSV data set must be a binary classification task with the target variable in the last column. The first row must contain the column names. If the file is copied to the subdirectory `userData`, the user can select the data set from the `Data` drop-down menu.

As an example, we have provided a CSV-version of the `Phishing` data set. The file is located in the `userData` subdirectory and is called `PhishingData.csv`. It contains the columns `empty_server_form_handler`, `popup_window`, `https`, `request_from_other_domain`, `anchor_from_other_domain`, `is_popular`, `long_url`, `age_of_domain`, `ip_in_url`, and `is_phishing`. The first few lines of the file are shown below (modified due to formatting reasons):

```
empty_server_form_handler,...,is_phishing
0.0,0.0,0.0,0.0,0.0,0.5,1.0,1,1,1
1.0,0.0,0.5,0.5,0.0,0.5,0.0,1,0,1
0.0,0.0,1.0,0.0,0.5,0.5,0.0,1,0,1
0.0,0.0,1.0,0.0,0.0,1.0,0.5,0,0,1
```

Based on the required format, we can see that `is_phishing` is the target column, because it is the last column of the data set.

37.3.1.3. Stream Data Sets

Forthcoming versions of the GUI will support stream data sets, e.g, the Friedman-Drift generator (Ikonomovska 2012) or the SEA-Drift generator (Street and Kim 2001). The Friedman-Drift generator was also used in the hyperparameter tuning study in Bartz-Beielstein (2024b).

37.3.1.4. Data Set Options

Currently, the user can select the following parameters for the data sets:

- `n_total`: The total number of instances. Since some data sets are quite large, the user can select a subset of the data set by specifying the `n_total` value.
- `test_size`: The size of the test set in percent (`0.0 - 1.0`). The training set will be `1.0 - test_size`.

The target column should be the last column of the data set. Future versions of the GUI will support the selection of the `target_column` from the GUI. Currently, the value from the field `target_column` has not effect.

To compare different data scaling methods, the user can select the preprocessing model from the `Preprocessing` drop-down menu. Currently, the following preprocessing models are available:

- `StandardScaler`: Standardize features by removing the mean and scaling to unit variance.
- `MinMaxScaler`: Scale features to a range.
- `None`: No scaling is performed.

The `spotRiverGUI` will not provide sophisticated data preprocessing methods. We assume that the data was preprocessed before it is copied into the `userData` subdirectory.

37.3.2. Experiment Options

Currently, the user can select the following options for specifying the experiment duration:

- `MAX_TIME`: The maximum time in minutes for the experiment.
- `FUN_EVALS`: The number of function evaluations for the experiment. This is the number of OML-models that are built and evaluated.

If the `MAX_TIME` is reached or `FUN_EVALS` OML models are evaluated, the experiment will be stopped.

> **Initial design is always evaluated**
>
> - The initial design will always be evaluated before one of the stopping criteria is reached.
> - If the initial design is very large or the model evaluations are very time-consuming, the runtime will be larger than the `MAX_TIME` value.

Based on the `INIT_SIZE`, the number of hyperparameter configurations for the initial design can be specified. The initial design is evaluated before the first surrogate model is built. A detailed description of the initial design and the surrogate model based hyperparameter tuning can be found in Bartz-Beielstein (2024a) and in Bartz-Beielstein and Zaefferer (2022). The `spotpython` package is used for the hyperparameter tuning process. It implements a robust surrogate model based optimization method (Forrester, Sóbester, and Keane 2008).

The `PREFIX` parameter can be used to specify the experiment name.

The `spotpython` hyperparameter tuning program allows the user to specify several options for the hyperparameter tuning process. The `spotRiverGUI` will support more options in future versions. Currently, the user can specify whether the outcome from the experiment is noisy or deterministic. The corresponding parameter is called `NOISE`. The reader is referred to Bartz-Beielstein (2024b) and to the chapter "Handling Noise" (https://sequential-parameter-optimization.github.io/Hyperparameter-Tuning-Cookbook/013_num_spot_noisy.html) for further information about the `NOISE` parameter.

37.3.3. Evaluation Options

The user can select one of the following evaluation metrics for binary classification tasks from the `metric` drop-down menu:

- `accuracy_score`
- `cohen_kappa_score`
- `f1_score`
- `hamming_loss`
- `hinge_loss`
- `jaccard_score`
- `matthews_corrcoef`
- `precision_score`
- `recall_score`
- `roc_auc_score`
- `zero_one_loss`

These metrics are based on the `scikit-learn` module (Pedregosa et al. 2011), which implements several loss, score, and utility functions to measure classification performance, see https://scikit-learn.org/stable/modules/model_evaluation.html#classification-metrics. `spotRiverGUI` supports metrics that are computed from the `y_pred` and the `y_true` values. The `y_pred` values are the predicted target values, and the `y_true` values are the true target values. The `y_pred` values are generated by the online machine learning model, and the `y_true` values are the true target values from the data set.

> **Evaluation Metrics: Minimization and Maximization**
>
> - Some metrics are minimized, and some are maximized. The `spotRiverGUI` will support the user in selecting the correct metric based on the task. For example, the `accuracy_score` is maximized, and the `hamming_loss` is minimized. The user can select the metric and `spotRiverGUI` will automatically determine whether the metric is minimized or maximized.

In addition to the evaluation metric results, `spotriver` considers the time and memory consumption of the online machine learning model. The `spotRiverGUI` will support the user in selecting the time and memory consumption as additional evaluation metrics. By modifying the weight vector, which is shown in the `weights: y, time, mem` field, the user can specify the importance of the evaluation metrics. For example, the weight vector `1,0,0` specifies that only the `y` metric (e.g., accuracy) is considered. The weight vector `0,1,0` specifies that only the time metric is considered. The weight vector `0,0,1` specifies that only the memory metric is considered. The weight vector `1,1,1` specifies that all metrics are considered. Any real values (also negative ones) are allowed for the weights.

> **The weight vector**
>
> - The specification of adequate weights is highly problem dependent.
> - There is no generic setting that fits to all problems.

As described in Bartz-Beielstein (2024a), a prediction horizon is used for the comparison of the online-machine learning algorithms. The `horizon` can be specified in the `spotRiverGUI` by the user and is highly problem dependent. The `spotRiverGUI` uses the `eval_oml_horizon` method from the `spotriver` package, which evaluates the online-machine learning model on a rolling horizon basis.

In addition to the `horizon` value, the user can specify the `oml_grace_period` value. During the `oml_grace_period`, the OML-model is trained on the (small) training data set. No predictions are made during this initial training phase, but the memory and computation time are measured. Then, the OML-model is evaluated on the test data set using a given (sklearn) evaluation metric. The default value of the `oml_grace_period` is `horizon`. For convenience, the value `horizon` is also selected when the user specifies the `oml_grace_period` value as `None`.

> **The oml_grace_period**
>
> - If the `oml_grace_period` is set to the size of the training data set, the OML-model is trained on the entire training data set and then evaluated on the test data set using a given (sklearn) evaluation metric.
> - This setting might be "unfair" in some cases, because the OML-model should learn online and not on the entire training data set.
> - Therefore, a small data set is recommended for the `oml_grace_period` setting and the prediction `horizon` is a recommended value for the `oml_grace_period` setting. The reader is referred to Bartz-Beielstein (2024a) for further information about the `oml_grace_period` setting.

37.3.4. Online Machine Learning Model Options

The user can select one of the following online machine learning models from the `coremodel` drop-down menu:

- `forest.AMFClassifier`: Aggregated Mondrian Forest classifier for online learning (Mourtada, Gaiffas, and Scornet 2019). This implementation is truly online, in the sense that a single pass is performed, and that predictions can be produced anytime. More: https://riverml.xyz/dev/api/forest/AMFClassifier/.
- `tree.ExtremelyFastDecisionTreeClassifier`: Extremely Fast Decision Tree (EFDT) classifier (Manapragada, Webb, and Salehi 2018). Also referred to as the Hoeffding AnyTime Tree (HATT) classifier. In practice, despite the name, EFDTs are typically slower than a vanilla Hoeffding Tree to process data. More: https://riverml.xyz/dev/api/tree/ExtremelyFastDecisionTreeClassifier/.
- `tree.HoeffdingTreeClassifier`: Hoeffding Tree or Very Fast Decision Tree classifier (Bifet et al. 2010a; Domingos and Hulten 2000). More: https://riverml.xyz/dev/api/tree/HoeffdingTreeClassifier/.
- `tree.HoeffdingAdaptiveTreeClassifier`: Hoeffding Adaptive Tree classifier (Bifet and Gavaldà 2009). More: https://riverml.xyz/dev/api/tree/HoeffdingAdaptiveTreeClassifier/.
- `linear_model.LogisticRegression`: Logistic regression classifier. More: hhttps://riverml.xyz/dev/api/linear-model/LogisticRegression/.

The `spotRiverGUI` automatically determines the hyperparameters for the selected online machine learning model and adapts the input fields to the model hyperparameters. The user can modify the hyperparameters in the GUI. Figure 37.2 shows the `spotRiverGUI` when the `forest.AMFClassifier` is selected and Figure 37.3 shows the `spotRiverGUI` when the `tree.HoeffdingTreeClassifier` is selected.

Figure 37.2.: `spotRiverGUI` when `forest.AMFClassifier` is selected

Numerical and categorical hyperparameters are treated differently in the `spotRiverGUI`:

Figure 37.3.: `spotRiverGUI` when `tree.HoeffdingAdaptiveTreeClassifier` is selected

- The user can modify the lower and upper bounds for the numerical hyperparameters.
- There are no upper or lower bounds for categorical hyperparameters. Instead, hyperparameter values for the categorical hyperparameters are considered as sets of values, e.g., the set of `ExhaustiveSplitter`, `HistogramSplitter`, `GaussianSplitter` is provided for the `splitter` hyperparameter of the `tree.HoeffdingAdaptiveTreeClassifier` model as can be seen in Figure 37.3. The user can select the full set or any subset of the set of values for the categorical hyperparameters.

In addition to the lower and upper bounds (or the set of values for the categorical hyperparameters), the `spotRiverGUI` provides information about the `Default values` and the `Transformation` function. If the `Transformation` function is set to `None`, the values of the hyperparameters are passed to the `spot` tuner as they are. If the `Transformation` function is set to `transform_power_2_int`, the value x is transformed to 2^x before it is passed to the `spot` tuner.

Modifications of the `Default values` and `Transformation` functions values in the `spotRiverGUI` have no effect on the hyperparameter tuning process. This is intensional. In future versions, the user will be able to add their own hyperparameter dictionaries to the `spotRiverGUI`, which allows the modification of `Default values` and `Transformation` functions values. Furthermore, the `spotRiverGUI` will support more online machine learning models in future versions.

37.4. Regression

Regression tasks will be supported soon. The same workflow as for the binary classification task will be used, i.e., the user can select the data set, the preprocessing model, the metric, and the online machine learning model.

37.5. Showing the Data

The `spotRiverGUI` provides the `Show Data` button, which opens a new window and shows information about the data set. The first figure (Figure 37.4) shows histograms of the target variables in the train and test data sets. The second figure (Figure 37.5) shows scatter plots of the features in the train data set. The third figure (Figure 37.6) shows the corresponding scatter plots of the features in the test data set.

Figure 37.4.: Output from the `spotRiverGUI` when `Bananas` data is selected for the `Show Data` option

> **Size of the Displayed Data Sets**
>
> - Some data sets are quite large and the display of the data sets might take some time.
> - Therefore, a random subset of 1000 instances of the data set is displayed if the data set is larger than 1000 instances.

Showing the data is important, especially for the new / unknown data sets as can be seen in Figure 37.7, Figure 37.8, and Figure 37.9: The target variable is highly biased. The user can check whether the data set is correctly formatted and whether the target variable is correctly specified.

Figure 37.5.: Visualization of the train data. Output from the `spotRiverGUI` when `Bananas` data is selected for the `Show Data` option

Figure 37.6.: Visualization of the test data. Output from the `spotRiverGUI` when `Bananas` data is selected for the `Show Data` option

Figure 37.7.: Output from the `spotRiverGUI` when `HTTP` data is selected for the `Show Data` option. The target variable is biased.

Figure 37.8.: Output from the `spotRiverGUI` when `HTTP` data is selected for the `Show Data` option. A subset of 1000 randomly chosen data points is shown. Only a few positive events are in the data.

Figure 37.9.: Output from the `spotRiverGUI` when `HTTP` data is selected for the `Show Data` option. The test data set shows the same structure as the train data set.

In addition to the histograms and scatter plots, the `spotRiverGUI` provides textual information about the data set in the console window. e.g., for the `Bananas` data set, the following information is shown:

```
Train data summary:
                 x1           x2            y
count  3710.000000  3710.000000  3710.000000
mean     -0.016243     0.002430     0.451482
std       0.995490     1.001150     0.497708
min      -3.089839    -2.385937     0.000000
25%      -0.764512    -0.914144     0.000000
50%      -0.027259    -0.033754     0.000000
75%       0.745066     0.836618     1.000000
max       2.754447     2.517112     1.000000

Test data summary:
                 x1           x2            y
count  1590.000000  1590.000000  1590.000000
mean      0.037900    -0.005670     0.440881
std       1.009744     0.997603     0.496649
min      -2.980834    -2.199138     0.000000
25%      -0.718710    -0.911151     0.000000
```

```
50%        0.034858    -0.046502     0.000000
75%        0.862049     0.806506     1.000000
max        2.813360     3.194302     1.000000
```

37.6. Saving and Loading

37.6.1. Saving the Experiment

If the experiment should not be started immediately, the user can save the experiment by clicking on the `Save Experiment` button. The `spotRiverGUI` will save the experiment as a pickle file. The file name is generated based on the `PREFIX` parameter. The pickle file contains a set of dictionaries, which are used to start the experiment.

`spotRiverGUI` shows a summary of the selected hyperparameters in the console window as can be seen in Table 37.1.

Table 37.1.: The hyperparameter values for the `tree.HoeffdingAdaptiveTreeClassifier` model.

name	type	default	lower	upper	transform
grace_period	int	200	10	1000	None
max_depth	int	20	2	20	transform_power_2_int
delta	float	1e-07	1e-08	1e-06	None
tau	float	0.05	0.01	0.1	None
leaf_prediction	factor	nba	0	2	None
nb_threshold	int	0	0	10	None
splitter	factor	GaussianSplitter	0	2	None
bootstrap_sampling	factor	0	0	1	None
drift_window_threshold	int	300	100	500	None
drift_detector	factor	ADWIN	0	0	None
switch_significance	float	0.05	0.01	0.1	None
binary_split	factor	0	0	1	None
max_size	float	100.0	100	1000	None
memory_estimate_period	int	1000000	100000	1e+06	None
stop_mem_management	factor	0	0	1	None

name	type	default	lower	upper	transform
remove_poor_attrs	factor	0	0	1	None
merit_preprune	factor	0	0	1	None

37.6.2. Loading an Experiment

Future versions of the `spotRiverGUI` will support the loading of experiments from the GUI. Currently, the user can load the experiment by executing the command `load_experiment`, see https://sequential-parameter-optimization.github.io/spotpython/reference/spotpython/utils/file/#spotpython.utils.file.load_experiment.

37.7. Running a New Experiment

An experiment can be started by clicking on the `Run Experiment` button. The GUI calls `run_spot_python_experiment` from `spotGUI.tuner.spotRun`. Output will be shown in the console window from which the GUI was started.

37.7.1. Starting and Stopping Tensorboard

Tensorboard (Abadi et al. 2016) is automatically started when an experiment is started. The tensorboard process can be observed in a browser by opening the http://localhost:6006 page. Tensorboard provides a visual representation of the hyperparameter tuning process. Figure 37.10 and Figure 37.11 show the tensorboard page when the `spotRiverGUI` is performing the tuning process.

`spotpython.utils.tensorboard` provides the methods `start_tensorboard` and `stop_tensorboard` to start and stop tensorboard as a background process. After the experiment is finished, the tensorboard process is stopped automatically.

37.8. Performing the Analysis

If the hyperparameter tuning process is finished, the user can analyze the results by clicking on the `Analysis` button. The following options are available:

- Progress plot
- Compare tuned versus default hyperparameters
- Importance of hyperparameters
- Contour plot

Figure 37.10.: Tensorboard visualization of the hyperparameter tuning process

Figure 37.11.: Tensorboard. Parallel coordinates plot

- Parallel coordinates plot

Figure 37.12 shows the progress plot of the hyperparameter tuning process. Black dots denote results from the initial design. Red dots illustrate the improvement found by the surrogate model based optimization. For binary classification tasks, the `roc_auc_score` can be used as the evaluation metric. The confusion matrix is shown in Figure 37.13. The default versus tuned hyperparameters are shown in Figure 37.14. The surrogate plot is shown in Figure 37.15, Figure 37.16, and Figure 37.17.

Figure 37.12.: Progress plot of the hyperparameter tuning process

Figure 37.13.: Confusion matrix

Furthermore, the tuned hyperparameters are shown in the console window. A typical output is shown below (modified due to formatting reasons):

```
|name    |type   |default |low | up |tuned |transf |importance|stars|
|--------|-------|--------|----|----|------|-------|----------|-----|
|n_estim |int    |    3.0 |2.0 |7.0 |  3.0 | pow_2 |      0.04|     |
|step    |float  |    1.0 |0.1 |10.0|  5.12| None  |      0.21| .   |
|use_agg |factor |    1.0 |0.0 |1.0 |  0.0 | None  |     10.17| *   |
|dirichl |float  |    0.5 |0.1 |0.75|  0.37| None  |     13.64| *   |
|split_p |factor |    0.0 |0.0 |1.0 |  0.0 | None  |    100.00| *** |
```

Figure 37.14.: Default versus tuned hyperparameters

Figure 37.15.: Surrogate plot based on the Kriging model. `x0` and `x1` plotted against each other.

Figure 37.16.: Surrogate plot based on the Kriging model. `x1` and `x2` plotted against each other.

Figure 37.17.: Surrogate plot based an the Kriging model. `x0` and `x2` plotted against each other.

In addition to the tuned parameters that are shown in the column `tuned`, the columns `importance` and `stars` are shown. Both columns show the most important hyperparameters based on information from the surrogate model. The `stars` column shows the importance of the hyperparameters in a graphical way. It is important to note that the results are based on a demo of the hyperparameter tuning process. The plots are not based on a real hyperparameter tuning process. The reader is referred to Bartz-Beielstein (2024b) for further information about the analysis of the hyperparameter tuning process.

37.9. Summary and Outlook

The `spotRiverGUI` provides a graphical user interface for the `spotriver` package. It releases the user from the burden of manually searching for the optimal hyperparameter setting. After copying a data set into the `userData` folder and starting `spotRiverGUI`, users can compare different OML algorithms from the powerful `river` package in a convenient way. Users can generate configurations on their local machines, which can be transferred to a remote machine for execution. Results from the remote machine can be copied back to the local machine for analysis.

! Benefits of the spotRiverGUI:

- Very easy to use (only the data must be provided in the correct format).
- Reproducible results.
- State-of-the-art hyperparameter tuning methods.
- Powerful analysis tools, e.g., Bayesian optimization (Forrester, Sóbester, and Keane 2008; Gramacy 2020).
- Visual representation of the hyperparameter tuning process with tensorboard.
- Most advanced online machine learning models from the `river` package.

The `river` package (Montiel et al. 2021), which is very well documented, can be downloaded from https://riverml.xyz/latest/.

The `spotRiverGUI` is under active development and new features will be added soon. It can be downloaded from GitHub: https://github.com/sequential-parameter-optimization/spotGUI.

Interactive Jupyter Notebooks and further material about OML are provided in the GitHub repository https://github.com/sn-code-inside/online-machine-learning. This material is part of the supplementary material of the book "Online Machine Learning - A Practical Guide with Examples in Python", see https://link.springer.com/book/9789819970063 and the forthcoming book "Online Machine Learning - Eine praxisorientierte Einführung", see https://link.springer.com/book/9783658425043.

37.10. Appendix

37.10.1. Adding new Tasks

Currently, three tasks are supported in the `spotRiverGUI`: `Binary Classification`, `Regression`, and `Rules`. `Rules` was added in ver 0.6.0. Here, we document how this task updated was implemented. Adding an additional task requires modifications in the following files:

- `spotRun.py`:
  - The `river`class `rules` must be imported, i.e., `from river import forest, tree, linear_model, rules`.
  - The method `get_river_rules_core_model_names()` must be modified.
  - The `get_scenario_dict()` method must be modified.
- `CTk.py`:
  - The `task_frame` must be extended.
  - The `change_task_event()` method must be modified.

In addition, the hyperparameter dictionary in `spotriver` must be updated. This is the only modification required in the `spotriver`package.

38. river Hyperparameter Tuning: Hoeffding Tree Regressor with Friedman Drift Data

This chapter demonstrates hyperparameter tuning for river's Hoeffding Tree Regressor (HTR) with the Friedman drift data set [SOURCE]. The Hoeffding Tree Regressor is a regression tree that uses the Hoeffding bound to limit the number of splits evaluated at each node, i.e., it predicts a real value for each sample.

38.1. The Friedman Drift Data Set

We will use the Friedman synthetic dataset with concept drifts, which is described in detail in Section E.2. The following parameters are used to generate and handle the data set:

- position: The positions of the concept drifts.
- n_train: The number of samples used for training.
- n_test: The number of samples used for testing.
- seed: The seed for the random number generator.
- target_column: The name of the target column.
- drift_type: The type of the concept drift.

We will use spotriver's convert_to_df function [SOURCE] to convert the river data set to a pandas data frame. Then we add column names x1 until x10 to the first 10 columns of the dataframe and the column name y to the last column of the dataframe.

This data generation is independently repeated for the training and test data sets, because the data sets are generated with concept drifts and the usual train-test split would not work.

```
from river.datasets import synth
import pandas as pd
import numpy as np
from spotriver.utils.data_conversion import convert_to_df
```

```
n_train = 6_000
n_test = 4_000
n_samples = n_train + n_test
target_column = "y"

dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=(n_train/4, n_train/2),
   seed=123
)

train = convert_to_df(dataset, n_total=n_train)
train.columns = [f"x{i}" for i in range(1, 11)] + [target_column]
```

```
dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=(n_test/4, n_test/2),
   seed=123
)
test = convert_to_df(dataset, n_total=n_test)
test.columns = [f"x{i}" for i in range(1, 11)] + [target_column]
```

> **The Data Set**
>
> Data sets that are available as `pandas` dataframes can easily be passed to the `spot` hyperparameter tuner. `spotpython` requires a `train` and a `test` data set, where the column names must be identical.

We combine the train and test data sets and save them to a csv file.

```
df = pd.concat([train, test])
df.to_csv("./userData/friedman.csv", index=False)
```

The Friedman Drift data set described in this section is avaialble as a `csv` data file and can be downloaded from github: friedman.csv.

38.2. Setup

38.2.1. General Experiment Setup

To keep track of the different experiments, we use a `PREFIX` for the experiment name. The `PREFIX` is used to create a unique experiment name. The `PREFIX` is also used to

create a unique TensorBoard folder, which is used to store the TensorBoard log files.

`spotpython` allows the specification of two different types of stopping criteria: first, the number of function evaluations (`fun_evals`), and second, the maximum run time in seconds (`max_time`). Here, we will set the number of function evaluations to infinity and the maximum run time to one minute.

Furthermore, we set the initial design size (`init_size`) to 10. The initial design is used to train the surrogate model. The surrogate model is used to predict the performance of the hyperparameter configurations. The initial design is also used to train the first model. Since the `init_size` belongs to the experimental design, it is set in the `design_control` dictionary, see [SOURCE].

`max_time` is set to one minute for demonstration purposes and `init_size` is set to 10 for demonstration purposes. For real experiments, these values should be increased. Note, the total run time may exceed the specified `max_time`, because the initial design is always evaluated, even if this takes longer than `max_time`.

Summary: General Experiment Setup

The following parameters are used to specify the general experiment setup:

```
PREFIX = "024"
fun_evals = inf
max_time = 1
init_size = 10
```

38.2.2. Data Setup

We use the `StandardScaler` [SOURCE] from `river` as the data-preprocessing model. The `StandardScaler` is used to standardize the data set, i.e., it has zero mean and unit variance.

The names of the training and test data sets are `train` and `test`, respectively. They are available as `pandas` dataframes. Both must use the same column names. The column names were set to `x1` to `x10` for the features and `y` for the target column during the data set generation in Section 38.1. Therefore, the `target_column` is set to `y` (as above).

Summary: Data Setup

The following parameters are used to specify the data setup:

```
prep_model_name = "StandardScaler"
test = test
train = train
target_column = "y"
```

38.2.3. Evaluation Setup

Here we use the `mean_absolute_error` [SOURCE] as the evaluation metric. Internally, this metric is passed to the objective (or loss) function `fun_oml_horizon` [SOURCE] and further to the evaluation function `eval_oml_horizon` [SOURCE].

`spotriver` also supports additional metrics. For example, the `metric_river` is used for the river based evaluation via `eval_oml_iter_progressive` [SOURCE]. The `metric_river` is implemented to simulate the behaviour of the "original" `river` metrics.

Summary: Evaluation Setup

The following parameter are used to select the evaluation metric:

```
metric_sklearn_name = "mean_absolute_error"
```

38.2.4. River-Specific Setup

In the online-machine-learning (OML) setup, the model is trained on a fixed number of observations and then evaluated on a fixed number of observations. The `horizon` defines the number of observations that are used for the evaluation. Here, a horizon of 7*24 is used, which corresponds to one week of data.

The `oml_grace_period` defines the number of observations that are used for the initial training of the model. This value is relatively small, since the online-machine-learning is trained on the incoming data and the model is updated continuously. However, it needs a certain number of observations to start the training process. Therefore, this short training period aka `oml_grace_period` is set to the horizon, i.e., the number of observations that are used for the evaluation. In this case, we use a horizon of 7*24.

The `weights` provide a flexible way to define specific requirements, e.g., if the memory is more important than the time, the weight for the memory can be increased. `spotriver` stores information about the model' s score (metric), memory, and time. The hyperparamter tuner requires a single objective. Therefore, a weighted sum of the metric, memory, and time is computed. The weights are defined in the `weights` array.

The `weights` provide a flexible way to define specific requirements, e.g., if the memory is more important than the time, the weight for the memory can be increased.

The `weight_coeff` defines a multiplier for the results: results are multiplied by (step/n_steps)**weight_coeff, where n_steps is the total number of iterations. Results from the beginning have a lower weight than results from the end if weight_coeff > 1. If weight_coeff == 0, all results have equal weight. Note, that the `weight_coeff` is only used internally for the tuner and does not affect the results that are used for the evaluation or comparisons.

Summary: River-Specific Setup

The following parameters are used:

```
horizon = 7*24
oml_grace_period = 7*24
weights = np.array([1, 0.01, 0.01])
weight_coeff = 0.0
```

38.2.5. Model Setup

By using `core_model_name = "tree.HoeffdingTreeRegressor"`, the `river` model class `HoeffdingTreeRegressor` [SOURCE] from the `tree` module is selected. For a given `core_model_name`, the corresponding hyperparameters are automatically loaded from the associated dictionary, which is stored as a JSON file. The JSON file contains hyperparameter type information, names, and bounds. For `river` models, the hyperparameters are stored in the `RiverHyperDict`, see [SOURCE]

Alternatively, you can load a local hyper_dict. Simply set `river_hyper_dict.json` as the filename. If `filename`is set to `None`, which is the default, the hyper_dict [SOURCE] is loaded from the `spotriver` package.

How hyperparameter levels can be modified is described in Section 12.19.1.

Summary: Model Setup

The following parameters are used for the model setup:

```
from spotriver.fun.hyperriver import HyperRiver
from spotriver.hyperdict.river_hyper_dict import RiverHyperDict
core_model_name = "tree.HoeffdingTreeRegressor"
hyperdict = RiverHyperDict
```

38.2.6. Objective Function Setup

The loss function (metric) values are passed to the objective function `fun_oml_horizon` [SOURCE], which combines information about the loss, required memory and time as described in Section 38.2.4.

Summary: Objective Function Setup

The following parameters are used:

```
fun = HyperRiver().fun_oml_horizon
```

38.2.7. Surrogate Model Setup

The default surrogate model is the `Kriging` model, see [SOURCE]. We specify `noise` as `True` to include noise in the model. An `anisotropic` kernel is used, which allows different length scales for each dimension. Furthermore, the interval for the `Lambda` value is set to `[1e-3, 1e2]`.
These parameters are set in the `surrogate_control` dictionary and therefore passed to the `surrogate_control_init` function [SOURCE].

```
noise = True
min_Lambda = 1e-3
max_Lambda = 10
```

38.2.8. Summary: Setting up the Experiment

At this stage, all required information is available to set up the dictionaries for the hyperparameter tuning. Altogether, the `fun_control`, `design_control`, `surrogate_control`, and `optimize_control` dictionaries are initialized as follows:

```
from spotpython.utils.init import fun_control_init, design_control_init, surrogate_con

fun = HyperRiver().fun_oml_horizon

fun_control = fun_control_init(
    PREFIX="024",
    fun_evals=inf,
    max_time=1,

    prep_model_name="StandardScaler",
    test=test,
```

```
    train=train,
    target_column=target_column,

    metric_sklearn_name="mean_absolute_error",
    horizon=7*24,
    oml_grace_period=7*24,
    weight_coeff=0.0,
    weights=np.array([1, 0.01, 0.01]),

    core_model_name="tree.HoeffdingTreeRegressor",
    hyperdict=RiverHyperDict,
   )


design_control = design_control_init(
    init_size=10,
)

surrogate_control = surrogate_control_init(
    method="regression",
    min_Lambda=1e-3,
    max_Lambda=10,
)

optimizer_control = optimizer_control_init()
```

38.2.9. Run the `Spot` Optimizer

The class `Spot` [SOURCE] is the hyperparameter tuning workhorse. It is initialized with the following parameters, which were specified above.

- `fun`: the objective function
- `fun_control`: the dictionary with the control parameters for the objective function
- `design_control`: the dictionary with the control parameters for the experimental design
- `surrogate_control`: the dictionary with the control parameters for the surrogate model
- `optimizer_control`: the dictionary with the control parameters for the optimizer

`spotpython` allows maximum flexibility in the definition of the hyperparameter tuning setup. Alternative surrogate models, optimizers, and experimental designs can be used. Thus, interfaces for the `surrogate` model, experimental `design`, and `optimizer` are

provided. The default surrogate model is the kriging model, the default optimizer is the differential evolution, and default experimental design is the Latin hypercube design.

Summary: Spot Setup

The following parameters are used for the Spot setup. These were specified above:

```
fun = fun
fun_control = fun_control
design_control = design_control
surrogate_control = surrogate_control
optimizer_control = optimizer_control
```

```
from spotpython.spot import Spot
spot_tuner = Spot(
    fun=fun,
    fun_control=fun_control,
    design_control=design_control,
    surrogate_control=surrogate_control,
    optimizer_control=optimizer_control,
)
res = spot_tuner.run()
```

```
spotpython tuning: 3.1966689842131513 [----------] 0.65%. Success rate: 0.00%
Using spacefilling design as fallback.
spotpython tuning: 2.236365395621708 [----------] 4.59%. Success rate: 50.00%
Using spacefilling design as fallback.
spotpython tuning: 2.236365395621708 [#---------] 5.91%. Success rate: 33.33%
Using spacefilling design as fallback.
spotpython tuning: 2.236365395621708 [#---------] 6.53%. Success rate: 25.00%
Using spacefilling design as fallback.
spotpython tuning: 2.236365395621708 [#---------] 8.76%. Success rate: 20.00%
Using spacefilling design as fallback.
spotpython tuning: 2.236365395621708 [#---------] 9.46%. Success rate: 16.67%
Using spacefilling design as fallback.
spotpython tuning: 2.236365395621708 [#---------] 10.23%. Success rate: 14.29%
spotpython tuning: 2.236365395621708 [#---------] 14.47%. Success rate: 12.50%
Using spacefilling design as fallback.
spotpython tuning: 2.236365395621708 [##--------] 16.14%. Success rate: 11.11%
Using spacefilling design as fallback.
spotpython tuning: 2.236365395621708 [##--------] 17.30%. Success rate: 10.00%
Using spacefilling design as fallback.
spotpython tuning: 2.1653877176929237 [##--------] 18.26%. Success rate: 18.18%
```

```
Using spacefilling design as fallback.
spotpython tuning: 2.1653877176929237 [##--------] 24.05%. Success rate: 16.67%
Using spacefilling design as fallback.
spotpython tuning: 2.1653877176929237 [###-------] 29.47%. Success rate: 15.38%
Using spacefilling design as fallback.
spotpython tuning: 2.1653877176929237 [###-------] 31.32%. Success rate: 14.29%
Using spacefilling design as fallback.
spotpython tuning: 2.1653877176929237 [####------] 36.86%. Success rate: 13.33%
spotpython tuning: 2.1653877176929237 [####------] 38.61%. Success rate: 12.50%
spotpython tuning: 2.1653877176929237 [####------] 39.67%. Success rate: 11.76%
spotpython tuning: 2.1653877176929237 [####------] 41.32%. Success rate: 11.11%
spotpython tuning: 2.1653877176929237 [####------] 42.20%. Success rate: 10.53%
Using spacefilling design as fallback.
spotpython tuning: 2.1653877176929237 [####------] 43.34%. Success rate: 10.00%
spotpython tuning: 2.1653877176929237 [####------] 44.45%. Success rate: 9.52%
spotpython tuning: 2.1653877176929237 [#####-----] 45.39%. Success rate: 9.09%
spotpython tuning: 2.1653877176929237 [#####-----] 46.49%. Success rate: 8.70%
spotpython tuning: 2.1653877176929237 [#####-----] 47.57%. Success rate: 8.33%
spotpython tuning: 2.1653877176929237 [#####-----] 48.64%. Success rate: 8.00%
spotpython tuning: 2.1653877176929237 [#####-----] 49.60%. Success rate: 7.69%
spotpython tuning: 2.1653877176929237 [#####-----] 50.54%. Success rate: 7.41%
spotpython tuning: 2.1653877176929237 [#####-----] 51.50%. Success rate: 7.14%
spotpython tuning: 2.1653877176929237 [#####-----] 52.56%. Success rate: 6.90%
spotpython tuning: 2.1653877176929237 [#####-----] 53.68%. Success rate: 6.67%
spotpython tuning: 2.1653877176929237 [#####-----] 54.72%. Success rate: 6.45%
spotpython tuning: 2.1653877176929237 [######----] 55.71%. Success rate: 6.25%
spotpython tuning: 2.1653877176929237 [######----] 56.85%. Success rate: 6.06%
spotpython tuning: 2.1653877176929237 [######----] 57.99%. Success rate: 5.88%
spotpython tuning: 2.1653877176929237 [######----] 59.33%. Success rate: 5.71%
spotpython tuning: 2.1653877176929237 [######----] 60.89%. Success rate: 5.56%
Using spacefilling design as fallback.
spotpython tuning: 2.1653877176929237 [######----] 62.49%. Success rate: 5.41%
Using spacefilling design as fallback.
spotpython tuning: 2.0538448046390014 [######----] 63.75%. Success rate: 7.89%
Using spacefilling design as fallback.
spotpython tuning: 2.0538448046390014 [######----] 64.84%. Success rate: 7.69%
Using spacefilling design as fallback.
spotpython tuning: 2.0538448046390014 [#######---] 66.66%. Success rate: 7.50%
spotpython tuning: 2.0538448046390014 [#######---] 68.03%. Success rate: 7.32%
spotpython tuning: 2.0538448046390014 [#######---] 69.72%. Success rate: 7.14%
spotpython tuning: 2.0538448046390014 [#######---] 71.22%. Success rate: 6.98%
spotpython tuning: 2.0538448046390014 [########--] 77.11%. Success rate: 6.82%
spotpython tuning: 2.0538448046390014 [########--] 78.86%. Success rate: 6.67%
spotpython tuning: 2.0538448046390014 [#########-] 85.23%. Success rate: 6.52%
spotpython tuning: 2.0538448046390014 [#########-] 87.01%. Success rate: 6.38%
```

```
spotpython tuning: 2.0538448046390014 [#########-] 88.72%. Success rate: 6.25%
spotpython tuning: 2.0538448046390014 [#########-] 90.68%. Success rate: 6.12%
spotpython tuning: 2.0538448046390014 [#########-] 92.48%. Success rate: 6.00%
spotpython tuning: 2.0538448046390014 [#########-] 94.24%. Success rate: 5.88%
spotpython tuning: 2.0538448046390014 [##########] 95.92%. Success rate: 5.77%
spotpython tuning: 2.0538448046390014 [##########] 97.66%. Success rate: 5.66%
spotpython tuning: 2.0538448046390014 [##########] 99.39%. Success rate: 5.56%
Using spacefilling design as fallback.
spotpython tuning: 2.0538448046390014 [##########] 100.00%. Success rate: 5.45% Done.

Experiment saved to 024_res.pkl
```

38.3. Using the spotgui

The spotgui [github] provides a convenient way to interact with the hyperparameter tuning process. To obtain the settings from Section 38.2.8, the spotgui can be started as shown in Figure 45.1.

Figure 38.1.: spotgui

38.4. Results

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress(log_y=True, filename=None)
```

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

```
| name                    | type    | default           |    lower |    upper | tuned                  | t
|-------------------------|---------|-------------------|----------|----------
|-------------------------|-------------------------|--------------
|---------|
| grace_period            | int     | 200               |     10.0 |   1000.0 | 220.0                  | N
| max_depth               | int     | 20                |      2.0 |     20.0 | 3.0                    | t
| delta                   | float   | 1e-07             |    1e-08 |    1e-
06 | 1.494407810763274e-07 | None                     |         1.59 | *        |
| tau                     | float   | 0.05              |     0.01 |      0.1 | 0.013189745147643657   | N
| leaf_prediction         | factor  | mean              |      0.0 |      2.0 | adaptive               | N
| leaf_model              | factor  | LinearRegression  |      0.0 |      2.0 | LinearRegression       | N
| model_selector_decay    | float   | 0.95              |      0.9 |     0.99 | 0.9874736389042145     | N
| splitter                | factor  | EBSTSplitter      |      0.0 |      2.0 | QOSplitter             | N
| min_samples_split       | int     | 5                 |      2.0 |     10.0 | 5.0                    | N
| binary_split            | factor  | 0                 |      0.0 |      1.0 | 0                      | N
| max_size                | float   | 500.0             |    100.0 |   1000.0 | 224.48451082605902     | N
```

```
| memory_estimate_period | int    | 6                |     3.0 |     8.0 | 4.0
| stop_mem_management    | factor | 0                |     0.0 |     1.0 | 1
| remove_poor_attrs      | factor | 0                |     0.0 |     1.0 | 1
| merit_preprune         | factor | 1                |     0.0 |     1.0 | 1
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=10.0)
```

38.5. Performance of the Model with Default Hyperparameters

38.5.1. Get Default Hyperparameters and Fit the Model

The default hyperparameters, which will be used for a comparion with the tuned hyperparameters, can be obtained with the following commands:

```
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
```

`spotpython` tunes numpy arrays, i.e., the hyperparameters are stored in a numpy array.

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
model_default = get_one_core_model_from_X(X_start, fun_control, default=True)
```

38.5.2. Evaluate the Model with Default Hyperparameters

The model with the default hyperparameters can be trained and evaluated. The evaluation function `eval_oml_horizon` [SOURCE] is the same function that was used for the hyperparameter tuning. During the hyperparameter tuning, the evaluation function was called from the objective (or loss) function `fun_oml_horizon` [SOURCE].

```
from spotriver.evaluation.eval_bml import eval_oml_horizon

df_eval_default, df_true_default = eval_oml_horizon(
                    model=model_default,
                    train=fun_control["train"],
                    test=fun_control["test"],
                    target_column=fun_control["target_column"],
                    horizon=fun_control["horizon"],
                    oml_grace_period=fun_control["oml_grace_period"],
                    metric=fun_control["metric_sklearn"],
                )
```

The three performance criteria, i.e., score (metric), runtime, and memory consumption, can be visualized with the following commands:

```
from spotriver.evaluation.eval_bml import plot_bml_oml_horizon_metrics, plot_bml_oml_horizon_predict
df_labels=["default"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default], log_y=False, df_labels=df_labels, metric=f
```

38.5.3. Show Predictions of the Model with Default Hyperparameters

- Select a subset of the data set for the visualization of the predictions:
  - We use the mean, m, of the data set as the center of the visualization.
  - We use 100 data points, i.e., $m \pm 50$ as the visualization window.

```
m = fun_control["test"].shape[0]
a = int(m/2)-50
b = int(m/2)+50
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b]], target_column=targe
```

38.6. Get SPOT Results

In a similar way, we can obtain the hyperparameters found by `spotpython`.

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
model_spot = get_one_core_model_from_X(X, fun_control)
```

```
df_eval_spot, df_true_spot = eval_oml_horizon(
                    model=model_spot,
                    train=fun_control["train"],
                    test=fun_control["test"],
                    target_column=fun_control["target_column"],
                    horizon=fun_control["horizon"],
                    oml_grace_period=fun_control["oml_grace_period"],
                    metric=fun_control["metric_sklearn"],
                )
```

```
df_labels=["default", "spot"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default, df_eval_spot], log_y=False, df_labels=df_la
```

```
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b], df_true_spot[a:b]],
```

```
from spotpython.plot.validation import plot_actual_vs_predicted
plot_actual_vs_predicted(y_test=df_true_default[target_column], y_pred=df_true_default
plot_actual_vs_predicted(y_test=df_true_spot[target_column], y_pred=df_true_spot["Pred
```

38.7. Visualize Regression Trees

```
dataset_f = dataset.take(n_samples)
print(f"n_samples: {n_samples}")
for x, y in dataset_f:
    model_default.learn_one(x, y)
```

```
n_samples: 10000
```

> **Caution: Large Trees**
>
> - Since the trees are large, the visualization is suppressed by default.
> - To visualize the trees, uncomment the following line.

```
# model_default.draw()
```

```
model_default.summary
```

```
{'n_nodes': 23,
 'n_branches': 11,
 'n_leaves': 12,
 'n_active_leaves': 12,
 'n_inactive_leaves': 0,
 'height': 7,
 'total_observed_weight': 14168.0}
```

38.7.1. Spot Model

```
print(f"n_samples: {n_samples}")
dataset_f = dataset.take(n_samples)
for x, y in dataset_f:
    model_spot.learn_one(x, y)
```

```
n_samples: 10000
```

> **Caution: Large Trees**
>
> - Since the trees are large, the visualization is suppressed by default.
> - To visualize the trees, uncomment the following line.

```
# model_spot.draw()
```

```
model_spot.summary
```

```
{'n_nodes': 27,
 'n_branches': 13,
 'n_leaves': 14,
 'n_active_leaves': 14,
 'n_inactive_leaves': 0,
 'height': 7,
 'total_observed_weight': 14168.0}
```

```
from spotpython.utils.eda import compare_two_tree_models
print(compare_two_tree_models(model_default, model_spot))
```

```
| Parameter              |   Default |   Spot |
|------------------------|-----------|--------|
| n_nodes                |        23 |     27 |
| n_branches             |        11 |     13 |
| n_leaves               |        12 |     14 |
| n_active_leaves        |        12 |     14 |
| n_inactive_leaves      |         0 |      0 |
| height                 |         7 |      7 |
| total_observed_weight |     14168 |  14168 |
```

38.8. Detailed Hyperparameter Plots

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
grace_period:  0.0010394208169376752
max_depth:  0.0010394208169376752
delta:  1.5907749782919138
tau:  0.017556458095247054
leaf_prediction:  0.0010587809450999033
leaf_model:  0.0014842189154865692
model_selector_decay:  0.03711035559987861
splitter:  0.03308540800698709
min_samples_split:  2.150500495722597
binary_split:  44.562251731663444
max_size:  0.16946859440714768
memory_estimate_period:  0.003876793359050844
stop_mem_management:  1.0347325623500703
remove_poor_attrs:  100.0
merit_preprune:  30.652890363099104
```

Prediction Surface
Prediction
remove_poor_attrs
merit_preprune
Prediction Error Surface
Error
remove_poor_attrs
merit_preprune
Prediction Contour
1e−13+4.523906224
merit_preprune
remove_poor_attrs
Error Contour
1e−13+2.660256877
merit_preprune
remove_poor_attrs

Prediction Surface
Prediction
binary_split
remove_poor_attrs
Prediction Error Surface
binary_split
remove_poor_attrs
Prediction Contour
remove_poor_attrs
binary_split
Error Contour
1e−5+2.6602
remove_poor_attrs
binary_split

38.9. Parallel Coordinates Plots

```
spot_tuner.parallel_plot()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

39. The Friedman Drift Data Set

This chapter demonstrates hyperparameter tuning for `river`'s `Mondrian Tree Regressor` [SOURCE] with the Friedman drift data set [SOURCE]. The `Mondrian Tree Regressor` is a regression tree, i.e., it predicts a real value for each sample.

The data set was introduced in Section 38.1.

```
from river.datasets import synth
import pandas as pd
import numpy as np
from spotriver.utils.data_conversion import convert_to_df
from spotpython.utils.eda import print_exp_table, print_res_table

n_train = 6_000
n_test = 4_000
n_samples = n_train + n_test
target_column = "y"

dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=(n_train/4, n_train/2),
   seed=123
)

train = convert_to_df(dataset, n_total=n_train)
train.columns = [f"x{i}" for i in range(1, 11)] + [target_column]

dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=(n_test/4, n_test/2),
   seed=123
)
test = convert_to_df(dataset, n_total=n_test)
test.columns = [f"x{i}" for i in range(1, 11)] + [target_column]
```

39.1. Setup

We will use a general experiment, data, evaluation, river-specific, objective-function, and surrogate setup similar to the setup from Section 38.2. Only the model setup differs from the setup in Section 38.2. Here we use the `Mondrian Tree Regressor` from `river`.

```
from spotriver.hyperdict.river_hyper_dict import RiverHyperDict
core_model_name = "forest.AMFRegressor"
hyperdict = RiverHyperDict
hyperdict
```

```
spotriver.hyperdict.river_hyper_dict.RiverHyperDict
```

39.1.1. Select a User Hyperdictionary

Alternatively, you can load a local hyper_dict from the "userModel" folder. Here, we have selected a copy of the JSON `MondrianHyperDict` hyperdictionary from [SOURCE] and the `MondrianHyperDict` class from [SOURCE]. The hyperparameters of the `Mondrian Tree Regressor` are defined in the `MondrianHyperDict` class, i.e., there is an key "AMFRegressor" in the `hyperdict` "mondrian_hyper_dict.json" file.

```
import sys
sys.path.insert(0, './userModel')
import mondrian_hyper_dict
hyperdict = mondrian_hyper_dict.MondrianHyperDict
hyperdict
```

```
mondrian_hyper_dict.MondrianHyperDict
```

```
from spotpython.utils.init import fun_control_init, design_control_init, surrogate_co
from spotriver.fun.hyperriver import HyperRiver

fun = HyperRiver().fun_oml_horizon

fun_control = fun_control_init(
    PREFIX="503",
    fun_evals=inf,
    max_time=1,
```

```
    prep_model_name="StandardScaler",
    test=test,
    train=train,
    target_column=target_column,

    metric_sklearn_name="mean_absolute_error",
    horizon=7*24,
    oml_grace_period=7*24,
    weight_coeff=0.0,
    weights=np.array([1, 0.01, 0.01]),

    core_model_name="forest.AMFRegressor",
    hyperdict=hyperdict,
   )


design_control = design_control_init(
    init_size=5,
)

surrogate_control = surrogate_control_init(
    method="regression",
    min_Lambda=1e-3,
    max_Lambda=10,
)

optimizer_control = optimizer_control_init()
```

39.2. Modify hyper_dict Hyperparameters for the Selected Algorithm aka core_model

After the core_model and the hyperdict are added to the fun_control dictionary, the hyperparameter tuning can be started. However, in some settings, the user wants to modify the hyperparameters. This can be done with the set_int_hyperparameter_values, set_float_hyperparameter_values, set_boolean_hyperparameter_values, and set_factor_hyperparameter_values functions, which can be imported from from spotpython.hyperparameters.values [SOURCE].

The following code shows how hyperparameter of type float and integer can be modified. Additional examples can be found in Section 12.19.1.

```
print_exp_table(fun_control)
```

```
| name            | type   |   default |   lower |   upper | transform             |
|-----------------|--------|-----------|---------|---------|-----------------------
|
| n_estimators    | int    |         3 |     2   |      10 | transform_power_2_int |
| step            | float  |         1 |     0.1 |      10 | None                  |
| use_aggregation | factor |         1 |     0   |       1 | None                  |
```

```
from spotpython.hyperparameters.values import set_int_hyperparameter_values, set_float
set_int_hyperparameter_values(fun_control, "n_estimators", 2, 7)
set_float_hyperparameter_values(fun_control, "step", 0.1, 15)
print_exp_table(fun_control)
```

```
Setting hyperparameter n_estimators to value [2, 7].
Variable type is int.
Core type is None.
Calling modify_hyper_parameter_bounds().
Setting hyperparameter step to value [0.1, 15].
Variable type is float.
Core type is None.
Calling modify_hyper_parameter_bounds().
| name            | type   |   default |   lower |   upper | transform             |
|-----------------|--------|-----------|---------|---------|-----------------------
|
| n_estimators    | int    |         3 |     2   |       7 | transform_power_2_int |
| step            | float  |         1 |     0.1 |      15 | None                  |
| use_aggregation | factor |         1 |     0   |       1 | None                  |
```

> **Note: Active and Inactive Hyperparameters**
>
> Hyperparameters can be excluded from the tuning procedure by selecting identical values for the lower and upper bounds.

39.2.1. Run the Spot Optimizer

```
from spotpython.spot import Spot
spot_tuner = Spot(
    fun=fun,
    fun_control=fun_control,
```

```
    design_control=design_control,
    surrogate_control=surrogate_control,
    optimizer_control=optimizer_control,
)
res = spot_tuner.run()
```

```
spotpython tuning: 2.788608300128256 [####------] 39.51%. Success rate: 100.00%
spotpython tuning: 2.788608300128256 [########--] 81.51%. Success rate: 50.00%
spotpython tuning: 2.788608300128256 [##########] 100.00%. Success rate: 33.33% Done...

Experiment saved to 503_res.pkl
```

We can start TensorBoard in the background with the following command, where ./runs is the default directory for the TensorBoard log files:

`tensorboard --logdir="./runs"`We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

39.3. Results

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress()
```

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

```
| name            | type    |   default |   lower |   upper |              tuned | trans
|-----------------|---------|-----------|---------|---------|--------------------
|-----------------------|--------------|---------|
| n_estimators    | int     |       3.0 |     2.0 |       7 |                5.0 | trans
| step            | float   |       1.0 |     0.1 |      15 | 4.884595795250249 | None
| use_aggregation | factor  |       1.0 |     0.0 |       1 |                1.0 | None
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=10.0)
```

39.4. Performance of the Model with Default Hyperparameters

39.4.1. Get Default Hyperparameters and Fit the Model

The default hyperparameters, which will be used for a comparion with the tuned hyperparameters, can be obtained with the following commands:

```
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
```

`spotpython` tunes numpy arrays, i.e., the hyperparameters are stored in a numpy array.

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
model_default = get_one_core_model_from_X(X_start, fun_control, default=True)
```

39.4.2. Evaluate the Model with Default Hyperparameters

The model with the default hyperparameters can be trained and evaluated. The evaluation function `eval_oml_horizon` [SOURCE] is the same function that was used for the hyperparameter tuning. During the hyperparameter tuning, the evaluation function was called from the objective (or loss) function `fun_oml_horizon` [SOURCE].

```
from spotriver.evaluation.eval_bml import eval_oml_horizon

df_eval_default, df_true_default = eval_oml_horizon(
                    model=model_default,
                    train=fun_control["train"],
                    test=fun_control["test"],
                    target_column=fun_control["target_column"],
                    horizon=fun_control["horizon"],
                    oml_grace_period=fun_control["oml_grace_period"],
                    metric=fun_control["metric_sklearn"],
                )
```

The three performance criteria, i.e., score (metric), runtime, and memory consumption, can be visualized with the following commands:

```
from spotriver.evaluation.eval_bml import plot_bml_oml_horizon_metrics, plot_bml_oml_horizon_predict
df_labels=["default"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default], log_y=False, df_labels=df_labels, metric=f
```

39.4.3. Show Predictions of the Model with Default Hyperparameters

- Select a subset of the data set for the visualization of the predictions:
  - We use the mean, m, of the data set as the center of the visualization.
  - We use 100 data points, i.e., $m \pm 50$ as the visualization window.

```
m = fun_control["test"].shape[0]
a = int(m/2)-50
b = int(m/2)+50
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b]], target_column=targe
```

39.5. Get SPOT Results

In a similar way, we can obtain the hyperparameters found by `spotpython`.

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
model_spot = get_one_core_model_from_X(X, fun_control)
```

```
df_eval_spot, df_true_spot = eval_oml_horizon(
                    model=model_spot,
                    train=fun_control["train"],
                    test=fun_control["test"],
                    target_column=fun_control["target_column"],
                    horizon=fun_control["horizon"],
                    oml_grace_period=fun_control["oml_grace_period"],
                    metric=fun_control["metric_sklearn"],
                )
```

```
df_labels=["default", "spot"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default, df_eval_spot], log_y=False, df_labels=df_la
```

```
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b], df_true_spot[a:b]],
```

```
from spotpython.plot.validation import plot_actual_vs_predicted
plot_actual_vs_predicted(y_test=df_true_default[target_column], y_pred=df_true_default
plot_actual_vs_predicted(y_test=df_true_spot[target_column], y_pred=df_true_spot["Pred
```

39.6. Detailed Hyperparameter Plots

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
n_estimators:  100.0
step:  2.424860713167851
use_aggregation:  3.9927534277372794
```

Prediction Surface
Prediction
n_estimators
use_aggregation
Prediction Error Surface
Error
n_estimators
use_aggregation
Prediction Contour
use_aggregation
n_estimators
Error Contour
use_aggregation
n_estimators

Prediction Surface
Prediction
n_estimators
step
Prediction Error Surface
Error
n_estimators
step
Prediction Contour
step
n_estimators
Error Contour
step
n_estimators

39.7. Parallel Coordinates Plots

```
spot_tuner.parallel_plot()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

Part IX.

Hyperparameter Tuning with PyTorch Lightning

40. Basic Lightning Module

40.1. Introduction

This chapter implements a basic Pytorch Lightning module. It is based on the Lightning documentation LIGHTNINGMODULE.

A `LightningModule` organizes your `PyTorch` code into six sections:

- Initialization (`__init__` and `setup()`).
- Train Loop (`training_step()`)
- Validation Loop (`validation_step()`)
- Test Loop (`test_step()`)
- Prediction Loop (`predict_step()`)
- Optimizers and LR Schedulers (`configure_optimizers()`)

The `Trainer` automates every required step in a clear and reproducible way. It is the most important part of PyTorch Lightning. It is responsible for training, testing, and validating the model. The `Lightning` core structure looks like this:

```
net = MyLightningModuleNet()
trainer = Trainer()
trainer.fit(net)
```

There are no `.cuda()` or `.to(device)` calls required. Lightning does these for you.

```
# don't do in Lightning
x = torch.Tensor(2, 3)
x = x.cuda()
x = x.to(device)

# do this instead
x = x  # leave it alone!

# or to init a new tensor
new_x = torch.Tensor(2, 3)
new_x = new_x.to(x)
```

A LightningModule is a `torch.nn.Module` but with added functionality. For example:

```
net = Net.load_from_checkpoint(PATH)
net.freeze()
out = net(x)
```

40.2. Starter Example: Transformer

Here are the only required methods for setting up a transfomer model:

```
import lightning as L
import torch

from lightning.pytorch.demos import Transformer


class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)

    def forward(self, inputs, target):
        return self.model(inputs, target)

    def training_step(self, batch, batch_idx):
        inputs, target = batch
        output = self(inputs, target)
        loss = torch.nn.functional.nll_loss(output, target.view(-1))
        return loss

    def configure_optimizers(self):
        return torch.optim.SGD(self.model.parameters(), lr=0.1)
```

The `LightningTransformer` class is a subclass of `LightningModule`. It can be trainted as follows:

```
from lightning.pytorch.demos import WikiText2
from torch.utils.data import DataLoader

dataset = WikiText2()
dataloader = DataLoader(dataset)
model = LightningTransformer(vocab_size=dataset.vocab_size)
```

```
trainer = L.Trainer(fast_dev_run=100)
trainer.fit(model=model, train_dataloaders=dataloader)
```

```
    Name    Type          Params   Mode    FLOPs

 0  model   Transformer   14.6 M   train       0


Trainable params: 14.6 M
Non-trainable params: 0
Total params: 14.6 M
Total estimated model params size (MB): 58
Modules in train mode: 60
Modules in eval mode: 0
Total FLOPs: 0


Output()
```

40.3. Lightning Core Methods

The LightningModule has many convenient methods, but the core ones you need to know about are shown in Table 40.1.

Table 40.1.: The core methods of a LightningModule

Method	Description
`__init__` and `setup`	Initializes the model.
`forward`	Performs a forward pass through the model. To run data through your model only (separate from `training_step`).
`training_step`	Performs a complete training step.
`validation_step`	Performs a complete validation step.
`test_step`	Performs a complete test step.
`predict_step`	Performs a complete prediction step.

Method	Description
`configure_optimizers`	Configures the optimizers and learning-rate schedulers.

We will take a closer look at thes methods.

40.3.1. Training Step

40.3.1.1. Basics

To activate the training loop, override the `training_step()` method. If you want to calculate epoch-level metrics and log them, use `log()`.

```
class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)

    def training_step(self, batch, batch_idx):
        inputs, target = batch
        output = self.model(inputs, target)
        loss = torch.nn.functional.nll_loss(output, target.view(-1))

        # logs metrics for each training_step,
        # and the average across the epoch, to the progress bar and logger
        self.log("train_loss", loss, on_step=True, on_epoch=True, prog_bar=True, logge
        return loss
```

The `log()` method automatically reduces the requested metrics across a complete epoch and devices.

40.3.1.2. Background

- Here is the pseudocode of what the `log()` method does under the hood:

```
outs = []
for batch_idx, batch in enumerate(train_dataloader):
    # forward
    loss = training_step(batch, batch_idx)
    outs.append(loss.detach())
```

```
    # clear gradients
    optimizer.zero_grad()
    # backward
    loss.backward()
    # update parameters
    optimizer.step()

# note: in reality, we do this incrementally, instead of keeping all outputs in memory
epoch_metric = torch.mean(torch.stack(outs))
```

- In the case that you need to make use of all the outputs from each `training_step()`, override the `on_train_epoch_end()` method.

```
class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)
        self.training_step_outputs = []

    def training_step(self, batch, batch_idx):
        inputs, target = batch
        output = self.model(inputs, target)
        loss = torch.nn.functional.nll_loss(output, target.view(-1))
        preds = ...
        self.training_step_outputs.append(preds)
        return loss

    def on_train_epoch_end(self):
        all_preds = torch.stack(self.training_step_outputs)
        # do something with all preds
        ...
        self.training_step_outputs.clear()  # free memory
```

40.3.2. Validation Step

40.3.2.1. Basics

To activate the validation loop while training, override the `validation_step()` method.

```
class LightningTransformer(L.LightningModule):
    def validation_step(self, batch, batch_idx):
        inputs, target = batch
```

```
        output = self.model(inputs, target)
        loss = F.cross_entropy(y_hat, y)
        self.log("val_loss", loss)
        return loss
```

40.3.2.2. Background

- You can also run just the validation loop on your validation dataloaders by overriding `validation_step()` and calling `validate()`.

```
model = LightningTransformer(vocab_size=dataset.vocab_size)
trainer = L.Trainer()
trainer.validate(model)
```

- In the case that you need to make use of all the outputs from each `validation_step()`, override the `on_validation_epoch_end()` method. Note that this method is called before `on_train_epoch_end()`.

```
class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)
        self.validation_step_outputs = []

    def validation_step(self, batch, batch_idx):
        x, y = batch
        inputs, target = batch
        output = self.model(inputs, target)
        loss = torch.nn.functional.nll_loss(output, target.view(-1))
        pred = ...
        self.validation_step_outputs.append(pred)
        return pred

    def on_validation_epoch_end(self):
        all_preds = torch.stack(self.validation_step_outputs)
        # do something with all preds
        ...
        self.validation_step_outputs.clear()  # free memory
```

40.3.3. Test Step

The process for enabling a test loop is the same as the process for enabling a validation loop. For this you need to override the `test_step()` method. The only difference is

that the test loop is only called when `test()` is used.

```
def test_step(self, batch, batch_idx):
    inputs, target = batch
    output = self.model(inputs, target)
    loss = F.cross_entropy(y_hat, y)
    self.log("test_loss", loss)
    return loss
```

40.3.4. Predict Step

40.3.4.1. Basics

By default, the `predict_step()` method runs the `forward()` method. In order to customize this behaviour, simply override the `predict_step()` method.

```
class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)

    def predict_step(self, batch):
        inputs, target = batch
        return self.model(inputs, target)
```

40.3.4.2. Background

- If you want to perform inference with the system, you can add a `forward` method to the LightningModule.
- When using forward, you are responsible to call `eval()` and use the `no_grad()` context manager.

```
class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)

    def forward(self, batch):
        inputs, target = batch
        return self.model(inputs, target)

    def training_step(self, batch, batch_idx):
```

```
        inputs, target = batch
        output = self.model(inputs, target)
        loss = torch.nn.functional.nll_loss(output, target.view(-1))
        return loss

    def configure_optimizers(self):
        return torch.optim.SGD(self.model.parameters(), lr=0.1)


model = LightningTransformer(vocab_size=dataset.vocab_size)

model.eval()
with torch.no_grad():
    batch = dataloader.dataset[0]
    pred = model(batch)
```

40.4. Lightning Extras

This section covers some additional features of Lightning.

40.4.1. Lightning: Save Hyperparameters

Often times we train many versions of a model. You might share that model or come back to it a few months later at which point it is very useful to know how that model was trained (i.e.: what learning rate, neural network, etc.).

Lightning has a standardized way of saving the information for you in checkpoints and YAML files. The goal here is to improve readability and reproducibility.

Use `save_hyperparameters()` within your `LightningModule`'s `__init__` method. It will enable Lightning to store all the provided arguments under the `self.hparams` attribute. These hyperparameters will also be stored within the model checkpoint, which simplifies model re-instantiation after training.

```
class LitMNIST(L.LightningModule):
    def __init__(self, layer_1_dim=128, learning_rate=1e-2):
        super().__init__()
        # call this to save (layer_1_dim=128, learning_rate=1e-4) to the checkpoint
        self.save_hyperparameters()

        # equivalent
        self.save_hyperparameters("layer_1_dim", "learning_rate")
```

```
        # Now possible to access layer_1_dim from hparams
        self.hparams.layer_1_dim
```

40.4.2. Lightning: Model Loading

LightningModules that have hyperparameters automatically saved with `save_hyperparameters()` can conveniently be loaded and instantiated directly from a checkpoint with `load_from_checkpoint()`:

```
# to load specify the other args
model = LitMNIST.load_from_checkpoint(PATH, loss_fx=torch.nn.SomeOtherLoss, generator_network=MyGene
```

40.5. Starter Example: Linear Neural Network

We will use the `LightningModule` to create a simple neural network for regression. It will be implemented as the `LightningBasic` class.

40.5.1. Hidden Layers

To specify the number of hidden layers, we will use the hyperparameter `l1` and the function `get_hidden_sizes()` [DOC] from the `spotpython` package.

```
from spotpython.hyperparameters.architecture import get_hidden_sizes
_L_in = 10
l1 = 20
max_n = 4
get_hidden_sizes(_L_in, l1, max_n)
```

```
[20, 10, 10, 5]
```

40.5.2. Hyperparameters

The argument `l1` will be treated as a hyperparameter, so it will be tuned in the following steps. Besides `l1`, additonal hyperparameters are `act_fn` and `dropout_prob`.

The arguments `_L_in`, `_L_out`, and `_torchmetric` are not hyperparameters, but are needed to create the network. The first two are specified by the data and the latter by user preferences (the desired evaluation metric).

40.5.3. The LightningBasic Class

```
import lightning as L
import torch
import torch.nn.functional as F
import torchmetrics.functional.regression
from torch import nn
from spotpython.hyperparameters.architecture import get_hidden_sizes

class LightningBasic(L.LightningModule):
    def __init__(
    self,
    l1: int,
    act_fn: nn.Module,
    dropout_prob: float,
    _L_in: int,
    _L_out: int,
    _torchmetric: str,
    *args,
    **kwargs):
        super().__init__()
        self._L_in = _L_in
        self._L_out = _L_out
        self._torchmetric = _torchmetric
        self.metric = getattr(torchmetrics.functional.regression, _torchmetric)
        # _L_in and _L_out are not hyperparameters, but are needed to create the netwo
        # _torchmetric is not a hyperparameter, but is needed to calculate the loss
        self.save_hyperparameters(ignore=["_L_in", "_L_out", "_torchmetric"])
        # set dummy input array for Tensorboard Graphs
        # set log_graph=True in Trainer to see the graph (in traintest.py)
        hidden_sizes = get_hidden_sizes(_L_in=self._L_in, l1=l1, max_n=4)
        # Create the network based on the specified hidden sizes
        layers = []
        layer_sizes = [self._L_in] + hidden_sizes
        layer_size_last = layer_sizes[0]
        for layer_size in layer_sizes[1:]:
            layers += [
                nn.Linear(layer_size_last, layer_size),
                self.hparams.act_fn,
                nn.Dropout(self.hparams.dropout_prob),
            ]
            layer_size_last = layer_size
        layers += [nn.Linear(layer_sizes[-1], self._L_out)]
        # nn.Sequential summarizes a list of modules into a single module,
```

```
        # applying them in sequence
        self.layers = nn.Sequential(*layers)

    def _calculate_loss(self, batch):
        x, y = batch
        y = y.view(len(y), 1)
        y_hat = self.layers(x)
        loss = self.metric(y_hat, y)
        return loss

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.layers(x)

    def training_step(self, batch: tuple) -> torch.Tensor:
        loss = self._calculate_loss(batch)
        self.log("train_loss", loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)
        return loss

    def validation_step(self, batch: tuple) -> torch.Tensor:
        loss = self._calculate_loss(batch)
        # logs metrics for each training_step,
        # and the average across the epoch, to the progress bar and logger
        self.log("val_loss", loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)
        return loss

    def test_step(self, batch, batch_idx):
        loss = self._calculate_loss(batch)
        # logs metrics for each training_step,
        # and the average across the epoch, to the progress bar and logger
        self.log("test_loss", loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)
        return loss

    def predict_step(self, batch, batch_idx, dataloader_idx=0):
        x, _ = batch
        y_hat = self.layers(x)
        return y_hat

    def configure_optimizers(self):
        return torch.optim.Adam(self.layers.parameters(), lr=0.02)
```

We can instantiate the `LightningBasic` class as follows:

```
model_base = LightningBasic(
    l1=20,
    act_fn=nn.ReLU(),
    dropout_prob=0.01,
    _L_in=10,
    _L_out=1,
    _torchmetric="mean_squared_error")
```

It has the following structure:

```
print(model_base)
```

```
LightningBasic(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=20, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.01, inplace=False)
    (3): Linear(in_features=20, out_features=10, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.01, inplace=False)
    (6): Linear(in_features=10, out_features=10, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.01, inplace=False)
    (9): Linear(in_features=10, out_features=5, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.01, inplace=False)
    (12): Linear(in_features=5, out_features=1, bias=True)
  )
)
```

```
from spotpython.plot.xai import viz_net
viz_net(net=model_base,
    device="cpu",
    filename="model_architecture700", format="png")
```

Figure 40.1.: Model architecture

40.5.4. The Data Set: Diabetes

We will use the `Diabetes` [DOC] data set from the `spotpython` package, which is a PyTorch Dataset for regression based on a data set from `scikit-learn`. It consists of DataFrame entries, which were converted to PyTorch tensors.

Ten baseline variables, age, sex, body mass index, average blood pressure, and six blood serum measurements were obtained for each of n = 442 diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline.

The `Diabetes` data set has the following properties:

- Number of Instances: 442
- Number of Attributes: First 10 columns are numeric predictive values.
- Target: Column 11 is a quantitative measure of disease progression one year after baseline.
- Attribute Information:
  - age age in years
  - sex
  - bmi body mass index
  - bp average blood pressure
  - s1 tc, total serum cholesterol
  - s2 ldl, low-density lipoproteins
  - s3 hdl, high-density lipoproteins
  - s4 tch, total cholesterol / HDL
  - s5 ltg, possibly log of serum triglycerides level
  - s6 glu, blood sugar level

```
from torch.utils.data import DataLoader
from spotpython.data.diabetes import Diabetes
import torch
dataset = Diabetes(feature_type=torch.float32, target_type=torch.float32)
# Set batch size for DataLoader to 2 for demonstration purposes
batch_size = 2
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=False)
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Batch Size: 2
```

```
---------------
Inputs: tensor([[ 0.0381,  0.0507,  0.0617,  0.0219, -0.0442, -0.0348, -
0.0434, -0.0026,
          0.0199, -0.0176],
        [-0.0019, -0.0446, -0.0515, -0.0263, -0.0084, -0.0192,  0.0744, -
0.0395,
         -0.0683, -0.0922]])
Targets: tensor([151.,  75.])
```

40.5.5. The DataLoaders

Before we can call the `Trainer` to fit, validate, and test the model, we need to create the `DataLoaders` for each of these steps. The `DataLoaders` are used to load the data into the model in batches and need the `batch_size`.

```
import torch
from spotpython.data.diabetes import Diabetes
from torch.utils.data import DataLoader

batch_size = 8

dataset = Diabetes(target_type=torch.float)
train1_set, test_set = torch.utils.data.random_split(dataset, [0.6, 0.4])
train_set, val_set = torch.utils.data.random_split(train1_set, [0.6, 0.4])
print(f"Full Data Set: {len(dataset)}")
print(f"Train Set: {len(train_set)}")
print(f"Validation Set: {len(val_set)}")
print(f"Test Set: {len(test_set)}")
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True, drop_last=True, pin_memory
test_loader = DataLoader(test_set, batch_size=batch_size)
val_loader = DataLoader(val_set, batch_size=batch_size)
```

```
Full Data Set: 442
Train Set: 160
Validation Set: 106
Test Set: 176
```

40.5.6. The Trainer

Now we are ready to train the model. We will use the `Trainer` class from the `lightning` package. For demonstration purposes, we will train the model for 100 epochs only.

```
epochs = 100

trainer = L.Trainer(max_epochs=epochs, enable_progress_bar=True)
trainer.fit(model=model_base, train_dataloaders=train_loader)
```

```
     Name     Type         Params   Mode    FLOPs

  0  layers   Sequential      601   train       0


Trainable params: 601
Non-trainable params: 0
Total params: 601
Total estimated model params size (MB): 0
Modules in train mode: 11
Modules in eval mode: 0
Total FLOPs: 0
```

Output()

```
trainer.validate(model_base, val_loader)
```

```
# automatically loads the best weights for you
out = trainer.test(model_base, test_loader, verbose=True)
```

Output()

```
         Test metric                DataLoader 0

       test_loss_epoch            3013.781005859375
```

```
yhat = trainer.predict(model_base, test_loader)
# convert the list of tensors to a numpy array
yhat = torch.cat(yhat).numpy()
yhat.shape
```

40.5.7. Using a DataModule

Instead of creating the three `DataLoaders` manually, we can use the `LightDataModule` class from the `spotpython` package.

```
from spotpython.data.lightdatamodule import LightDataModule
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
```

There is a minor difference in the sizes of the data sets due to the random split as can be seen in the following code:

```
print(f"Full Data Set: {len(dataset)}")
print(f"Training set size: {len(data_module.data_train)}")
print(f"Validation set size: {len(data_module.data_val)}")
print(f"Test set size: {len(data_module.data_test)}")
```

```
Full Data Set: 442
Training set size: 160
Validation set size: 106
Test set size: 177
```

The `DataModule` can be used to train the model as follows:

```
trainer = L.Trainer(max_epochs=epochs, enable_progress_bar=False)
trainer.fit(model=model_base, datamodule=data_module)
```

```
     Name     Type         Params   Mode    FLOPs

  0  layers   Sequential      601   train       0
```

```
Trainable params: 601
Non-trainable params: 0
Total params: 601
Total estimated model params size (MB): 0
Modules in train mode: 11
Modules in eval mode: 0
Total FLOPs: 0
```

```
trainer.validate(model=model_base, datamodule=data_module, verbose=True, ckpt_path=Nor
```

```
       Validate metric              DataLoader 0

       val_loss_epoch             3244.82373046875
```

```
[{'val_loss_epoch': 3244.82373046875}]
```

```
trainer.test(model=model_base, datamodule=data_module, verbose=True, ckpt_path=None)
```

```
         Test metric                DataLoader 0

       test_loss_epoch               2346.109375
```

```
[{'test_loss_epoch': 2346.109375}]
```

40.6. Using spotpython with Pytorch Lightning

```
import os
from math import inf
import warnings
warnings.filterwarnings("ignore")
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
```

```
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_control_init)
from spotpython.utils.eda import print_exp_table, print_res_table
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename
```

```
PREFIX="700"
data_set = Diabetes()
fun_control = fun_control_init(
    PREFIX=PREFIX,
    save_experiment=True,
    fun_evals=inf,
    fun_repeats=2,
    max_time=1,
    data_set=data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    noise=True,
    ocba_delta = 1,  )
fun = HyperLight().fun
from spotpython.hyperparameters.values import set_hyperparameter
set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,7])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])

design_control = design_control_init(init_size=10, repeats=2)

print_exp_table(fun_control)

spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
res = spot_tuner.run()
spot_tuner.plot_progress()
print_res_table(spot_tuner)
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_11_04_21_05_36_0
```

```
Created spot_tensorboard_path: runs/spot_logs/700_maans08_2025-11-
04_21-05-36 for SummaryWriter()
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
| name           | type   | default   |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------
|
| l1             | int    | 3         |     3   |   4     | transform_power_2_int |
| epochs         | int    | 4         |     3   |   7     | transform_power_2_int |
| batch_size     | int    | 4         |     4   |  11     | transform_power_2_int |
| act_fn         | factor | ReLU      |     0   |   5     | None                  |
| optimizer      | factor | SGD       |     0   |   2     | None                  |
| dropout_prob   | float  | 0.01      |     0   |   0.025 | None                  |
| lr_mult        | float  | 1.0       |     0.1 |  10     | None                  |
| patience       | int    | 2         |     2   |   3     | transform_power_2_int |
| batch_norm     | factor | 0         |     0   |   1     | None                  |
| initialization | factor | Default   |     0   |   4     | None                  |
Experiment saved to 700_exp.pkl
```

```
     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    202 K   train   826 M   [2048, 10]   [2048, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 23075.09765625, 'hp_metric': 23075.09765625}
```

```
     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    202 K   train   826 M   [2048, 10]   [2048, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 23175.75, 'hp_metric': 23175.75}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   6.6 M   [64, 10]     [64, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 3110.947021484375, 'hp_metric': 3110.947021484375}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   6.6 M   [64, 10]     [64, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M
```

```
train_model result: {'val_loss': 3212.389404296875, 'hp_metric': 3212.389404296875}


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 4957.79296875, 'hp_metric': 4957.79296875}


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 4993.95263671875, 'hp_metric': 4993.95263671875}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]
```

```
Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 24098.71484375, 'hp_metric': 24098.71484375}



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 23989.546875, 'hp_metric': 23989.546875}



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M
```

```
train_model result: {'val_loss': 22617.146484375, 'hp_metric': 22617.146484375}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 22949.05859375, 'hp_metric': 22949.05859375}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 4575.27001953125, 'hp_metric': 4575.27001953125}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 4273.98681640625, 'hp_metric': 4273.98681640625}



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 21430.103515625, 'hp_metric': 21430.103515625}



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 52.5 M
```

```
train_model result: {'val_loss': 21671.5234375, 'hp_metric': 21671.5234375}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 3739.078369140625, 'hp_metric': 3739.078369140625}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 4217.5869140625, 'hp_metric': 4217.5869140625}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   13.1 M   [128, 10]    [128, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 13.1 M

train_model result: {'val_loss': 20498.853515625, 'hp_metric': 20498.853515625}



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   13.1 M   [128, 10]    [128, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 13.1 M

train_model result: {'val_loss': 20785.01953125, 'hp_metric': 20785.01953125}



     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   413 M   [1024, 10]   [1024, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 413 M
```

```
train_model result: {'val_loss': 24013.759765625, 'hp_metric': 24013.759765625}


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   413 M   [1024, 10]   [1024, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 413 M

train_model result: {'val_loss': 23966.04296875, 'hp_metric': 23966.04296875}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 5028.01416015625, 'hp_metric': 5028.01416015625}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 5419.3017578125, 'hp_metric': 5419.3017578125}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 5504.8271484375, 'hp_metric': 5504.8271484375}
spotpython tuning: 3110.947021484375 [##--------] 16.20%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M
```

```
train_model result: {'val_loss': 4030.219970703125, 'hp_metric': 4030.219970703125}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 24031.546875, 'hp_metric': 24031.546875}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 24042.513671875, 'hp_metric': 24042.513671875}
spotpython tuning: 3110.947021484375 [##--------] 22.22%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 3744.834716796875, 'hp_metric': 3744.834716796875}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   6.6 M   [64, 10]     [64, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 23541.51171875, 'hp_metric': 23541.51171875}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   6.6 M   [64, 10]     [64, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.6 M
```

```
train_model result: {'val_loss': 23640.212890625, 'hp_metric': 23640.212890625}
spotpython tuning: 3110.947021484375 [###-------] 29.95%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 4495.7265625, 'hp_metric': 4495.7265625}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 6404.63134765625, 'hp_metric': 6404.63134765625}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 5502.2958984375, 'hp_metric': 5502.2958984375}
spotpython tuning: 3110.947021484375 [####------] 37.05%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 4089.298095703125, 'hp_metric': 4089.298095703125}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M
```

```
train_model result: {'val_loss': 3133.743896484375, 'hp_metric': 3133.743896484375}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 3266.07080078125, 'hp_metric': 3266.07080078125}
spotpython tuning: 3110.947021484375 [####------] 44.72%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 3263.72216796875, 'hp_metric': 3263.72216796875}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   6.5 M   [16, 10]     [16, 1]
```

```
Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 17595.814453125, 'hp_metric': 17595.814453125}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 18742.83984375, 'hp_metric': 18742.83984375}
spotpython tuning: 3110.947021484375 [#####-----] 54.84%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   6.6 M   [64, 10]     [64, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M
```

```
train_model result: {'val_loss': 3179.1845703125, 'hp_metric': 3179.1845703125}


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   6.6 M   [64, 10]     [64, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 23752.625, 'hp_metric': 23752.625}


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   6.6 M   [64, 10]     [64, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 24077.103515625, 'hp_metric': 24077.103515625}
spotpython tuning: 3110.947021484375 [######----] 59.97%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 3202.749755859375, 'hp_metric': 3202.749755859375}



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 14215.9052734375, 'hp_metric': 14215.9052734375}



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M
```

```
train_model result: {'val_loss': 23243.51953125, 'hp_metric': 23243.51953125}
spotpython tuning: 3110.947021484375 [######----] 64.80%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   6.6 M   [64, 10]     [64, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 3123.379638671875, 'hp_metric': 3123.379638671875}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3469.1591796875, 'hp_metric': 3469.1591796875}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3059.779296875, 'hp_metric': 3059.779296875}
spotpython tuning: 3059.779296875 [#######---] 71.19%. Success rate: 3.70%



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3426.83935546875, 'hp_metric': 3426.83935546875}



     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   104 M   [1024, 10]   [1024, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 104 M
```

```
train_model result: {'val_loss': 22599.34375, 'hp_metric': 22599.34375}


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': 22829.173828125, 'hp_metric': 22829.173828125}
spotpython tuning: 3059.779296875 [#######---] 74.43%. Success rate: 3.33%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   6.6 M   [64, 10]     [64, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 3055.126220703125, 'hp_metric': 3055.126220703125}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 2985.070068359375, 'hp_metric': 2985.070068359375}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 2955.3466796875, 'hp_metric': 2955.3466796875}
spotpython tuning: 2955.3466796875 [########--] 78.78%. Success rate: 12.12%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M
```

```
train_model result: {'val_loss': 3040.13720703125, 'hp_metric': 3040.13720703125}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3145.708984375, 'hp_metric': 3145.708984375}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 15273.0732421875, 'hp_metric': 15273.0732421875}
spotpython tuning: 2955.3466796875 [########--] 84.65%. Success rate: 11.11%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]
```

```
Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3289.585205078125, 'hp_metric': 3289.585205078125}



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3477.06591796875, 'hp_metric': 3477.06591796875}



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M
```

```
train_model result: {'val_loss': 3473.568359375, 'hp_metric': 3473.568359375}
spotpython tuning: 2955.3466796875 [##########-] 94.04%. Success rate: 10.26%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3620.5595703125, 'hp_metric': 3620.5595703125}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 6685.09130859375, 'hp_metric': 6685.09130859375}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]
```

```
Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 7005.31640625, 'hp_metric': 7005.31640625}
spotpython tuning: 2955.3466796875 [##########] 100.00%. Success rate: 9.52% Done...

Experiment saved to 700_res.pkl
```

```
| name           | type   | default   |   lower |   upper | tuned                | transform
|----------------|--------|-----------|---------|---------|----------------------
|----------------------|--------------|---------|
| l1             | int    | 3         |     3.0 |     4.0 | 3.0                  | transform_power_2
| epochs         | int    | 4         |     3.0 |     7.0 | 6.0                  | transform_power_2
| batch_size     | int    | 4         |     4.0 |    11.0 | 5.0                  | transform_power_2
| act_fn         | factor | ReLU      |     0.0 |     5.0 | LeakyReLU            | None
| optimizer      | factor | SGD       |     0.0 |     2.0 | Adamax               | None
| dropout_prob   | float  | 0.01      |     0.0 |   0.025 | 0.022934782244241836 | None
| lr_mult        | float  | 1.0       |     0.1 |    10.0 | 2.8837852741429577   | None
| patience       | int    | 2         |     2.0 |     3.0 | 3.0                  | transform_power_2
| batch_norm     | factor | 0         |     0.0 |     1.0 | 0                    | None
| initialization | factor | Default   |     0.0 |     4.0 | kaiming_uniform      | None
```

41. Details of the Lightning Module Integration in spotpython

41.1. Introduction

Based on the Diabetes Data set and the `NNLinearRegressor` model, we will provide details on the integration of the Lightning module in spotpython.

- Section 41.2: The `Hyperlight` class provides the `fun` method, which takes `X` and `fun_control` as arguments. It calls the `train_model` method.
- Section 41.3: The `train_model` method trains the model and returns the loss.
- Section 41.4: The `Trainer` class is used to train the model and validate it. It also uses the `LightDataModule` class to load the data.

41.2. 1. spotpython.fun.hyperlight.HyperLight.fun()

The class `Hyperlight` provides the method `fun`, which takes `X` (`np.ndarray`) and `fun_control` (`dict`) as arguments. It calls the

```
from math import inf
import numpy as np
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import fun_control_init
from spotpython.utils.eda import print_exp_table
from spotpython.spot import Spot
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array

PREFIX="000"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
```

```
    save_experiment=True,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    seed=42,)

print_exp_table(fun_control)

X = get_default_hyperparameters_as_array(fun_control)
# set epochs to 2^8:
# X[0, 1] = 8
# set patience to 2^10:
# X[0, 7] = 10

print(f"X: {X}")
# combine X and X to a np.array with shape (2, n_hyperparams)
# so that two values are returned
X = np.vstack((X, X, X))
print(f"X: {X}")
```

```
hyper_light = HyperLight(seed=125, log_level=50)
hyper_light.fun(X, fun_control)
```

- Using the same seed:

```
hyper_light = HyperLight(seed=125, log_level=50)
hyper_light.fun(X, fun_control)
```

- Using a different seed:

```
hyper_light = HyperLight(seed=123, log_level=50)
hyper_light.fun(X, fun_control)
```

41.3. 2. spotpython.light.trainmodel.train_model()

```
from math import inf
import numpy as np
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.utils.eda import print_exp_table
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
from spotpython.hyperparameters.values import assign_values, generate_one_config_from_var_dict, get_
from spotpython.light.trainmodel import train_model
import pprint

PREFIX="000"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
    save_experiment=True,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    seed=42,)

print_exp_table(fun_control)

X = get_default_hyperparameters_as_array(fun_control)
# set epochs to 2^8:
# X[0, 1] = 8
# set patience to 2^10:
# X[0, 7] = 10

print(f"X: {X}")
# combine X and X to a np.array with shape (2, n_hyperparams)
# so that two values are returned
X = np.vstack((X, X))
var_dict = assign_values(X, get_var_name(fun_control))
for config in generate_one_config_from_var_dict(var_dict, fun_control):
```

```
    pprint.pprint(config)
    y = train_model(config, fun_control)
```

41.4. 3. Trainer: fit and validate

- Generate the `config` dictionary:

```
from math import inf
import numpy as np
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.utils.eda import print_exp_table
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
from spotpython.hyperparameters.values import assign_values, generate_one_config_from_
from spotpython.light.trainmodel import train_model

PREFIX="000"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
    save_experiment=True,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    seed=42,)
print_exp_table(fun_control)
X = get_default_hyperparameters_as_array(fun_control)
# set epochs to 2^8:
X[0, 1] = 10
# set patience to 2^10:
X[0, 7] = 10
print(f"X: {X}")
var_dict = assign_values(X, get_var_name(fun_control))
```

```
config = list(generate_one_config_from_var_dict(var_dict, fun_control))[0]
config
```

```
_L_in = 10
_L_out = 1
_L_cond = 0
_torchmetric = "mean_squared_error"
```

41.4.1. Commented: Using the fun_control dictionary

```
# model = fun_control["core_model"](**config, _L_in=_L_in, _L_out=_L_out, _L_cond=_L_cond, _torchmet
# model
```

41.4.2. Using the source code:

```
import lightning as L
import torch
from torch import nn
from spotpython.hyperparameters.optimizer import optimizer_handler
import torchmetrics.functional.regression
import torch.optim as optim
from spotpython.hyperparameters.architecture import get_hidden_sizes


class NNLinearRegressor(L.LightningModule):
    def __init__(
        self,
        l1: int,
        epochs: int,
        batch_size: int,
        initialization: str,
        act_fn: nn.Module,
        optimizer: str,
        dropout_prob: float,
        lr_mult: float,
        patience: int,
        batch_norm: bool,
        _L_in: int,
        _L_out: int,
```

```
        _torchmetric: str,
        *args,
        **kwargs,
    ):
        super().__init__()
        # Attribute 'act_fn' is an instance of `nn.Module` and is already saved during
        # checkpointing. It is recommended to ignore them
        # using `self.save_hyperparameters(ignore=['act_fn'])`
        # self.save_hyperparameters(ignore=["act_fn"])
        #
        self._L_in = _L_in
        self._L_out = _L_out
        if _torchmetric is None:
            _torchmetric = "mean_squared_error"
        self._torchmetric = _torchmetric
        self.metric = getattr(torchmetrics.functional.regression, _torchmetric)
        # _L_in and _L_out are not hyperparameters, but are needed to create the netwo
        # _torchmetric is not a hyperparameter, but is needed to calculate the loss
        self.save_hyperparameters(ignore=["_L_in", "_L_out", "_torchmetric"])
        # set dummy input array for Tensorboard Graphs
        # set log_graph=True in Trainer to see the graph (in traintest.py)
        self.example_input_array = torch.zeros((batch_size, self._L_in))
        if self.hparams.l1 < 4:
            raise ValueError("l1 must be at least 4")
        hidden_sizes = get_hidden_sizes(_L_in=self._L_in, l1=l1, n=10)

        if batch_norm:
            # Add batch normalization layers
            layers = []
            layer_sizes = [self._L_in] + hidden_sizes
            for i in range(len(layer_sizes) - 1):
                current_layer_size = layer_sizes[i]
                next_layer_size = layer_sizes[i + 1]
                layers += [
                    nn.Linear(current_layer_size, next_layer_size),
                    nn.BatchNorm1d(next_layer_size),
                    self.hparams.act_fn,
                    nn.Dropout(self.hparams.dropout_prob),
                ]
            layers += [nn.Linear(layer_sizes[-1], self._L_out)]
        else:
            layers = []
            layer_sizes = [self._L_in] + hidden_sizes
            for i in range(len(layer_sizes) - 1):
```

```
            current_layer_size = layer_sizes[i]
            next_layer_size = layer_sizes[i + 1]
            layers += [
                nn.Linear(current_layer_size, next_layer_size),
                self.hparams.act_fn,
                nn.Dropout(self.hparams.dropout_prob),
            ]
        layers += [nn.Linear(layer_sizes[-1], self._L_out)]

    # Wrap the layers into a sequential container
    self.layers = nn.Sequential(*layers)

    # Initialization (Xavier, Kaiming, or Default)
    self.apply(self._init_weights)

def _init_weights(self, module):
    if isinstance(module, nn.Linear):
        if self.hparams.initialization == "xavier_uniform":
            nn.init.xavier_uniform_(module.weight)
        elif self.hparams.initialization == "xavier_normal":
            nn.init.xavier_normal_(module.weight)
        elif self.hparams.initialization == "kaiming_uniform":
            nn.init.kaiming_uniform_(module.weight)
        elif self.hparams.initialization == "kaiming_normal":
            nn.init.kaiming_normal_(module.weight)
        else:  # "Default"
            nn.init.uniform_(module.weight)
        if module.bias is not None:
            nn.init.zeros_(module.bias)

def forward(self, x: torch.Tensor) -> torch.Tensor:
    """
    Performs a forward pass through the model.

    Args:
        x (torch.Tensor): A tensor containing a batch of input data.

    Returns:
        torch.Tensor: A tensor containing the output of the model.

    """
    x = self.layers(x)
    return x
```

```
    def _calculate_loss(self, batch):
        """
        Calculate the loss for the given batch.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            mode (str, optional): The mode of the model. Defaults to "train".

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        x, y = batch
        y = y.view(len(y), 1)
        y_hat = self(x)
        loss = self.metric(y_hat, y)
        return loss

    def training_step(self, batch: tuple) -> torch.Tensor:
        """
        Performs a single training step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        loss = self._calculate_loss(batch)
        self.log("train_loss", loss, on_step=False, on_epoch=True, prog_bar=False)
        return loss

    def validation_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) ->
        """
        Performs a single validation step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults t

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.
```

```
        """
        loss = self._calculate_loss(batch)
        self.log("val_loss", loss, on_step=False, on_epoch=True, prog_bar=prog_bar)
        self.log("hp_metric", loss, on_step=False, on_epoch=True, prog_bar=prog_bar)
        return loss

    def test_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch.Tensor:
        """
        Performs a single test step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults to False.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.
        """
        loss = self._calculate_loss(batch)
        self.log("val_loss", loss, on_step=False, on_epoch=True, prog_bar=prog_bar)
        self.log("hp_metric", loss, on_step=False, on_epoch=True, prog_bar=prog_bar)
        return loss

    def predict_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch.Tensor:
        """
        Performs a single prediction step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults to False.

        Returns:
            torch.Tensor: A tensor containing the prediction for this batch.
        """
        x, y = batch
        yhat = self(x)
        y = y.view(len(y), 1)
        yhat = yhat.view(len(yhat), 1)
        print(f"Predict step x: {x}")
        print(f"Predict step y: {y}")
        print(f"Predict step y_hat: {yhat}")
        # pred_loss = F.mse_loss(y_hat, y)
        # pred loss not registered
```

```
        # self.log("pred_loss", pred_loss, prog_bar=prog_bar)
        # self.log("hp_metric", pred_loss, prog_bar=prog_bar)
        # MisconfigurationException: You are trying to `self.log()`
        # but the loop's result collection is not registered yet.
        # This is most likely because you are trying to log in a `predict` hook, but i
        # If you want to manually log, please consider using `self.log_dict({'pred_los
        return (x, y, yhat)

    def configure_optimizers(self) -> torch.optim.Optimizer:
        """
        Configures the optimizer for the model.

        Notes:
            The default Lightning way is to define an optimizer as
            `optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)`.
            spotpython uses an optimizer handler to create the optimizer, which
            adapts the learning rate according to the lr_mult hyperparameter as
            well as other hyperparameters. See `spotpython.hyperparameters.optimizer.p

        Returns:
            torch.optim.Optimizer: The optimizer to use during training.

        """
        # optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)
        optimizer = optimizer_handler(optimizer_name=self.hparams.optimizer, params=se

        num_milestones = 3  # Number of milestones to divide the epochs
        milestones = [int(self.hparams.epochs / (num_milestones + 1) * (i + 1)) for i
        scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=milestones, g

        lr_scheduler_config = {
            "scheduler": scheduler,
            "interval": "epoch",
            "frequency": 1,
        }

        return {"optimizer": optimizer, "lr_scheduler": lr_scheduler_config}
```

```
model = NNLinearRegressor(**config, _L_in=_L_in, _L_out=_L_out, _L_cond=_L_cond, _torc
model
```

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
```

```
data_set = Diabetes()
dm = LightDataModule(
    dataset=data_set,
    batch_size=config["batch_size"],
    test_size=fun_control["test_size"],
    test_seed=fun_control["test_seed"],
    scaler=None,
)
```

- Using `callbacks` for early stopping:

```
from lightning.pytorch.callbacks.early_stopping import EarlyStopping
callbacks = [EarlyStopping(monitor="val_loss", patience=config["patience"], mode="min", strict=False
```

```
timestamp = True

from lightning.pytorch.callbacks import ModelCheckpoint
if not timestamp:
    # add ModelCheckpoint only if timestamp is False
    callbacks.append(ModelCheckpoint(dirpath=os.path.join(fun_control["CHECKPOINT_PATH"], config_id)
```

```
from spotpython.utils.eda import generate_config_id
if timestamp:
    # config id is unique. Since the model is not loaded from a checkpoint,
    # the config id is generated here with a timestamp.
    config_id = generate_config_id(config, timestamp=True)
else:
    # config id is not time-dependent and therefore unique,
    # so that the model can be loaded from a checkpoint,
    # the config id is generated here without a timestamp.
    config_id = generate_config_id(config, timestamp=False) + "_TRAIN"
```

```
from pytorch_lightning.loggers import TensorBoardLogger
import lightning as L
import os
trainer = L.Trainer(
    # Where to save models
    default_root_dir=os.path.join(fun_control["CHECKPOINT_PATH"], config_id),
    max_epochs=model.hparams.epochs,
    accelerator=fun_control["accelerator"],
    devices=fun_control["devices"],
    strategy=fun_control["strategy"],
    num_nodes=fun_control["num_nodes"],
```

```
    precision=fun_control["precision"],
    logger=TensorBoardLogger(save_dir=fun_control["TENSORBOARD_PATH"], version=config_
    callbacks=callbacks,
    enable_progress_bar=False,
    num_sanity_val_steps=fun_control["num_sanity_val_steps"],
    log_every_n_steps=fun_control["log_every_n_steps"],
    gradient_clip_val=None,
    gradient_clip_algorithm="norm",
)
```

```
trainer.fit(model=model, datamodule=dm, ckpt_path=None)
```

```
trainer.validate(model=model, datamodule=dm, verbose=True, ckpt_path=None)
```

41.4.3. DataModule

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.csvdataset import CSVDataset
import torch
dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', feature_type=torc
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.5)
data_module.setup()
print(f"Training set size: {len(data_module.data_train)}")
```

- Generate the `config` dictionary:

```
from math import inf
import lightning as L
import numpy as np
import os
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.utils.eda import print_exp_table
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
from spotpython.hyperparameters.values import assign_values, generate_one_config_from_
from spotpython.light.trainmodel import train_model, generate_config_id_with_timestamp
from pytorch_lightning.loggers import TensorBoardLogger
from lightning.pytorch.callbacks.early_stopping import EarlyStopping
from spotpython.data.lightdatamodule import LightDataModule
PREFIX="000"
```

```
data_set = Diabetes()
fun_control = fun_control_init(
    PREFIX=PREFIX,
    save_experiment=True,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    seed=42,)
print_exp_table(fun_control)
X = get_default_hyperparameters_as_array(fun_control)
# set epochs to 2^8:
X[0, 1] = 10
# set patience to 2^10:
X[0, 7] = 10
print(f"X: {X}")
var_dict = assign_values(X, get_var_name(fun_control))
config = list(generate_one_config_from_var_dict(var_dict, fun_control))[0]
_L_in = fun_control["_L_in"]
_L_out = fun_control["_L_out"]
_L_cond = fun_control["_L_cond"]
_torchmetric = fun_control["_torchmetric"]
model = fun_control["core_model"](**config, _L_in=_L_in, _L_out=_L_out, _L_cond=_L_cond, _torchmetri
dm = LightDataModule(
    dataset=fun_control["data_set"],
    batch_size=config["batch_size"],
    num_workers=fun_control["num_workers"],
    test_size=fun_control["test_size"],
    test_seed=fun_control["test_seed"],
    scaler=fun_control["scaler"],
)
config_id = generate_config_id_with_timestamp(config, timestamp=True)
callbacks = [EarlyStopping(monitor="val_loss", patience=config["patience"], mode="min", strict=False
trainer = L.Trainer(
    default_root_dir=os.path.join(fun_control["CHECKPOINT_PATH"], config_id),
    max_epochs=model.hparams.epochs,
    accelerator=fun_control["accelerator"],
    devices=fun_control["devices"],
    strategy=fun_control["strategy"],
```

```
    num_nodes=fun_control["num_nodes"],
    precision=fun_control["precision"],
    logger=TensorBoardLogger(save_dir=fun_control["TENSORBOARD_PATH"], version=config_
    callbacks=callbacks,
    enable_progress_bar=False,
    num_sanity_val_steps=fun_control["num_sanity_val_steps"],
    log_every_n_steps=fun_control["log_every_n_steps"],
    gradient_clip_val=None,
    gradient_clip_algorithm="norm",
)
trainer.fit(model=model, datamodule=dm, ckpt_path=None)
```

```
from math import inf
import lightning as L
import numpy as np
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import assign_values, generate_one_config_from_
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.utils.scaler import TorchStandardScaler
PREFIX="000"
data_set = Diabetes()
fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)
X = np.array([[3.0e+00, 5.0, 4.0e+00, 2.0e+00, 1.1e+01, 1.0e-02, 1.0e+00, 1.0e+01, 0.0
  0.0e+00]])
var_dict = assign_values(X, get_var_name(fun_control))
config = list(generate_one_config_from_var_dict(var_dict, fun_control))[0]
_torchmetric = "mean_squared_error"
model = fun_control["core_model"](**config, _L_in=10, _L_out=1, _L_cond=None, _torchme
dm = LightDataModule(
    dataset=data_set,
    batch_size=16,
    test_size=0.6,
    scaler=TorchStandardScaler())
trainer = L.Trainer(
```

```
    max_epochs=32,
    enable_progress_bar=False,
)
trainer.fit(model=model, datamodule=dm, ckpt_path=None)
trainer.validate(model=model, datamodule=dm, ckpt_path=None)
```

42. User Specified Basic Lightning Module With spotpython

42.1. Introduction

This chapter implements a user-defined DataModule and a user-defined neural network. Remember, that a `LightningModule` organizes your `PyTorch` code into six sections:

- Initialization (`__init__` and `setup()`).
- Train Loop (`training_step()`)
- Validation Loop (`validation_step()`)
- Test Loop (`test_step()`)
- Prediction Loop (`predict_step()`)
- Optimizers and LR Schedulers (`configure_optimizers()`)

The `Trainer` automates every required step in a clear and reproducible way. It is the most important part of PyTorch Lightning. It is responsible for training, testing, and validating the model. The `Lightning` core structure looks like this:

```
import pandas as pd
df = pd.read_pickle("./userData/Turbo_Charger_Data.pkl")
df = df.drop(columns=["M", "R"])
print(f"Features des DataFrames: {df.columns}")
print(df.shape)
```

42.1.1. Dataset

```
from sklearn.preprocessing import LabelEncoder
from lightning import LightningDataModule
import torch
from torch.utils.data import Dataset, DataLoader, random_split

class UserDataset(Dataset):
    def __init__(self, data, y_varname="N", x_varnames=None, dtype=torch.float32):
        """
```

```
        Args:
            data (pd.DataFrame):
                The user data. for example,
                generated by the `preprocess_data` function.
            y_varname (str):
                The name of the target variable.
                Default is "N".
            x_varnames (list):
                The names of the input variables.
                Default is `None`, which means all columns
                except the target variable are used.
            dtype (torch.dtype):
                The data type for the tensors.
                Default is `torch.float32`.

        Examples:
            >>> dataset = UserDataset(data)
            >>> x, y = dataset[0]
        """
        self.data = data.reset_index(drop=True)
        if x_varnames is not None:
            self.x_varnames = x_varnames
        else:
            self.x_varnames = [col for col in self.data.columns if col != y_varname]
        print(f"X variables: {self.x_varnames}")
        print(f"Y variable: {y_varname}")
        self.y_varname = y_varname
        self.dtype = dtype
        self.encoders = {}

        for var in self.x_varnames:
            if self.data[var].dtype == "object" or isinstance(self.data[var][0], str)
                le = LabelEncoder()
                self.data[var] = le.fit_transform(self.data[var])
                self.encoders[var] = le

        if self.data[self.y_varname].dtype == "object" or isinstance(self.data[self.y_
            le = LabelEncoder()
            self.data[self.y_varname] = le.fit_transform(self.data[self.y_varname])
            self.encoders[self.y_varname] = le

        # Convert entire dataset to tensors
        self.features = torch.tensor(self.data[self.x_varnames].values, dtype=self.dty
        self.targets = torch.tensor(self.data[self.y_varname].values, dtype=self.dtype
```

```
    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.features[idx], self.targets[idx]
```

```
dataset = UserDataset(df)
x, y = dataset[0]
print(x)
print(y)
```

42.1.2. DataModule

```
import lightning as L
import torch
from torch.utils.data import DataLoader, random_split, TensorDataset
from typing import Optional
from math import floor


class LightDataModule(L.LightningDataModule):
    """
    A LightningDataModule for handling data.

    Args:
        batch_size (int):
            The batch size. Required.
        dataset (torch.utils.data.Dataset, optional):
            The dataset from the torch.utils.data Dataset class.
            It must implement three functions: __init__, __len__, and __getitem__.
        test_size (float, optional):
            The test size. If test_size is float, then train_size is 1 - test_size.
            If test_size is int, then train_size is len(data_full) - test_size.
        test_seed (int):
            The test seed. Defaults to 42.
        num_workers (int):
            The number of workers. Defaults to 0.
        verbosity (int):
            The verbosity level. Defaults to 0.

    Examples:
```

```
        >>> from spotpython.data.lightdatamodule import LightDataModule
            from spotpython.data.csvdataset import CSVDataset
            from spotpython.utils.scaler import TorchStandardScaler
            import torch
            # data.csv is simple csv file with 11 samples
            dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', featu
            scaler = TorchStandardScaler()
            data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.5
            data_module.setup()
            print(f"Training set size: {len(data_module.data_train)}")
            print(f"Validation set size: {len(data_module.data_val)}")
            print(f"Test set size: {len(data_module.data_test)}")
            full_train_size: 0.5
            val_size: 0.25
            train_size: 0.25
            test_size: 0.5
            Training set size: 3
            Validation set size: 3
            Test set size: 6

    References:
        See https://lightning.ai/docs/pytorch/stable/data/datamodule.html

    """

    def __init__(
        self,
        batch_size: int,
        dataset: Optional[object] = None,
        test_size: Optional[float] = None,
        test_seed: int = 42,
        num_workers: int = 0,
        verbosity: int = 0,
    ):
        super().__init__()
        self.batch_size = batch_size
        self.data_full = dataset
        self.test_size = test_size
        self.test_seed = test_seed
        self.num_workers = num_workers
        self.verbosity = verbosity

    def prepare_data(self) -> None:
        """Prepares the data for use."""
```

```
        # download
        pass

    def _setup_full_data_provided(self, stage) -> None:
        full_size = len(self.data_full)
        test_size = self.test_size

        # consider the case when test_size is a float
        if isinstance(self.test_size, float):
            full_train_size = 1.0 - self.test_size
            val_size = full_train_size * self.test_size
            train_size = full_train_size - val_size
        else:
            # test_size is an int, training size calculation directly based on it
            full_train_size = full_size - self.test_size
            val_size = floor(full_train_size * self.test_size / full_size)
            train_size = full_size - val_size - test_size

        # Assign train/val datasets for use in dataloaders
        if stage == "fit" or stage is None:
            generator_fit = torch.Generator().manual_seed(self.test_seed)
            self.data_train, self.data_val, _ = random_split(self.data_full, [train_size, val_size,
            if self.verbosity > 0:
                print(f"train_size: {train_size}, val_size: {val_size}, test_sie: {test_size} for sp
                print(f"train samples: {len(self.data_train)}, val samples: {len(self.data_val)} gen

        # Assign test dataset for use in dataloader(s)
        if stage == "test" or stage is None:
            generator_test = torch.Generator().manual_seed(self.test_seed)
            self.data_test, _, _ = random_split(self.data_full, [test_size, train_size, val_size], g
            if self.verbosity > 0:
                print(f"train_size: {train_size}, val_size: {val_size}, test_sie: {test_size} for sp
                print(f"test samples: {len(self.data_test)} generated for test data.")

        # Assign pred dataset for use in dataloader(s)
        if stage == "predict" or stage is None:
            generator_predict = torch.Generator().manual_seed(self.test_seed)
            self.data_predict, _, _ = random_split(self.data_full, [test_size, train_size, val_size]
            if self.verbosity > 0:
                print(f"train_size: {train_size}, val_size: {val_size}, test_size (= predict_size):
                print(f"predict samples: {len(self.data_predict)} generated for train & val data.")


    def setup(self, stage: Optional[str] = None) -> None:
```

```
        """
        Splits the data for use in training, validation, and testing.
        Uses torch.utils.data.random_split() to split the data.
        Splitting is based on the test_size and test_seed.
        The test_size can be a float or an int.
        If a spotpython scaler object is defined, the data will be scaled.

        Args:
            stage (Optional[str]):
                The current stage. Can be "fit" (for training and validation), "test"
                or None (for all three stages). Defaults to None.

        Examples:
            >>> from spotpython.data.lightdatamodule import LightDataModule
                from spotpython.data.csvdataset import CSVDataset
                import torch
                dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', i
                data_module = LightDataModule(dataset=dataset, batch_size=5, test_size
                data_module.setup()
                print(f"Training set size: {len(data_module.data_train)}")
                Training set size: 3

        """
        self._setup_full_data_provided(stage)


    def train_dataloader(self) -> DataLoader:
        """
        Returns the training dataloader, i.e., a pytorch DataLoader instance
        using the training dataset.

        Returns:
            DataLoader: The training dataloader.

        Examples:
            >>> from spotpython.data.lightdatamodule import LightDataModule
                from spotpython.data.csvdataset import CSVDataset
                import torch
                dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', i
                data_module = LightDataModule(dataset=dataset, batch_size=5, test_size
                data_module.setup()
                print(f"Training set size: {len(data_module.data_train)}")
                Training set size: 3
```

```
        """
        if self.verbosity > 0:
            print(f"LightDataModule.train_dataloader(). data_train size: {len(self.data_train)}")
        return DataLoader(self.data_train, batch_size=self.batch_size, num_workers=self.num_workers)

    def val_dataloader(self) -> DataLoader:
        """
        Returns the validation dataloader, i.e., a pytorch DataLoader instance
        using the validation dataset.

        Returns:
            DataLoader: The validation dataloader.

        Examples:
            >>> from spotpython.data.lightdatamodule import LightDataModule
                from spotpython.data.csvdataset import CSVDataset
                import torch
                dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', feature_type=to
                data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.5)
                data_module.setup()
                print(f"Training set size: {len(data_module.data_val)}")
                Training set size: 3
        """
        if self.verbosity > 0:
            print(f"LightDataModule.val_dataloader(). Val. set size: {len(self.data_val)}")
        return DataLoader(self.data_val, batch_size=self.batch_size, num_workers=self.num_workers)

    def test_dataloader(self) -> DataLoader:
        """
        Returns the test dataloader, i.e., a pytorch DataLoader instance
        using the test dataset.

        Returns:
            DataLoader: The test dataloader.

        Examples:
            >>> from spotpython.data.lightdatamodule import LightDataModule
                from spotpython.data.csvdataset import CSVDataset
                import torch
                dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', feature_type=to
                data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.5)
                data_module.setup()
                print(f"Test set size: {len(data_module.data_test)}")
                Test set size: 6
```

```
        """
        if self.verbosity > 0:
            print(f"LightDataModule.test_dataloader(). Test set size: {len(self.data_t
        return DataLoader(self.data_test, batch_size=self.batch_size, num_workers=seli

    def predict_dataloader(self) -> DataLoader:
        """
        Returns the predict dataloader, i.e., a pytorch DataLoader instance
        using the predict dataset.

        Returns:
            DataLoader: The predict dataloader.

        Examples:
            >>> from spotpython.data.lightdatamodule import LightDataModule
                from spotpython.data.csvdataset import CSVDataset
                import torch
                dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', f
                data_module = LightDataModule(dataset=dataset, batch_size=5, test_size
                data_module.setup()
                print(f"Predict set size: {len(data_module.data_predict)}")
                Predict set size: 6

        """
        if self.verbosity > 0:
            print(f"LightDataModule.predict_dataloader(). Predict set size: {len(self
        return DataLoader(self.data_predict, batch_size=len(self.data_predict), num_wo
```

```
data_module = LightDataModule(batch_size=2, dataset=dataset, test_size=0.2)
data_module.setup()
for batch in data_module.train_dataloader():
    print(batch)
    print(f"Number of input features: {batch[0][1].shape}")
    break
```

42.2. The Neural Network: MyRegressor

```
import lightning as L
import torch
from torch import nn
from spotpython.hyperparameters.optimizer import optimizer_handler
```

```python
import torchmetrics.functional.regression
from math import ceil

class MyRegressor(L.LightningModule):
    """
    A LightningModule class for a regression neural network model.

    Attributes:
        l1 (int):
            The number of neurons in the first hidden layer.
        epochs (int):
            The number of epochs to train the model for.
        batch_size (int):
            The batch size to use during training.
        initialization (str):
            The initialization method to use for the weights.
        act_fn (nn.Module):
            The activation function to use in the hidden layers.
        optimizer (str):
            The optimizer to use during training.
        dropout_prob (float):
            The probability of dropping out a neuron during training.
        lr_mult (float):
            The learning rate multiplier for the optimizer.
        patience (int):
            The number of epochs to wait before early stopping.
        _L_in (int):
            The number of input features.
        _L_out (int):
            The number of output classes.
        _torchmetric (str):
            The metric to use for the loss function. If `None`,
            then "mean_squared_error" is used.
        layers (nn.Sequential):
            The neural network model.

    """

    def __init__(
        self,
        l1: int,
        epochs: int,
        batch_size: int,
        initialization: str,
```

```
        act_fn: nn.Module,
        optimizer: str,
        dropout_prob: float,
        lr_mult: float,
        patience: int,
        _L_in: int,
        _L_out: int,
        _torchmetric: str,
        *args,
        **kwargs,
    ):
        """
        Initializes the MyRegressor object.

        Args:
            l1 (int):
                The number of neurons in the first hidden layer.
            epochs (int):
                The number of epochs to train the model for.
            batch_size (int):
                The batch size to use during training.
            initialization (str):
                The initialization method to use for the weights.
            act_fn (nn.Module):
                The activation function to use in the hidden layers.
            optimizer (str):
                The optimizer to use during training.
            dropout_prob (float):
                The probability of dropping out a neuron during training.
            lr_mult (float):
                The learning rate multiplier for the optimizer.
            patience (int):
                The number of epochs to wait before early stopping.
            _L_in (int):
                The number of input features. Not a hyperparameter, but needed to crea
            _L_out (int):
                The number of output classes. Not a hyperparameter, but needed to crea
            _torchmetric (str):
                The metric to use for the loss function. If `None`,
                then "mean_squared_error" is used.

        Returns:
            (NoneType): None
```

```
        Raises:
            ValueError: If l1 is less than 4.

        """
        super().__init__()
        self._L_in = _L_in
        self._L_out = _L_out
        if _torchmetric is None:
            _torchmetric = "mean_squared_error"
        self._torchmetric = _torchmetric
        self.metric = getattr(torchmetrics.functional.regression, _torchmetric)
        # _L_in and _L_out are not hyperparameters, but are needed to create the network
        # _torchmetric is not a hyperparameter, but is needed to calculate the loss
        self.save_hyperparameters(ignore=["_L_in", "_L_out", "_torchmetric"])
        # set dummy input array for Tensorboard Graphs
        # set log_graph=True in Trainer to see the graph (in traintest.py)
        self.example_input_array = torch.zeros((batch_size, self._L_in))
        if self.hparams.l1 < 4:
            raise ValueError("l1 must be at least 4")
        hidden_sizes = [l1 * 2, l1, ceil(l1/2)]
        # Create the network based on the specified hidden sizes
        layers = []
        layer_sizes = [self._L_in] + hidden_sizes
        layer_size_last = layer_sizes[0]
        for layer_size in layer_sizes[1:]:
            layers += [
                nn.Linear(layer_size_last, layer_size),
                self.hparams.act_fn,
                nn.Dropout(self.hparams.dropout_prob),
            ]
            layer_size_last = layer_size
        layers += [nn.Linear(layer_sizes[-1], self._L_out)]
        self.layers = nn.Sequential(*layers)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Performs a forward pass through the model.

        Args:
            x (torch.Tensor): A tensor containing a batch of input data.

        Returns:
            torch.Tensor: A tensor containing the output of the model.
```

```
        """
        x = self.layers(x)
        return x

    def _calculate_loss(self, batch):
        """
        Calculate the loss for the given batch.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        x, y = batch
        y = y.view(len(y), 1)
        y_hat = self(x)
        loss = self.metric(y_hat, y)
        return loss

    def training_step(self, batch: tuple) -> torch.Tensor:
        """
        Performs a single training step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        val_loss = self._calculate_loss(batch)
        return val_loss

    def validation_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) ->
        """
        Performs a single validation step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults t
```

```
        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        val_loss = self._calculate_loss(batch)
        self.log("val_loss", val_loss, prog_bar=prog_bar)
        self.log("hp_metric", val_loss, prog_bar=prog_bar)
        return val_loss

    def test_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch.Tensor:
        """
        Performs a single test step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults to False.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.
        """
        val_loss = self._calculate_loss(batch)
        self.log("val_loss", val_loss, prog_bar=prog_bar)
        self.log("hp_metric", val_loss, prog_bar=prog_bar)
        return val_loss

    def predict_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch.Tensor:
        """
        Performs a single prediction step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults to False.

        Returns:
            A tuple containing the input data, the true labels, and the predicted values.
        """
        x, y = batch
        yhat = self(x)
        y = y.view(len(y), 1)
        yhat = yhat.view(len(yhat), 1)
        print(f"Predict step x: {x}")
        print(f"Predict step y: {y}")
```

```
        print(f"Predict step y_hat: {yhat}")
        return (x, y, yhat)

    def configure_optimizers(self) -> torch.optim.Optimizer:
        """
        Configures the optimizer for the model.
        Simple examples use the following code here:
        `optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)`

        Notes:
            The default Lightning way is to define an optimizer as
            `optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)`.
            spotpython uses an optimizer handler to create the optimizer, which
            adapts the learning rate according to the lr_mult hyperparameter as
            well as other hyperparameters. See `spotpython.hyperparameters.optimizer.p

        Returns:
            torch.optim.Optimizer: The optimizer to use during training.

        """
        # optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)
        optimizer = optimizer_handler(
            optimizer_name=self.hparams.optimizer, params=self.parameters(), lr_mult=s
        )
        return optimizer
```

42.3. Calling the Neural Network With spotpython

```
PREFIX="702_lightning_user_datamodule"
```

```
import sys
sys.path.insert(0, './userModel')
import my_regressor
import my_hyper_dict
```

```
from spotpython.hyperparameters.values import add_core_model_to_fun_control
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_co
from spotpython.hyperparameters.values import set_hyperparameter
from spotpython.spot import Spot
```

```
from math import inf
fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
    fun_repeats=1,
    max_time=5,
    accelerator="cpu",
    data_module=data_module,
    _L_in=dataset[0][0].shape[0],
    _L_out=1,
    noise=False,
    ocba_delta=0,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    _torchmetric="mean_squared_error",
    log_level=50,
    save_experiment=True,
    verbosity=1)

add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=my_regressor.MyRegressor,
                              hyper_dict=my_hyper_dict.MyHyperDict)

set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "act_fn", [ "ReLU", "Swish", "LeakyReLU"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,5])
set_hyperparameter(fun_control, "batch_size", [1,5])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])
# set_hyperparameter(fun_control, "initialization", ["Default"])

design_control = design_control_init(init_size=5, repeats=1)
surrogate_control = surrogate_control_init(method="regression")

fun = HyperLight().fun

spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control, surrogate_control=
```

```
import os
from spotpython.utils.file import load_experiment
if os.path.exists("spot_" + PREFIX + "_experiment.pickle"):
    (spot_tuner, fun_control, design_control,
```

```
        surrogate_control, optimizer_control) = load_experiment(PREFIX=PREFIX)
else:
    res = spot_tuner.run()
```

42.4. Looking at the Results

42.4.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress()
```

42.4.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=1.0)
```

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

42.4.3. Get the Tuned Architecture

```
import pprint
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner)
pprint.pprint(config)
```

43. HPT PyTorch Lightning: Data

In this tutorial, we will show how `spotpython` can be integrated into the `PyTorch` Lightning training workflow.

This chapter describes the data preparation and processing in `spotpython`. The Diabetes data set is used as an example. This is a PyTorch Dataset for regression. A toy data set from scikit-learn. Ten baseline variables, age, sex, body mass index, average blood pressure, and six blood serum measurements were obtained for each of n = 442 diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline.

43.1. Setup

- Before we consider the detailed experimental setup, we select the parameters that affect run time, initial design size, etc.
- The parameter `WORKERS` specifies the number of workers.
- The prefix `PREFIX` is used for the experiment name and the name of the log file.
- The parameter `DEVICE` specifies the device to use for training.

```
import torch
from spotpython.utils.device import getDevice
from math import inf
WORKERS = 0
PREFIX="030"
DEVICE = getDevice()
DEVICES = 1
TEST_SIZE = 0.4
```

> **Note: Device selection**
>
> - Although there are no .cuda() or .to(device) calls required, because Lightning does these for you, see LIGHTNINGMODULE, we would like to know which device is used. Threrefore, we imitate the LightningModule behaviour which selects the highest device.
> - The method `spotpython.utils.device.getDevice()` returns the device that is used by Lightning.

43.2. Initialization of the `fun_control` Dictionary

`spotpython` uses a Python dictionary for storing the information required for the hyperparameter tuning process.

```
from spotpython.utils.init import fun_control_init
import numpy as np
fun_control = fun_control_init(
    _L_in=10,
    _L_out=1,
    _torchmetric="mean_squared_error",
    PREFIX=PREFIX,
    device=DEVICE,
    enable_progress_bar=False,
    num_workers=WORKERS,
    show_progress=True,
    test_size=TEST_SIZE,
    )
```

43.3. Loading the Diabetes Data Set

Here, we load the Diabetes data set from `spotpython`'s `data` module.

```
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
print(len(dataset))
```

```
442
```

43.3.1. Data Set and Data Loader

As shown below, a DataLoader from `torch.utils.data` can be used to check the data.

```
# Set batch size for DataLoader
batch_size = 5
# Create DataLoader
from torch.utils.data import DataLoader
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=False)

# Iterate over the data in the DataLoader
```

```
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0381,  0.0507,  0.0617,  0.0219, -0.0442, -0.0348, -
0.0434, -0.0026,
          0.0199, -0.0176],
        [-0.0019, -0.0446, -0.0515, -0.0263, -0.0084, -0.0192,  0.0744, -
0.0395,
         -0.0683, -0.0922],
        [ 0.0853,  0.0507,  0.0445, -0.0057, -0.0456, -0.0342, -
0.0324, -0.0026,
          0.0029, -0.0259],
        [-0.0891, -0.0446, -0.0116, -0.0367,  0.0122,  0.0250, -
0.0360,  0.0343,
          0.0227, -0.0094],
        [ 0.0054, -0.0446, -0.0364,  0.0219,  0.0039,  0.0156,  0.0081, -
0.0026,
         -0.0320, -0.0466]])
Targets: tensor([151.,  75., 141., 206., 135.])
```

43.3.2. Preparing Training, Validation, and Test Data

The following code shows how to split the data into training, validation, and test sets. Then a Lightning Trainer is used to train (`fit`) the model, validate it, and test it.

```
from torch.utils.data import DataLoader
from spotpython.data.diabetes import Diabetes
from spotpython.light.regression.netlightregression import NetLightRegression
from torch import nn
import lightning as L
import torch
BATCH_SIZE = 8
```

```
dataset = Diabetes(target_type=torch.float)
train1_set, test_set = torch.utils.data.random_split(dataset, [0.6, 0.4])
train_set, val_set = torch.utils.data.random_split(train1_set, [0.6, 0.4])
train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True, drop_last=Tr
test_loader = DataLoader(test_set, batch_size=BATCH_SIZE)
val_loader = DataLoader(val_set, batch_size=BATCH_SIZE)
batch_x, batch_y = next(iter(train_loader))
print(f"batch_x.shape: {batch_x.shape}")
print(f"batch_y.shape: {batch_y.shape}")
net_light_base = NetLightRegression(l1=128,
                                    epochs=10,
                                    batch_size=BATCH_SIZE,
                                    initialization='Default',
                                    act_fn=nn.ReLU(),
                                    optimizer='Adam',
                                    dropout_prob=0.1,
                                    lr_mult=0.1,
                                    patience=5,
                                    _L_in=10,
                                    _L_out=1,
                                    _torchmetric="mean_squared_error")
trainer = L.Trainer(max_epochs=10,  enable_progress_bar=False)
trainer.fit(net_light_base, train_loader)
trainer.validate(net_light_base, val_loader)
trainer.test(net_light_base, test_loader)
```

```
batch_x.shape: torch.Size([8, 10])
batch_y.shape: torch.Size([8])
```

	Name	Type	Params	Mode	FLOPs	In sizes	Out sizes
0	layers	Sequential	15.9 K	train	250 K	[8, 10]	[8, 1]

```
Trainable params: 15.9 K
Non-trainable params: 0
Total params: 15.9 K
Total estimated model params size (MB): 0
Modules in train mode: 11
Modules in eval mode: 0
Total FLOPs: 250 K
```

```
       Validate metric              DataLoader 0

          hp_metric                32421.490234375
          val_loss                 32421.490234375



         Test metric                DataLoader 0

          hp_metric                 25781.0234375
          val_loss                  25781.0234375


[{'val_loss': 25781.0234375, 'hp_metric': 25781.0234375}]
```

43.3.3. Dataset for spotpython

`spotpython` handles the data set, which is added to the `fun_control` dictionary with the key `data_set` as follows:

```
from spotpython.hyperparameters.values import set_control_key_value
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
set_control_key_value(control_dict=fun_control,
                        key="data_set",
                        value=dataset,
                        replace=True)
print(len(dataset))
```

```
442
```

If the data set is in the `fun_control` dictionary, it is used to create a `LightDataModule` object. This object is used to create the data loaders for the training, validation, and test sets. Therefore, the following information must be provided in the `fun_control` dictionary:

- `data_set`: the data set
- `batch_size`: the batch size
- `num_workers`: the number of workers
- `test_size`: the size of the test set
- `test_seed`: the seed for the test set

```
from spotpython.utils.init import fun_control_init
import numpy as np
fun_control = fun_control_init(
    data_set=dataset,
    device="cpu",
    enable_progress_bar=False,
    num_workers=0,
    show_progress=True,
    test_size=0.4,
    test_seed=42,
    )
```

```
from spotpython.data.lightdatamodule import LightDataModule
dm = LightDataModule(
    dataset=fun_control["data_set"],
    batch_size=8,
    num_workers=fun_control["num_workers"],
    test_size=fun_control["test_size"],
    test_seed=fun_control["test_seed"],
)
dm.setup()
print(f"train_model(): Test set size: {len(dm.data_test)}")
print(f"train_model(): Train set size: {len(dm.data_train)}")
```

```
train_model(): Test set size: 177
train_model(): Train set size: 160
```

43.4. The LightDataModule

The steps described above are handled by the `LightDataModule` class. This class is used to create the data loaders for the training, validation, and test sets. The `LightDataModule` class is part of the `spotpython` package. The `LightDataModule` class provides the following methods:

- `prepare_data()`: This method is used to prepare the data set.
- `setup()`: This method is used to create the data loaders for the training, validation, and test sets.
- `train_dataloader()`: This method is used to return the data loader for the training set.
- `val_dataloader()`: This method is used to return the data loader for the validation set.

- `test_dataloader()`: This method is used to return the data loader for the test set.
- `predict_dataloader()`: This method is used to return the data loader for the prediction set.

43.4.1. The `prepare_data()` Method

The `prepare_data()` method is used to prepare the data set. This method is called only once and on a single process. It can be used to download the data set. In our case, the data set is already available, so this method uses a simple `pass` statement.

43.4.2. The `setup()` Method

Splits the data for use in training, validation, and testing. It uses `torch.utils.data.random_split()` to split the data. Splitting is based on the `test_size` and `test_seed`. The `test_size` can be a float or an int.

43.4.2.1. Determine the Sizes of the Data Sets

```
from torch.utils.data import random_split
data_full = dataset
test_size = fun_control["test_size"]
test_seed=fun_control["test_seed"]
# if test_size is float, then train_size is 1 - test_size
if isinstance(test_size, float):
    full_train_size = round(1.0 - test_size, 2)
    val_size = round(full_train_size * test_size, 2)
    train_size = round(full_train_size - val_size, 2)
else:
    # if test_size is int, then train_size is len(data_full) - test_size
    full_train_size = len(data_full) - test_size
    val_size = int(full_train_size * test_size / len(data_full))
    train_size = full_train_size - val_size

print(f"LightDataModule setup(): full_train_size: {full_train_size}")
print(f"LightDataModule setup(): val_size: {val_size}")
print(f"LightDataModule setup(): train_size: {train_size}")
print(f"LightDataModule setup(): test_size: {test_size}")
```

```
LightDataModule setup(): full_train_size: 0.6
LightDataModule setup(): val_size: 0.24
```

```
LightDataModule setup(): train_size: 0.36
LightDataModule setup(): test_size: 0.4
```

`stage` is used to define the data set to be returned. The `stage` can be `None`, `fit`, `test`, or `predict`. If `stage` is `None`, the method returns the training (`fit`), testing (`test`) and prediction (`predict`) data sets.

43.4.2.2. Stage "fit"

```
stage = "fit"
if stage == "fit" or stage is None:
    generator_fit = torch.Generator().manual_seed(test_seed)
    data_train, data_val, _ = random_split(data_full, [train_size, val_size, test_size
print(f"LightDataModule setup(): Train set size: {len(data_train)}")
print(f"LightDataModule setup(): Validation set size: {len(data_val)}")
```

```
LightDataModule setup(): Train set size: 160
LightDataModule setup(): Validation set size: 106
```

43.4.2.3. Stage "test"

```
stage = "test"
if stage == "test" or stage is None:
    generator_test = torch.Generator().manual_seed(test_seed)
    data_test, _ = random_split(data_full, [test_size, full_train_size], generator=gen
print(f"LightDataModule setup(): Test set size: {len(data_test)}")
# Set batch size for DataLoader
batch_size = 5
# Create DataLoader
from torch.utils.data import DataLoader
dataloader = DataLoader(data_test, batch_size=batch_size, shuffle=False)
# Iterate over the data in the DataLoader
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
LightDataModule setup(): Test set size: 177
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0562, -0.0446, -0.0579, -0.0080,  0.0521,  0.0491,  0.0560, -
0.0214,
         -0.0283,  0.0445],
        [ 0.0018, -0.0446, -0.0709, -0.0229, -0.0016, -0.0010,  0.0266, -
0.0395,
         -0.0225,  0.0072],
        [-0.0527, -0.0446,  0.0542, -0.0263, -0.0552, -0.0339, -
0.0139, -0.0395,
         -0.0741, -0.0591],
        [ 0.0054, -0.0446, -0.0482, -0.0126,  0.0012, -0.0066,  0.0634, -
0.0395,
         -0.0514, -0.0591],
        [-0.0527, -0.0446, -0.0094, -0.0057,  0.0397,  0.0447,  0.0266, -
0.0026,
         -0.0181, -0.0135]])
Targets: tensor([158.,  49., 142.,  96.,  59.])
```

43.4.2.4. Stage "predict"

Prediction and testing use the same data set.

```python
stage = "predict"
if stage == "predict" or stage is None:
    generator_predict = torch.Generator().manual_seed(test_seed)
    data_predict, _ = random_split(
        data_full, [test_size, full_train_size], generator=generator_predict
    )
print(f"LightDataModule setup(): Predict set size: {len(data_predict)}")
# Set batch size for DataLoader
batch_size = 5
# Create DataLoader
from torch.utils.data import DataLoader
dataloader = DataLoader(data_predict, batch_size=batch_size, shuffle=False)
# Iterate over the data in the DataLoader
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
```

```
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
LightDataModule setup(): Predict set size: 177
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0562, -0.0446, -0.0579, -0.0080,  0.0521,  0.0491,  0.0560, -
0.0214,
         -0.0283,  0.0445],
        [ 0.0018, -0.0446, -0.0709, -0.0229, -0.0016, -0.0010,  0.0266, -
0.0395,
         -0.0225,  0.0072],
        [-0.0527, -0.0446,  0.0542, -0.0263, -0.0552, -0.0339, -
0.0139, -0.0395,
         -0.0741, -0.0591],
        [ 0.0054, -0.0446, -0.0482, -0.0126,  0.0012, -0.0066,  0.0634, -
0.0395,
         -0.0514, -0.0591],
        [-0.0527, -0.0446, -0.0094, -0.0057,  0.0397,  0.0447,  0.0266, -
0.0026,
         -0.0181, -0.0135]])
Targets: tensor([158.,  49., 142.,  96.,  59.])
```

43.4.3. The `train_dataloader()` Method

Returns the training dataloader, i.e., a Pytorch DataLoader instance using the training dataset. It simply returns a DataLoader with the `data_train` set that was created in the `setup()` method as described in Section 43.4.2.2.

```
def train_dataloader(self) -> DataLoader:
    return DataLoader(self.data_train, batch_size=self.batch_size, num_workers=self.nu
```

The `train_dataloader()` method can be used as follows:

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
```

```
data_module.setup()
print(f"Training set size: {len(data_module.data_train)}")
dl = data_module.train_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Training set size: 160
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0708,  0.0507, -0.0310,  0.0219, -0.0373, -0.0470,  0.0339, -
0.0395,
         -0.0150, -0.0011],
        [ 0.0344,  0.0507, -0.0299,  0.0047,  0.0934,  0.0870,  0.0339, -
0.0026,
          0.0241, -0.0384],
        [ 0.0090, -0.0446,  0.0552, -0.0057,  0.0576,  0.0447, -
0.0029,  0.0232,
          0.0557,  0.1066],
        [ 0.0417, -0.0446, -0.0644,  0.0356,  0.0122, -0.0580,  0.1812, -
0.0764,
         -0.0006, -0.0508],
        [ 0.0381,  0.0507,  0.0164,  0.0219,  0.0397,  0.0450, -
0.0434,  0.0712,
          0.0498,  0.0155]])
Targets: tensor([ 66.,  69., 173., 170., 212.])
```

43.4.4. The `val_dataloader()` Method

Returns the validation dataloader, i.e., a Pytorch DataLoader instance using the validation dataset. It simply returns a DataLoader with the `data_val` set that was created in the `setup()` method as desccribed in Section 43.4.2.2.

```
def val_dataloader(self) -> DataLoader:
    return DataLoader(self.data_val, batch_size=self.batch_size, num_workers=self.num_
```

The `val_dataloader()` method can be used as follows:

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Validation set size: {len(data_module.data_val)}")
dl = data_module.val_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Validation set size: 106
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0163, -0.0446,  0.0736, -0.0412, -0.0043, -0.0135, -
0.0139, -0.0011,
          0.0429,  0.0445],
        [ 0.0453, -0.0446,  0.0714,  0.0012, -0.0098, -0.0010,  0.0155, -
0.0395,
         -0.0412, -0.0715],
        [ 0.0308,  0.0507,  0.0326,  0.0494, -0.0401, -0.0436, -
0.0692,  0.0343,
          0.0630,  0.0031],
        [ 0.0235,  0.0507, -0.0396, -0.0057, -0.0484, -0.0333,  0.0118, -
0.0395,
         -0.1016, -0.0674],
        [-0.0091,  0.0507,  0.0013, -0.0022,  0.0796,  0.0701,  0.0339, -
0.0026,
          0.0267,  0.0818]])
Targets: tensor([275., 141., 208.,  78., 142.])
```

43.4.5. The `test_dataloader()` Method

Returns the test dataloader, i.e., a Pytorch DataLoader instance using the test dataset. It simply returns a DataLoader with the `data_test` set that was created in the `setup()` method as described in Section 43.4.2.3.

```
def test_dataloader(self) -> DataLoader:
    return DataLoader(self.data_test, batch_size=self.batch_size, num_workers=self.num_workers)
```

The `test_dataloader()` method can be used as follows:

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Test set size: {len(data_module.data_test)}")
dl = data_module.test_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Test set size: 177
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0562, -0.0446, -0.0579, -0.0080,  0.0521,  0.0491,  0.0560, -
0.0214,
         -0.0283,  0.0445],
        [ 0.0018, -0.0446, -0.0709, -0.0229, -0.0016, -0.0010,  0.0266, -
0.0395,
         -0.0225,  0.0072],
        [-0.0527, -0.0446,  0.0542, -0.0263, -0.0552, -0.0339, -
0.0139, -0.0395,
         -0.0741, -0.0591],
```

```
        [ 0.0054, -0.0446, -0.0482, -0.0126,  0.0012, -0.0066,  0.0634, -
0.0395,
         -0.0514, -0.0591],
        [-0.0527, -0.0446, -0.0094, -0.0057,  0.0397,  0.0447,  0.0266, -
0.0026,
         -0.0181, -0.0135]])
Targets: tensor([158.,  49., 142.,  96.,  59.])
```

43.4.6. The `predict_dataloader()` Method

Returns the prediction dataloader, i.e., a Pytorch DataLoader instance using the prediction dataset. It simply returns a DataLoader with the `data_predict` set that was created in the `setup()` method as described in Section 43.4.2.4.

> **Warning**
>
> The `batch_size` is set to the length of the `data_predict` set.

```
def predict_dataloader(self) -> DataLoader:
    return DataLoader(self.data_predict, batch_size=len(self.data_predict), num_worker
```

The `predict_dataloader()` method can be used as follows:

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Test set size: {len(data_module.data_predict)}")
dl = data_module.predict_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Test set size: 177
```

```
Batch Size: 177
Inputs Shape: torch.Size([177, 10])
Targets Shape: torch.Size([177])
---------------
Inputs: tensor([[ 0.0562, -0.0446, -0.0579,  ..., -0.0214, -0.0283,  0.0445],
        [ 0.0018, -0.0446, -0.0709,  ..., -0.0395, -0.0225,  0.0072],
        [-0.0527, -0.0446,  0.0542,  ..., -0.0395, -0.0741, -0.0591],
        ...,
        [ 0.0090, -0.0446, -0.0321,  ..., -0.0764, -0.0119, -0.0384],
        [-0.0273, -0.0446, -0.0666,  ..., -0.0395, -0.0358, -0.0094],
        [ 0.0817,  0.0507,  0.0067,  ...,  0.0919,  0.0547,  0.0072]])
Targets: tensor([158.,  49., 142.,  96.,  59.,  74., 137., 136.,  39.,  66., 310., 198.,
        235., 116.,  55., 177.,  59., 246.,  53., 135.,  88., 198., 186., 217.,
         51., 118., 153., 180.,  51., 229.,  84.,  72., 237., 142., 185.,  91.,
         88., 148., 179., 144.,  25.,  89.,  42.,  60., 124., 170., 215., 263.,
        178., 245., 202.,  97., 321.,  71., 123., 220., 132., 243.,  61., 102.,
        187.,  70., 242., 134.,  63.,  72.,  88., 219., 127., 146., 122., 143.,
        220., 293.,  59., 317.,  60., 140.,  65., 277.,  90.,  96., 109., 190.,
         90.,  52., 160., 233., 230., 175.,  68., 272., 144.,  70.,  68., 163.,
         71.,  93., 263., 118., 220.,  90., 232., 120., 163.,  88.,  85.,  52.,
        181., 232., 212., 332.,  81., 214., 145., 268., 115.,  93.,  64., 156.,
        128., 200., 281., 103., 220.,  66.,  48., 246.,  42., 150., 125., 109.,
        129.,  97., 265.,  97., 173., 216., 237., 121.,  42., 151.,  31.,  68.,
        137., 221., 283., 124., 243., 150.,  69., 306., 182., 252., 132., 258.,
        121., 110., 292., 101., 275., 141., 208.,  78., 142., 185., 167., 258.,
        144.,  89., 225., 140., 303., 236.,  87.,  77., 131.])
```

43.5. Using the LightDataModule in the train_model() Method

First, a LightDataModule object is created and the setup() method is called.

```
dm = LightDataModule(
    dataset=fun_control["data_set"],
    batch_size=config["batch_size"],
    num_workers=fun_control["num_workers"],
    test_size=fun_control["test_size"],
    test_seed=fun_control["test_seed"],
)
dm.setup()
```

Then, the Trainer is initialized.

```
# Init trainer
trainer = L.Trainer(
    default_root_dir=os.path.join(fun_control["CHECKPOINT_PATH"], config_id),
    max_epochs=model.hparams.epochs,
    accelerator=fun_control["accelerator"],
    devices=fun_control["devices"],
    logger=TensorBoardLogger(
        save_dir=fun_control["TENSORBOARD_PATH"],
        version=config_id,
        default_hp_metric=True,
        log_graph=fun_control["log_graph"],
    ),
    callbacks=[
        EarlyStopping(monitor="val_loss", patience=config["patience"], mode="min", str
    ],
    enable_progress_bar=enable_progress_bar,
)
```

Next, the `fit()` method is called to train the model.

```
# Pass the datamodule as arg to trainer.fit to override model hooks :)
trainer.fit(model=model, datamodule=dm)
```

Finally, the `validate()` method is called to validate the model. The `validate()` method returns the validation loss.

```
# Test best model on validation and test set
# result = trainer.validate(model=model, datamodule=dm, ckpt_path="last")
result = trainer.validate(model=model, datamodule=dm)
# unlist the result (from a list of one dict)
result = result[0]
return result["val_loss"]
```

43.6. Further Information

43.6.1. Preprocessing

Preprocessing is handled by `Lightning` and `PyTorch`. It is described in the LIGHTNINGDATAMODULE documentation. Here you can find information about the `transforms` methods.

44. Hyperparameter Tuning with spotpython and PyTorch Lightning for the Diabetes Data Set

In this section, we will show how spotpython can be integrated into the PyTorch Lightning training workflow for a regression task. It demonstrates how easy it is to use spotpython to tune hyperparameters for a PyTorch Lightning model.

44.1. The Basic Setting

```
import os
from math import inf
import warnings
warnings.filterwarnings("ignore")
```

After importing the necessary libraries, the fun_control dictionary is set up via the fun_control_init function. The fun_control dictionary contains

- PREFIX: a unique identifier for the experiment
- fun_evals: the number of function evaluations
- max_time: the maximum run time in minutes
- data_set: the data set. Here we use the Diabetes data set that is provided by spotpython.
- core_model_name: the class name of the neural network model. This neural network model is provided by spotpython.
- hyperdict: the hyperparameter dictionary. This dictionary is used to define the hyperparameters of the neural network model. It is also provided by spotpython.
- _L_in: the number of input features. Since the Diabetes data set has 10 features, _L_in is set to 10.
- _L_out: the number of output features. Since we want to predict a single value, _L_out is set to 1.

The HyperLight class is used to define the objective function fun. It connects the PyTorch and the spotpython methods and is provided by spotpython.

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_co
from spotpython.utils.eda import print_exp_table, print_res_table
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename

PREFIX="601"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)

fun = HyperLight().fun
```

```
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
```

The method `set_hyperparameter` allows the user to modify default hyperparameter settings. Here we modify some hyperparameters to keep the model small and to decrease the tuning time.

```
from spotpython.hyperparameters.values import set_hyperparameter
set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,7])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])

design_control = design_control_init(init_size=10)
print_exp_table(fun_control)
```

```
| name           | type   | default   |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|----------------------
|
| l1             | int    | 3         |     3   |   4     | transform_power_2_int |
| epochs         | int    | 4         |     3   |   7     | transform_power_2_int |
| batch_size     | int    | 4         |     4   |  11     | transform_power_2_int |
| act_fn         | factor | ReLU      |     0   |   5     | None                  |
| optimizer      | factor | SGD       |     0   |   2     | None                  |
| dropout_prob   | float  | 0.01      |     0   |   0.025 | None                  |
| lr_mult        | float  | 1.0       |     0.1 |  10     | None                  |
| patience       | int    | 2         |     2   |   3     | transform_power_2_int |
| batch_norm     | factor | 0         |     0   |   1     | None                  |
| initialization | factor | Default   |     0   |   4     | None                  |
```

Finally, a `Spot` object is created. Calling the method `run()` starts the hyperparameter tuning process.

```
S = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
S.run()
```

```
     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    202 K   train   826 M   [2048, 10]   [2048, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 23075.09765625, 'hp_metric': 23075.09765625}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   6.6 M   [64, 10]     [64, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 3466.626220703125, 'hp_metric': 3466.626220703125}



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 4775.9482421875, 'hp_metric': 4775.9482421875}



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M
```

```
train_model result: {'val_loss': 23975.263671875, 'hp_metric': 23975.263671875}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 22921.740234375, 'hp_metric': 22921.740234375}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 4060.3369140625, 'hp_metric': 4060.3369140625}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   52.5 M   [512, 10]    [512, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 20739.09375, 'hp_metric': 20739.09375}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 3873.864990234375, 'hp_metric': 3873.864990234375}



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   13.1 M   [128, 10]    [128, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 13.1 M
```

```
train_model result: {'val_loss': 20911.953125, 'hp_metric': 20911.953125}


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   413 M   [1024, 10]   [1024, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 413 M

train_model result: {'val_loss': 23951.484375, 'hp_metric': 23951.484375}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3849.736083984375, 'hp_metric': 3849.736083984375}
spotpython tuning: 3466.626220703125 [----------] 3.11%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 2951.248291015625, 'hp_metric': 2951.248291015625}
spotpython tuning: 2951.248291015625 [#---------] 6.14%. Success rate: 50.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 3231.650634765625, 'hp_metric': 3231.650634765625}
spotpython tuning: 2951.248291015625 [#---------] 9.60%. Success rate: 33.33%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M
```

```
train_model result: {'val_loss': 4529.53173828125, 'hp_metric': 4529.53173828125}
spotpython tuning: 2951.248291015625 [#---------] 13.76%. Success rate: 25.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 3681.19580078125, 'hp_metric': 3681.19580078125}
spotpython tuning: 2951.248291015625 [##--------] 16.89%. Success rate: 20.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 3431.694580078125, 'hp_metric': 3431.694580078125}
spotpython tuning: 2951.248291015625 [##--------] 20.69%. Success rate: 16.67%
```

```
     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3088.29150390625, 'hp_metric': 3088.29150390625}
spotpython tuning: 2951.248291015625 [##--------] 23.80%. Success rate: 14.29%



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   206 M   [512, 10]    [512, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M

train_model result: {'val_loss': 23220.31640625, 'hp_metric': 23220.31640625}
spotpython tuning: 2951.248291015625 [###-------] 26.78%. Success rate: 12.50%



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3073.880859375, 'hp_metric': 3073.880859375}
spotpython tuning: 2951.248291015625 [###-------] 30.18%. Success rate: 11.11%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 3184.729736328125, 'hp_metric': 3184.729736328125}
spotpython tuning: 2951.248291015625 [###-------] 34.02%. Success rate: 10.00%


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   103 M   [256, 10]    [256, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 103 M
```

```
train_model result: {'val_loss': 14631.6435546875, 'hp_metric': 14631.6435546875}
spotpython tuning: 2951.248291015625 [####------] 36.18%. Success rate: 9.09%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0   layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3149.479248046875, 'hp_metric': 3149.479248046875}
spotpython tuning: 2951.248291015625 [####------] 38.86%. Success rate: 8.33%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0   layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 3920.870849609375, 'hp_metric': 3920.870849609375}
spotpython tuning: 2951.248291015625 [####------] 42.49%. Success rate: 7.69%
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 8323.37890625, 'hp_metric': 8323.37890625}
spotpython tuning: 2951.248291015625 [#####-----] 45.53%. Success rate: 7.14%



    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3683.1748046875, 'hp_metric': 3683.1748046875}
spotpython tuning: 2951.248291015625 [#####-----] 48.72%. Success rate: 6.67%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 22978.158203125, 'hp_metric': 22978.158203125}
spotpython tuning: 2951.248291015625 [#####-----] 50.94%. Success rate: 6.25%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 20163.91015625, 'hp_metric': 20163.91015625}
spotpython tuning: 2951.248291015625 [#####-----] 53.45%. Success rate: 5.88%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M
```

```
train_model result: {'val_loss': 3537.125244140625, 'hp_metric': 3537.125244140625}
spotpython tuning: 2951.248291015625 [######----] 56.57%. Success rate: 5.56%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 9505.927734375, 'hp_metric': 9505.927734375}
spotpython tuning: 2951.248291015625 [######----] 59.62%. Success rate: 5.26%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 23010.025390625, 'hp_metric': 23010.025390625}
spotpython tuning: 2951.248291015625 [#######---] 65.12%. Success rate: 5.00%
```

```
     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 3156.76513671875, 'hp_metric': 3156.76513671875}
spotpython tuning: 2951.248291015625 [#######---] 66.82%. Success rate: 4.76%

Using spacefilling design as fallback.


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 2951.248291015625 [#######---] 67.95%. Success rate: 4.76%

Using spacefilling design as fallback.


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 2951.248291015625 [#######---] 69.03%. Success rate: 4.76%

Using spacefilling design as fallback.


     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K  train   104 M   [1024, 10]   [1024, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K  train   104 M   [1024, 10]   [1024, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 2951.248291015625 [#######---] 70.09%. Success rate: 4.76%

Using spacefilling design as fallback.
```

```
    Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


    Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 2951.248291015625 [#######---] 71.19%. Success rate: 4.76%

Using spacefilling design as fallback.


    Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 2951.248291015625 [#######---] 72.27%. Success rate: 4.76%

Using spacefilling design as fallback.


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 2951.248291015625 [#######---] 73.33%. Success rate: 4.76%

Using spacefilling design as fallback.


     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 2951.248291015625 [#######---] 74.44%. Success rate: 4.76%

Using spacefilling design as fallback.


     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M


train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 2951.248291015625 [########--] 75.48%. Success rate: 4.76%


Using spacefilling design as fallback.



     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M


train_model result: {'val_loss': nan, 'hp_metric': nan}


Using spacefilling design as fallback.



     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 2951.248291015625 [########--] 76.54%. Success rate: 4.76%

Using spacefilling design as fallback.


     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 2951.248291015625 [########--] 77.59%. Success rate: 4.76%

Using spacefilling design as fallback.


     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K  train   104 M   [1024, 10]   [1024, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   51.9 K  train   104 M   [1024, 10]   [1024, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 2951.248291015625 [########--] 78.65%. Success rate: 4.76%

Using spacefilling design as fallback.
```

```
    Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   104 M   [1024, 10]   [1024, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': 14037.4052734375, 'hp_metric': 14037.4052734375}
spotpython tuning: 2951.248291015625 [########--] 79.58%. Success rate: 4.55%



    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 4839.69775390625, 'hp_metric': 4839.69775390625}
spotpython tuning: 2951.248291015625 [########--] 82.35%. Success rate: 4.35%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   1.6 M   [16, 10]     [16, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 1.6 M

train_model result: {'val_loss': 3723.298095703125, 'hp_metric': 3723.298095703125}
spotpython tuning: 2951.248291015625 [#########-] 85.89%. Success rate: 4.17%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 4467.46923828125, 'hp_metric': 4467.46923828125}
spotpython tuning: 2951.248291015625 [#########-] 88.54%. Success rate: 4.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   1.6 M   [16, 10]     [16, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 1.6 M
```

```
train_model result: {'val_loss': 3218.08349609375, 'hp_metric': 3218.08349609375}
spotpython tuning: 2951.248291015625 [##########-] 91.20%. Success rate: 3.85%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0   layers   Sequential    202 K   train   25.8 M   [64, 10]     [64, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 18673.591796875, 'hp_metric': 18673.591796875}
spotpython tuning: 2951.248291015625 [###########] 95.46%. Success rate: 3.70%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0   layers   Sequential   53.2 K   train   6.6 M   [64, 10]     [64, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 22501.916015625, 'hp_metric': 22501.916015625}
spotpython tuning: 2951.248291015625 [###########] 97.96%. Success rate: 3.57%
```

```
    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   1.6 M   [16, 10]     [16, 1]




Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 1.6 M



train_model result: {'val_loss': 3526.53564453125, 'hp_metric': 3526.53564453125}
spotpython tuning: 2951.248291015625 [##########] 100.00%. Success rate: 3.45% Done...

Experiment saved to 601_res.pkl




<spotpython.spot.spot.Spot at 0x11956f0e0>
```

44.2. Looking at the Results

44.2.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
S.plot_progress()
```

44.2.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
print_res_table(S)
```

```
| name           | type    | default    |   lower |    upper | tuned              | transf
|----------------|--------|-----------|---------|---------|-------------------
|-----------------------|--------------|---------|
| l1             | int     | 3          |     3.0 |      4.0 | 4.0                | transf
| epochs         | int     | 4          |     3.0 |      7.0 | 6.0                | transf
| batch_size     | int     | 4          |     4.0 |     11.0 | 4.0                | transf
| act_fn         | factor  | ReLU       |     0.0 |      5.0 | Swish              | None
| optimizer      | factor  | SGD        |     0.0 |      2.0 | Adamax             | None
| dropout_prob   | float   | 0.01       |     0.0 |    0.025 | 0.025              | None
| lr_mult        | float   | 1.0        |     0.1 |     10.0 | 5.159276791225372  | None
| patience       | int     | 2          |     2.0 |      3.0 | 3.0                | transf
| batch_norm     | factor  | 0          |     0.0 |      1.0 | 0                  | None
| initialization | factor  | Default    |     0.0 |      4.0 | xavier_normal      | None
```

A histogram can be used to visualize the most important hyperparameters.

```
S.plot_importance(threshold=1.0)
```

```
S.plot_important_hyperparameter_contour(max_imp=3)
```

```
l1:  50.91556329257147
epochs:  0.20008042990698388
batch_size:  2.5922275993291755
act_fn:  46.61926797122009
optimizer:  48.155844478234236
dropout_prob:  0.7645543976398174
lr_mult:  0.0629238653975717
patience:  0.0629238653975717
batch_norm:  100.0
initialization:  43.844337411799785
```

Prediction Surface
Prediction
22000
20000
18000
16000
14000
12000
10000
8000
l1
3.0
3.2
3.4
3.6
3.8
4.0
batch_norm
0.0
0.2
0.4
0.6
0.8
1.0
Prediction Error Surface
Error
5600
5400
5200
5000
4800
4600
4400
Prediction Contour
22200
20400
18600
16800
15000
13200
11400
9600
7800
Error Contour
5550
5400
5250
5100
4950
4800
4650
4500
4350

Prediction Surface
Prediction
14000
12000
10000
8000
6000
3.0
3.2
3.4
3.6
3.8
4.0
I1
0.0
0.5
1.0
1.5
2.0
optimizer
Prediction Error Surface
Error
5500
5000
4500
4000
3500
3000
2500
Prediction Contour
optimizer
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
14800
13600
12400
11200
10000
8800
7600
6400
5200
Error Contour
5550
5100
4650
4200
3750
3300
2850
2400

44.2.3. Get the Tuned Architecture

```
import pprint
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(S)
pprint.pprint(config)
```

```
{'act_fn': Swish(),
 'batch_norm': False,
 'batch_size': 16,
 'dropout_prob': 0.025,
 'epochs': 64,
 'initialization': 'xavier_normal',
 'l1': 16,
 'lr_mult': 5.159276791225372,
```

```
 'optimizer': 'Adamax',
 'patience': 8}
```

44.2.4. Test on the full data set

```
# set the value of the key "TENSORBOARD_CLEAN" to True in the fun_control dictionary and use the upd
fun_control.update({"TENSORBOARD_CLEAN": True})
fun_control.update({"tensorboard_log": True})
```

```
from spotpython.light.testmodel import test_model
from spotpython.utils.init import get_feature_names

test_model(config, fun_control)
get_feature_names(fun_control)
```

```
     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M


         Test metric                DataLoader 0

          hp_metric               4396.61572265625
          val_loss                4396.61572265625


test_model result: {'val_loss': 4396.61572265625, 'hp_metric': 4396.61572265625}

['age',
 'sex',
 'bmi',
```

```
 'bp',
 's1_tc',
 's2_ldl',
 's3_hdl',
 's4_tch',
 's5_ltg',
 's6_glu']
```

44.3. Cross Validation With Lightning

- The `KFold` class from `sklearn.model_selection` is used to generate the folds for cross-validation.
- These mechanism is used to generate the folds for the final evaluation of the model.
- The `CrossValidationDataModule` class [SOURCE] is used to generate the folds for the hyperparameter tuning process.
- It is called from the `cv_model` function [SOURCE].

```
config
```

```
{'l1': 16,
 'epochs': 64,
 'batch_size': 16,
 'act_fn': Swish(),
 'optimizer': 'Adamax',
 'dropout_prob': 0.025,
 'lr_mult': 5.159276791225372,
 'patience': 8,
 'batch_norm': False,
 'initialization': 'xavier_normal'}
```

```
from spotpython.light.cvmodel import cv_model
fun_control.update({"k_folds": 2})
fun_control.update({"test_size": 0.6})
cv_model(config, fun_control)
```

```
k: 0
```

	Name	Type	Params	Mode	FLOPs	In sizes	Out sizes
0	layers	Sequential	202 K	train	6.5 M	[16, 10]	[16, 1]

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M


train_model result: {'val_loss': 4136.17431640625, 'hp_metric': 4136.17431640625}
k: 1



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M


train_model result: {'val_loss': 3132.096923828125, 'hp_metric': 3132.096923828125}


3634.1356201171875
```

44.4. Extending the Basic Setup

This basic setup can be adapted to user-specific needs in many ways. For example, the user can specify a custom data set, a custom model, or a custom loss function. The following sections provide more details on how to customize the hyperparameter tuning process. Before we proceed, we will provide an overview of the basic settings of the hyperparameter tuning process and explain the parameters used so far.

44.4.1. General Experiment Setup

To keep track of the different experiments, we use a PREFIX for the experiment name. The PREFIX is used to create a unique experiment name. The PREFIX is also used to create a unique TensorBoard folder, which is used to store the TensorBoard log files.

spotpython allows the specification of two different types of stopping criteria: first, the number of function evaluations (fun_evals), and second, the maximum run time in seconds (max_time). Here, we will set the number of function evaluations to infinity and the maximum run time to one minute.

max_time is set to one minute for demonstration purposes. For real experiments, this value should be increased. Note, the total run time may exceed the specified max_time, because the initial design is always evaluated, even if this takes longer than max_time.

44.4.2. Data Setup

Here, we have provided the Diabetes data set class, which is a subclass of torch.utils.data.Dataset. Data preprocessing is handled by Lightning and PyTorch. It is described in the LIGHTNINGDATAMODULE documentation.

The data splitting, i.e., the generation of training, validation, and testing data, is handled by Lightning.

44.4.3. Objective Function fun

The objective function fun from the class HyperLight [SOURCE] is selected next. It implements an interface from PyTorch's training, validation, and testing methods to spotpython.

44.4.4. Core-Model Setup

By using core_model_name = "light.regression.NNLinearRegressor", the spotpython model class NetLightRegression [SOURCE] from the light.regression module is selected.

44.4.5. Hyperdict Setup

For a given `core_model_name`, the corresponding hyperparameters are automatically loaded from the associated dictionary, which is stored as a JSON file. The JSON file contains hyperparameter type information, names, and bounds. For `spotpython` models, the hyperparameters are stored in the `LightHyperDict`, see [SOURCE] Alternatively, you can load a local hyper_dict. The `hyperdict` uses the default hyperparameter settings. These can be modified as described in Section 12.19.1.

44.4.6. Other Settings

There are several additional parameters that can be specified, e.g., since we did not specify a loss function, `mean_squared_error` is used, which is the default loss function. These will be explained in more detail in the following sections.

44.5. Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard, if the argument `tensorboard_log` to `fun_control_init()` is set to `True`. The Tensorboard log files are stored in the `runs` folder. To start Tensorboard, run the following command in the terminal:

```
tensorboard --logdir="runs/"
```

Further information can be found in the PyTorch Lightning documentation for Tensorboard.

44.6. Loading the Saved Experiment and Getting the Hyperparameters of the Tuned Model

To get the tuned hyperparameters as a dictionary, the `get_tuned_architecture` function can be used.

```
from spotpython.utils.file import load_result
spot_tuner = load_result(PREFIX=PREFIX)
config = get_tuned_architecture(spot_tuner)
config
```

```
Loaded experiment from 601_res.pkl
```

```
{'l1': 16,
 'epochs': 64,
 'batch_size': 16,
 'act_fn': Swish(),
 'optimizer': 'Adamax',
 'dropout_prob': 0.025,
 'lr_mult': 5.159276791225372,
 'patience': 8,
 'batch_norm': False,
 'initialization': 'xavier_normal'}
```

44.7. Using the spotgui

The spotgui [github] provides a convenient way to interact with the hyperparameter tuning process. To obtain the settings from Section 44.1, the spotgui can be started as shown in Figure 45.1.

Figure 44.1.: spotgui

44.8. Summary

This section presented an introduction to the basic setup of hyperparameter tuning with `spotpython` and `PyTorch` Lightning.

45. Early Stopping Explained: HPT with spotpython and PyTorch Lightning for the Diabetes Data Set

In this section, we will show how early stopping can be integrated into the PyTorch Lightning training workflow for a regression task. We will use the setting described in Chapter 44, i.e., the Diabetes data set, which is provided by spotpython, and the HyperLight class to define the objective function.

45.1. The Basic Setting

```
import os
from math import inf
import warnings
warnings.filterwarnings("ignore")
```

After importing the necessary libraries, the fun_control dictionary is set up via the fun_control_init function. The fun_control dictionary contains the same parameters as in Chapter 44, i.e., it contains the following parameters:

- PREFIX: a unique identifier for the experiment
- fun_evals: the number of function evaluations
- max_time: the maximum run time in minutes
- data_set: the data set. Here we use the Diabetes data set that is provided by spotpython.
- core_model_name: the class name of the neural network model. This neural network model is provided by spotpython.
- hyperdict: the hyperparameter dictionary. This dictionary is used to define the hyperparameters of the neural network model. It is also provided by spotpython.
- _L_in: the number of input features. Since the Diabetes data set has 10 features, _L_in is set to 10.
- _L_out: the number of output features. Since we want to predict a single value, _L_out is set to 1.

In addition, the `fun_control` dictionary contains the following parameters that are specific to the early-stopping mechanism:

- `divergence_threshold`: Stop training as soon as the monitored quantity becomes worse than this threshold.
- `check_finite`: When set True, stops training when the monitor becomes NaN or infinite.
- `stopping_threshold`: Stop training immediately once the monitored quantity reaches this threshold.

> **`divergence_threshold`**
>
> - We will set the `divergence_threshold` to 25,000, because good values are in a range around 15,000. This means that the training will be stopped if the monitored quantity becomes worse than this threshold.

Furthermore, the `patience` parameter can be used as a hyperparameter to control the early stopping mechanism. It defines how many validation checks with no improvement are allowed before training is stopped.

> **patience**
>
> It must be noted that the patience parameter counts the number of validation checks with no improvement, and not the number of training epochs. Therefore, with parameters check_val_every_n_epoch=10 and patience=3, the trainer will perform at least 40 training epochs before being stopped.

The `HyperLight` class is used to define the objective function `fun`. It connects the `PyTorch` and the `spotpython` methods and is provided by `spotpython`.

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_co
from spotpython.utils.eda import print_exp_table, print_res_table
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename

PREFIX="601-early_stopping"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
```

```
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    divergence_threshold=25_000,
    _L_in=10,
    _L_out=1)

fun = HyperLight().fun
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_11_04_21_11_53_0
Created spot_tensorboard_path: runs/spot_logs/601-early_stopping_maans08_2025-
11-04_21-11-53 for SummaryWriter()
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
```

The method `set_hyperparameter` allows the user to modify default hyperparameter settings. Here we modify some hyperparameters to keep the model small and to decrease the tuning time.

```
from spotpython.hyperparameters.values import set_hyperparameter
set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,7])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])

design_control = design_control_init(init_size=10)
print_exp_table(fun_control)
```

```
| name           | type   | default    |   lower |   upper | transform             |
|----------------|--------|------------|---------|---------|-----------------------
|
| l1             | int    | 3          |     3   |   4     | transform_power_2_int |
| epochs         | int    | 4          |     3   |   7     | transform_power_2_int |
| batch_size     | int    | 4          |     4   |  11     | transform_power_2_int |
| act_fn         | factor | ReLU       |     0   |   5     | None                  |
| optimizer      | factor | SGD        |     0   |   2     | None                  |
| dropout_prob   | float  | 0.01       |     0   |   0.025 | None                  |
```

```
| lr_mult        | float  | 1.0       |     0.1 |  10     | None                  |
| patience       | int    | 2         |     2   |   3     | transform_power_2_int |
| batch_norm     | factor | 0         |     0   |   1     | None                  |
| initialization | factor | Default   |     0   |   4     | None                  |
```

Finally, a `Spot` object is created. Calling the method `run()` starts the hyperparameter tuning process.

```
S = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
S.run()
```

```
     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    202 K   train   826 M   [2048, 10]   [2048, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 23075.09765625, 'hp_metric': 23075.09765625}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   6.6 M   [64, 10]     [64, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M
```

```
train_model result: {'val_loss': 3466.626220703125, 'hp_metric': 3466.626220703125}


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 44729112.0, 'hp_metric': 44729112.0}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 24005.33984375, 'hp_metric': 24005.33984375}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]
```

```
Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 22881.826171875, 'hp_metric': 22881.826171875}



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 4215.99072265625, 'hp_metric': 4215.99072265625}



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 52.5 M
```

```
train_model result: {'val_loss': 20590.513671875, 'hp_metric': 20590.513671875}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 3969.725830078125, 'hp_metric': 3969.725830078125}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   13.1 M   [128, 10]    [128, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 13.1 M

train_model result: {'val_loss': 20699.109375, 'hp_metric': 20699.109375}


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   413 M   [1024, 10]   [1024, 1]
```

```
Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 413 M

train_model result: {'val_loss': 23871.84765625, 'hp_metric': 23871.84765625}



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   206 M   [512, 10]    [512, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M

train_model result: {'val_loss': 23622.703125, 'hp_metric': 23622.703125}
spotpython tuning: 3466.626220703125 [----------] 2.12%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   103 M   [256, 10]    [256, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 103 M
```

```
train_model result: {'val_loss': 22528.830078125, 'hp_metric': 22528.830078125}
spotpython tuning: 3466.626220703125 [#---------] 5.13%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    202 K   train   826 M   [2048, 10]   [2048, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 22143.693359375, 'hp_metric': 22143.693359375}
spotpython tuning: 3466.626220703125 [#---------] 7.45%. Success rate: 0.00%



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 23968.31640625, 'hp_metric': 23968.31640625}
spotpython tuning: 3466.626220703125 [#---------] 9.25%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   13.1 M   [128, 10]    [128, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 13.1 M

train_model result: {'val_loss': 21933.705078125, 'hp_metric': 21933.705078125}
spotpython tuning: 3466.626220703125 [#---------] 11.80%. Success rate: 0.00%


    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   26.2 M   [256, 10]    [256, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 26.2 M

train_model result: {'val_loss': 23276.97265625, 'hp_metric': 23276.97265625}
spotpython tuning: 3466.626220703125 [#---------] 14.01%. Success rate: 0.00%


    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 21300.08984375, 'hp_metric': 21300.08984375}
spotpython tuning: 3466.626220703125 [##--------] 16.67%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 23967.828125, 'hp_metric': 23967.828125}
spotpython tuning: 3466.626220703125 [##--------] 18.66%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   52.5 M   [512, 10]    [512, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 52.5 M
```

```
train_model result: {'val_loss': 3467.520263671875, 'hp_metric': 3467.520263671875}
spotpython tuning: 3466.626220703125 [##--------] 22.06%. Success rate: 0.00%


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 24110.126953125, 'hp_metric': 24110.126953125}
spotpython tuning: 3466.626220703125 [###-------] 25.88%. Success rate: 0.00%


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 798487.5, 'hp_metric': 798487.5}
spotpython tuning: 3466.626220703125 [###-------] 28.61%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 21710.76171875, 'hp_metric': 21710.76171875}
spotpython tuning: 3466.626220703125 [###-------] 32.84%. Success rate: 0.00%



    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 4556.01904296875, 'hp_metric': 4556.01904296875}
spotpython tuning: 3466.626220703125 [####------] 36.59%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    205 K   train   6.5 M   [16, 10]     [16, 1]
```

```
Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 23870.84375, 'hp_metric': 23870.84375}
spotpython tuning: 3466.626220703125 [####------] 41.05%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   413 M   [1024, 10]   [1024, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 413 M

train_model result: {'val_loss': 23618.93359375, 'hp_metric': 23618.93359375}
spotpython tuning: 3466.626220703125 [####------] 44.92%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   6.5 M   [16, 10]     [16, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.5 M
```

```
train_model result: {'val_loss': 23767.53515625, 'hp_metric': 23767.53515625}
spotpython tuning: 3466.626220703125 [#####-----] 50.00%. Success rate: 0.00%



     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 24076.083984375, 'hp_metric': 24076.083984375}
spotpython tuning: 3466.626220703125 [#####-----] 53.81%. Success rate: 0.00%



     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   13.1 M   [128, 10]    [128, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 13.1 M

train_model result: {'val_loss': 22512.0, 'hp_metric': 22512.0}
spotpython tuning: 3466.626220703125 [######----] 58.40%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 23715.1171875, 'hp_metric': 23715.1171875}
spotpython tuning: 3466.626220703125 [#######---] 65.65%. Success rate: 0.00%



    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   26.2 M   [256, 10]    [256, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 26.2 M

train_model result: {'val_loss': 23959.41015625, 'hp_metric': 23959.41015625}
spotpython tuning: 3466.626220703125 [#######---] 70.44%. Success rate: 0.00%



    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]
```

```
Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 23177.203125, 'hp_metric': 23177.203125}
spotpython tuning: 3466.626220703125 [########--] 75.42%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   6.6 M   [64, 10]     [64, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 24018.24609375, 'hp_metric': 24018.24609375}
spotpython tuning: 3466.626220703125 [########--] 81.91%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   26.2 M   [256, 10]    [256, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 26.2 M
```

```
train_model result: {'val_loss': 23840.671875, 'hp_metric': 23840.671875}
spotpython tuning: 3466.626220703125 [##########-] 85.89%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0   layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 20081.701171875, 'hp_metric': 20081.701171875}
spotpython tuning: 3466.626220703125 [##########-] 92.04%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0   layers   Sequential    202 K   train   25.8 M   [64, 10]     [64, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 2976.613525390625, 'hp_metric': 2976.613525390625}
spotpython tuning: 2976.613525390625 [###########] 96.29%. Success rate: 4.00%
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   206 M   [512, 10]    [512, 1]




Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M



train_model result: {'val_loss': 23504.173828125, 'hp_metric': 23504.173828125}
spotpython tuning: 2976.613525390625 [##########] 100.00%. Success rate: 3.85% Done...

Experiment saved to 601-early_stopping_res.pkl




<spotpython.spot.spot.Spot at 0x1102230e0>
```

45.2. Looking at the Results

45.2.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performance values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
S.plot_progress(log_y=True)
```

45.2.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
print_res_table(S)
```

```
| name            | type    | default    |   lower |    upper | tuned                 | tra
|----------------|--------|-----------|---------|---------|----------------------
|-----------------------|--------------|---------|
| l1              | int     | 3          |     3.0 |     4.0 | 4.0                   | tra
| epochs          | int     | 4          |     3.0 |     7.0 | 7.0                   | tra
| batch_size      | int     | 4          |     4.0 |    11.0 | 6.0                   | tra
| act_fn          | factor  | ReLU       |     0.0 |     5.0 | LeakyReLU             | Nor
| optimizer       | factor  | SGD        |     0.0 |     2.0 | Adamax                | Nor
| dropout_prob    | float   | 0.01       |     0.0 |   0.025 | 0.004766429873889258  | Nor
| lr_mult         | float   | 1.0        |     0.1 |    10.0 | 2.6506652267065136    | Nor
| patience        | int     | 2          |     2.0 |     3.0 | 2.0                   | tra
| batch_norm      | factor  | 0          |     0.0 |     1.0 | 0                     | Nor
| initialization  | factor  | Default    |     0.0 |     4.0 | xavier_uniform        | Nor
```

A histogram can be used to visualize the most important hyperparameters.

```
S.plot_importance(threshold=1.0)
```

```
S.plot_important_hyperparameter_contour(max_imp=3)
```

```
l1:  0.08168308593570597
epochs:  0.08168308593570597
batch_size:  0.08168308593570597
act_fn:  0.08168308593570597
optimizer:  0.08168308593570597
dropout_prob:  0.5085867740038202
lr_mult:  0.08168308593570597
patience:  0.08168308593570597
batch_norm:  0.08168308593570597
initialization:  100.0
```

Prediction Surface
Prediction
1e7
dropout_prob
initialization
Prediction Error Surface
Error
1e6
dropout_prob
initialization
Prediction Contour
1e7
initialization
dropout_prob
Error Contour
1e6
initialization
dropout_prob

Prediction Surface
Prediction
1e7
initialization
l1
Prediction Error Surface
Error
1e6
initialization
l1
Prediction Contour
1e7
initialization
l1
Error Contour
1e6
initialization
l1

45.2.3. Get the Tuned Architecture

```
import pprint
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(S)
pprint.pprint(config)
```

```
{'act_fn': LeakyReLU(),
 'batch_norm': False,
 'batch_size': 64,
 'dropout_prob': 0.004766429873889258,
 'epochs': 128,
 'initialization': 'xavier_uniform',
 'l1': 16,
 'lr_mult': 2.6506652267065136,
```

```
 'optimizer': 'Adamax',
 'patience': 4}
```

45.2.4. Test on the full data set

```
fun_control.update({"TENSORBOARD_CLEAN": True})
fun_control.update({"tensorboard_log": True})
```

```
from spotpython.light.testmodel import test_model
from spotpython.utils.init import get_feature_names

test_model(config, fun_control)
get_feature_names(fun_control)
```

```
     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   25.8 M   [64, 10]     [64, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 25.8 M


         Test metric                DataLoader 0

          hp_metric               3278.120849609375
          val_loss                3278.120849609375


test_model result: {'val_loss': 3278.120849609375, 'hp_metric': 3278.120849609375}

['age',
 'sex',
 'bmi',
 'bp',
```

```
 's1_tc',
 's2_ldl',
 's3_hdl',
 's4_tch',
 's5_ltg',
 's6_glu']
```

45.3. Cross Validation With Lightning

- The `KFold` class from `sklearn.model_selection` is used to generate the folds for cross-validation.
- These mechanism is used to generate the folds for the final evaluation of the model.
- The `CrossValidationDataModule` class [SOURCE] is used to generate the folds for the hyperparameter tuning process.
- It is called from the `cv_model` function [SOURCE].

```
config
```

```
{'l1': 16,
 'epochs': 128,
 'batch_size': 64,
 'act_fn': LeakyReLU(),
 'optimizer': 'Adamax',
 'dropout_prob': 0.004766429873889258,
 'lr_mult': 2.6506652267065136,
 'patience': 4,
 'batch_norm': False,
 'initialization': 'xavier_uniform'}
```

```
from spotpython.light.cvmodel import cv_model
fun_control.update({"k_folds": 2})
fun_control.update({"test_size": 0.6})
cv_model(config, fun_control)
```

```
k: 0


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   25.8 M   [64, 10]     [64, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 25.8 M


train_model result: {'val_loss': 2898.214111328125, 'hp_metric': 2898.214111328125}
k: 1



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   25.8 M   [64, 10]     [64, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 25.8 M


train_model result: {'val_loss': 3062.371337890625, 'hp_metric': 3062.371337890625}


2980.292724609375
```

45.4. Extending the Basic Setup

This basic setup can be adapted to user-specific needs in many ways. For example, the user can specify a custom data set, a custom model, or a custom loss function. The following sections provide more details on how to customize the hyperparameter tuning process. Before we proceed, we will provide an overview of the basic settings of the hyperparameter tuning process and explain the parameters used so far.

45.4.1. General Experiment Setup

To keep track of the different experiments, we use a `PREFIX` for the experiment name. The `PREFIX` is used to create a unique experiment name. The `PREFIX` is also used to create a unique TensorBoard folder, which is used to store the TensorBoard log files.

`spotpython` allows the specification of two different types of stopping criteria: first, the number of function evaluations (`fun_evals`), and second, the maximum run time in seconds (`max_time`). Here, we will set the number of function evaluations to infinity and the maximum run time to one minute.

`max_time` is set to one minute for demonstration purposes. For real experiments, this value should be increased. Note, the total run time may exceed the specified `max_time`, because the initial design is always evaluated, even if this takes longer than `max_time`.

45.4.2. Data Setup

Here, we have provided the `Diabetes` data set class, which is a subclass of `torch.utils.data.Dataset`. Data preprocessing is handled by `Lightning` and `PyTorch`. It is described in the LIGHTNINGDATAMODULE documentation.

The data splitting, i.e., the generation of training, validation, and testing data, is handled by `Lightning`.

45.4.3. Objective Function `fun`

The objective function `fun` from the class `HyperLight` [SOURCE] is selected next. It implements an interface from `PyTorch`'s training, validation, and testing methods to `spotpython`.

45.4.4. Core-Model Setup

By using `core_model_name = "light.regression.NNLinearRegressor"`, the `spotpython` model class `NetLightRegression` [SOURCE] from the `light.regression` module is selected.

45.4.5. Hyperdict Setup

For a given `core_model_name`, the corresponding hyperparameters are automatically loaded from the associated dictionary, which is stored as a JSON file. The JSON file contains hyperparameter type information, names, and bounds. For `spotpython` models, the hyperparameters are stored in the `LightHyperDict`, see [SOURCE] Alternatively, you can load a local hyper_dict. The `hyperdict` uses the default hyperparameter settings. These can be modified as described in Section 12.19.1.

45.4.6. Other Settings

There are several additional parameters that can be specified, e.g., since we did not specify a loss function, `mean_squared_error` is used, which is the default loss function. These will be explained in more detail in the following sections.

45.5. Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard, if the argument `tensorboard_log` to `fun_control_init()` is set to `True`. The Tensorboard log files are stored in the `runs` folder. To start Tensorboard, run the following command in the terminal:

```
tensorboard --logdir="runs/"
```

Further information can be found in the PyTorch Lightning documentation for Tensorboard.

45.6. Loading the Saved Experiment and Getting the Hyperparameters of the Tuned Model

To get the tuned hyperparameters as a dictionary, the `get_tuned_architecture` function can be used.

```
from spotpython.utils.file import load_result
spot_tuner = load_result(PREFIX=PREFIX)
config = get_tuned_architecture(spot_tuner)
config
```

```
Loaded experiment from 601-early_stopping_res.pkl
```

```
{'l1': 16,
 'epochs': 128,
 'batch_size': 64,
 'act_fn': LeakyReLU(),
 'optimizer': 'Adamax',
 'dropout_prob': 0.004766429873889258,
 'lr_mult': 2.6506652267065136,
 'patience': 4,
 'batch_norm': False,
 'initialization': 'xavier_uniform'}
```

45.7. Using the spotgui

The spotgui [github] provides a convenient way to interact with the hyperparameter tuning process. To obtain the settings from Section 44.1, the spotgui can be started as shown in Figure 45.1.

Figure 45.1.: spotgui

45.8. Summary

This section presented an introduction to the basic setup of hyperparameter tuning with `spotpython` and `PyTorch` Lightning.

46. Hyperparameter Tuning with PyTorch Lightning and User Data Sets

In this section, we will show how user specfied data can be used for the `PyTorch` Lightning hyperparameter tuning workflow with `spotpython`.

46.1. Loading a User Specified Data Set

Using a user-specified data set is straightforward.

The user simply needs to provide a data set and loads is as a `spotpython` `CVSDataset()` class by specifying the path, filename, and target column.

Consider the following example, where the user has a data set stored in the `userData` directory. The data set is stored in a file named `data.csv`. The target column is named `target`. To show the data, it is loaded as a `pandas` data frame and the first 5 rows are displayed. This step is not necessary for the hyperparameter tuning process, but it is useful for understanding the data.

```
# load the csv data set as a pandas dataframe and dislay the first 5 rows
import pandas as pd
data = pd.read_csv("./userData/data.csv")
print(data.head())
```

```
        age       sex       bmi        bp        s1        s2        s3  \
0  0.038076  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.043401
1 -0.001882 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163  0.074412
2  0.085299  0.050680  0.044451 -0.005670 -0.045599 -0.034194 -0.032356
3 -0.089063 -0.044642 -0.011595 -0.036656  0.012191  0.024991 -0.036038
4  0.005383 -0.044642 -0.036385  0.021872  0.003935  0.015596  0.008142

         s4        s5        s6  target
0 -0.002592  0.019907 -0.017646   151.0
1 -0.039493 -0.068332 -0.092204    75.0
```

```
2 -0.002592  0.002861 -0.025930   141.0
3  0.034309  0.022688 -0.009362   206.0
4 -0.002592 -0.031988 -0.046641   135.0
```

Next, the data set is loaded as a `spotpython` `CSVDataset()` class. This step is necessary for the hyperparameter tuning process.

```
from spotpython.data.csvdataset import CSVDataset
import torch
data_set = CSVDataset(directory="./userData/",
                      filename="data.csv",
                      target_column="target",
                      feature_type=torch.float32,
                      target_type=torch.float32,
                      rmNA=True)
print(len(data_set))
```

```
442
```

The following step is not necessary for the hyperparameter tuning process, but it is useful for understanding the data. The data set is loaded as a `DataLoader` from `torch.utils.data` to check the data.

```
# Set batch size for DataLoader
batch_size = 5
# Create DataLoader
from torch.utils.data import DataLoader
dataloader = DataLoader(data_set, batch_size=batch_size, shuffle=False)

# Iterate over the data in the DataLoader
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
```

```
---------------
Inputs: tensor([[ 0.0381,  0.0507,  0.0617,  0.0219, -0.0442, -0.0348, -
0.0434, -0.0026,
          0.0199, -0.0176],
        [-0.0019, -0.0446, -0.0515, -0.0263, -0.0084, -0.0192,  0.0744, -
0.0395,
         -0.0683, -0.0922],
        [ 0.0853,  0.0507,  0.0445, -0.0057, -0.0456, -0.0342, -
0.0324, -0.0026,
          0.0029, -0.0259],
        [-0.0891, -0.0446, -0.0116, -0.0367,  0.0122,  0.0250, -
0.0360,  0.0343,
          0.0227, -0.0094],
        [ 0.0054, -0.0446, -0.0364,  0.0219,  0.0039,  0.0156,  0.0081, -
0.0026,
         -0.0320, -0.0466]])
Targets: tensor([151.,  75., 141., 206., 135.])
```

Similar to the setting from Section 44.1, the hyperparameter tuning setup is defined. Instead of using the `Diabetes` data set, the user data set is used. The `data_set` parameter is set to the user data set. The `fun_control` dictionary is set up via the `fun_control_init` function.

Note, that we have modified the `fun_evals` parameter to 12 and the `init_size` to 7 to reduce the computational time for this example. The `divergence_threshold` is set to 5,000, which is based on some pre-experiments with the user data set.

```
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_control_init)
from spotpython.utils.eda import print_res_table
from spotpython.hyperparameters.values import set_hyperparameter
from spotpython.spot import Spot

fun_control = fun_control_init(
    PREFIX="601",
    fun_evals=12,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    divergence_threshold=5_000,
    _L_in=10,
    _L_out=1)
```

```
design_control = design_control_init(init_size=7)

set_hyperparameter(fun_control, "initialization", ["Default"])

fun = HyperLight().fun

spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
```

```
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
```

```
res = spot_tuner.run()
print_res_table(spot_tuner)
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   409 K    [4, 10]      [4, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 409 K

train_model(): trainer.fit failed with exception: SparseAdam does not support dense gr
train_model result: {'val_loss': 23320.16015625, 'hp_metric': 23320.16015625}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   12.8 M   train   203 M    [8, 10]      [8, 1]
```

```
Trainable params: 12.8 M
Non-trainable params: 0
Total params: 12.8 M
Total estimated model params size (MB): 51
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 203 M

train_model result: {'val_loss': nan, 'hp_metric': nan}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   1.6 M    [4, 10]      [4, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 1.6 M

train_model result: {'val_loss': 22162.58984375, 'hp_metric': 22162.58984375}



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    3.2 M   train   51.0 M    [8, 10]      [8, 1]



Trainable params: 3.2 M
Non-trainable params: 0
Total params: 3.2 M
Total estimated model params size (MB): 12
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.0 M
```

```
train_model result: {'val_loss': 16100.7783203125, 'hp_metric': 16100.7783203125}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    802 K   train   12.8 M    [8, 10]      [8, 1]



Trainable params: 802 K
Non-trainable params: 0
Total params: 802 K
Total estimated model params size (MB): 3
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.8 M

train_model result: {'val_loss': nan, 'hp_metric': nan}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   50.9 M   train   203 M    [2, 10]      [2, 1]



Trainable params: 50.9 M
Non-trainable params: 0
Total params: 50.9 M
Total estimated model params size (MB): 203
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 203 M

train_model result: {'val_loss': nan, 'hp_metric': nan}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    807 K   train   25.6 M   [16, 10]     [16, 1]
```

```
Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.6 M

train_model result: {'val_loss': 24040.451171875, 'hp_metric': 24040.451171875}



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    3.2 M   train   51.0 M    [8, 10]      [8, 1]



Trainable params: 3.2 M
Non-trainable params: 0
Total params: 3.2 M
Total estimated model params size (MB): 12
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.0 M

train_model result: {'val_loss': 16582.51171875, 'hp_metric': 16582.51171875}
spotpython tuning: 16100.7783203125 [####------] 41.67%. Success rate: 0.00%



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    3.2 M   train   51.0 M    [8, 10]      [8, 1]



Trainable params: 3.2 M
Non-trainable params: 0
Total params: 3.2 M
Total estimated model params size (MB): 12
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.0 M
```

```
train_model result: {'val_loss': 22252.875, 'hp_metric': 22252.875}
spotpython tuning: 16100.7783203125 [#####-----] 50.00%. Success rate: 0.00%



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0   layers   Sequential    3.2 M   train   51.0 M    [8, 10]      [8, 1]



Trainable params: 3.2 M
Non-trainable params: 0
Total params: 3.2 M
Total estimated model params size (MB): 12
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.0 M


train_model result: {'val_loss': 14519.490234375, 'hp_metric': 14519.490234375}
spotpython tuning: 14519.490234375 [######----] 58.33%. Success rate: 33.33%



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0   layers   Sequential   12.8 M   train   50.9 M    [2, 10]      [2, 1]



Trainable params: 12.8 M
Non-trainable params: 0
Total params: 12.8 M
Total estimated model params size (MB): 51
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 50.9 M


train_model result: {'val_loss': 5972.04296875, 'hp_metric': 5972.04296875}
spotpython tuning: 5972.04296875 [#######---] 66.67%. Success rate: 50.00%
```

```
    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    205 K   train   3.2 M    [8, 10]      [8, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.2 M

train_model(): trainer.fit failed with exception: SparseAdam does not support dense gradients, pleas
train_model result: {'val_loss': 22818.576171875, 'hp_metric': 22818.576171875}
spotpython tuning: 5972.04296875 [########--] 75.00%. Success rate: 40.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   50.9 M   train   203 M    [2, 10]      [2, 1]



Trainable params: 50.9 M
Non-trainable params: 0
Total params: 50.9 M
Total estimated model params size (MB): 203
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 203 M

train_model result: {'val_loss': 6787.0361328125, 'hp_metric': 6787.0361328125}
spotpython tuning: 5972.04296875 [########--] 83.33%. Success rate: 33.33%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   50.9 M   train   203 M    [2, 10]      [2, 1]
```

```
Trainable params: 50.9 M
Non-trainable params: 0
Total params: 50.9 M
Total estimated model params size (MB): 203
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 203 M

train_model result: {'val_loss': 6273.56005859375, 'hp_metric': 6273.56005859375}
spotpython tuning: 5972.04296875 [#########-] 91.67%. Success rate: 28.57%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   50.9 M  train   203 M    [2, 10]      [2, 1]


Trainable params: 50.9 M
Non-trainable params: 0
Total params: 50.9 M
Total estimated model params size (MB): 203
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 203 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K  train   409 K    [4, 10]      [4, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 409 K
```

```
train_model result: {'val_loss': 21099.947265625, 'hp_metric': 21099.947265625}
spotpython tuning: 5972.04296875 [##########] 100.00%. Success rate: 28.57% Done...

Experiment saved to 601_res.pkl
| name           | type   | default   |   lower |   upper | tuned                | transform
|----------------|--------|-----------|---------|---------|---------------------
|------------------------|--------------|---------|
| l1             | int    | 3         |     3.0 |     8.0 | 7.0                  | transform_power_2_
| epochs         | int    | 4         |     4.0 |     9.0 | 7.0                  | transform_power_2_
| batch_size     | int    | 4         |     1.0 |     4.0 | 1.0                  | transform_power_2_
| act_fn         | factor | ReLU      |     0.0 |     5.0 | Sigmoid              | None
| optimizer      | factor | SGD       |     0.0 |    11.0 | Adagrad              | None
| dropout_prob   | float  | 0.01      |     0.0 |    0.25 | 0.06316793576530161 | None
| lr_mult        | float  | 1.0       |     0.1 |    10.0 | 9.004716312092247    | None
| patience       | int    | 2         |     2.0 |     6.0 | 3.0                  | transform_power_2_
| batch_norm     | factor | 0         |     0.0 |     1.0 | 1                    | None
| initialization | factor | Default   |     0.0 |     0.0 | Default              | None
l1:  2.9079631170943667
epochs:  0.07976513592982082
batch_size:  4.767854437417129
act_fn:  11.347665219718287
optimizer:  0.07976513592982082
dropout_prob:  100.0
lr_mult:  0.07976513592982082
patience:  0.07976513592982082
batch_norm:  0.07976513592982082
```

Prediction Surface
Prediction
16000
15000
14000
13000
12000
act_fn
dropout_prob
Prediction Error Surface
Error
2600
2500
2400
2300
2200
2100
2000
1900
Prediction Contour
dropout_prob
act_fn
16750
16000
15250
14500
13750
13000
12250
11500
Error Contour
2640
2520
2400
2280
2160
2040
1920
1800

Prediction Surface
Prediction
batch_size
dropout_prob
Prediction Error Surface
Error
batch_size
dropout_prob
Prediction Contour
dropout_prob
batch_size
Error Contour
dropout_prob
batch_size

46.2. Summary

This section showed how to use user-specified data sets for the hyperparameter tuning process with `spotpython`. The user needs to provide the data set and load it as a `spotpython CSVDataset()` class.

47. Hyperparameter Tuning with PyTorch Lightning and User Models

In this section, we will show how a user defined model can be used for the `PyTorch` Lightning hyperparameter tuning workflow with `spotpython`.

47.1. Using a User Specified Model

As templates, we provide the following three files that allow the user to specify a model in the `/userModel` directory:

- `my_regressor.py`, see Section 47.2.4
- `my_hyperdict.json`, see Section 47.2.3
- `my_hyperdict.py`, see Section 47.2.2.

The `my_regressor.py` file contains the model class, which is a subclass of `nn.Module`. The `my_hyperdict.json` file contains the hyperparameter settings as a dictionary, which are loaded via the `my_hyperdict.py` file.

Note, that we have to add the path to the `userModel` directory to the `sys.path` list as shown below. The `divergence_threshold` is set to 3,000, which is based on some pre-experiments with the user data set.

```
import sys
sys.path.insert(0, './userModel')
import my_regressor
import my_hyper_dict
from spotpython.hyperparameters.values import add_core_model_to_fun_control

from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, design_control_init)
from spotpython.utils.eda import print_res_table
from spotpython.hyperparameters.values import set_hyperparameter
from spotpython.spot import Spot
```

```
fun_control = fun_control_init(
    PREFIX="601-user-model",
    fun_evals=inf,
    max_time=1,
    data_set = Diabetes(),
    divergence_threshold=3_000,
    _L_in=10,
    _L_out=1)

add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=my_regressor.MyRegressor,
                              hyper_dict=my_hyper_dict.MyHyperDict)

design_control = design_control_init(init_size=7)

fun = HyperLight().fun

spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
```

```
res = spot_tuner.run()
print_res_table(spot_tuner)
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential      353   train   2.6 K    [4, 10]      [4, 1]



Trainable params: 353
Non-trainable params: 0
Total params: 353
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 2.6 K

train_model(): trainer.fit failed with exception: SparseAdam does not support dense gr
train_model result: {'val_loss': 23988.361328125, 'hp_metric': 23988.361328125}
```

```
    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   44.0 K   train   697 K    [8, 10]      [8, 1]



Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 697 K

train_model result: {'val_loss': nan, 'hp_metric': nan}



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    1.0 K   train   7.7 K    [4, 10]      [4, 1]



Trainable params: 1.0 K
Non-trainable params: 0
Total params: 1.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 7.7 K

train_model result: {'val_loss': 3563.595703125, 'hp_metric': 3563.595703125}



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   11.8 K   train   184 K    [8, 10]      [8, 1]
```

```
Trainable params: 11.8 K
Non-trainable params: 0
Total params: 11.8 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 184 K

train_model result: {'val_loss': 17523.4453125, 'hp_metric': 17523.4453125}



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    3.3 K   train   51.5 K    [8, 10]      [8, 1]



Trainable params: 3.3 K
Non-trainable params: 0
Total params: 3.3 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 51.5 K

train_model result: {'val_loss': nan, 'hp_metric': nan}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]



Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K
```

```
train_model result: {'val_loss': 3854.307861328125, 'hp_metric': 3854.307861328125}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    3.3 K   train   102 K   [16, 10]     [16, 1]


Trainable params: 3.3 K
Non-trainable params: 0
Total params: 3.3 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 102 K

train_model result: {'val_loss': 23849.71875, 'hp_metric': 23849.71875}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    3.3 K   train   25.7 K    [4, 10]      [4, 1]


Trainable params: 3.3 K
Non-trainable params: 0
Total params: 3.3 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 25.7 K

train_model result: {'val_loss': 15952.798828125, 'hp_metric': 15952.798828125}
spotpython tuning: 3563.595703125 [----------] 0.84%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]
```

```
Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K

train_model result: {'val_loss': 3994.522216796875, 'hp_metric': 3994.522216796875}
spotpython tuning: 3563.595703125 [----------] 2.42%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.0 K   train   7.7 K    [4, 10]      [4, 1]


Trainable params: 1.0 K
Non-trainable params: 0
Total params: 1.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 7.7 K

train_model result: {'val_loss': 4304.8701171875, 'hp_metric': 4304.8701171875}
spotpython tuning: 3563.595703125 [----------] 3.58%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   44.0 K   train   174 K    [2, 10]      [2, 1]


Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 174 K
```

```
train_model result: {'val_loss': 20643.490234375, 'hp_metric': 20643.490234375}
spotpython tuning: 3563.595703125 [#---------] 5.00%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]



Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K

train_model result: {'val_loss': 4006.954833984375, 'hp_metric': 4006.954833984375}
spotpython tuning: 3563.595703125 [#---------] 6.48%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]



Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K

train_model result: {'val_loss': 7049.56787109375, 'hp_metric': 7049.56787109375}
spotpython tuning: 3563.595703125 [#---------] 8.08%. Success rate: 0.00%
```

```
     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]



Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K

train_model result: {'val_loss': 3479.590576171875, 'hp_metric': 3479.590576171875}
spotpython tuning: 3479.590576171875 [#---------] 9.99%. Success rate: 14.29%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]



Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K

train_model result: {'val_loss': 3936.7490234375, 'hp_metric': 3936.7490234375}
spotpython tuning: 3479.590576171875 [#---------] 11.52%. Success rate: 12.50%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.0 K   train   7.7 K    [4, 10]      [4, 1]
```

```
Trainable params: 1.0 K
Non-trainable params: 0
Total params: 1.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 7.7 K

train_model result: {'val_loss': 3786.134033203125, 'hp_metric': 3786.134033203125}
spotpython tuning: 3479.590576171875 [#---------] 12.57%. Success rate: 11.11%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]


Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K

train_model result: {'val_loss': 3156.9912109375, 'hp_metric': 3156.9912109375}
spotpython tuning: 3156.9912109375 [#---------] 13.88%. Success rate: 20.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]


Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K
```

```
train_model result: {'val_loss': 4475.7841796875, 'hp_metric': 4475.7841796875}
spotpython tuning: 3156.9912109375 [##--------] 15.14%. Success rate: 18.18%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]



Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K


train_model result: {'val_loss': 3225.822509765625, 'hp_metric': 3225.822509765625}
spotpython tuning: 3156.9912109375 [##--------] 17.12%. Success rate: 16.67%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.0 K   train   7.7 K    [4, 10]      [4, 1]



Trainable params: 1.0 K
Non-trainable params: 0
Total params: 1.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 7.7 K


train_model result: {'val_loss': 4072.851318359375, 'hp_metric': 4072.851318359375}
spotpython tuning: 3156.9912109375 [##--------] 18.17%. Success rate: 15.38%
```

```
    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    1.0 K   train   7.7 K    [4, 10]      [4, 1]



Trainable params: 1.0 K
Non-trainable params: 0
Total params: 1.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 7.7 K

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    1.0 K   train   31.0 K   [16, 10]     [16, 1]



Trainable params: 1.0 K
Non-trainable params: 0
Total params: 1.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 31.0 K

train_model result: {'val_loss': 23563.7734375, 'hp_metric': 23563.7734375}
spotpython tuning: 3156.9912109375 [##--------] 19.99%. Success rate: 15.38%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]
```

```
Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K

train_model result: {'val_loss': 3087.028564453125, 'hp_metric': 3087.028564453125}
spotpython tuning: 3087.028564453125 [##--------] 22.19%. Success rate: 21.43%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]



Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K

train_model result: {'val_loss': 9860.0400390625, 'hp_metric': 9860.0400390625}
spotpython tuning: 3087.028564453125 [##--------] 24.35%. Success rate: 20.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]



Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K
```

```
train_model result: {'val_loss': 3075.575439453125, 'hp_metric': 3075.575439453125}
spotpython tuning: 3075.575439453125 [###-------] 26.67%. Success rate: 25.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]



Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K

train_model result: {'val_loss': 4069.6005859375, 'hp_metric': 4069.6005859375}
spotpython tuning: 3075.575439453125 [###-------] 28.69%. Success rate: 23.53%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.0 K   train   7.7 K    [4, 10]      [4, 1]



Trainable params: 1.0 K
Non-trainable params: 0
Total params: 1.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 7.7 K

train_model result: {'val_loss': 4133.14013671875, 'hp_metric': 4133.14013671875}
spotpython tuning: 3075.575439453125 [###-------] 30.38%. Success rate: 22.22%
```

```
    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]



Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential      353   train   2.6 K    [4, 10]      [4, 1]



Trainable params: 353
Non-trainable params: 0
Total params: 353
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 2.6 K

train_model result: {'val_loss': 23675.81640625, 'hp_metric': 23675.81640625}
spotpython tuning: 3075.575439453125 [###-------] 32.85%. Success rate: 22.22%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    169 K   train   1.4 M    [4, 10]      [4, 1]
```

```
Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 1.4 M

train_model result: {'val_loss': 3314.97900390625, 'hp_metric': 3314.97900390625}
spotpython tuning: 3075.575439453125 [###-------] 34.99%. Success rate: 21.05%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   1.4 M    [4, 10]      [4, 1]


Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 1.4 M

train_model result: {'val_loss': 3199.306396484375, 'hp_metric': 3199.306396484375}
spotpython tuning: 3075.575439453125 [####------] 36.79%. Success rate: 20.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   2.7 M    [8, 10]      [8, 1]


Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 2.7 M
```

```
train_model result: {'val_loss': 4840.58740234375, 'hp_metric': 4840.58740234375}
spotpython tuning: 3075.575439453125 [####------] 38.34%. Success rate: 19.05%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0   layers   Sequential    1.0 K   train   3.9 K    [2, 10]      [2, 1]



Trainable params: 1.0 K
Non-trainable params: 0
Total params: 1.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 3.9 K


train_model result: {'val_loss': 4428.2646484375, 'hp_metric': 4428.2646484375}
spotpython tuning: 3075.575439453125 [####------] 39.98%. Success rate: 18.18%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0   layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]



Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K


train_model result: {'val_loss': 4892.3759765625, 'hp_metric': 4892.3759765625}
spotpython tuning: 3075.575439453125 [####------] 42.55%. Success rate: 17.39%
```

```
    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   44.0 K  train   174 K    [2, 10]      [2, 1]



Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 174 K

train_model result: {'val_loss': 3043.216064453125, 'hp_metric': 3043.216064453125}
spotpython tuning: 3043.216064453125 [#####-----] 45.32%. Success rate: 20.83%



    Name     Type        Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential   11.8 K  train   46.2 K    [2, 10]      [2, 1]



Trainable params: 11.8 K
Non-trainable params: 0
Total params: 11.8 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 46.2 K

train_model result: {'val_loss': 3677.52001953125, 'hp_metric': 3677.52001953125}
spotpython tuning: 3043.216064453125 [#####-----] 47.87%. Success rate: 20.00%



    Name     Type        Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    3.3 K  train   12.9 K    [2, 10]      [2, 1]
```

```
Trainable params: 3.3 K
Non-trainable params: 0
Total params: 3.3 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 12.9 K

train_model result: {'val_loss': 3510.701416015625, 'hp_metric': 3510.701416015625}
spotpython tuning: 3043.216064453125 [#####-----] 51.28%. Success rate: 19.23%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    3.3 K   train   12.9 K    [2, 10]      [2, 1]


Trainable params: 3.3 K
Non-trainable params: 0
Total params: 3.3 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 12.9 K

train_model result: {'val_loss': 3345.48681640625, 'hp_metric': 3345.48681640625}
spotpython tuning: 3043.216064453125 [#####-----] 53.92%. Success rate: 18.52%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    3.3 K   train   12.9 K    [2, 10]      [2, 1]


Trainable params: 3.3 K
Non-trainable params: 0
Total params: 3.3 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 12.9 K
```

```
train_model result: {'val_loss': 3664.25732421875, 'hp_metric': 3664.25732421875}
spotpython tuning: 3043.216064453125 [######----] 56.62%. Success rate: 17.86%



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    3.3 K   train   12.9 K    [2, 10]      [2, 1]



Trainable params: 3.3 K
Non-trainable params: 0
Total params: 3.3 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 12.9 K

train_model result: {'val_loss': 3435.54296875, 'hp_metric': 3435.54296875}
spotpython tuning: 3043.216064453125 [######----] 59.41%. Success rate: 17.24%



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    3.3 K   train   12.9 K    [2, 10]      [2, 1]



Trainable params: 3.3 K
Non-trainable params: 0
Total params: 3.3 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 12.9 K

train_model result: {'val_loss': 3732.949951171875, 'hp_metric': 3732.949951171875}
spotpython tuning: 3043.216064453125 [######----] 62.22%. Success rate: 16.67%
```

```
    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    3.3 K   train   12.9 K    [2, 10]      [2, 1]



Trainable params: 3.3 K
Non-trainable params: 0
Total params: 3.3 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 12.9 K

train_model result: {'val_loss': 3427.607421875, 'hp_metric': 3427.607421875}
spotpython tuning: 3043.216064453125 [#######---] 65.07%. Success rate: 16.13%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    1.0 K   train   3.9 K    [2, 10]      [2, 1]



Trainable params: 1.0 K
Non-trainable params: 0
Total params: 1.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 3.9 K

train_model result: {'val_loss': 5749.81884765625, 'hp_metric': 5749.81884765625}
spotpython tuning: 3043.216064453125 [#######---] 68.13%. Success rate: 15.62%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   44.0 K   train   174 K    [2, 10]      [2, 1]
```

```
Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 174 K

train_model result: {'val_loss': 4987.91845703125, 'hp_metric': 4987.91845703125}
spotpython tuning: 3043.216064453125 [#######---] 71.05%. Success rate: 15.15%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    169 K   train   676 K    [2, 10]      [2, 1]


Trainable params: 169 K
Non-trainable params: 0
Total params: 169 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 676 K

train_model result: {'val_loss': 4062.236083984375, 'hp_metric': 4062.236083984375}
spotpython tuning: 3043.216064453125 [#######---] 74.14%. Success rate: 14.71%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   44.0 K   train   174 K    [2, 10]      [2, 1]


Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 174 K
```

```
train_model result: {'val_loss': 3284.009521484375, 'hp_metric': 3284.009521484375}
spotpython tuning: 3043.216064453125 [########--] 77.08%. Success rate: 14.29%



     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  44.0 K   train   174 K    [2, 10]      [2, 1]



Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 174 K

train_model result: {'val_loss': 3233.1904296875, 'hp_metric': 3233.1904296875}
spotpython tuning: 3043.216064453125 [########--] 80.03%. Success rate: 13.89%



     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  44.0 K   train   174 K    [2, 10]      [2, 1]



Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 174 K

train_model result: {'val_loss': 3699.428955078125, 'hp_metric': 3699.428955078125}
spotpython tuning: 3043.216064453125 [########--] 83.15%. Success rate: 13.51%
```

```
    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   44.0 K   train   174 K    [2, 10]      [2, 1]


Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 174 K

train_model result: {'val_loss': 3574.594970703125, 'hp_metric': 3574.594970703125}
spotpython tuning: 3043.216064453125 [#########-] 86.50%. Success rate: 13.16%

Using spacefilling design as fallback.


    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   44.0 K   train   348 K    [4, 10]      [4, 1]


Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 348 K

train_model(): trainer.fit failed with exception: SparseAdam does not support dense gradients, pleas
train_model result: {'val_loss': 24037.681640625, 'hp_metric': 24037.681640625}
spotpython tuning: 3043.216064453125 [#########-] 89.28%. Success rate: 12.82%


    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   44.0 K   train   174 K    [2, 10]      [2, 1]
```

```
Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 174 K

train_model result: {'val_loss': 3110.500244140625, 'hp_metric': 3110.500244140625}
spotpython tuning: 3043.216064453125 [#########-] 92.52%. Success rate: 12.50%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  44.0 K   train   174 K    [2, 10]      [2, 1]


Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 174 K

train_model result: {'val_loss': 3255.001953125, 'hp_metric': 3255.001953125}
spotpython tuning: 3043.216064453125 [##########] 95.70%. Success rate: 12.20%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  44.0 K   train   174 K    [2, 10]      [2, 1]


Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 174 K
```

```
train_model result: {'val_loss': 3795.85107421875, 'hp_metric': 3795.85107421875}
spotpython tuning: 3043.216064453125 [##########] 98.89%. Success rate: 11.90%



     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0   layers   Sequential   44.0 K   train   174 K    [2, 10]      [2, 1]



Trainable params: 44.0 K
Non-trainable params: 0
Total params: 44.0 K
Total estimated model params size (MB): 0
Modules in train mode: 9
Modules in eval mode: 0
Total FLOPs: 174 K


train_model result: {'val_loss': 3514.47705078125, 'hp_metric': 3514.47705078125}
spotpython tuning: 3043.216064453125 [##########] 100.00%. Success rate: 11.63% Done...

Experiment saved to 601-user-model_res.pkl
| name           | type   | default   |   lower |   upper | tuned                | transform
|----------------|--------|-----------|---------|---------|----------------------
|------------------------|--------------|---------|
| l1             | int    | 3         |     3.0 |     8.0 | 7.0                  | transform_power_2_
| epochs         | int    | 4         |     4.0 |     9.0 | 7.0                  | transform_power_2_
| batch_size     | int    | 4         |     1.0 |     4.0 | 1.0                  | transform_power_2_
| act_fn         | factor | ReLU      |     0.0 |     5.0 | LeakyReLU            | None
| optimizer      | factor | SGD       |     0.0 |    11.0 | NAdam                | None
| dropout_prob   | float  | 0.01      |     0.0 |    0.25 | 0.21387297529421007  | None
| lr_mult        | float  | 1.0       |     0.1 |    10.0 | 6.539174191414857    | None
| patience       | int    | 2         |     2.0 |     6.0 | 4.0                  | transform_power_2_
| initialization | factor | Default   |     0.0 |     4.0 | kaiming_uniform      | None
l1:  0.010026791708976695
epochs:  0.0012575814595942338
batch_size:  0.0012575814595942338
act_fn:  100.0
optimizer:  1.0668824372305763
dropout_prob:  0.11289337720667261
lr_mult:  0.0012575814595942338
patience:  0.19681880524639017
initialization:  0.012230054219095239
```

Prediction Surface
Prediction
22500
20000
17500
15000
12500
10000
7500
5000
act_fn
optimizer
Prediction Error Surface
Error
5000
4000
3000
2000
1000
act_fn
optimizer
Prediction Contour
optimizer
act_fn
23200
20800
18400
16000
13600
11200
8800
6400
4000
Error Contour
optimizer
act_fn
5600
5000
4400
3800
3200
2600
2000
1400
800

Prediction Surface
Prediction
act_fn
patience
Prediction Error Surface
Error
act_fn
patience
Prediction Contour
patience
act_fn
Error Contour
patience
act_fn

47.2. Details

47.2.1. Model Setup

By using `core_model_name = "my_regressor.MyRegressor"`, the user specified model class `MyRegressor` [SOURCE] is selected. For this given `core_model_name`, the local hyper_dict is loaded using the `my_hyper_dict.py` file as shown below.

47.2.2. The `my_hyper_dict.py` File

The `my_hyper_dict.py` file must be placed in the `/userModel` directory. It provides a convenience function to load the hyperparameters from user specified the `my_hyper_dict.json` file, see Section 47.2.2. The user does not need to modify this file,

if the JSON file is stored as `my_hyper_dict.json`. Alternative filenames can be specified via the `filename` argument (which is default set to `"my_hyper_dict.json"`).

47.2.3. The `my_hyper_dict.json` File

The `my_hyper_dict.json` file contains the hyperparameter settings as a dictionary, which are loaded via the `my_hyper_dict.py` file. The example below shows the content of the `my_hyper_dict.json` file.

```
{
    "MyRegressor": {
        "l1": {
            "type": "int",
            "default": 3,
            "transform": "transform_power_2_int",
            "lower": 3,
            "upper": 8
        },
        "epochs": {
            "type": "int",
            "default": 4,
            "transform": "transform_power_2_int",
            "lower": 4,
            "upper": 9
        },
        "batch_size": {
            "type": "int",
            "default": 4,
            "transform": "transform_power_2_int",
            "lower": 1,
            "upper": 4
        },
        "act_fn": {
            "levels": [
                "Sigmoid",
                "Tanh",
                "ReLU",
                "LeakyReLU",
                "ELU",
                "Swish"
            ],
            "type": "factor",
            "default": "ReLU",
            "transform": "None",
```

```
            "class_name": "spotpython.torch.activation",
            "core_model_parameter_type": "instance()",
            "lower": 0,
            "upper": 5
        },
        "optimizer": {
            "levels": [
                "Adadelta",
                "Adagrad",
                "Adam",
                "AdamW",
                "SparseAdam",
                "Adamax",
                "ASGD",
                "NAdam",
                "RAdam",
                "RMSprop",
                "Rprop",
                "SGD"
            ],
            "type": "factor",
            "default": "SGD",
            "transform": "None",
            "class_name": "torch.optim",
            "core_model_parameter_type": "str",
            "lower": 0,
            "upper": 11
        },
        "dropout_prob": {
            "type": "float",
            "default": 0.01,
            "transform": "None",
            "lower": 0.0,
            "upper": 0.25
        },
        "lr_mult": {
            "type": "float",
            "default": 1.0,
            "transform": "None",
            "lower": 0.1,
            "upper": 10.0
        },
        "patience": {
            "type": "int",
```

```
            "default": 2,
            "transform": "transform_power_2_int",
            "lower": 2,
            "upper": 6
        },
        "initialization": {
            "levels": [
                "Default",
                "Kaiming",
                "Xavier"
            ],
            "type": "factor",
            "default": "Default",
            "transform": "None",
            "core_model_parameter_type": "str",
            "lower": 0,
            "upper": 2
        }
    }
}
```

47.2.4. The `my_regressor.py` File

The `my_regressor.py` file contains [SOURCE] the model class, which is a subclass of `nn.Module`. It must implement the following methods:

- `__init__(self, **kwargs)`: The constructor of the model class. The hyperparameters are passed as keyword arguments.
- `forward(self, x: torch.Tensor) -> torch.Tensor`: The forward pass of the model. The input `x` is passed through the model and the output is returned.
- `training_step(self, batch, batch_idx) -> torch.Tensor`: The training step of the model. It takes a batch of data and the batch index as input and returns the loss.
- `validation_step(self, batch, batch_idx) -> torch.Tensor`: The validation step of the model. It takes a batch of data and the batch index as input and returns the loss.
- `test_step(self, batch, batch_idx) -> torch.Tensor`: The test step of the model. It takes a batch of data and the batch index as input and returns the loss.
- `predict(self, x: torch.Tensor) -> torch.Tensor`: The prediction method of the model. It takes an input `x` and returns the prediction.
- `configure_optimizers(self) -> torch.optim.Optimizer`: The method to configure the optimizer of the model. It returns the optimizer.

The file `my_regressor.py` must be placed in the `/userModel` directory. The user can modify the model class to implement a custom model architecture.

We will take a closer look at the methods defined in the `my_regressor.py` file in the next subsections.

47.2.4.1. The `__init__` Method

`__init__()` initializes the `MyRegressor` object. It takes the following arguments:

- `l1` (int): The number of neurons in the first hidden layer.
- `epochs` (int): The number of epochs to train the model for.
- `batch_size` (int): The batch size to use during training.
- `initialization` (str): The initialization method to use for the weights.
- `act_fn` (nn.Module): The activation function to use in the hidden layers.
- `optimizer` (str): The optimizer to use during training.
- `dropout_prob` (float): The probability of dropping out a neuron during training.
- `lr_mult` (float): The learning rate multiplier for the optimizer.
- `patience` (int): The number of epochs to wait before early stopping.
- `_L_in` (int): The number of input features. Not a hyperparameter, but needed to create the network.
- `_L_out` (int): The number of output classes. Not a hyperparameter, but needed to create the network.
- `_torchmetric` (str): The metric to use for the loss function. If `None`, then "mean_squared_error" is used.

It is implemented as follows:

```
class MyRegressor(L.LightningModule):
        def __init__(
        self,
        l1: int,
        epochs: int,
        batch_size: int,
        initialization: str,
        act_fn: nn.Module,
        optimizer: str,
        dropout_prob: float,
        lr_mult: float,
        patience: int,
        _L_in: int,
        _L_out: int,
        _torchmetric: str,
        *args,
        **kwargs,
```

```
    ):
        super().__init__()
        self._L_in = _L_in
        self._L_out = _L_out
        if _torchmetric is None:
            _torchmetric = "mean_squared_error"
        self._torchmetric = _torchmetric
        self.metric = getattr(torchmetrics.functional.regression, _torchmetric)
        # _L_in and _L_out are not hyperparameters, but are needed to create the network
        # _torchmetric is not a hyperparameter, but is needed to calculate the loss
        self.save_hyperparameters(ignore=["_L_in", "_L_out", "_torchmetric"])
        # set dummy input array for Tensorboard Graphs
        # set log_graph=True in Trainer to see the graph (in traintest.py)
        self.example_input_array = torch.zeros((batch_size, self._L_in))
        if self.hparams.l1 < 4:
            raise ValueError("l1 must be at least 4")
        hidden_sizes = [l1 * 2, l1, ceil(l1/2)]
        # Create the network based on the specified hidden sizes
        layers = []
        layer_sizes = [self._L_in] + hidden_sizes
        layer_size_last = layer_sizes[0]
        for layer_size in layer_sizes[1:]:
            layers += [
                nn.Linear(layer_size_last, layer_size),
                self.hparams.act_fn,
                nn.Dropout(self.hparams.dropout_prob),
            ]
            layer_size_last = layer_size
        layers += [nn.Linear(layer_sizes[-1], self._L_out)]
        # nn.Sequential summarizes a list of modules into a single module,
        # applying them in sequence
        self.layers = nn.Sequential(*layers)
```

47.2.4.2. The `hidden_sizes`

`__init__()` uses the `hidden_sizes` list to define the sizes of the hidden layers in the network. The hidden sizes are calculated based on the `l1` hyperparameter. The hidden sizes can be computed in different ways, depending on the problem and the desired network architecture. We recommend using a separate function to calculate the hidden sizes based on the hyperparameters.

47.2.4.3. The `forward` Method

The `forward()` method defines the forward pass of the model. It takes an input tensor `x` and passes it through the network layers to produce an output tensor. It is implemented as follows:

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    return self.layers(x)
```

47.2.4.4. The `_calculate_loss` Method

The `_calculate_loss()` method calculates the loss based on the predicted output and the target values. It uses the specified metric to calculate the loss. It takes the following arguments:

- `batch (tuple)`: A tuple containing a batch of input data and labels.

It is implemented as follows:

```
def _calculate_loss(self, batch):
    x, y = batch
    y = y.view(len(y), 1)
    y_hat = self(x)
    loss = self.metric(y_hat, y)
    return loss
```

47.2.4.5. The `training_step` Method

The `training_step()` method defines the training step of the model. It takes a batch of data and returns the loss. It is implemented as follows:

```
def training_step(self, batch: tuple) -> torch.Tensor:
    val_loss = self._calculate_loss(batch)
    return val_loss
```

47.2.4.6. The `validation_step` Method

The `validation_step()` method defines the validation step of the model. It takes a batch of data and returns the loss. It is implemented as follows:

```
def validation_step(self, batch: tuple) -> torch.Tensor:
    val_loss = self._calculate_loss(batch)
    return val_loss
```

47.2.4.7. The `test_step` Method

The `test_step()` method defines the test step of the model. It takes a batch of data and returns the loss. It is implemented as follows:

```
def test_step(self, batch: tuple) -> torch.Tensor:
    val_loss = self._calculate_loss(batch)
    return val_loss
```

47.2.4.8. The `predict` Method

The `predict()` method defines the prediction method of the model. It takes an input tensor `x` and returns a tuple with the input tensor `x`, the target tensor `y`, and the predicted tensor `y_hat`.

It is implemented as follows:

```
def predict(self, x: torch.Tensor) -> torch.Tensor:
    x, y = batch
    yhat = self(x)
    y = y.view(len(y), 1)
    yhat = yhat.view(len(yhat), 1)
    return (x, y, yhat)
```

47.2.4.9. The `configure_optimizers` Method

The `configure_optimizers()` method defines the optimizer to use during training. It uses the `optimizer_handler` from `spotpython.hyperparameter.optimizer` to create the optimizer based on the specified optimizer name, parameters, and learning rate multiplier. It is implemented as follows:

```
def configure_optimizers(self) -> torch.optim.Optimizer:
    optimizer = optimizer_handler(
        optimizer_name=self.hparams.optimizer, params=self.parameters(), lr_mult=self.hparams.lr_mul
    )
    return optimizer
```

Note, the default Lightning way is to define an optimizer as `optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)`. `spotpython` uses an optimizer handler to create the optimizer, which adapts the learning rate according to the `lr_mult` hyperparameter as well as other hyperparameters. See `spotpython.hyperparameters.optimizer.py` [SOURCE] for details.

47.3. Connection with the LightDataModule

The steps described in Section 47.2.4 are connected to the `LightDataModule` class [DOC]. This class is used to create the data loaders for the training, validation, and test sets. The `LightDataModule` class is part of the `spotpython` package and class provides the following methods [SOURCE]:

- `prepare_data()`: This method is used to prepare the data set.
- `setup()`: This method is used to create the data loaders for the training, validation, and test sets.
- `train_dataloader()`: This method is used to return the data loader for the training set.
- `val_dataloader()`: This method is used to return the data loader for the validation set.
- `test_dataloader()`: This method is used to return the data loader for the test set.
- `predict_dataloader()`: This method is used to return the data loader for the prediction set.

47.3.1. The `prepare_data()` Method

The `prepare_data()` method is used to prepare the data set. This method is called only once and on a single process. It can be used to download the data set. In our case, the data set is already available, so this method uses a simple `pass` statement.

47.3.2. The `setup()` Method

The `stage` is used to define the data set to be returned. It can be `None`, `fit`, `test`, or `predict`. If `stage` is `None`, the method returns the training (`fit`), testing (`test`), and prediction (`predict`) data sets.

The `setup` methods splits the data based on the `stage` setting for use in training, validation, and testing. It uses `torch.utils.data.random_split()` to split the data.

Splitting is based on the `test_size` and `test_seed`. The `test_size` can be a float or an int.

First, the data set sizes are determined as described in Section 47.3.2.1. Then, the data sets are split based on the `stage` setting. `spotpython`'s `LightDataModule` class uses the following sizes:

- `full_train_size`: The size of the full training data set. This data set is splitted into the final training data set and a validation data set.
- `val_size`: The size of the validation data set. The validation data set is used to validate the model during training.

- `train_size`: The size of the training data set. The training data set is used to train the model.
- `test_size`: The size of the test data set. The test data set is used to evaluate the model after training. It is not used during training ("hyperparameter tuning"). Only after everything is finished, the model is evaluated on the test data set.

47.3.2.1. Determine the Sizes of the Data Sets

```
import torch
from torch.utils.data import random_split
data_full = Diabetes()
test_size = fun_control["test_size"]
test_seed=fun_control["test_seed"]
# if test_size is float, then train_size is 1 - test_size
if isinstance(test_size, float):
    full_train_size = round(1.0 - test_size, 2)
    val_size = round(full_train_size * test_size, 2)
    train_size = round(full_train_size - val_size, 2)
else:
    # if test_size is int, then train_size is len(data_full) - test_size
    full_train_size = len(data_full) - test_size
    val_size = int(full_train_size * test_size / len(data_full))
    train_size = full_train_size - val_size

print(f"LightDataModule setup(): full_train_size: {full_train_size}")
print(f"LightDataModule setup(): val_size: {val_size}")
print(f"LightDataModule setup(): train_size: {train_size}")
print(f"LightDataModule setup(): test_size: {test_size}")
```

```
LightDataModule setup(): full_train_size: 0.6
LightDataModule setup(): val_size: 0.24
LightDataModule setup(): train_size: 0.36
LightDataModule setup(): test_size: 0.4
```

47.3.2.2. The "setup" Method: Stage "fit"

Here, `train_size` and `val_size` are used to split the data into training and validation sets.

```
stage = "fit"
scaler = None
# Assign train/val datasets for use in dataloaders
if stage == "fit" or stage is None:
    print(f"train_size: {train_size}, val_size: {val_size} used for train & val data.
    generator_fit = torch.Generator().manual_seed(test_seed)
    data_train, data_val, _ = random_split(
        data_full, [train_size, val_size, test_size], generator=generator_fit
    )
    if scaler is not None:
        # Fit the scaler on training data and transform both train and val data
        scaler_train_data = torch.stack([data_train[i][0] for i in range(len(data_trai
        # train_val_data = data_train[:,0]
        print(scaler_train_data.shape)
        scaler.fit(scaler_train_data)
        data_train = [(scaler.transform(data), target) for data, target in data_train
        data_tensors_train = [data.clone().detach() for data, target in data_train]
        target_tensors_train = [target.clone().detach() for data, target in data_trai
        data_train = TensorDataset(
            torch.stack(data_tensors_train).squeeze(1), torch.stack(target_tensors_tra
        )
        # print(data_train)
        data_val = [(scaler.transform(data), target) for data, target in data_val]
        data_tensors_val = [data.clone().detach() for data, target in data_val]
        target_tensors_val = [target.clone().detach() for data, target in data_val]
        data_val = TensorDataset(torch.stack(data_tensors_val).squeeze(1), torch.stack
```

```
train_size: 0.36, val_size: 0.24 used for train & val data.
```

The `data_train` and `data_val` data sets are further used to create the training and validation data loaders as described in Section 47.3.3 and Section 47.3.4, respectively.

47.3.2.3. The "setup" Method: Stage "test"

Here, the test data set, which is based on the `test_size`, is created.

```
stage = "test"
# Assign test dataset for use in dataloader(s)
if stage == "test" or stage is None:
    print(f"test_size: {test_size} used for test dataset.")
    # get test data set as test_abs percent of the full dataset
    generator_test = torch.Generator().manual_seed(test_seed)
```

```
    data_test, _ = random_split(data_full, [test_size, full_train_size], generator=generator_test)
    if scaler is not None:
        data_test = [(scaler.transform(data), target) for data, target in data_test]
        data_tensors_test = [data.clone().detach() for data, target in data_test]
        target_tensors_test = [target.clone().detach() for data, target in data_test]
        data_test = TensorDataset(
            torch.stack(data_tensors_test).squeeze(1), torch.stack(target_tensors_test)
        )
print(f"LightDataModule setup(): Test set size: {len(data_test)}")
# Set batch size for DataLoader
batch_size = 5
# Create DataLoader
from torch.utils.data import DataLoader
dataloader = DataLoader(data_test, batch_size=batch_size, shuffle=False)
# Iterate over the data in the DataLoader
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
test_size: 0.4 used for test dataset.
LightDataModule setup(): Test set size: 177
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0490, -0.0446, -0.0418,  0.1045,  0.0356, -0.0257,  0.1775, -
0.0764,
         -0.0129,  0.0155],
        [-0.0273,  0.0507, -0.0159, -0.0298,  0.0039, -0.0007,  0.0413, -
0.0395,
         -0.0236,  0.0113],
        [ 0.0708,  0.0507, -0.0170,  0.0219,  0.0438,  0.0563,  0.0376, -
0.0026,
         -0.0702, -0.0176],
        [-0.0382,  0.0507,  0.0714, -0.0573,  0.1539,  0.1559,  0.0008,  0.0719,
          0.0503,  0.0693],
        [ 0.0453, -0.0446,  0.0391,  0.0460,  0.0067, -0.0242,  0.0081, -
0.0126,
```

```
          0.0643,  0.0569]])
Targets: tensor([103.,  53.,  80., 220., 246.])
```

47.3.2.4. The "setup" Method: Stage "predict"

Prediction and testing use the same data set. The prediction data set is created based on the `test_size`.

```
stage = "predict"
if stage == "predict" or stage is None:
    print(f"test_size: {test_size} used for predict dataset.")
    # get test data set as test_abs percent of the full dataset
    generator_predict = torch.Generator().manual_seed(test_seed)
    data_predict, _ = random_split(
        data_full, [test_size, full_train_size], generator=generator_predict
    )
    if scaler is not None:
        data_predict = [(scaler.transform(data), target) for data, target in data_prec
        data_tensors_predict = [data.clone().detach() for data, target in data_predict
        target_tensors_predict = [target.clone().detach() for data, target in data_pre
        data_predict = TensorDataset(
            torch.stack(data_tensors_predict).squeeze(1), torch.stack(target_tensors_p
        )
print(f"LightDataModule setup(): Predict set size: {len(data_predict)}")
# Set batch size for DataLoader
batch_size = 5
# Create DataLoader
from torch.utils.data import DataLoader
dataloader = DataLoader(data_predict, batch_size=batch_size, shuffle=False)
# Iterate over the data in the DataLoader
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
test_size: 0.4 used for predict dataset.
LightDataModule setup(): Predict set size: 177
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
```

```
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0490, -0.0446, -0.0418,  0.1045,  0.0356, -0.0257,  0.1775, -
0.0764,
         -0.0129,  0.0155],
        [-0.0273,  0.0507, -0.0159, -0.0298,  0.0039, -0.0007,  0.0413, -
0.0395,
         -0.0236,  0.0113],
        [ 0.0708,  0.0507, -0.0170,  0.0219,  0.0438,  0.0563,  0.0376, -
0.0026,
         -0.0702, -0.0176],
        [-0.0382,  0.0507,  0.0714, -0.0573,  0.1539,  0.1559,  0.0008,  0.0719,
          0.0503,  0.0693],
        [ 0.0453, -0.0446,  0.0391,  0.0460,  0.0067, -0.0242,  0.0081, -
0.0126,
          0.0643,  0.0569]])
Targets: tensor([103.,  53.,  80., 220., 246.])
```

47.3.3. The `train_dataloader()` Method

The method '`train_dataloader` returns the training dataloader, i.e., a Pytorch DataLoader instance using the training dataset. It simply returns a DataLoader with the `data_train` set that was created in the `setup()` method as described in Section 47.3.2.2.

```
def train_dataloader(self) -> DataLoader:
    return DataLoader(data_train, batch_size=batch_size, num_workers=num_workers)
```

Using the `train_dataloader()` Method

The `train_dataloader()` method can be used as follows:

```python
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Training set size: {len(data_module.data_train)}")
dl = data_module.train_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Training set size: 160
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[-0.0491, -0.0446, -0.0569, -0.0435, -0.0456, -
0.0433,  0.0008, -0.0395,
         -0.0119,  0.0155],
        [ 0.0344,  0.0507,  0.0283, -0.0332, -0.0456, -0.0098, -
0.0508, -0.0026,
         -0.0595, -0.0218],
        [-0.0600,  0.0507, -0.0472, -0.0229, -0.0717, -0.0577, -
0.0066, -0.0395,
         -0.0629, -0.0549],
        [-0.0128,  0.0507, -0.0558, -0.0022, -0.0277, -
0.0292,  0.0192, -0.0395,
         -0.0171,  0.0445],
        [ 0.0671,  0.0507, -0.0148,  0.0586, -0.0594, -0.0345, -
0.0618,  0.0129,
         -0.0051,  0.0486]])
Targets: tensor([ 68., 145.,  72., 135., 124.])
```

47.3.4. The `val_dataloader()` Method

Returns the validation dataloader, i.e., a Pytorch DataLoader instance using the validation dataset. It simply returns a DataLoader with the `data_val` set that was created in the `setup()` method as desccribed in Section 47.3.2.2.

```
def val_dataloader(self) -> DataLoader:
    return DataLoader(data_val, batch_size=batch_size, num_workers=num_workers)
```

Using the `val_dataloader()` Method

The `val_dataloader()` method can be used as follows:

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Validation set size: {len(data_module.data_val)}")
dl = data_module.val_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Validation set size: 106
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0163, -0.0446,  0.0736, -0.0412, -0.0043, -
0.0135, -0.0139, -0.0011,
          0.0429,  0.0445],
        [ 0.0453, -0.0446,  0.0714,  0.0012, -0.0098, -
0.0010,  0.0155, -0.0395,
         -0.0412, -0.0715],
        [ 0.0308,  0.0507,  0.0326,  0.0494, -0.0401, -0.0436, -
0.0692,  0.0343,
```

```
          0.0630,  0.0031],
        [ 0.0235,  0.0507, -0.0396, -0.0057, -0.0484, -
0.0333,  0.0118, -0.0395,
         -0.1016, -0.0674],
        [-0.0091,  0.0507,  0.0013, -0.0022,  0.0796,  0.0701,  0.0339, -
0.0026,
          0.0267,  0.0818]])
Targets: tensor([275., 141., 208.,  78., 142.])
```

47.3.5. The `test_dataloader()` Method

Returns the test dataloader, i.e., a Pytorch DataLoader instance using the test dataset. It simply returns a DataLoader with the `data_test` set that was created in the `setup()` method as described in Section 43.4.2.3.

```
def test_dataloader(self) -> DataLoader:
    return DataLoader(data_test, batch_size=batch_size, num_workers=num_workers)
```

Using the `test_dataloader()` Method

The `test_dataloader()` method can be used as follows:

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Test set size: {len(data_module.data_test)}")
dl = data_module.test_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Test set size: 177
Batch Size: 5
```

```
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0562, -0.0446, -0.0579, -
0.0080,  0.0521,  0.0491,  0.0560, -0.0214,
         -0.0283,  0.0445],
        [ 0.0018, -0.0446, -0.0709, -0.0229, -0.0016, -
0.0010,  0.0266, -0.0395,
         -0.0225,  0.0072],
        [-0.0527, -0.0446,  0.0542, -0.0263, -0.0552, -0.0339, -
0.0139, -0.0395,
         -0.0741, -0.0591],
        [ 0.0054, -0.0446, -0.0482, -0.0126,  0.0012, -
0.0066,  0.0634, -0.0395,
         -0.0514, -0.0591],
        [-0.0527, -0.0446, -0.0094, -0.0057,  0.0397,  0.0447,  0.0266, -
0.0026,
         -0.0181, -0.0135]])
Targets: tensor([158.,  49., 142.,  96.,  59.])
```

47.3.6. The `predict_dataloader()` Method

Returns the prediction dataloader, i.e., a Pytorch DataLoader instance using the prediction dataset. It simply returns a DataLoader with the `data_predict` set that was created in the `setup()` method as described in Section 43.4.2.4.

> **Warning**
>
> The `batch_size` is set to the length of the `data_predict` set.

```python
def predict_dataloader(self) -> DataLoader:
    return DataLoader(data_predict, batch_size=len(data_predict), num_workers=num_workers)
```

> **Using the `predict_dataloader()` Method**
>
> The `predict_dataloader()` method can be used as follows:

```python
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Test set size: {len(data_module.data_predict)}")
dl = data_module.predict_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Test set size: 177
Batch Size: 177
Inputs Shape: torch.Size([177, 10])
Targets Shape: torch.Size([177])
---------------
Inputs: tensor([[ 0.0562, -0.0446, -0.0579,  ..., -0.0214, -
0.0283,  0.0445],
        [ 0.0018, -0.0446, -0.0709,  ..., -0.0395, -
0.0225,  0.0072],
        [-0.0527, -0.0446,  0.0542,  ..., -0.0395, -0.0741, -
0.0591],
        ...,
        [ 0.0090, -0.0446, -0.0321,  ..., -0.0764, -0.0119, -
0.0384],
        [-0.0273, -0.0446, -0.0666,  ..., -0.0395, -0.0358, -
0.0094],
        [ 0.0817,  0.0507,  0.0067,  ...,  0.0919,  0.0547,  0.0072]])
Targets: tensor([158.,  49., 142.,  96.,  59.,  74., 137., 136.,  39.,  66., 310.,
        235., 116.,  55., 177.,  59., 246.,  53., 135.,  88., 198., 186., 217.,
         51., 118., 153., 180.,  51., 229.,  84.,  72., 237., 142., 185.,  91.,
         88., 148., 179., 144.,  25.,  89.,  42.,  60., 124., 170., 215., 263.,
        178., 245., 202.,  97., 321.,  71., 123., 220., 132., 243.,  61., 102.,
        187.,  70., 242., 134.,  63.,  72.,  88., 219., 127., 146., 122., 143.,
        220., 293.,  59., 317.,  60., 140.,  65., 277.,  90.,  96., 109., 190.,
         90.,  52., 160., 233., 230., 175.,  68., 272., 144.,  70.,  68., 163.,
```

```
         71.,  93., 263., 118., 220.,  90., 232., 120., 163.,  88.,  85.,  52.,
        181., 232., 212., 332.,  81., 214., 145., 268., 115.,  93.,  64., 156.,
        128., 200., 281., 103., 220.,  66.,  48., 246.,  42., 150., 125., 109.,
        129.,  97., 265.,  97., 173., 216., 237., 121.,  42., 151.,  31.,  68.,
        137., 221., 283., 124., 243., 150.,  69., 306., 182., 252., 132., 258.,
        121., 110., 292., 101., 275., 141., 208.,  78., 142., 185., 167., 258.,
        144.,  89., 225., 140., 303., 236.,  87.,  77., 131.])
```

47.4. Using the LightDataModule in the train_model() Method

The methods discussed so far are used in spotpython's train_model() method [DOC] to train the model. It is implemented as follows [SOURCE].

First, a LightDataModule object is created and the setup() method is called.

```
dm = LightDataModule(
    dataset=fun_control["data_set"],
    batch_size=config["batch_size"],
    num_workers=fun_control["num_workers"],
    test_size=fun_control["test_size"],
    test_seed=fun_control["test_seed"],
)
dm.setup()
```

Then, the Trainer is initialized.

```
# Init trainer
trainer = L.Trainer(
    default_root_dir=os.path.join(fun_control["CHECKPOINT_PATH"], config_id),
    max_epochs=model.hparams.epochs,
    accelerator=fun_control["accelerator"],
    devices=fun_control["devices"],
    logger=TensorBoardLogger(
        save_dir=fun_control["TENSORBOARD_PATH"],
        version=config_id,
        default_hp_metric=True,
        log_graph=fun_control["log_graph"],
    ),
    callbacks=[
        EarlyStopping(monitor="val_loss", patience=config["patience"], mode="min", strict=False, ver
```

```
    ],
    enable_progress_bar=enable_progress_bar,
)
```

Next, the `fit()` method is called to train the model.

```
# Pass the datamodule as arg to trainer.fit to override model hooks :)
trainer.fit(model=model, datamodule=dm)
```

Finally, the `validate()` method is called to validate the model. The `validate()` method returns the validation loss.

```
# Test best model on validation and test set
result = trainer.validate(model=model, datamodule=dm)
# unlist the result (from a list of one dict)
result = result[0]
return result["val_loss"]
```

47.5. The Last Connection: The `HyperLight` Class

The method `train_model()` is part of the `HyperLight` class [DOC]. It is called from `spotpython` as an objective function to train the model and return the validation loss.

The `HyperLight` class is implemented as follows [SOURCE].

```
class HyperLight:
    def fun(self, X: np.ndarray, fun_control: dict = None) -> np.ndarray:
        z_res = np.array([], dtype=float)
        self.check_X_shape(X=X, fun_control=fun_control)
        var_dict = assign_values(X, get_var_name(fun_control))
        for config in generate_one_config_from_var_dict(var_dict, fun_control):
            df_eval = train_model(config, fun_control)
            z_val = fun_control["weights"] * df_eval
            z_res = np.append(z_res, z_val)
        return z_res
```

47.6. Further Information

47.6.1. Preprocessing

Preprocessing is handled by `Lightning` and `PyTorch`. It is described in the LIGHTNINGDATAMODULE documentation. Here you can find information about the `transforms` methods.

48. Hyperparameter Tuning with PyTorch Lightning: ResNets

Neural ODEs are related to Residual Neural Networks (ResNets). We consider ResNets in Section 48.1.

48.1. Residual Neural Networks

He et al. (2015) introduced Residual Neural Networks (ResNets).

48.1.1. Residual Connections

Residual connections are a key component of ResNets. They are used to stabilize the training of very deep networks. The idea is to learn a residual mapping instead of the full mapping. The residual mapping is defined as:

Definition 48.1 (Residual Connection)**.** Let F denote a non-linear mapping (usually a sequence of NN modules likes convolutions, activation functions, and normalizations).

Instead of modeling

$$x_{l+1} = F(x_l),$$

residual connections model

$$x_{l+1} = x_l + F(x_l). \tag{48.1}$$

This is illustrated in Figure 48.1.

Applying backpropagation to the residual mapping results in the following gradient calculation:

$$\frac{\partial x_{l+1}}{\partial x_l} = \mathbf{I} + \frac{\partial F(x_l)}{\partial x_l}, \tag{48.2}$$

where $\mathbf{I}$ is the identity matrix. The identity matrix is added to the gradient, which helps to stabilize the training of very deep networks. The identity matrix ensures that the gradient is not too small, which can happen if the gradient of F is close to zero.

Figure 48.1.: Residual Connection. Figure credit He et al. (2015)

This is especially important for very deep networks, where the gradient can vanish quickly.

The bias towards the identity matrix guarantees a stable gradient propagation being less effected by F itself.

There have been many variants of ResNet proposed, which mostly concern the function F, or operations applied on the sum. Figure 48.2 shows two different ResNet blocks:

- the original ResNet block, which applies a non-linear activation function, usually ReLU, after the skip connection. and
- the pre-activation ResNet block, which applies the non-linearity at the beginning of F.

For very deep network the pre-activation ResNet has shown to perform better as the gradient flow is guaranteed to have the identity matrix as shown in Equation 48.2, and is not harmed by any non-linear activation applied to it.

48.1.2. Implementation of the Original ResNet Block

One special case we have to handle is when we want to reduce the image dimensions in terms of width and height. The basic ResNet block requires $F(x_l)$ to be of the same shape as x_l. Thus, we need to change the dimensionality of x_l as well before adding to $F(x_l)$. The original implementation used an identity mapping with stride 2 and padded additional feature dimensions with 0. However, the more common implementation is to use a 1x1 convolution with stride 2 as it allows us to change the feature dimensionality while being efficient in parameter and computation cost. The code for the ResNet block is relatively simple, and shown below:

```
import torch
import torch.nn as nn
```

Figure 48.2.: ResNet Block. Left: original Residual block in He et al. (2015). Right: pre-activation block. BN describes batch-normalization. Figure credit He et al. (2016)

```
class ResNetBlock(nn.Module):
    def __init__(self, c_in, act_fn, subsample=False, c_out=-1):
        """
        Inputs:
            c_in - Number of input features
            act_fn - Activation class constructor (e.g. nn.ReLU)
            subsample - If True, we need to apply a transformation inside the block to change the fe
            c_out - Number of output features. Note that this is only relevant if subsample is True,
        """
        super().__init__()
        if not subsample:
            c_out = c_in

        # Network representing F
        self.net = nn.Sequential(
            nn.Linear(c_in, c_out, bias=False),  # Linear layer for feature transformation
            nn.BatchNorm1d(c_out),               # Batch normalization for stable learning
            act_fn(),                            # Activation function
            nn.Linear(c_out, c_out, bias=False), # Second linear layer
            nn.BatchNorm1d(c_out)                # Batch normalization
```

```
        )

        # If subsampling, adjust the input feature dimensionality using a linear layer
        self.downsample = nn.Linear(c_in, c_out) if subsample else None
        self.act_fn = act_fn()

    def forward(self, x):
        z = self.net(x)  # Apply the main network
        if self.downsample is not None:
            x = self.downsample(x)  # Adjust dimensionality if necessary
        out = z + x  # Residual connection
        out = self.act_fn(out)  # Apply activation function
        return out

class ResNetRegression(nn.Module):
    def __init__(self, input_dim, output_dim, block, num_blocks=1, hidden_dim=64, act_
        super().__init__()
        self.input_layer = nn.Linear(input_dim, hidden_dim)  # Input layer transformat
        self.blocks = nn.ModuleList([block(hidden_dim, act_fn) for _ in range(num_bloc
        self.output_layer = nn.Linear(hidden_dim, output_dim)  # Output layer for regr

    def forward(self, x):
        x = self.input_layer(x)  # Apply input layer
        for block in self.blocks:
            x = block(x)  # Apply each block
        x = self.output_layer(x)  # Get final output
        return x
```

```
input_dim = 10
output_dim = 1
hidden_dim = 64
model = ResNetRegression(input_dim, output_dim, ResNetBlock, num_blocks=2, hidden_dim=
model
```

```
ResNetRegression(
  (input_layer): Linear(in_features=10, out_features=64, bias=True)
  (blocks): ModuleList(
    (0-1): 2 x ResNetBlock(
      (net): Sequential(
        (0): Linear(in_features=64, out_features=64, bias=False)
        (1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (2): ReLU()
        (3): Linear(in_features=64, out_features=64, bias=False)
        (4): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
```

```
      )
      (act_fn): ReLU()
    )
  )
  (output_layer): Linear(in_features=64, out_features=1, bias=True)
)
```

```python
# Create a sample input tensor with a batch size of 2
from torchviz import make_dot
sample_input = torch.randn(2, input_dim)

# Generate the visualization
output = model(sample_input)
dot = make_dot(output, params=dict(model.named_parameters()))

# Save and render the visualization
dot.format = 'png'
dot.render('./figures_static/resnet_regression')
```

```
'figures_static/resnet_regression.png'
```

Figure 48.3.: ResNet Regression

48.1.3. Implementation of the Pre-Activation ResNet Block

The second block we implement is the pre-activation ResNet block. For this, we have to change the order of layer in `self.net`, and do not apply an activation function on the output. Additionally, the downsampling operation has to apply a non-linearity as well as the input, x_l, has not been processed by a non-linearity yet. Hence, the block looks as follows:

```python
import torch
import torch.nn as nn

class PreActResNetBlock(nn.Module):
    def __init__(self, c_in, act_fn, subsample=False, c_out=-1):
        super().__init__()
        if not subsample:
            c_out = c_in
        self.net = nn.Sequential(
            nn.LayerNorm(c_in),  # Replacing BatchNorm1d with LayerNorm
            act_fn(),
            nn.Linear(c_in, c_out, bias=False),
            nn.LayerNorm(c_out),
            act_fn(),
            nn.Linear(c_out, c_out, bias=False)
        )
        self.downsample = nn.Sequential(
            nn.LayerNorm(c_in),
            act_fn(),
            nn.Linear(c_in, c_out, bias=False)
        ) if subsample else None

    def forward(self, x):
        z = self.net(x)
        if self.downsample is not None:
            x = self.downsample(x)
        out = z + x
        return out

class PreActResNetRegression(nn.Module):
    def __init__(self, input_dim, output_dim, block, num_blocks=1, hidden_dim=64, act_fn=nn.ReLU):
        super().__init__()
        self.input_layer = nn.Linear(input_dim, hidden_dim)
        self.blocks = nn.ModuleList([block(hidden_dim, act_fn) for _ in range(num_blocks)])
        self.output_layer = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.input_layer(x)
        for block in self.blocks:
            x = block(x)
        x = self.output_layer(x)
        return x
```

```
input_dim = 10
output_dim = 1
hidden_dim = 64
model = PreActResNetRegression(input_dim, output_dim, PreActResNetBlock, num_blocks=2
model
```

```
PreActResNetRegression(
  (input_layer): Linear(in_features=10, out_features=64, bias=True)
  (blocks): ModuleList(
    (0-1): 2 x PreActResNetBlock(
      (net): Sequential(
        (0): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
        (1): ReLU()
        (2): Linear(in_features=64, out_features=64, bias=False)
        (3): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
        (4): ReLU()
        (5): Linear(in_features=64, out_features=64, bias=False)
      )
    )
  )
  (output_layer): Linear(in_features=64, out_features=1, bias=True)
)
```

```
from torchviz import make_dot
# Create a sample input tensor
sample_input = torch.randn(1, input_dim)

# Generate the visualization
output = model(sample_input)
dot = make_dot(output, params=dict(model.named_parameters()))

# Save and render the visualization
dot.format = 'png'
dot.render('./figures_static/preact_resnet_regression')
```

```
'figures_static/preact_resnet_regression.png'
```

Figure 48.4.: Pre-Activation ResNet Regression

48.1.4. The Overall ResNet Architecture

The overall ResNet architecture for regression consists of stacking multiple ResNet blocks, of which some are downsampling the input. When discussing ResNet blocks within the entire network, they are usually grouped by output shape. If we describe the

ResNet as having `[3,3,3]` blocks, it means there are three groups of ResNet blocks, each containing three blocks, with downsampling occurring in the first block of the second and third groups. The final layer produces continuous outputs suitable for regression tasks.

Figure 48.5.: ResNet Notation. Figure credit Lippe (2022)

The `output_dim` parameter is used to determine the number of outputs for regression. This is set to 1 for a single regression target by default, but can be adjusted for multiple targets. Note, a final layer without a softmax or similar classification layer has to be added for regression tasks. A similar notation is used by many other implementations such as in the torchvision library from PyTorch.

Example 48.1 (Example ResNet Model)**.**

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import make_regression
from types import SimpleNamespace

def get_resnet_blocks_by_name():
    return {"ResNetBlock": ResNetBlock}

def get_act_fn_by_name():
    return {"relu": nn.ReLU}

# Define a simple ResNetBlock for fully connected layers
class ResNetBlock(nn.Module):
    def __init__(self, c_in, act_fn, subsample=False, c_out=-1):
        super().__init__()
        if not subsample:
            c_out = c_in
```

```
        self.net = nn.Sequential(
            nn.Linear(c_in, c_out, bias=False),
            nn.BatchNorm1d(c_out),
            act_fn(),
            nn.Linear(c_out, c_out, bias=False),
            nn.BatchNorm1d(c_out)
        )

        self.downsample = nn.Linear(c_in, c_out) if subsample else None
        self.act_fn = act_fn()

    def forward(self, x):
        z = self.net(x)
        if self.downsample is not None:
            x = self.downsample(x)
        out = z + x
        out = self.act_fn(out)
        return out

# Generate a simple random dataset for regression
num_samples = 100
num_features = 20  # Number of features, typical in a regression dataset
X, y = make_regression(n_samples=num_samples, n_features=num_features, noise=0.1)

# Convert to PyTorch tensors
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32).unsqueeze(1)  # Add a dimension for compatibility

# Define the ResNet model for regression
class ResNet(nn.Module):
    def __init__(self, input_dim, output_dim, num_blocks=[3, 3, 3], c_hidden=[64, 64, 64], act_fn_na
        super().__init__()
        resnet_blocks_by_name = get_resnet_blocks_by_name()
        act_fn_by_name = get_act_fn_by_name()
        assert block_name in resnet_blocks_by_name
        self.hparams = SimpleNamespace(output_dim=output_dim,
                                       c_hidden=c_hidden,
                                       num_blocks=num_blocks,
                                       act_fn_name=act_fn_name,
                                       act_fn=act_fn_by_name[act_fn_name],
                                       block_class=resnet_blocks_by_name[block_name])
        self._create_network(input_dim)
        self._init_params()
```

```
    def _create_network(self, input_dim):
        c_hidden = self.hparams.c_hidden
        self.input_net = nn.Sequential(
            nn.Linear(input_dim, c_hidden[0]),
            self.hparams.act_fn()
        )

        blocks = []
        for block_idx, block_count in enumerate(self.hparams.num_blocks):
            for bc in range(block_count):
                subsample = (bc == 0 and block_idx > 0)
                blocks.append(
                    self.hparams.block_class(c_in=c_hidden[block_idx if not subsample
                                             act_fn=self.hparams.act_fn,
                                             subsample=subsample,
                                             c_out=c_hidden[block_idx])
                )
        self.blocks = nn.Sequential(*blocks)

        self.output_net = nn.Linear(c_hidden[-1], self.hparams.output_dim)

    def _init_params(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm1d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        x = self.input_net(x)
        x = self.blocks(x)
        x = self.output_net(x)
        return x

# Instantiate the model
model = ResNet(input_dim=num_features, output_dim=1)

# Define a loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)

# Example training loop
num_epochs = 10
```

```python
for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()

    # Forward pass
    output = model(X_tensor)

    # Compute loss
    loss = criterion(output, y_tensor)

    # Backward pass and optimization
    loss.backward()
    optimizer.step()

    print(f'Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}')
```

```
Epoch 1/10, Loss: 15636.9345703125
Epoch 2/10, Loss: 12827.517578125
Epoch 3/10, Loss: 11187.2978515625
Epoch 4/10, Loss: 9906.5654296875
Epoch 5/10, Loss: 8869.42578125
Epoch 6/10, Loss: 7931.02490234375
Epoch 7/10, Loss: 7057.416015625
Epoch 8/10, Loss: 6238.6005859375
Epoch 9/10, Loss: 5459.04736328125
Epoch 10/10, Loss: 4747.82373046875
```

49. Neural ODEs

Neural ODEs are related to Residual Neural Networks (ResNets). We consider ResNets in Section 48.1.

49.1. Neural Ordinary Differential Equations

Neural Ordinary Differential Equations (Neural ODEs) are a class of models that are based on ordinary differential equations (ODEs). They are a generalization of ResNets, where the depth of the network is treated as a continuous parameter. Neural ODEs have been introduced by R. T. Q. Chen et al. (2018). We will consider dynamical systems first.

Definition 49.1. A dynamical system is a triple

$$(\mathcal{S}, \mathcal{T}, \Phi)$$

where

- $\mathcal{S}$ is the *state space*
- $\mathcal{T}$ is the *parameter space*, and
- $\Phi : (\mathcal{T} \times \mathcal{S}) \longrightarrow \mathcal{S}$ is the evolution.

Definition 49.1 is a very general definition that includes all sort of dynamical systems. We deal with ODEs where Φ plays the role of the *general solution*: indeed a 1-parameter family of transformations of the state space. $\mathcal{T} = \mathbb{R}_+$ is the time, and usually, $\mathcal{S} = \mathbb{R}^n$ is the state space. The evolution takes a point in space (initial value), a point in time, and returns the a point in space. A general solution to an ODE is a function $y : I \times \mathbb{R}^n \longrightarrow \mathbb{R}^n$: a 1-parameter (usually time is the parameter) family of transformations of the state space. A 1-parameter family of transformations is often called a *flow*.

First-order Ordinary Differential Equations (ODEs) can be defined as follows:

Definition 49.2 (First-Order Ordinary Differential Equation (ODE))**.**

$$\dot{\mathbf{y}}(t) = f(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = y_0, \quad f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$$

The solution of the ODE is the function $\mathbf{y}(t)$ that satisfies the ODE and the initial condition, which can be stated as an initial value problems (IVP), i.e. predict $\mathbf{y}(t_1)$ given $\mathbf{y}(t_0)$.

Definition 49.3 (Initial Value Problem (IVP))**.**

$$\mathbf{y}(t_1) = \mathbf{y}(t_0) + \int_{t_0}^{t_1} f(\mathbf{y}(t), t)\mathrm{d}t = \mathrm{ODESolve}(\mathbf{y}(t_0), f, t_0, t_1) \tag{49.1}$$

The existence and uniqueness of solutions to an IVP is ensured by the Picard-Lindelöf theorem, provided the RHS of the ODE is *Lipschitz continuous*. Lipschitz continuity is a property that pops up quite often in ODE-related results in ML.

Definition 49.4 (Lipschitz Continuity)**.** A function $f : X \subset \mathbb{R}^n \longrightarrow \mathbb{R}^n$ is called *Lipschitz continuous* (with constant λ) if

$$||f(x_1) - f(x_2)|| \leq \lambda ||x_1 - x_2|| \quad \forall x_1, x_2 \in X.$$

Note that Lipschitz continuity is a stronger condition than just continuity.

Numerical solvers can be used to perform the forward pass and solve the IVP. If we use, for example, Euler's method, we have the following update rule:

$$\mathbf{y}(t + h) = \mathbf{y}(t) + hf(\mathbf{y}(t), t) \tag{49.2}$$

where h is the step size. The update rule is applied iteratively to solve the IVP. The solution is a discrete approximation of the continuous function $\mathbf{y}(t)$.

Equation 49.2 looks almost identical to a ResNet block (see Equation 48.1). This was one of the main motivations for Neural ODEs (R. T. Q. Chen et al. 2018).

ResNets update hidden states by employing residual connections:

$$\mathbf{y}_{l+1} = \mathbf{y}_l + f(\mathbf{y}_l, \theta_l)$$

where f is a neural network with parameters θ_l, and $\mathbf{y}_l$ and $\mathbf{y}_{l+1}$ are the hidden states at subsequent layers, $l \in \{0, \dots, L\}$.

These updates can be seen as Euler discretizations of continuous transformations.

$$\dot{\mathbf{y}} = f(\mathbf{y}, t, \theta) \tag{49.3}$$

$$\Big\downarrow \text{ Euler Discretization} \tag{49.4}$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + hf(\mathbf{y}_n, t_n, \theta) \tag{49.5}$$

What happens in a residual network (with step sizes h) if we consider the continuous limit of each discrete layer in the network? What happens as we add more layers and take smaller steps? The answer seems rather astounding: instead of having a discrete number of layers between the input and output domains, we allow the evolution of the hidden states to become continuous.

Figure 49.1.: A residual network defines a discrete sequence of finite transformations. Circles represent evaluation locations. Figure credit R. T. Q. Chen et al. (2018).

The main technical difficulty in training continuous-depth networks is performing backpropagation through the ODE solver. Differentiating through the operations of the forward pass is straightforward, but incurs a high memory cost and introduces additional numerical error.

Figure 49.2.: An ODE network defines a vector field, which continuously transforms the state. Circles represent evaluation locations. Figure credit R. T. Q. Chen et al. (2018).

Pontryagin (1987) treated the ODE solver as a black box, and computed gradients using the adjoint sensitivity method. This approach computes gradients by solving a second, augmented ODE backwards in time, and is applicable to all ODE solvers. It scales linearly with problem size, has low memory cost, and explicitly controls numerical error.

Consider optimizing a scalar-valued loss function $L()$, whose input is the result of an ODE solver:

$$L(y(t_1) = L\left(y(t_0) + \int_{t_0}^{t_1} f(y(t), t, \theta)dt\right) = L\left(\text{ODESolve}(y(t_0), f, t_0, t_1, \theta)\right) \quad (49.6)$$

Equation 49.6 is related to {Equation 49.1}. To optimize L, we require gradients with respect to θ.

Similar to standard neural networks, we start with determining how the gradient of the loss depends on the hidden state $y(t)$ at each instant. This quantity is called the adjoint $a(t) = \frac{\partial L}{\partial y(t)}$. It satisfies the following IVP:

$$\dot{\mathbf{a}}(t) = -\mathbf{a}(t)^\top \frac{\partial f(\mathbf{x}(t), t, \theta)}{\partial \mathbf{x}}, \quad \mathbf{a}(t_1) = \frac{\partial L}{\partial \mathbf{x}(t_1)}.$$

Its dynamics are given by another ODE, which can be thought of as the instantaneous analog of the chain rule:

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(y(t), t, \theta)}{\partial y}.$$

Thus, starting from the initial (remember we are running backwards) value $\mathbf{a}(t_1) = \frac{\partial L}{\partial \mathbf{x}(t_1)}$, we can compute $\mathbf{a}(t_0) = \frac{\partial L}{\partial \mathbf{x}(t_0)}$ by another call to an ODE solver.

Finally, computing the gradients with respect to the parameters θ requires evaluating a third integral, which depends on both $\mathbf{x}(t)$ and $\mathbf{a}(t)$:

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^\top \frac{\partial f}{\partial \theta} \mathrm{d}t,$$

So this method trades off computation for memory—in fact the memory requirement for this gradient calculation is only $\mathcal{O}(1)$ with respect to the number of layers. The corresponding algorithm is described in R. T. Q. Chen et al. (2018), see also Figure 49.3.

Here you can find a very good explanation of the following result based on Lagrange multipliers.

Figure 49.3.: Reverse-mode differentiation of an ODE solution. The adjoint sensitivity method solves an augmented ODE backwards in time. The augmented system contains both the original state and the sensitivity of the loss with respect to the state. If the loss depends directly on the state at multiple observation times, the adjoint state must be updated in the direction of the partial derivative of the loss with respect to each observation. Figure credit R. T. Q. Chen et al. (2018).

49.2. Regression Example

To illustrate this concept, we will consider a simple regression example. This example is based on the Neural-ODEs tutorial from Neural Ordinary Differential Equations, which is provided by R. T. Q. Chen et al. (2018). We will use the ODE solvers from Torchdiffeq.

Neural ODEs, or ODE-Nets, build complex models by chaining together simple building blocks, similar to residual networks. Here, our base layer will define the dynamics of an ODE, which will be interconnected using an ODE solver to form the complete neural network model.

49.2.1. Specifying the Dynamics Layer

The dynamics of an ODE can be captured by the equation:

$$\dot{y}(t) = f(y(t), t, \theta), \qquad y(0) = y_0,$$

where the initial value $y_0 \in \mathbb{R}^n$. The θ parameters were added to the dynamics, so the dynamics function has the dimensions $f : \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^{|\theta|} \to \mathbb{R}^n$, where $|\theta|$ is the number of parameters we've added to f.

We need the dynamics function to take in the current state $y(t)$ of the ODE, the current time t, and some parameters θ, and output $\frac{\partial y(t)}{\partial t}$, which has the same shape as $y(t)$. They are passed as input to a multi-layer perceptron (MLP). Multiple evaluations of this dynamics layer can be combined using any suitable ODE solver, such as the adaptive-step Dormand-Price solver implemented in the `torchdiffeq` library's `odeint` function.

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import torchdiffeq
```

Let's start by defining an MLP class to serve as the building block of our models.

```
class MLP(nn.Module):
    def __init__(self, layer_sizes):
        super(MLP, self).__init__()
        layers = []
        for i in range(len(layer_sizes) - 1):
            layers.append(nn.Linear(layer_sizes[i], layer_sizes[i+1]))
```

```
            if i < len(layer_sizes) - 2:
                layers.append(nn.Tanh())
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)
```

Next, we'll define a ResNet class that uses the MLP as its inner component.

```
class ResNet(nn.Module):
    def __init__(self, layer_sizes, depth):
        super(ResNet, self).__init__()
        self.mlp = MLP(layer_sizes)
        self.depth = depth

    def forward(self, x):
        for _ in range(self.depth):
            x = self.mlp(x) + x
        return x
```

- `ODEFunc` defines how the system evolves over time using the MLP to approximate derivatives $\dot{y}(t)$.
- `ODEBlock` specifies the network structure. It uses `torchdiffeq.odeint` to integrate these dynamics over time.

```
class ODEFunc(nn.Module):
    def __init__(self, layer_sizes):
        super(ODEFunc, self).__init__()
        self.mlp = MLP(layer_sizes)

    def forward(self, t, y):
        t_expanded = t.expand_as(y)
        state_and_time = torch.cat([y, t_expanded], dim=1)
        return self.mlp(state_and_time)

class ODEBlock(nn.Module):
    def __init__(self, odefunc):
        super(ODEBlock, self).__init__()
        self.odefunc = odefunc

    def forward(self, x):
        t = torch.tensor([0.0, 1.0])
        out = torchdiffeq.odeint(self.odefunc, x, t, atol=1e-3, rtol=1e-3)
        return out[1]
```

Generate a toy 1D dataset.

```
inputs = torch.linspace(-2.0, 2.0, 10).reshape(10, 1)
targets = inputs**3 +  0.1 * inputs
```

We specify the hyperparameters for the ResNet and ODE-Net.

```
layer_sizes = [1, 25, 1]
param_scale = 1.0
step_size = 0.01
train_iters = 1000
resnet_depth = 3
```

Initialize and train the ResNet.

```
resnet = ResNet(layer_sizes, resnet_depth)
criterion = nn.MSELoss()
optimizer = optim.SGD(resnet.parameters(), lr=step_size)

for _ in range(train_iters):
    optimizer.zero_grad()
    outputs = resnet(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()
```

```
# We need to change the input dimension to 2, to allow time-dependent dynamics.
odenet_layer_sizes = [2, 25, 1]

# Initialize and train ODE-Net.
odefunc = ODEFunc(odenet_layer_sizes)
odenet = ODEBlock(odefunc)
optimizer = optim.SGD(odenet.parameters(), lr=step_size)

for _ in range(train_iters):
    optimizer.zero_grad()
    outputs = odenet(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()
```

Finally, plot the predictions of both models.

```
fine_inputs = np.linspace(-3.0, 3.0, 100).reshape(-1, 1)
fine_inputs_tensor = torch.from_numpy(fine_inputs).float()
plt.figure(figsize=(6, 4), dpi=150)
plt.scatter(inputs, targets, color='green', label='Targets')
plt.plot(fine_inputs, resnet(fine_inputs_tensor).detach().numpy(), color='blue', label

plt.plot(fine_inputs, odenet(fine_inputs_tensor).detach().numpy(), color='red', label=
plt.xlabel('Input')
plt.ylabel('Output')
plt.legend()
plt.show()
```

49.3. Further Reading

Neural ODEs have received a lot of attention in the past few years, ever since their introduction in Neurips 2018. Some of many many work in this field include:

- Neural Stochastic Differential Equations (Neural SDEs),
- Neural Controlled Differential Equations (Neural CDEs),
- Graph ODEs,

- Hamiltonial Neural Networks, and
- Lagrangian Neural Networks.

Michael Poli maintains the excellent Awesome Neural ODE, a collection of resources regarding the interplay between neural differential equations, dynamical systems, deep learning, control, numerical methods and scientific machine learning.

Torchdyn is an excellent library for Neural Differential Equations.

Implicit Layers is a list of tutorials on implicit functions and automatic differentiation, Neural ODEs, and Deep Equilibrium Models.

Understanding Neural ODE's is an excellent blogpost on ODEs and Neural ODEs.

Patrick Kidger's doctoral dissertation is an excellent textbook on Neural Differential Equations, see Kidger (2022).

50. Neural ODE Example

50.1. Implementation of a Neural ODE

The following example is based on the "UvA Deep Learning Tutorials" (Lippe 2022).

Example 50.1 (Example: Neural ODE)**.**

```
%matplotlib inline
import time
import logging
import statistics
from typing import Optional, List

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation
from IPython.display import HTML

import torch
import torch.nn as nn
import torch.utils.data as data
import torch.nn.functional as F
from torch.utils.data import Dataset

try:
    import torchdiffeq
except ModuleNotFoundError:
    !pip install --quiet torchdiffeq
    import torchdiffeq

try:
    import rich
except ModuleNotFoundError:
    !pip install --quiet rich
```

```
    import rich

try:
    # import pytorch_lightning as pl
    import lightning as pl
except ModuleNotFoundError:
    !pip install --quiet pytorch-lightning>=1.4
    # import pytorch_lightning as pl
    import lightning as pl
from torchmetrics.classification import Accuracy

pl.seed_everything(42)

torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")

from torchmetrics.functional import accuracy
```

```
Device: cpu
```

First, we define the core of our Neural ODE model.

```
class _ODEFunc(nn.Module):
    def __init__(self, module, autonomous=True):
        super().__init__()
        self.module = module
        self.autonomous = autonomous

    def forward(self, t, x):
        if not self.autonomous:
            x = torch.cat([torch.ones_like(x[:, [0]]) * t, x], 1)
        return self.module(x)


class ODEBlock(nn.Module):
    def __init__(self, odefunc: nn.Module, solver: str = 'dopri5',
                 rtol: float = 1e-4, atol: float = 1e-4, adjoint: bool = True,
                 autonomous: bool = True):
        super().__init__()
        self.odefunc = _ODEFunc(odefunc, autonomous=autonomous)
```

```
        self.rtol = rtol
        self.atol = atol
        self.solver = solver
        self.use_adjoint = adjoint
        self.integration_time = torch.tensor([0, 1], dtype=torch.float32)

    @property
    def ode_method(self):
        return torchdiffeq.odeint_adjoint if self.use_adjoint else torchdiffeq.odeint

    def forward(self, x: torch.Tensor, adjoint: bool = True, integration_time=None):
        integration_time = self.integration_time if integration_time is None else integration_time
        integration_time = integration_time.to(x.device)
        ode_method =  torchdiffeq.odeint_adjoint if adjoint else torchdiffeq.odeint
        out = ode_method(
            self.odefunc, x, integration_time, rtol=self.rtol,
            atol=self.atol, method=self.solver)
        return out
```

Next, we will wrap everything together in a LightningModule.

```
class Learner(pl.LightningModule):
    def __init__(self, model:nn.Module, t_span:torch.Tensor, learning_rate:float=5e-3):
        super().__init__()
        self.model = model
        self.t_span = t_span
        self.learning_rate = learning_rate
        # self.accuracy = Accuracy(num_classes=2)
        self.accuracy = accuracy

    def forward(self, x):
        return self.model(x)

    def inference(self, x, time_span):
        return self.model(x, adjoint=False, integration_time=time_span)

    def inference_no_projection(self, x, time_span):
        return self.model.forward_no_projection(x, adjoint=False, integration_time=time_span)

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_pred = self(x)
        y_pred = y_pred[-1]  # select last point of solution trajectory
        loss = nn.CrossEntropyLoss()(y_pred, y)
```

```
        self.log('train_loss', loss, prog_bar=True, logger=True)
        return loss

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_pred = self(x)
        y_pred = y_pred[-1]  # select last point of solution trajectory
        loss = nn.CrossEntropyLoss()(y_pred, y)
        self.log('val_loss', loss, prog_bar=True, logger=True)
        acc = self.accuracy(y_pred.softmax(dim=-1), y, num_classes=2, task="MULTICLASS
        self.log('val_accuracy', acc, prog_bar=True, logger=True)
        return loss

    def test_step(self, batch, batch_idx):
        x, y = batch
        y_pred = self(x)
        y_pred = y_pred[-1]  # select last point of solution trajectory
        loss = nn.CrossEntropyLoss()(y_pred, y)
        self.log('test_loss', loss, prog_bar=True, logger=True)
        acc = self.accuracy(y_pred.softmax(dim=-1), y, num_classes=2, task="MULTICLASS
        self.log('test_accuracy', acc, prog_bar=True, logger=True)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.model.parameters(), lr=self.learning_rate)
        return optimizer
```

We will be working will Half Moons Dataset, a non-linearly separable, binary classification dataset. The code is based on the excellent TorchDyn tutorials (https://github.com/DiffEqML/torchdyn), as well as the original TorchDiffEq examples (https://github.com/rtqichen/torchdiffeq).

```
class MoonsDataset(Dataset):
    """Half Moons Classification Dataset

    Adapted from https://github.com/DiffEqML/torchdyn
    """
    def __init__(self, num_samples=100, noise_std=1e-4):
        self.num_samples = num_samples
        self.noise_std = noise_std
        self.X, self.y = self.generate_moons(num_samples, noise_std)

    @staticmethod
    def generate_moons(num_samples=100, noise_std=1e-4):
```

```
        """Creates a *moons* dataset of `num_samples` data points.
        :param num_samples: number of data points in the generated dataset
        :type num_samples: int
        :param noise_std: standard deviation of noise magnitude added to each data point
        :type noise_std: float
        """
        num_samples_out = num_samples // 2
        num_samples_in = num_samples - num_samples_out
        theta_out = np.linspace(0, np.pi, num_samples_out)
        theta_in = np.linspace(0, np.pi, num_samples_in)
        outer_circ_x = np.cos(theta_out)
        outer_circ_y = np.sin(theta_out)
        inner_circ_x = 1 - np.cos(theta_in)
        inner_circ_y = 1 - np.sin(theta_in) - 0.5

        X = np.vstack([np.append(outer_circ_x, inner_circ_x),
                       np.append(outer_circ_y, inner_circ_y)]).T
        y = np.hstack([np.zeros(num_samples_out), np.ones(num_samples_in)])

        if noise_std is not None:
            X += noise_std * np.random.rand(num_samples, 2)

        X = torch.Tensor(X)
        y = torch.LongTensor(y)
        return X, y

    def __len__(self):
        return self.num_samples

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

```
def plot_binary_classification_dataset(X, y, title=None):
    CLASS_COLORS = ['coral', 'darkviolet']
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.scatter(X[:, 0], X[:, 1], color=[CLASS_COLORS[yi.int()] for yi in y], alpha=0.6)
    ax.set_aspect('equal')
    if title is not None:
        ax.set_title(title)

    return fig, ax
```

Let's create a sample dataset and visualize it.

```
sample_dataset = MoonsDataset(num_samples=400, noise_std=1e-1)
fig, ax = plot_binary_classification_dataset(sample_dataset.X, sample_dataset.y, title
```

Let's now create the train, validation, and test sets, with their corresponding data loaders. We will create a single big dataset and randomly split it in train, val, and test sets.

```
def split_dataset(dataset_size:int, split_percentages:List[float]) -> List[int]:
    split_sizes = [int(pi * dataset_size) for pi in split_percentages]
    split_sizes[0] += dataset_size - sum(split_sizes)
    return split_sizes


class ToyDataModule(pl.LightningDataModule):
    def __init__(self, dataset_size:int, split_percentages:Optional[float]=None):
        super().__init__()
        self.dataset_size = dataset_size
        if split_percentages is None:
            split_percentages = [0.8, 0.1, 0.1]
        self.split_sizes = split_dataset(self.dataset_size, split_percentages)

    def prepare_data(self):
        pass

    def setup(self, stage: Optional[str] = None):
        pass
```

```
    def train_dataloader(self):
        train_loader = torch.utils.data.DataLoader(self.train_set, batch_size=len(self.train_set), s
        return train_loader

    def val_dataloader(self):
        val_loader = torch.utils.data.DataLoader(self.val_set, batch_size=len(self.val_set), shuffle
        return val_loader

    def test_dataloader(self):
        test_loader = torch.utils.data.DataLoader(self.test_set, batch_size=len(self.test_set), shuf
        return test_loader


class HalfMoonsDataModule(ToyDataModule):
    def __init__(self, dataset_size:int, split_percentages:Optional[float]=None):
        super().__init__(dataset_size, split_percentages=split_percentages)

    def setup(self, stage: Optional[str] = None):
        dataset = MoonsDataset(num_samples=self.dataset_size, noise_std=1e-1)
        self.train_set, self.val_set, self.test_set = torch.utils.data.random_split(dataset, self.sp
```

We define a Neural ODE and train it. We will use a simple 2-layer MLP with a *tanh* activation and 64 hidden dimensions. We will train the model using the adjoint method for backpropagation.

A quick note on the architectural choices for our model. The **Picard-Lindelöf theorem** (Coddington and Levinson, 1955) states that the solution to an initial value problem **exists and is unique** if the differential equation is *uniformly Lipschitz continuous* in $\mathbf{z}$ and *continuous* in t. It turns out that this theorem holds for our model if the neural network has finite weights and uses Lipschitz nonlinearities, such as tanh or relu. However, not all tools are our deep learning arsenal is c. For example, as shown in **The Lipschitz Constant of Self-Attention** by Hyunjik Kim et al., standard self-attention is ***not*** Lipschitz. The authors propose alternative forms of self-attention that are Lipschitz.

```
import torch
from lightning.pytorch import Trainer
from lightning.pytorch.callbacks import ModelCheckpoint, RichProgressBar


adjoint = True
data_module = HalfMoonsDataModule(1000)
t_span = torch.linspace(0, 1, 2)
f = nn.Sequential(
```

```
    nn.Linear(2, 64),
    nn.Tanh(),
    nn.Linear(64, 2))
model = ODEBlock(f, adjoint=adjoint)
learner = Learner(model, t_span)

trainer = Trainer(
    max_epochs=200,
    accelerator="gpu" if torch.cuda.is_available() else "cpu",
    devices=1,
    callbacks=[
        ModelCheckpoint(mode="max", monitor="val_accuracy"),
        RichProgressBar(),
    ],
    log_every_n_steps=1,
)
trainer.fit(learner, datamodule=data_module)
val_result = trainer.validate(learner, datamodule=data_module, verbose=True)
test_result = trainer.test(learner, datamodule=data_module, verbose=True)
```

```
     Name    Type       Params   Mode    FLOPs

  0  model   ODEBlock      322   train       0


Trainable params: 322
Non-trainable params: 0
Total params: 322
Total estimated model params size (MB): 0
Modules in train mode: 6
Modules in eval mode: 0
Total FLOPs: 0


Output()


/Users/bartz/miniforge3/envs/spot313/lib/python3.13/site-packages/lightning/pytorch/tr
or.py:434: The 'val_dataloader' does not have many workers which may be a bottleneck.
of the `num_workers` argument` to `num_workers=23` in the `DataLoader` to improve peri
```

```
/Users/bartz/miniforge3/envs/spot313/lib/python3.13/site-packages/lightning/pytorch/trainer/connecto
or.py:434: The 'train_dataloader' does not have many workers which may be a bottleneck. Consider inc
value of the `num_workers` argument` to `num_workers=23` in the `DataLoader` to improve performance.
```

```
Output()
```

```
       Validate metric             DataLoader 0

        val_accuracy                    1.0
          val_loss             0.002127713290974498
```

```
Output()
```

```
         Test metric               DataLoader 0

        test_accuracy                   1.0
          test_loss            0.0018282054224982858
```

It seems that in less that 200 epochs we have achieved perfect validation accuracy. Let's now use the trained model to run inference and visualize the trajectories using a dense time span of 100 timesteps.

```
@torch.no_grad()
def run_inference(learner, data_loader, time_span):
    learner.to(device)
    trajectories = []
    classes = []
    time_span = torch.from_numpy(time_span).to(device)
    for data, target in data_loader:
        data = data.to(device)
        traj = learner.inference(data, time_span).cpu().numpy()
        trajectories.append(traj)
        classes.extend(target.numpy())
    trajectories = np.concatenate(trajectories, 1)
    return trajectories, classes

time_span = np.linspace(0.0, 1.0, 100)
trajectories, classes = run_inference(learner, data_module.train_dataloader(), time_sp

colors = ['coral', 'darkviolet']
class_colors = [colors[ci] for ci in classes]
```

We will now define a few functions to visualize the learned trajectories, the state-space, and the learned vector field.

Before we visualize the trajectories, let's plot the (training) data once again:

```
fig, ax = plot_binary_classification_dataset(*data_module.train_set[:], title='Half Mo
```

Below we visualize the evolution for each of the 2 inputs dimensions as a function of time (depth):

```
plot_trajectories(time_span, trajectories, class_colors)
```

And the same evolution combined in a single plot:

```
plot_trajectories_3d(time_span, trajectories, class_colors)
```

The 3D plot can be somewhat complicated to decipher. Thus, we also plot an animated version of the evolution. Each timestep of the animation is a slice on the temporal axis of the figure above.

```
anim = plot_trajectories_animation(time_span, trajectories, colors, classes, lim=8.0)
HTML(anim.to_html5_video())
```

```
<IPython.core.display.HTML object>
```

Finally, we can visualize the state-space diagram and the learned vector field:

```
fig, ax = plt.subplots(2, 1, figsize=(16, 8))
plot_state_space(trajectories, class_colors, ax=ax[0])
plot_static_vector_field(model, trajectories, ax=ax[1], device=device)
```

51. Physics Informed Neural Networks

51.1. PINNs

```
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as thdat
import functools
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_theme()
torch.manual_seed(42)

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# boundaries for the frequency range
a = 0
b = 500
```

51.2. Generation and Visualization of the Training Data and the Ground Truth (Function)

- Definition of the (unknown) differential equation:

```
def ode(frequency, loc, sigma, R):
    """Computes the amplitude. Defining equation, used
    to generate data and train models.
    The equation itself is not known to the model.

    Args:
        frequency: (N,) array-like
```

```
        loc: float
        sigma: float
        R: float

    Returns:
        (N,) array-like

    Examples:
        >>> ode(0, 25, 100, 0.005)
        100.0
    """
    A = np.exp(-R * (frequency - loc)**2/sigma**2)
    return A
```

- Setting the parameters for the ode

```
np.random.seed(10)
loc = 250
sigma = 100
R = 0.5
```

- Generating the data

```
frequencies = np.linspace(a, b, 1000)
eq = functools.partial(ode, loc=loc, sigma=sigma, R=R)
amplitudes = eq(frequencies)
```

- Now we have the ground truth for the full frequency range and can take a look at the first 10 values:

```
import pandas as pd
df = pd.DataFrame({'Frequency': frequencies[:10], 'Amplitude': amplitudes[:10]})
print(df)
```

```
   Frequency  Amplitude
0   0.000000   0.043937
1   0.500501   0.044490
2   1.001001   0.045048
3   1.501502   0.045612
4   2.002002   0.046183
5   2.502503   0.046759
6   3.003003   0.047341
7   3.503504   0.047929
8   4.004004   0.048524
9   4.504505   0.049124
```

- We generate the training data as a subset of the full frequency range and add some noise:

```
t = np.linspace(a, 2*b/3, 10)
A = eq(t) +  0.2 * np.random.randn(10)
```

- Plot of the training data and the ground truth:

```
plt.plot(frequencies, amplitudes)
plt.plot(t, A, 'o')
plt.legend(['Equation (ground truth)', 'Training data'])
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

(a)

(b)

Figure 51.1.

51.3. Gradient With Autograd

```
def grad(outputs, inputs):
    """Computes the partial derivative of
    an output with respect to an input.

    Args:
        outputs: (N, 1) tensor
        inputs: (N, D) tensor

    Returns:
        (N, D) tensor

    Examples:
        >>> x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
        >>> y = x**2
        >>> grad(y, x)
        tensor([2., 4., 6.])
    """
    return torch.autograd.grad(
        outputs, inputs, grad_outputs=torch.ones_like(outputs), create_graph=True
    )
```

- Autograd example:

```
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x**2
grad(y, x)
```

```
(tensor([2., 4., 6.], grad_fn=<MulBackward0>),)
```

51.4. Network

```
def numpy2torch(x):
    """Converts a numpy array to a pytorch tensor.

    Args:
        x: (N, D) array-like
```

```
    Returns:
        (N, D) tensor

    Examples:
        >>> numpy2torch(np.array([1,2,3]))
        tensor([1., 2., 3.])
    """
    n_samples = len(x)
    return torch.from_numpy(x).to(torch.float).to(DEVICE).reshape(n_samples, -1)
```

```
class Net(nn.Module):
    def __init__(
        self,
        input_dim,
        output_dim,
        n_units=100,
        epochs=1000,
        loss=nn.MSELoss(),
        lr=1e-3,
        loss2=None,
        loss2_weight=0.1,
    ) -> None:
        super().__init__()

        self.epochs = epochs
        self.loss = loss
        self.loss2 = loss2
        self.loss2_weight = loss2_weight
        self.lr = lr
        self.n_units = n_units

        self.layers = nn.Sequential(
            nn.Linear(input_dim, self.n_units),
            nn.ReLU(),
            nn.Linear(self.n_units, self.n_units),
            nn.ReLU(),
            nn.Linear(self.n_units, self.n_units),
            nn.ReLU(),
            nn.Linear(self.n_units, self.n_units),
            nn.ReLU(),
        )
        self.out = nn.Linear(self.n_units, output_dim)
```

```
    def forward(self, x):
        h = self.layers(x)
        out = self.out(h)
        return out

    def fit(self, X, y):
        Xt = numpy2torch(X)
        yt = numpy2torch(y)

        optimiser = optim.Adam(self.parameters(), lr=self.lr)
        self.train()
        losses = []
        for ep in range(self.epochs):
            optimiser.zero_grad()
            outputs = self.forward(Xt)
            loss = self.loss(yt, outputs)
            if self.loss2:
                loss += self.loss2_weight + self.loss2_weight * self.loss2(self)
            loss.backward()
            optimiser.step()
            losses.append(loss.item())
            if ep % int(self.epochs / 10) == 0:
                print(f"Epoch {ep}/{self.epochs}, loss: {losses[-1]:.2f}")
        return losses

    def predict(self, X):
        self.eval()
        out = self.forward(numpy2torch(X))
        return out.detach().cpu().numpy()
```

- Extended network for parameter estimation of parameter `r`:

```
class PINNParam(Net):
    def __init__(
        self,
        input_dim,
        output_dim,
        n_units=100,
        epochs=1000,
        loss=nn.MSELoss(),
        lr=0.001,
        loss2=None,
        loss2_weight=0.1,
    ) -> None:
```

```
        super().__init__(
            input_dim, output_dim, n_units, epochs, loss, lr, loss2, loss2_weight
        )

        self.r = nn.Parameter(data=torch.tensor([1.]))
        self.sigma = nn.Parameter(data=torch.tensor([100.]))
        self.loc = nn.Parameter(data=torch.tensor([100.]))
```

51.5. Basic Neutral Network

- Network without regularization:

```
net = Net(1,1, loss2=None, epochs=2000, lr=1e-5).to(DEVICE)

losses = net.fit(t, A)

plt.plot(losses)
plt.yscale('log')
```

```
Epoch 0/2000, loss: 6.59
Epoch 200/2000, loss: 0.06
Epoch 400/2000, loss: 0.05
Epoch 600/2000, loss: 0.05
Epoch 800/2000, loss: 0.05
Epoch 1000/2000, loss: 0.05
Epoch 1200/2000, loss: 0.05
Epoch 1400/2000, loss: 0.05
Epoch 1600/2000, loss: 0.05
Epoch 1800/2000, loss: 0.05
```

Figure 51.2.

- Adding L2 regularization:

```
def l2_reg(model: torch.nn.Module):
    """L2 regularization for the model parameters.

    Args:
        model: torch.nn.Module

    Returns:
        torch.Tensor

    Examples:
        >>> l2_reg(Net(1,1))
        tensor(0.0001, grad_fn=<SumBackward0>)
    """
    return torch.sum(sum([p.pow(2.) for p in model.parameters()]))
```

```
netreg = Net(1,1, loss2=l2_reg, epochs=20000, lr=1e-5, loss2_weight=.1).to(DEVICE)
losses = netreg.fit(t, A)
plt.plot(losses)
plt.yscale('log')
```

```
Epoch 0/20000, loss: 662.07
Epoch 2000/20000, loss: 612.81
Epoch 4000/20000, loss: 571.31
Epoch 6000/20000, loss: 533.74
Epoch 8000/20000, loss: 499.32
Epoch 10000/20000, loss: 467.44
Epoch 12000/20000, loss: 437.53
Epoch 14000/20000, loss: 409.15
Epoch 16000/20000, loss: 382.10
Epoch 18000/20000, loss: 356.29
```

Figure 51.3.

```
predsreg = netreg.predict(frequencies)
preds = net.predict(frequencies)
plt.plot(frequencies, amplitudes, alpha=0.8)
plt.plot(t, A, 'o')
plt.plot(frequencies, preds, alpha=0.8)
plt.plot(frequencies, predsreg, alpha=0.8)

plt.legend(labels=['Equation','Training data', 'Network', 'L2 Regularization Network'])
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

(a)

(b)

Figure 51.4.

51.6. PINNs

- Calculate the physics-informed loss (similar to the L2 regularization):

```
def physics_loss(model: torch.nn.Module):
    """Computes the physics-informed loss for the model.

    Args:
        model: torch.nn.Module

    Returns:
        torch.Tensor

    Examples:
        >>> physics_loss(Net(1,1))
```

```
        tensor(0.0001, grad_fn=<MeanBackward0>)
    """
    ts = torch.linspace(a, b, steps=1000).view(-1,1).requires_grad_(True).to(DEVICE)
    amplitudes = model(ts)
    dT = grad(amplitudes, ts)[0]
    ode = -2*R*(ts-loc)/ sigma**2 * amplitudes - dT
    return torch.mean(ode**2)
```

- Train the network with the physics-informed loss and plot the training error:

```
net_pinn = Net(1,1, loss2=physics_loss, epochs=2000, loss2_weight=1, lr=1e-5).to(DEVICE)
losses = net_pinn.fit(t, A)
plt.plot(losses)
plt.yscale('log')
```

```
Epoch 0/2000, loss: 12.23
Epoch 200/2000, loss: 1.05
Epoch 400/2000, loss: 1.05
Epoch 600/2000, loss: 1.05
Epoch 800/2000, loss: 1.05
Epoch 1000/2000, loss: 1.05
Epoch 1200/2000, loss: 1.05
Epoch 1400/2000, loss: 1.05
Epoch 1600/2000, loss: 1.05
Epoch 1800/2000, loss: 1.05
```

Figure 51.5.

- Predict the amplitude and plot the results:

```
preds_pinn = net_pinn.predict(frequencies)
plt.plot(frequencies, amplitudes, alpha=0.8)
plt.plot(t, A, 'o')
plt.plot(frequencies, preds, alpha=0.8)
plt.plot(frequencies, predsreg, alpha=0.8)
plt.plot(frequencies, preds_pinn, alpha=0.8)
plt.legend(labels=['Equation','Training data', 'NN', "R2", 'PINN'])
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

(a)

(b)

Figure 51.6.

51.6.1. PINNs: Parameter Estimation

```
def physics_loss_estimation(model: torch.nn.Module):
    ts = torch.linspace(a, b, steps=1000,).view(-1,1).requires_grad_(True).to(DEVICE)
    amplitudes = model(ts)
    dT = grad(amplitudes, ts)[0]
    ode = -2*model.r*(ts-model.loc)/ (model.sigma)**2 * amplitudes - dT
    return torch.mean(ode**2)
```

```
pinn_param = PINNParam(1, 1, loss2=physics_loss_estimation, loss2_weight=1, epochs=4000, lr= 5e-6).t
losses = pinn_param.fit(t, A)
plt.plot(losses)
plt.yscale('log')
```

```
Epoch 0/4000, loss: 15.06
```

```
Epoch 400/4000, loss: 1.06
Epoch 800/4000, loss: 1.06
Epoch 1200/4000, loss: 1.06
Epoch 1600/4000, loss: 1.05
Epoch 2000/4000, loss: 1.05
Epoch 2400/4000, loss: 1.05
Epoch 2800/4000, loss: 1.05
Epoch 3200/4000, loss: 1.05
Epoch 3600/4000, loss: 1.05
```

Figure 51.7.

```python
preds_disc = pinn_param.predict(frequencies)
print(f"Estimated r: {pinn_param.r}")
print(f"Estimated sigma: {pinn_param.sigma}")
print(f"Estimated loc: {pinn_param.loc}")
```

```
Estimated r: Parameter containing:
tensor([0.9893], requires_grad=True)
Estimated sigma: Parameter containing:
tensor([100.0067], requires_grad=True)
Estimated loc: Parameter containing:
tensor([100.0065], requires_grad=True)
```

```
plt.plot(frequencies, amplitudes, alpha=0.8)
plt.plot(t, A, 'o')
plt.plot(frequencies, preds_disc, alpha=0.8)
plt.legend(labels=['Equation','Training data', 'estimation PINN'])
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

(a)

(b)

Figure 51.8.

```
plt.plot(frequencies, amplitudes, alpha=0.8)
plt.plot(t, A, 'o')
plt.plot(frequencies, preds, alpha=0.8)
plt.plot(frequencies, predsreg, alpha=0.8)
plt.plot(frequencies, preds_pinn, alpha=0.8)
plt.plot(frequencies, preds_disc, alpha=0.8)
plt.legend(labels=['Equation','Training data', 'NN', "R2", 'PINN', 'paramPINN'])
plt.ylabel('Amplitude')
```

```
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

(a)

(b)

Figure 51.9.

```
plt.plot(frequencies, amplitudes, alpha=0.8)
plt.plot(t, A, 'o')
plt.plot(frequencies, preds, alpha=0.8)
plt.plot(frequencies, predsreg, alpha=0.8)
plt.plot(frequencies, preds_disc, alpha=0.8)
plt.legend(labels=['Equation','Training data', 'NN', "R2", 'paramPINN'])
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

(a)

(b)

Figure 51.10.

```
plt.plot(frequencies, amplitudes, alpha=0.8)
plt.plot(t, A, 'o')
plt.plot(frequencies, preds, alpha=0.8)
plt.plot(frequencies, predsreg, alpha=0.8)
plt.plot(frequencies, preds_disc, alpha=0.8)
plt.legend(labels=['Grundwahrheit','Trainingsdaten', 'NN', "NN+R2", 'PINN'])
plt.ylabel('Amplitude')
plt.xlabel('Frequenz')
# save the plot as a pdf
plt.savefig('pinns.pdf')
plt.savefig('pinns.png')
```

Figure 51.11.

51.7. Summary

- Results strongly depend on the parametrization(s)
- PINN parameter estimation not robust
- Hyperparameter tuning is crucial
- Use SPOT before further analysis is done

52. Hyperparameter Tuning with PyTorch Lightning: Physics Informed Neural Networks

52.1. PINNs

In this section, we will show how to set up PINN hyperparameter tuner from scratch based on the `spotpython` programs from Chapter 47.

52.1.1. The Ground Truth Model

Definition of the (unknown) differential equation:

```
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as thdat
import functools
import matplotlib.pyplot as plt
import seaborn as sns
# boundaries for the frequency range
a = 0
b = 500

def ode(frequency, loc, sigma, R):
    """Computes the amplitude. Defining equation, used
    to generate data and train models.
    The equation itself is not known to the model.

    Args:
        frequency: (N,) array-like
        loc: float
        sigma: float
```

```
        R: float

    Returns:
        (N,) array-like

    Examples:
        >>> ode(0, 25, 100, 0.005)
        100.0
    """
    A = np.exp(-R * (frequency - loc)**2/sigma**2)
    return A
```

Setting the parameters for the ode

```
np.random.seed(10)
loc = 250
sigma = 100
R = 0.5
```

- Generating the data

```
frequencies = np.linspace(a, b, 1000)
eq = functools.partial(ode, loc=loc, sigma=sigma, R=R)
amplitudes = eq(frequencies)
```

- Now we have the ground truth for the full frequency range and can take a look at the first 10 values:

```
df = pd.DataFrame({'Frequency': frequencies[:10], 'Amplitude': amplitudes[:10]})
print(df)
```

```
   Frequency  Amplitude
0   0.000000   0.043937
1   0.500501   0.044490
2   1.001001   0.045048
3   1.501502   0.045612
4   2.002002   0.046183
5   2.502503   0.046759
6   3.003003   0.047341
7   3.503504   0.047929
8   4.004004   0.048524
9   4.504505   0.049124
```

- We generate the training data as a subset of the full frequency range and add some noise:

```
# Make training data
t = np.linspace(a, 2*b/3, 10)
A = eq(t) +  0.2 * np.random.randn(10)
```

- Plot of the training data and the ground truth:

```
plt.plot(frequencies, amplitudes)
plt.plot(t, A, 'o')
plt.legend(['Equation (ground truth)', 'Training data'])
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

52.1.2. Required Files

We use the files from the `/userModel` directory as templates. They are renamed as follows:

- `my_regressor.py` ⇒ `pinn_regressor.py`, see Section 52.1.4
- `my_hyperdict.json` ⇒ `pinn_hyperdict.py`, see Section 52.1.5
- `my_hyperdict.py` ⇒ `pinn_hyperdict.py`, see Section 52.1.3.

52.1.3. The New `pinn_hyperdict.py` File

Modifying the `pin_hyperdict.py` file is very easy. We simply have to change the class-name `MyHyperDict` to `PINNHyperDict` and the `filename` from `"my_hyper_dict.json"` to `"pinn_hyper_dict.json"`. The file is shown below.

```
import json
from spotpython.data import base
import pathlib

class PINNHyperDict(base.FileConfig):
    def __init__(
        self,
        filename: str = "pinn_hyper_dict.json",
        directory: None = None,
    ) -> None:
        super().__init__(filename=filename, directory=directory)
        self.filename = filename
        self.directory = directory
        self.hyper_dict = self.load()

    @property
    def path(self):
        if self.directory:
            return pathlib.Path(self.directory).joinpath(self.filename)
        return pathlib.Path(__file__).parent.joinpath(self.filename)

    def load(self) -> dict:
        with open(self.path, "r") as f:
            d = json.load(f)
        return d
```

52.1.4. The New `pinn_regressor.py` File

> **Warning**
>
> The document is not complete. The code below is a template and needs to be modified to work with the PINN model.

```
import lightning as L
import torch
from torch import nn
from spotpython.hyperparameters.optimizer import optimizer_handler
import torchmetrics.functional.regression

class PINNRegressor(L.LightningModule):
    """
    A LightningModule class for a regression neural network model.

    Attributes:
        l1 (int):
            The number of neurons in the first hidden layer.
        epochs (int):
            The number of epochs to train the model for.
        batch_size (int):
            The batch size to use during training.
        initialization (str):
            The initialization method to use for the weights.
        act_fn (nn.Module):
            The activation function to use in the hidden layers.
        optimizer (str):
            The optimizer to use during training.
        dropout_prob (float):
            The probability of dropping out a neuron during training.
        lr_mult (float):
            The learning rate multiplier for the optimizer.
        patience (int):
            The number of epochs to wait before early stopping.
        _L_in (int):
            The number of input features.
        _L_out (int):
            The number of output classes.
        _torchmetric (str):
            The metric to use for the loss function. If `None`,
            then "mean_squared_error" is used.
        layers (nn.Sequential):
```

```
            The neural network model.

    """

    def __init__(
        self,
        l1: int,
        epochs: int,
        batch_size: int,
        initialization: str,
        act_fn: nn.Module,
        optimizer: str,
        dropout_prob: float,
        lr_mult: float,
        patience: int,
        _L_in: int,
        _L_out: int,
        _torchmetric: str,
    ):
        """
        Initializes the MyRegressor object.

        Args:
            l1 (int):
                The number of neurons in the first hidden layer.
            epochs (int):
                The number of epochs to train the model for.
            batch_size (int):
                The batch size to use during training.
            initialization (str):
                The initialization method to use for the weights.
            act_fn (nn.Module):
                The activation function to use in the hidden layers.
            optimizer (str):
                The optimizer to use during training.
            dropout_prob (float):
                The probability of dropping out a neuron during training.
            lr_mult (float):
                The learning rate multiplier for the optimizer.
            patience (int):
                The number of epochs to wait before early stopping.
            _L_in (int):
                The number of input features. Not a hyperparameter, but needed to crea
            _L_out (int):
```

```
            The number of output classes. Not a hyperparameter, but needed to create the network
        _torchmetric (str):
            The metric to use for the loss function. If `None`,
            then "mean_squared_error" is used.

    Returns:
        (NoneType): None

    Raises:
        ValueError: If l1 is less than 4.

    """
    super().__init__()
    # Attribute 'act_fn' is an instance of `nn.Module` and is already saved during
    # checkpointing. It is recommended to ignore them
    # using `self.save_hyperparameters(ignore=['act_fn'])`
    # self.save_hyperparameters(ignore=["act_fn"])
    #
    self._L_in = _L_in
    self._L_out = _L_out
    if _torchmetric is None:
        _torchmetric = "mean_squared_error"
    self._torchmetric = _torchmetric
    self.metric = getattr(torchmetrics.functional.regression, _torchmetric)
    # _L_in and _L_out are not hyperparameters, but are needed to create the network
    # _torchmetric is not a hyperparameter, but is needed to calculate the loss
    self.save_hyperparameters(ignore=["_L_in", "_L_out", "_torchmetric"])
    # set dummy input array for Tensorboard Graphs
    # set log_graph=True in Trainer to see the graph (in traintest.py)
    self.example_input_array = torch.zeros((batch_size, self._L_in))
    if self.hparams.l1 < 4:
        raise ValueError("l1 must be at least 4")
    hidden_sizes = self._get_hidden_sizes()
    # Create the network based on the specified hidden sizes
    layers = []
    layer_sizes = [self._L_in] + hidden_sizes
    layer_size_last = layer_sizes[0]
    for layer_size in layer_sizes[1:]:
        layers += [
            nn.Linear(layer_size_last, layer_size),
            self.hparams.act_fn,
            nn.Dropout(self.hparams.dropout_prob),
        ]
        layer_size_last = layer_size
```

```
        layers += [nn.Linear(layer_sizes[-1], self._L_out)]
        # nn.Sequential summarizes a list of modules into a single module, applying th
        self.layers = nn.Sequential(*layers)

    def _generate_div2_list(self, n, n_min) -> list:
        """
        Generate a list of numbers from n to n_min (inclusive) by dividing n by 2
        until the result is less than n_min.
        This function starts with n and keeps dividing it by 2 until n_min is reached
        The number of times each value is added to the list is determined by n // curr
        No more than 4 repeats of the same value (`max_repeats` below) are added to th

        Args:
            n (int): The number to start with.
            n_min (int): The minimum number to stop at.

        Returns:
            list: A list of numbers from n to n_min (inclusive).

        Examples:
            _generate_div2_list(10, 1)
            [10, 5, 5, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
            _ generate_div2_list(10, 2)
            [10, 5, 5, 2, 2, 2, 2, 2]
        """
        result = []
        current = n
        repeats = 1
        max_repeats = 4
        while current >= n_min:
            result.extend([current] * min(repeats, max_repeats))
            current = current // 2
            repeats = repeats + 1
        return result

    def _get_hidden_sizes(self):
        """
        Generate the hidden layer sizes for the network.

        Returns:
            list: A list of hidden layer sizes.

        """
        n_low = self._L_in // 4
```

```
        n_high = max(self.hparams.l1, 2 * n_low)
        hidden_sizes = self._generate_div2_list(n_high, n_low)
        return hidden_sizes

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Performs a forward pass through the model.

        Args:
            x (torch.Tensor): A tensor containing a batch of input data.

        Returns:
            torch.Tensor: A tensor containing the output of the model.

        """
        x = self.layers(x)
        return x

    def _calculate_loss(self, batch):
        """
        Calculate the loss for the given batch.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        x, y = batch
        y = y.view(len(y), 1)
        y_hat = self(x)
        loss = self.metric(y_hat, y)
        return loss

    def training_step(self, batch: tuple) -> torch.Tensor:
        """
        Performs a single training step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.
```

```
        """
        val_loss = self._calculate_loss(batch)
        # self.log("train_loss", val_loss, on_step=True, on_epoch=True, prog_bar=True)
        # self.log("train_mae_loss", mae_loss, on_step=True, on_epoch=True, prog_bar=T
        return val_loss

    def validation_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) ->
        """
        Performs a single validation step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults t

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        val_loss = self._calculate_loss(batch)
        # self.log("val_loss", val_loss, on_step=False, on_epoch=True, prog_bar=prog_b
        self.log("val_loss", val_loss, prog_bar=prog_bar)
        self.log("hp_metric", val_loss, prog_bar=prog_bar)
        return val_loss

    def test_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch
        """
        Performs a single test step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults t

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.
        """
        val_loss = self._calculate_loss(batch)
        self.log("val_loss", val_loss, prog_bar=prog_bar)
        self.log("hp_metric", val_loss, prog_bar=prog_bar)
        return val_loss

    def predict_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> to
        """
```

```
        Performs a single prediction step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults to False.

        Returns:
            A tuple containing the input data, the true labels, and the predicted values.
        """
        x, y = batch
        yhat = self(x)
        y = y.view(len(y), 1)
        yhat = yhat.view(len(yhat), 1)
        print(f"Predict step x: {x}")
        print(f"Predict step y: {y}")
        print(f"Predict step y_hat: {yhat}")
        # pred_loss = F.mse_loss(y_hat, y)
        # pred loss not registered
        # self.log("pred_loss", pred_loss, prog_bar=prog_bar)
        # self.log("hp_metric", pred_loss, prog_bar=prog_bar)
        # MisconfigurationException: You are trying to `self.log()`
        # but the loop's result collection is not registered yet.
        # This is most likely because you are trying to log in a `predict` hook, but it doesn't supp
        # If you want to manually log, please consider using `self.log_dict({'pred_loss': pred_loss}
        return (x, y, yhat)

    def configure_optimizers(self) -> torch.optim.Optimizer:
        """
        Configures the optimizer for the model.

        Notes:
            The default Lightning way is to define an optimizer as
            `optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)`.
            spotpython uses an optimizer handler to create the optimizer, which
            adapts the learning rate according to the lr_mult hyperparameter as
            well as other hyperparameters. See `spotpython.hyperparameters.optimizer.py` for details

        Returns:
            torch.optim.Optimizer: The optimizer to use during training.

        """
        # optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)
        optimizer = optimizer_handler(
```

```
            optimizer_name=self.hparams.optimizer, params=self.parameters(), lr_mult=s
        )
        return optimizer
```

52.1.5. The New `pinn_hyperdict.json` File

53. Explainable AI with SpotPython and Pytorch

Note, the `divergence_threshold` is set to 3,000, which is based on some pre-experiments with the user data set.

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_control_init)
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename
from spotpython.hyperparameters.values import set_hyperparameter
from math import inf

PREFIX="602_12_1"

data_set = Diabetes()

fun_control = fun_control_init(
    save_experiment=True,
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    divergence_threshold=3_000,
    _L_in=10,
    _L_out=1)

fun = HyperLight().fun


set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,7])
set_hyperparameter(fun_control, "epochs", [10,12])
```

```
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,9])

design_control = design_control_init(init_size=7)

S = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
```

```
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
Experiment saved to 602_12_1_exp.pkl
```

53.1. Running the Hyperparameter Tuning or Loading the Existing Model

```
S.run()
```

```
     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 24008.5703125, 'hp_metric': 24008.5703125}


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    3.2 M   train   816 M   [128, 10]    [128, 1]
```

```
Trainable params: 3.2 M
Non-trainable params: 0
Total params: 3.2 M
Total estimated model params size (MB): 12
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 816 M

train_model result: {'val_loss': 25638.1328125, 'hp_metric': 25638.1328125}



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 4298.646484375, 'hp_metric': 4298.646484375}



     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    802 K   train   1.6 B   [1024, 10]   [1024, 1]



Trainable params: 802 K
Non-trainable params: 0
Total params: 802 K
Total estimated model params size (MB): 3
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 1.6 B
```

```
train_model result: {'val_loss': 23521.087890625, 'hp_metric': 23521.087890625}


     Name     Type         Params   Mode     FLOPs     In sizes   Out sizes

  0  layers   Sequential   12.7 M   train   52.1 B   [2048, 10]   [2048, 1]


Trainable params: 12.7 M
Non-trainable params: 0
Total params: 12.7 M
Total estimated model params size (MB): 50
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 52.1 B

train_model result: {'val_loss': 234333.578125, 'hp_metric': 234333.578125}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 24054.228515625, 'hp_metric': 24054.228515625}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    807 K   train   51.2 M   [32, 10]     [32, 1]
```

```
Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.2 M

train_model result: {'val_loss': 24212.8515625, 'hp_metric': 24212.8515625}



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    807 K   train   25.6 M   [16, 10]     [16, 1]



Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.6 M

train_model result: {'val_loss': 15141.951171875, 'hp_metric': 15141.951171875}
spotpython tuning: 4298.646484375 [----------] 1.30%. Success rate: 0.00%



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    807 K   train   25.6 M   [16, 10]     [16, 1]



Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.6 M
```

```
train_model result: {'val_loss': 10541.05859375, 'hp_metric': 10541.05859375}
spotpython tuning: 4298.646484375 [----------] 3.02%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 22962.96484375, 'hp_metric': 22962.96484375}
spotpython tuning: 4298.646484375 [----------] 4.72%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   12.7 M   train   814 M   [32, 10]     [32, 1]



Trainable params: 12.7 M
Non-trainable params: 0
Total params: 12.7 M
Total estimated model params size (MB): 50
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 814 M

train_model result: {'val_loss': 77827.0234375, 'hp_metric': 77827.0234375}
spotpython tuning: 4298.646484375 [#---------] 6.92%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    807 K   train   51.2 M   [32, 10]     [32, 1]



Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.2 M

train_model result: {'val_loss': 23592.322265625, 'hp_metric': 23592.322265625}
spotpython tuning: 4298.646484375 [#---------] 7.50%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 6308.0634765625, 'hp_metric': 6308.0634765625}
spotpython tuning: 4298.646484375 [#---------] 8.54%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 24040.986328125, 'hp_metric': 24040.986328125}
spotpython tuning: 4298.646484375 [#---------] 9.87%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 20776.10546875, 'hp_metric': 20776.10546875}
spotpython tuning: 4298.646484375 [#---------] 11.57%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   25.8 M   [64, 10]     [64, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 25.8 M
```

```
train_model result: {'val_loss': 23944.212890625, 'hp_metric': 23944.212890625}
spotpython tuning: 4298.646484375 [#---------] 12.54%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 24245.560546875, 'hp_metric': 24245.560546875}
spotpython tuning: 4298.646484375 [#---------] 13.67%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    802 K   train   102 M   [64, 10]     [64, 1]



Trainable params: 802 K
Non-trainable params: 0
Total params: 802 K
Total estimated model params size (MB): 3
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 102 M

train_model result: {'val_loss': 23191.6796875, 'hp_metric': 23191.6796875}
spotpython tuning: 4298.646484375 [##--------] 15.20%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 22550.796875, 'hp_metric': 22550.796875}
spotpython tuning: 4298.646484375 [##--------] 16.81%. Success rate: 0.00%



    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    202 K   train   25.8 M   [64, 10]     [64, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 23998.98046875, 'hp_metric': 23998.98046875}
spotpython tuning: 4298.646484375 [##--------] 18.80%. Success rate: 0.00%



    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 7605.974609375, 'hp_metric': 7605.974609375}
spotpython tuning: 4298.646484375 [##--------] 20.29%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   6.6 M   [64, 10]     [64, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 24050.08984375, 'hp_metric': 24050.08984375}
spotpython tuning: 4298.646484375 [##--------] 22.40%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M
```

```
train_model result: {'val_loss': 13179.0244140625, 'hp_metric': 13179.0244140625}
spotpython tuning: 4298.646484375 [##--------] 23.71%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3.0394979793716095e+24, 'hp_metric': 3.0394979793716
spotpython tuning: 4298.646484375 [##--------] 24.62%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 24071.0078125, 'hp_metric': 24071.0078125}
spotpython tuning: 4298.646484375 [###-------] 27.18%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    807 K   train   51.2 M   [32, 10]     [32, 1]



Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.2 M

train_model result: {'val_loss': 22922.708984375, 'hp_metric': 22922.708984375}
spotpython tuning: 4298.646484375 [###-------] 27.93%. Success rate: 0.00%



    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    807 K   train   51.2 M   [32, 10]     [32, 1]



Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.2 M

train_model result: {'val_loss': 21665.01953125, 'hp_metric': 21665.01953125}
spotpython tuning: 4298.646484375 [###-------] 32.64%. Success rate: 0.00%



    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    807 K   train   51.2 M   [32, 10]     [32, 1]
```

```
Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.2 M

train_model result: {'val_loss': 24376.826171875, 'hp_metric': 24376.826171875}
spotpython tuning: 4298.646484375 [####------] 35.83%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   52.5 M   [512, 10]    [512, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 23619.865234375, 'hp_metric': 23619.865234375}
spotpython tuning: 4298.646484375 [####------] 37.80%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    802 K   train   819 M   [512, 10]    [512, 1]


Trainable params: 802 K
Non-trainable params: 0
Total params: 802 K
Total estimated model params size (MB): 3
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 819 M
```

```
train_model result: {'val_loss': 23855.919921875, 'hp_metric': 23855.919921875}
spotpython tuning: 4298.646484375 [####------] 41.26%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    802 K   train   204 M   [128, 10]    [128, 1]



Trainable params: 802 K
Non-trainable params: 0
Total params: 802 K
Total estimated model params size (MB): 3
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 204 M

train_model result: {'val_loss': 18489.638671875, 'hp_metric': 18489.638671875}
spotpython tuning: 4298.646484375 [####------] 43.86%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   6.6 M   [64, 10]     [64, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 24000.38671875, 'hp_metric': 24000.38671875}
spotpython tuning: 4298.646484375 [#####-----] 46.53%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    807 K   train   51.2 M   [32, 10]     [32, 1]



Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.2 M

train_model result: {'val_loss': 23436.99609375, 'hp_metric': 23436.99609375}
spotpython tuning: 4298.646484375 [#####-----] 50.98%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 24025.77734375, 'hp_metric': 24025.77734375}
spotpython tuning: 4298.646484375 [#####-----] 52.87%. Success rate: 0.00%



    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]
```

```
Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 24050.146484375, 'hp_metric': 24050.146484375}
spotpython tuning: 4298.646484375 [######----] 55.41%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   25.8 M   [64, 10]     [64, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 23898.478515625, 'hp_metric': 23898.478515625}
spotpython tuning: 4298.646484375 [######----] 58.03%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    807 K   train   51.2 M   [32, 10]     [32, 1]


Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.2 M
```

```
train_model result: {'val_loss': 20080.798828125, 'hp_metric': 20080.798828125}
spotpython tuning: 4298.646484375 [######----] 59.70%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   13.1 M   [128, 10]    [128, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.1 M

train_model result: {'val_loss': 24015.955078125, 'hp_metric': 24015.955078125}
spotpython tuning: 4298.646484375 [######----] 62.53%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 24050.486328125, 'hp_metric': 24050.486328125}
spotpython tuning: 4298.646484375 [#######---] 66.48%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    3.2 M   train   102 M   [16, 10]     [16, 1]



Trainable params: 3.2 M
Non-trainable params: 0
Total params: 3.2 M
Total estimated model params size (MB): 12
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 102 M

train_model result: {'val_loss': 17813.669921875, 'hp_metric': 17813.669921875}
spotpython tuning: 4298.646484375 [#######---] 70.10%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 22139.86328125, 'hp_metric': 22139.86328125}
spotpython tuning: 4298.646484375 [#######---] 73.53%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    202 K   train   6.5 M   [16, 10]     [16, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 24036.263671875, 'hp_metric': 24036.263671875}
spotpython tuning: 4298.646484375 [########--] 77.23%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  51.9 K   train   1.6 M   [16, 10]     [16, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 1.6 M

train_model result: {'val_loss': 21281.712890625, 'hp_metric': 21281.712890625}
spotpython tuning: 4298.646484375 [########--] 80.25%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  51.9 K   train   1.6 M   [16, 10]     [16, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 1.6 M
```

```
train_model result: {'val_loss': 5334.50048828125, 'hp_metric': 5334.50048828125}
spotpython tuning: 4298.646484375 [########--] 83.57%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   6.5 M   [16, 10]     [16, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.5 M

train_model result: {'val_loss': 24019.35546875, 'hp_metric': 24019.35546875}
spotpython tuning: 4298.646484375 [#########-] 87.30%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   1.6 M   [16, 10]     [16, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 1.6 M

train_model result: {'val_loss': 5027.46240234375, 'hp_metric': 5027.46240234375}
spotpython tuning: 4298.646484375 [#########-] 90.71%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    3.2 M   train   816 M   [128, 10]    [128, 1]



Trainable params: 3.2 M
Non-trainable params: 0
Total params: 3.2 M
Total estimated model params size (MB): 12
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 816 M

train_model result: {'val_loss': 724579.1875, 'hp_metric': 724579.1875}
spotpython tuning: 4298.646484375 [##########-] 91.99%. Success rate: 0.00%



    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    807 K   train   25.6 M   [16, 10]     [16, 1]



Trainable params: 807 K
Non-trainable params: 0
Total params: 807 K
Total estimated model params size (MB): 3
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.6 M

train_model result: {'val_loss': 24025.4453125, 'hp_metric': 24025.4453125}
spotpython tuning: 4298.646484375 [##########-] 94.72%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]
```

```
Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 23999.83203125, 'hp_metric': 23999.83203125}
spotpython tuning: 4298.646484375 [##########] 96.48%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    802 K   train   25.6 M   [16, 10]     [16, 1]


Trainable params: 802 K
Non-trainable params: 0
Total params: 802 K
Total estimated model params size (MB): 3
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 25.6 M

train_model result: {'val_loss': 19088.296875, 'hp_metric': 19088.296875}
spotpython tuning: 4298.646484375 [##########] 98.50%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   25.8 M   [64, 10]     [64, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.8 M
```

```
train_model result: {'val_loss': 24011.466796875, 'hp_metric': 24011.466796875}
spotpython tuning: 4298.646484375 [##########] 100.00%. Success rate: 0.00% Done...

Experiment saved to 602_12_1_res.pkl


<spotpython.spot.spot.Spot at 0x1433db620>
```

53.2. Results from the Hyperparameter Tuning Experiment

- After the hyperparameter tuning is finished, the following information is available:
  - the `S` object and the associated
  - `fun_control` dictionary

```
S.print_results(print_screen=True)
```

```
min y: 4298.646484375
l1: 4.0
epochs: 12.0
batch_size: 5.0
act_fn: 2.0
optimizer: 1.0
dropout_prob: 0.0035182324401420283
lr_mult: 5.090825861997777
patience: 2.0
batch_norm: 0.0
initialization: 1.0


[['l1', np.float64(4.0)],
 ['epochs', np.float64(12.0)],
 ['batch_size', np.float64(5.0)],
 ['act_fn', np.float64(2.0)],
 ['optimizer', np.float64(1.0)],
 ['dropout_prob', np.float64(0.0035182324401420283)],
 ['lr_mult', np.float64(5.090825861997777)],
 ['patience', np.float64(2.0)],
 ['batch_norm', np.float64(0.0)],
 ['initialization', np.float64(1.0)]]
```

```
S.plot_progress()
```

53.2.1. Getting the Best Model, i.e, the Tuned Architecture

- The method `get_tuned_architecture` [DOC] returns the best model architecture found during the hyperparameter tuning.
- It returns the transformed values, i.e., `batch_size = 2^x` if the hyperparameter `batch_size` was transformed with the `transform_power_2_int` function.

```
from spotpython.hyperparameters.values import get_tuned_architecture
import pprint
config = get_tuned_architecture(S)
pprint.pprint(config)
```

```
{'act_fn': ReLU(),
 'batch_norm': False,
 'batch_size': 32,
 'dropout_prob': 0.0035182324401420283,
 'epochs': 4096,
 'initialization': 'kaiming_uniform',
 'l1': 16,
 'lr_mult': 5.090825861997777,
 'optimizer': 'Adam',
 'patience': 4}
```

- Note: `get_tuned_architecture` has the option `force_minX` which does not have any effect in this case.

```
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(S, force_minX=True)
pprint.pprint(config)
```

```
{'act_fn': ReLU(),
 'batch_norm': False,
 'batch_size': 32,
 'dropout_prob': 0.0035182324401420283,
 'epochs': 4096,
 'initialization': 'kaiming_uniform',
 'l1': 16,
 'lr_mult': 5.090825861997777,
 'optimizer': 'Adam',
 'patience': 4}
```

53.3. Training the Tuned Architecture on the Test Data

- Since we are interested in the explainability of the model, we will train the tuned architecture on the test data.
- `spotpythons`'s `test_model` function [DOC] is used to train the model on the test data.
- Note: Until now, we do not use any information about the NN's weights and biases. Only the architecture, which is available as the `config`, is used.
- `spotpython` used the TensorBoard logger to save the training process in the `./runs` directory. Therefore, we have to enable the TensorBoard logger in the `fun_control` dictionary. To get a clean start, we remove an existing `runs` folder.

```
from spotpython.light.testmodel import test_model
from spotpython.light.loadmodel import load_light_from_checkpoint
fun_control.update({"tensorboard_log": True})
test_model(config, fun_control)
```

```
    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 202 K
Non-trainable params: 0
```

```
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M
```

Test metric	DataLoader 0
hp_metric	4042.448486328125
val_loss	4042.448486328125

```
test_model result: {'val_loss': 4042.448486328125, 'hp_metric': 4042.448486328125}
```

```
(4042.448486328125, 4042.448486328125)
```

```
model = load_light_from_checkpoint(config, fun_control)
```

```
config: {'l1': 16, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'Adam', 'dropout
Loading model with 16_4096_32_ReLU_Adam_0.0035_5.0908_4_False_kaiming_uniform_TEST from runs/saved_m
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0035182324401420283, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0035182324401420283, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.0035182324401420283, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.0035182324401420283, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): ReLU()
    (14): Dropout(p=0.0035182324401420283, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): ReLU()
    (17): Dropout(p=0.0035182324401420283, inplace=False)
```

```
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): ReLU()
    (20): Dropout(p=0.0035182324401420283, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
```

53.3.0.1. Details of the Training Process on the Test Data

- The `test_model` method initializes the model with the tuned architecture as follows:

```
model = fun_control["core_model"](**config, _L_in=_L_in, _L_out=_L_out, _torchmetric=
```

- Then, the Lightning Trainer is initialized with the `fun_control` dictionary and the model as follows:

```
    trainer = L.Trainer(
    default_root_dir=os.path.join(fun_control["CHECKPOINT_PATH"], config_id),
    max_epochs=model.hparams.epochs,
    accelerator=fun_control["accelerator"],
    devices=fun_control["devices"],
    logger=TensorBoardLogger(
        save_dir=fun_control["TENSORBOARD_PATH"],
        version=config_id,
        default_hp_metric=True,
        log_graph=fun_control["log_graph"],
    ),
    callbacks=[
        EarlyStopping(monitor="val_loss", patience=config["patience"], mode="min"
        ModelCheckpoint(
            dirpath=os.path.join(fun_control["CHECKPOINT_PATH"], config_id), save
        ),
    ],
    enable_progress_bar=enable_progress_bar,
)
trainer.fit(model=model, datamodule=dm)
test_result = trainer.test(datamodule=dm, ckpt_path="last")
```

- As shown in the code above, the last checkpoint ist saved.
- `spotpython`'s method `load_light_from_checkpoint` is used to load the last checkpoint and to get the model's weights and biases. It requires the `fun_control` dictionary and the `config_id` as input to find the correct checkpoint.
- Now, the model is trained and the weights and biases are available.

53.4. Visualizing the Neural Network Architecture

```
# get the device
from spotpython.utils.device import getDevice
device = getDevice()
```

```
from spotpython.plot.xai import viz_net
viz_net(model, device=device)
```

Figure 53.1.: architecture

53.5. XAI Methods

- `spotpython` provides methods to explain the model's predictions. The following neural network elements can be analyzed:

53.5.1. Weights

- Weights are the parameters of the neural network that are learned from the data during training. They connect neurons between layers and determine the strength and direction of the signal sent from one neuron to another. The network adjusts the weights during training to minimize the error between the predicted output and the actual output.
- Interpretation of the weights: A high weight value indicates a strong influence of the input neuron on the output. Positive weights suggest a positive correlation, whereas negative weights suggest an inverse relationship between neurons.

53.5.2. Activations

- Activations are the outputs produced by neurons after applying an activation function to the weighted sum of inputs. The activation function (e.g., ReLU, sigmoid, tanh) adds non-linearity to the model, allowing it to learn more complex relationships.
- Interpretation of the activations: The value of activations indicates the intensity of the signal passed to the next layer. Certain activation patterns can highlight which features or parts of the data the network is focusing on.

53.5.3. Gradients

- Gradients are the partial derivatives of the loss function with respect to different parameters (weights) of the network. During backpropagation, gradients are used to update the weights in the direction that reduces the loss by methods like gradient descent.
- Interpretation of the gradients: The magnitude of the gradient indicates how much a parameter should change to reduce the error. A large gradient implies a steeper slope and a bigger update, while a small gradient suggests that the parameter is near an optimal point. If gradients are too small (vanishing gradient problem), the network may learn slowly or stop learning. If they are too large (exploding gradient problem), the updates may be unstable.
- `sptpython` provides the method `get_gradients` to get the gradients of the model.

```
from spotpython.plot.xai import (get_activations, get_gradients, get_weights, visualize_weights, vis
batch_size = config["batch_size"]
```

53.5.4. Getting the Weights

```
from spotpython.plot.xai import sort_layers
weights, _ = get_weights(model)
# sort_layers(weights)
```

```
visualize_weights(model, absolute=True, cmap="GreenYellowRed", figsize=(6, 6))
```

```
3200 values in Layer Layer 0. Geometry: (320, 10)
```

```
51200 values in Layer Layer 3. Geometry: (160, 320)
```

```
51200 values in Layer Layer 6. Geometry: (320, 160)
```

```
51200 values in Layer Layer 9. Geometry: (160, 320)
```

```
25600 values in Layer Layer 12. Geometry: (160, 160)
```

```
12800 values in Layer Layer 15. Geometry: (80, 160)
```

```
6400 values in Layer Layer 18. Geometry: (80, 80)
```

```
80 values in Layer Layer 21. Geometry: (1, 80)
```

```
visualize_weights_distributions(model, color=f"C{0}", columns=4)
```

```
n:8
```

53.5.5. Getting the Activations

```
from spotpython.plot.xai import get_activations
activations, mean_activations, layer_sizes = get_activations(net=model, fun_control=fun_control, bat
```

```
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting train & val data.
train samples: 160, val samples: 106 generated for train & val data.
LightDataModule.train_dataloader(). data_train size: 160
```

```
visualize_mean_activations(mean_activations, layer_sizes=layer_sizes, absolute=True, cmap="GreenYell
```

```
320 values in Layer 0. Geometry: (1, 320)
```

```
160 values in Layer 3. Geometry: (1, 160)
```

```
320 values in Layer 6. Geometry: (1, 320)
```

```
160 values in Layer 9. Geometry: (1, 160)
```

```
160 values in Layer 12. Geometry: (1, 160)
```

```
80 values in Layer 15. Geometry: (1, 80)
```

```
80 values in Layer 18. Geometry: (1, 80)
```

```
visualize_activations_distributions(activations=activations,
                                    net=model, color="C0", columns=4)
```

53.5.6. Getting the Gradients

```
gradients, _ = get_gradients(net=model, fun_control=fun_control, batch_size=batch_size
```

```
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting train & val data.
train samples: 160, val samples: 106 generated for train & val data.
LightDataModule.train_dataloader(). data_train size: 160
```

```
visualize_gradients(model, fun_control, batch_size, absolute=True, cmap="GreenYellowRed", figsize=(6
```

```
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting train & val data.
train samples: 160, val samples: 106 generated for train & val data.
LightDataModule.train_dataloader(). data_train size: 160
3200 values in Layer layers.0.weight. Geometry: (320, 10)
```

```
51200 values in Layer layers.3.weight. Geometry: (160, 320)
```

```
51200 values in Layer layers.6.weight. Geometry: (320, 160)
```

```
51200 values in Layer layers.9.weight. Geometry: (160, 320)
```

```
25600 values in Layer layers.12.weight. Geometry: (160, 160)
```

```
12800 values in Layer layers.15.weight. Geometry: (80, 160)
```

```
6400 values in Layer layers.18.weight. Geometry: (80, 80)
```

```
80 values in Layer layers.21.weight. Geometry: (1, 80)
```

```
visualize_gradient_distributions(model, fun_control, batch_size=batch_size, color=f"C-
```

```
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting train & val data.
train samples: 160, val samples: 106 generated for train & val data.
LightDataModule.train_dataloader(). data_train size: 160
n:8
```

53.6. Feature Attributions

53.6.1. Integrated Gradients

```
from spotpython.plot.xai import get_attributions, plot_attributions
df_att = get_attributions(S, fun_control, attr_method="IntegratedGradients", n_rel=10)
plot_attributions(df_att, attr_method="IntegratedGradients")
```

```
     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M


train_model result: {'val_loss': 20680.001953125, 'hp_metric': 20680.001953125}
config: {'l1': 16, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'Adam', 'dropout
Loading model with 16_4096_32_ReLU_Adam_0.0035_5.0908_4_False_kaiming_uniform_TRAIN from runs/saved_
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0035182324401420283, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0035182324401420283, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.0035182324401420283, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.0035182324401420283, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
```

```
    (13): ReLU()
    (14): Dropout(p=0.0035182324401420283, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): ReLU()
    (17): Dropout(p=0.0035182324401420283, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): ReLU()
    (20): Dropout(p=0.0035182324401420283, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting test data.
test samples: 177 generated for test data.
LightDataModule.test_dataloader(). Test set size: 177
```

53.6.2. Deep Lift

```
df_lift = get_attributions(S, fun_control, attr_method="DeepLift",n_rel=10)
print(df_lift)
plot_attributions(df_lift,  attr_method="DeepLift")
```

```
    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M


train_model result: {'val_loss': 4244.9228515625, 'hp_metric': 4244.9228515625}
config: {'l1': 16, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'Adam', 'dropout
Loading model with 16_4096_32_ReLU_Adam_0.0035_5.0908_4_False_kaiming_uniform_TRAIN from runs/saved_
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0035182324401420283, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0035182324401420283, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.0035182324401420283, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.0035182324401420283, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): ReLU()
    (14): Dropout(p=0.0035182324401420283, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): ReLU()
    (17): Dropout(p=0.0035182324401420283, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): ReLU()
    (20): Dropout(p=0.0035182324401420283, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
```

```
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting test data.
test samples: 177 generated for test data.
LightDataModule.test_dataloader(). Test set size: 177
   Feature Index Feature  DeepLiftAttribution
0              1     sex          1026.236694
1              9  s6_glu          1026.086426
2              0     age          1012.324463
3              6  s3_hdl           988.167114
4              3      bp           953.655762
5              4   s1_tc           916.110718
6              8  s5_ltg           913.878418
7              7  s4_tch           854.368530
8              5  s2_ldl           848.338867
9              2     bmi           782.169434
```

53.6.3. Feature Ablation

```
df_fl = get_attributions(S, fun_control, attr_method="FeatureAblation",n_rel=10)
```

```
     Name      Type          Params   Mode     FLOPs   In sizes   Out sizes
```

```
  0   layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 6053.70849609375, 'hp_metric': 6053.70849609375}
config: {'l1': 16, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'Adam', 'dropout
Loading model with 16_4096_32_ReLU_Adam_0.0035_5.0908_4_False_kaiming_uniform_TRAIN from runs/saved_
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0035182324401420283, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0035182324401420283, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.0035182324401420283, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.0035182324401420283, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): ReLU()
    (14): Dropout(p=0.0035182324401420283, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): ReLU()
    (17): Dropout(p=0.0035182324401420283, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): ReLU()
    (20): Dropout(p=0.0035182324401420283, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting test data.
test samples: 177 generated for test data.
LightDataModule.test_dataloader(). Test set size: 177
```

```
print(df_fl)
plot_attributions(df_fl, attr_method="FeatureAblation")
```

```
   Feature Index Feature  FeatureAblationAttribution
0                9  s6_glu                  577.062134
1                0     age                  556.625305
2                6  s3_hdl                  554.604370
3                3      bp                  528.497559
4                8  s5_ltg                  527.454163
5                4   s1_tc                  523.947937
6                1     sex                  519.197693
7                5  s2_ldl                  463.207764
8                7  s4_tch                  440.091187
9                2     bmi                  371.574615
```

53.7. Conductance

```
from spotpython.plot.xai import plot_conductance_last_layer, get_weights_conductance_l
weights_last, layer_conductance_last = get_weights_conductance_last_layer(S, fun_contr
plot_conductance_last_layer(weights_last, layer_conductance_last, figsize=(6, 6))
```

```
    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 11901.6708984375, 'hp_metric': 11901.6708984375}
config: {'l1': 16, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'Adam', 'dropout
Loading model with 16_4096_32_ReLU_Adam_0.0035_5.0908_4_False_kaiming_uniform_TRAIN from runs/saved_

Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0035182324401420283, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0035182324401420283, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.0035182324401420283, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.0035182324401420283, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): ReLU()
    (14): Dropout(p=0.0035182324401420283, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): ReLU()
    (17): Dropout(p=0.0035182324401420283, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): ReLU()
    (20): Dropout(p=0.0035182324401420283, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
```

```
     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 14553.958984375, 'hp_metric': 14553.958984375}
config: {'l1': 16, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'A
Loading model with 16_4096_32_ReLU_Adam_0.0035_5.0908_4_False_kaiming_uniform_TRAIN fr
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0035182324401420283, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0035182324401420283, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.0035182324401420283, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.0035182324401420283, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): ReLU()
    (14): Dropout(p=0.0035182324401420283, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): ReLU()
    (17): Dropout(p=0.0035182324401420283, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): ReLU()
    (20): Dropout(p=0.0035182324401420283, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
Conductance analysis for layer:  Linear(in_features=80, out_features=1, bias=True)
```

Layer Conductance vs. Weights
Weights
Layer Conductance
Normalized Value
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
10
20
30
40
50
60
70
80
Weight Index

54. HPT PyTorch Lightning Transformer: Introduction

In this chapter, we will introduce transformer. The transformer architecture is a neural network architecture that is based on the attention mechanism (Vaswani et al. 2017). It is particularly well suited for sequence-to-sequence tasks, such as machine translation, text summarization, and more. The transformer architecture has been a breakthrough in the field of natural language processing (NLP) and has been the basis for many state-of-the-art models in the field.

We start with a description of the transformer basics in Section 54.1. Section 54.2 provides a detailed description of the implementation of the transformer architecture. Finally, an example of a transformer implemented in PyTorch Lightning in presented in Section 54.3.

54.1. Transformer Basics

54.1.1. Embedding

Word embedding is a technique where words or phrases (so-called tokens) from the vocabulary are mapped to vectors of real numbers. These vectors capture the semantic properties of the words. Words that are similar in meaning are mapped to vectors that are close to each other in the vector space, and words that are dissimilar are mapped to vectors that are far apart. Word embeddings are needed for transformers for several reasons:

- Dimensionality reduction: Word embeddings reduce the dimensionality of the data. Instead of dealing with high-dimensional sparse vectors (like one-hot encoded vectors), we deal with dense vectors of much lower dimensionality.
- Capturing semantic similarities: Word embeddings capture semantic similarities between words. This is crucial for tasks like text classification, sentiment analysis, etc., where the meaning of the words is important.
- Handling unknown words: If a word is not present in the training data but appears in the test data, one-hot encoding cannot handle it. But word embeddings can handle such situations by mapping the unknown word to a vector that is similar to known words.

- Input to neural networks: Transformers, like other neural networks, work with numerical data. Word embeddings provide a way to convert text data into numerical form that can be fed into these networks.

In the context of transformers, word embeddings are used as the initial input representation. The transformer then learns more complex representations by considering the context in which each token appears.

54.1.1.1. Neural Network for Embeddings

Idea for word embeddings: use a relatively simple NN that has one input for every token (word, symbol) in the vocabulary. The output of the NN is a vector of a fixed size, which is the word embedding. The network that is used in this chapter is visualized in Figure 54.1. For simplicity, a 2-dimensional output vector is used in this visualization. The weights of the NN are randomly initialized, and are learned during training.

All tokens are embedded in this way. For each token there are two numerical values, the embedding vector. The same network is used for embedding all tokens. If a longer input is added, it can be embedded with the same net.

54.1.1.2. Positional Encoding for the Embeddings

Positional encoding is added to the input embeddings to give the model some information about the relative or absolute position of the tokens in the sequence. The positional encodings have the same dimension as the embeddings so that the two can be summed.

If a token occurs several times, it is embedded several times and receives different embedding vectors, as the position is taken into account by the positional encoding.

54.1.2. Attention

Attention describes how similar is each token to itself and to all other tokens in the input, e.g., in a sentence. The attention mechanism can be implemented as a set of layers in neural networks. There are a lot of different possible definitions of "attention" in the literature, but the one we will use here is the following: *the attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and elements' keys* (Lippe 2022).

The goal is to take an average over the features of multiple elements. However, instead of weighting each element equally, we want to weight them depending on their actual values. In other words, we want to dynamically decide on which inputs we want to "attend" more than others.

Calculation of the self-attention:

Figure 54.1.: Transformer. Computation of the self attention. In this example, we consider two inputs, i.e., (1,0) and (0,1). For each input, there are two values, which results in a 2×2 matrix. In general, when there are T inputs, a $T \times T$ matrix will be generated. Figure credits: Starmer, Josh: Decoder-Only Transformers, ChatGPTs specific Transformer, Clearly Explained.

1. Queries: Calculate two new values from the (two) values of the embedding vector using an NN, which are referred to as query values.
2. Keys: Calculate two new values, called key values, from the (two) values of the embedding vector using an NN.
3. Dot product: Calculate the dot product of the query values and the key values. This is a measure of the similarity of the query and key values.
4. Softmax: Apply the softmax function to the outputs from the dot product. This is a measure of the attention that a token pays to other tokens.
5. Values: Calculate two new values from the (two) values of the embedding vector using an NN, which are referred to as value values.
6. The values are multiplied (weighted) by the values of the softmax function.
7. The weighted values are summed. Now we have the self attention value for the token.

54.1.3. Self-Attention

Most attention mechanisms differ in terms of what queries they use, how the key and value vectors are defined, and what score function is used. The attention applied inside the Transformer architecture is called "self-attention". In self-attention, each sequence element provides a key, value, and query. For each element, we perform an attention layer where based on its query, we check the similarity of the all sequence elements' keys, and returned a different, averaged value vector for each element.

54.1.4. Masked Self-Attention

Masked self-attention is a variant of the self-attention method described in Section 54.1.3. It asks the question: How similar is each token to itself and to all preceding tokens in the input (sentence)? Masked self-attention is an autoregressive mechanism, which means that the attention mechanism is only allowed to look at the tokens that have already been processed. Calculation of the mask self-attention is identical to the self-attention, but the attention is only calculated for the tokens that have already been processed. If the masked self-attention method is applied to the first token, the masked self-attention value is exactly the value of the first token, as it only takes itself into account. For the other tokens, the masked self-attention value is a weighted sum of the values of the previous tokens. The weighting is determined by the similarity of the query values and the key values (dot product and softmax).

54.1.5. Generation of Outputs

To calculate the output, we use a residual connector that adds the output of the neural network and the output of the masked self-attention method. We thus obtain the residual connection values. The residual connector is used to facilitate training.

To generate the next token, we use another neural network that calculates the output from the (two) residual connection values. The input layer of the neural network has the size of the residual connection values, the output layer has the number of tokens in the vocabulary as a dimension.

If we now enter the residual connection value of the first token, we receive the token (or the probabilities using Softmax) that is to come next as the output of the neural network. This makes sense even if we already know the second token (as with the first token): We can use it to calculate the error of the neural network and train the network. In addition, the decoder-transformer uses the masked self-attention method to calculate the output, i.e. the encoding and generation of new tokens is done with exactly the same elements of the network.

Note: ChatGPT does not use a new neural network, but the same network that was already used to calculate the embedding. The network is therefore used for embedding, masked self-attention and calculating the output. In the last calculation, the network is inverted, i.e. it is run in the opposite direction to obtain the tokens and not the embeddings as in the original run.

54.1.6. End-Of-Sequence-Token

The end-of-sequence token is used to signal the end of the input and also to start generating new tokens after the input. The EOS token recognizes all other tokens, as it comes after all tokens. When generating tokens, it is important to consider the relationships between the input tokens and the generation of new tokens.

54.2. Details of the Implementation

We will now go into a bit more detail by first looking at the specific implementation of the attention mechanism which is in the Transformer case the (scaled) dot product attention. The variables shown in Table 54.1 are used in the Transformer architecture.

Table 54.1.: Variables used in the Transformer architecture.

Symbol	Variable	Description
Q	`query`	The query vectors.
K	`key`	The key vectors.
V	`value`	The value vectors.
d_{model}	`d_model`	The dimensionality of the input and output features of the Transformer.
d_k	`d_k`	The hidden dimensionality of the key and query vectors.

Symbol	Variable	Description
d_v	`d_v`	The hidden dimensionality of the value vectors.
h	`num_heads`	The number of heads in the Multi-Head Attention layer.
B	`batch_size`	The batch size.
T	`seq_length`	The sequence length.
X	`x`	The input features (input elements in the sequence).
W^Q	`qkv_proj`	The weight matrix to transform the input to the query vectors.
W^K	`qkv_proj`	The weight matrix to transform the input to the key vectors.
W^V	`qkv_proj`	The weight matrix to transform the input to the value vectors.
W^O	`o_proj`	The weight matrix to transform the concatenated output of the Multi-Head Attention layer to the final output.
N	`num_layers`	The number of layers in the Transformer.
$PE_{(pos,i)}$	`positional_encoding`	The positional encoding for position pos and hidden dimensionality i.

Summarizing the ideas from Section 54.1, an attention mechanism has usually four parts we need to specify (Lippe 2022):

- *Query*: The query is a feature vector that describes what we are looking for in the sequence, i.e., what would we maybe want to pay attention to.
- *Keys*: For each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is "offering", or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to based on the query.
- *Score function*: To rate which elements we want to pay attention to, we need to specify a score function f_{attn}. The score function takes the query and a key as input, and output the score/attention weight of the query-key pair. It is usually implemented by simple similarity metrics like a dot product, or a small MLP.
- *Values*: For each input element, we also have a value vector. This feature vector is the one we want to average over.

The weights of the average are calculated by a softmax over all score function outputs. Hence, we assign those value vectors a higher weight whose corresponding key is most similar to the query. If we try to describe it with pseudo-math, we can write:

$$\alpha_i = \frac{\exp\left(f_{attn}\left(\text{key}_i, \text{query}\right)\right)}{\sum_j \exp\left(f_{attn}\left(\text{key}_j, \text{query}\right)\right)}, \qquad \text{out} = \sum_i \alpha_i \cdot \text{value}_i$$

Visually, we can show the attention over a sequence of words as follows:

Figure 54.2.: Attention over a sequence of words. For every word, we have one key and one value vector. The query is compared to all keys with a score function (in this case the dot product) to determine the weights. The softmax is not visualized for simplicity. Finally, the value vectors of all words are averaged using the attention weights. Figure taken from Lippe (2022)

54.2.1. Dot Product Attention

Our goal is to have an attention mechanism with which any element in a sequence can attend to any other while still being efficient to compute. The dot product attention takes as input a set of queries $Q \in \mathbb{R}^{T \times d_k}$, keys $K \in \mathbb{R}^{T \times d_k}$ and values $V \in \mathbb{R}^{T \times d_v}$ where T is the sequence length, and d_k and d_v are the hidden dimensionality for queries/keys and values respectively. For simplicity, we neglect the batch dimension for now. The attention value from element i to j is based on its similarity of the query Q_i and key K_j, using the dot product as the similarity metric (in Figure 54.1, we considered Q_2 and K_1 as well as Q_2 and K_2). The dot product attention is calculated as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(QK^T\right)V \tag{54.1}$$

The matrix multiplication QK^T performs the dot product for every possible pair of queries and keys, resulting in a matrix of the shape $T \times T$. Each row represents the attention logits for a specific element i to all other elements in the sequence. On these, we apply a softmax and multiply with the value vector to obtain a weighted mean (the weights being determined by the attention).

54.2.2. Scaled Dot Product Attention

An additional aspect is the scaling of the dot product using a scaling factor of $1/\sqrt{d_k}$. This scaling factor is crucial to maintain an appropriate variance of attention values after initialization. We initialize our layers with the intention of having equal variance throughout the model, and hence, Q and K might also have a variance close to 1. However, performing a dot product over two vectors with a variance σ^2 results in a scalar having d_k-times higher variance:

$$q_i \sim \mathcal{N}(0, \sigma^2), k_i \sim \mathcal{N}(0, \sigma^2) \rightarrow \text{Var}\left(\sum_{i=1}^{d_k} q_i \cdot k_i\right) = \sigma^4 \cdot d_k$$

If we do not scale down the variance back to $\sim \sigma^2$, the softmax over the logits will already saturate to 1 for one random element and 0 for all others. The gradients through the softmax will be close to zero so that we can't learn the parameters appropriately. Note that the extra factor of σ^2, i.e., having σ^4 instead of σ^2, is usually not an issue, since we keep the original variance σ^2 close to 1 anyways. Equation 54.1 can be modified as follows to calculate the dot product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V.$$

Another perspective on this scaled dot product attention mechanism offers the computation graph which is visualized in Figure 54.3.

Figure 54.3.: Scaled dot product attention. Figure credit Vaswani et al. (2017)

The block `Mask (opt.)` in the diagram above represents the optional masking of specific entries in the attention matrix. This is for instance used if we stack multiple sequences with different lengths into a batch. To still benefit from parallelization in

PyTorch, we pad the sentences to the same length and mask out the padding tokens during the calculation of the attention values. This is usually done by setting the respective attention logits to a very low value.

After we have discussed the details of the scaled dot product attention block, we can write a function below which computes the output features given the triple of queries, keys, and values:

54.3. Example: Transformer in Lightning

The following code is based on https://github.com/phlippe/uvadlc_notebooks/tree/master (Author: Phillip Lippe)

First, we import the necessary libraries and download the pretrained models.

```
import os
import numpy as np
import random
import math
import json
from functools import partial
import matplotlib.pyplot as plt
from matplotlib.colors import to_rgb
import matplotlib
import seaborn as sns

## tqdm for loading bars
from tqdm.notebook import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim

# PyTorch Lightning
import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
```

```
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial6"

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
```

```
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

```
from spotpython.utils.device import getDevice
device = getDevice()
print("Device:", device)
```

```
Device: mps
```

```
# Setting the seed
pl.seed_everything(42)
```

```
42
```

Two pre-trained models are downloaded below. Make sure to have adjusted your `CHECKPOINT_PATH` before running this code if not already done.

```
import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial6/"
# Files to download
pretrained_files = ["ReverseTask.ckpt", "SetAnomalyTask.ckpt"]

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)
```

54.3.1. Downloading the Pretrained Models

```
# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/",1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Error:\n", e)
```

54.3.2. The Transformer Architecture

We will implement the Transformer architecture by hand. As the architecture is so popular, there already exists a Pytorch module `nn.Transformer` (documentation) and a tutorial on how to use it for next token prediction. However, we will implement it here ourselves, to get through to the smallest details.

54.3.3. Attention Mechanism

```
def scaled_dot_product(q, k, v, mask=None):
    """
    Compute scaled dot product attention.
    Args:
        q: Queries
        k: Keys
        v: Values
        mask: Mask to apply to the attention logits

    Returns:
        Tuple of (Values, Attention weights)

    Examples:
    >>> seq_len, d_k = 1, 2
        pl.seed_everything(42)
        q = torch.randn(seq_len, d_k)
        k = torch.randn(seq_len, d_k)
        v = torch.randn(seq_len, d_k)
        values, attention = scaled_dot_product(q, k, v)
        print("Q\n", q)
        print("K\n", k)
        print("V\n", v)
        print("Values\n", values)
        print("Attention\n", attention)
    """
    d_k = q.size()[-1]
    attn_logits = torch.matmul(q, k.transpose(-2, -1))
    attn_logits = attn_logits / math.sqrt(d_k)
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 0, -9e15)
    attention = F.softmax(attn_logits, dim=-1)
    values = torch.matmul(attention, v)
    return values, attention
```

Note that our code above supports any additional dimensionality in front of the sequence length so that we can also use it for batches. However, for a better understanding, let's generate a few random queries, keys, and value vectors, and calculate the attention outputs:

```
seq_len, d_k = 1, 2
pl.seed_everything(42)
q = torch.randn(seq_len, d_k)
k = torch.randn(seq_len, d_k)
v = torch.randn(seq_len, d_k)
values, attention = scaled_dot_product(q, k, v)
print("Q\n", q)
print("K\n", k)
print("V\n", v)
print("Values\n", values)
print("Attention\n", attention)
```

```
Q
 tensor([[0.3367, 0.1288]])
K
 tensor([[0.2345, 0.2303]])
V
 tensor([[-1.1229, -0.1863]])
Values
 tensor([[-1.1229, -0.1863]])
Attention
 tensor([[1.]])
```

54.3.4. Multi-Head Attention

The scaled dot product attention allows a network to attend over a sequence. However, often there are multiple different aspects a sequence element wants to attend to, and a single weighted average is not a good option for it. This is why we extend the attention mechanisms to multiple heads, i.e. multiple different query-key-value triplets on the same features. Specifically, given a query, key, and value matrix, we transform those into h sub-queries, sub-keys, and sub-values, which we pass through the scaled dot product attention independently. Afterward, we concatenate the heads and combine them with a final weight matrix. Mathematically, we can express this operation as:

$$\begin{aligned}\text{Multihead}(Q,K,V) &= \text{Concat}(\text{head}_1,...,\text{head}_h)W^O\\ \text{where head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)\end{aligned}$$

We refer to this as Multi-Head Attention layer with the learnable parameters $W^Q_{1\ldots h} \in \mathbb{R}^{D\times d_k}$, $W^K_{1\ldots h} \in \mathbb{R}^{D\times d_k}$, $W^V_{1\ldots h} \in \mathbb{R}^{D\times d_v}$, and $W^O \in \mathbb{R}^{h\cdot d_v\times d_{out}}$ (D being the input dimensionality). Expressed in a computational graph, we can visualize it as in Figure 54.4.

Figure 54.4.: Multi-Head Attention. Figure taken from Vaswani et al. (2017)

How are we applying a Multi-Head Attention layer in a neural network, where we do not have an arbitrary query, key, and value vector as input? Looking at the computation graph in Figure 54.4, a simple but effective implementation is to set the current feature map in a NN, $X \in \mathbb{R}^{B\times T\times d_{\text{model}}}$, as Q, K and V (B being the batch size, T the sequence length, d_{model} the hidden dimensionality of X). The consecutive weight matrices W^Q, W^K, and W^V can transform X to the corresponding feature vectors that represent the queries, keys, and values of the input. Using this approach, we can implement the Multi-Head Attention module below.

As a consequence, if the embedding dimension is 4, then 1, 2 or 4 heads can be used, but not 3. If 4 heads are used, then the dimension of the query, key and value vectors is 1. If 2 heads are used, then the dimension of the query, key and value vectors is $D = 2$. If 1 head is used, then the dimension of the query, key and value vectors is $D = 4$. The number of heads is a hyperparameter that can be adjusted. The number of heads is usually 8 or 16.

```
# Helper function to support different mask shapes.
# Output shape supports (batch_size, number of heads, seq length, seq length)
# If 2D: broadcasted over batch size and number of heads
# If 3D: broadcasted over number of heads
# If 4D: leave as is
def expand_mask(mask):
    assert mask.ndim >= 2, "Mask must be >= 2-dim. with seq_length x seq_length"
```

```
    if mask.ndim == 3:
        mask = mask.unsqueeze(1)
    while mask.ndim < 4:
        mask = mask.unsqueeze(0)
    return mask
```

```
class MultiheadAttention(nn.Module):

    def __init__(self, input_dim, embed_dim, num_heads):
        super().__init__()
        assert embed_dim % num_heads == 0, "Embedding dim. must be 0 modulo number of

        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        # Stack all weight matrices 1...h together for efficiency
        # Note that in many implementations you see "bias=False" which is optional
        self.qkv_proj = nn.Linear(input_dim, 3*embed_dim)
        self.o_proj = nn.Linear(embed_dim, embed_dim)

        self._reset_parameters()

    def _reset_parameters(self):
        # Original Transformer initialization, see PyTorch documentation
        nn.init.xavier_uniform_(self.qkv_proj.weight)
        self.qkv_proj.bias.data.fill_(0)
        nn.init.xavier_uniform_(self.o_proj.weight)
        self.o_proj.bias.data.fill_(0)

    def forward(self, x, mask=None, return_attention=False):
        batch_size, seq_length, _ = x.size()
        if mask is not None:
            mask = expand_mask(mask)
        qkv = self.qkv_proj(x)

        # Separate Q, K, V from linear output
        qkv = qkv.reshape(batch_size, seq_length, self.num_heads, 3*self.head_dim)
        qkv = qkv.permute(0, 2, 1, 3) # [Batch, Head, SeqLen, Dims]
        q, k, v = qkv.chunk(3, dim=-1)

        # Determine value outputs
        values, attention = scaled_dot_product(q, k, v, mask=mask)
        values = values.permute(0, 2, 1, 3) # [Batch, SeqLen, Head, Dims]
```

```
        values = values.reshape(batch_size, seq_length, self.embed_dim)
        o = self.o_proj(values)

        if return_attention:
            return o, attention
        else:
            return o
```

54.3.5. Permutation Equivariance

One crucial characteristic of the multi-head attention is that it is permutation-equivariant with respect to its inputs. This means that if we switch two input elements in the sequence, e.g. $X_1 \leftrightarrow X_2$ (neglecting the batch dimension for now), the output is exactly the same besides the elements 1 and 2 switched. Hence, the multi-head attention is actually looking at the input not as a sequence, but as a set of elements. This property makes the multi-head attention block and the Transformer architecture so powerful and widely applicable! But what if the order of the input is actually important for solving the task, like language modeling? The answer is to encode the position in the input features, which we will take a closer look in Section 54.3.8.

54.3.6. Transformer Encoder

Next, we will look at how to apply the multi-head attention block inside the Transformer architecture. Originally, the Transformer model was designed for machine translation. Hence, it got an encoder-decoder structure where the encoder takes as input the sentence in the original language and generates an attention-based representation. On the other hand, the decoder attends over the encoded information and generates the translated sentence in an autoregressive manner, as in a standard RNN. While this structure is extremely useful for Sequence-to-Sequence tasks with the necessity of autoregressive decoding, we will focus here on the encoder part. Many advances in NLP have been made using pure encoder-based Transformer models (if interested, models include the BERT-family (Devlin et al. 2018), the Vision Transformer (Dosovitskiy et al. 2020), and more). We will also mainly focus on the encoder part. If you have understood the encoder architecture, the decoder is a very small step to implement as well. The full Transformer architecture looks as shown in Figure 54.5.

The encoder consists of N identical blocks that are applied in sequence. Taking as input x, it is first passed through a Multi-Head Attention block as we have implemented above. The output is added to the original input using a residual connection, and we apply a consecutive Layer Normalization on the sum. Overall, it calculates

$$\text{LayerNorm}(x + \text{Multihead}(x, x, x))$$

Figure 54.5.: Transformer architecture. Figure credit: Vaswani et al. (2017)

(x being Q, K and V input to the attention layer). The residual connection is crucial in the Transformer architecture for two reasons:

1. Similar to ResNets, Transformers are designed to be very deep. Some models contain more than 24 blocks in the encoder. Hence, the residual connections are crucial for enabling a smooth gradient flow through the model.
2. Without the residual connection, the information about the original sequence is lost. Remember that the Multi-Head Attention layer ignores the position of elements in a sequence, and can only learn it based on the input features. Removing the residual connections would mean that this information is lost after the first attention layer (after initialization), and with a randomly initialized query and key vector, the output vectors for position i has no relation to its original input. All outputs of the attention are likely to represent similar/same information, and there is no chance for the model to distinguish which information came from which input element. An alternative option to residual connection would be to fix at least one head to focus on its original input, but this is very inefficient and does not have the benefit of the improved gradient flow.

54.3.7. Layer Normalization and Feed-Forward Network

The Layer Normalization also plays an important role in the Transformer architecture as it enables faster training and provides small regularization. Additionally, it ensures that the features are in a similar magnitude among the elements in the sequence.

We are not using Batch Normalization because it depends on the batch size which is often small with Transformers (they require a lot of GPU memory), and BatchNorm has shown to perform particularly bad in language as the features of words tend to have a much higher variance (there are many, very rare words which need to be considered for a good distribution estimate).

Additionally to the Multi-Head Attention, a small fully connected feed-forward network is added to the model, which is applied to each position separately and identically. Specifically, the model uses a Linear→ReLU→Linear MLP. The full transformation including the residual connection can be expressed as:

$$\begin{aligned} \mathrm{FFN}(x) &= \max(0, xW_1 + b_1)W_2 + b_2 \\ x &= \mathrm{LayerNorm}(x + \mathrm{FFN}(x)) \end{aligned}$$

This MLP adds extra complexity to the model and allows transformations on each sequence element separately. You can imagine as this allows the model to "post-process" the new information added by the previous Multi-Head Attention, and prepare it for the next attention block. Usually, the inner dimensionality of the MLP is 2-8× larger than d_{model}, i.e. the dimensionality of the original input x. The general advantage of a wider layer instead of a narrow, multi-layer MLP is the faster, parallelizable execution.

Finally, after looking at all parts of the encoder architecture, we can start implementing it below. We first start by implementing a single encoder block. Additionally to the layers described above, we will add dropout layers in the MLP and on the output of the MLP and Multi-Head Attention for regularization.

```
class EncoderBlock(nn.Module):

    def __init__(self, input_dim, num_heads, dim_feedforward, dropout=0.0):
        """
        Inputs:
            input_dim - Dimensionality of the input
            num_heads - Number of heads to use in the attention block
            dim_feedforward - Dimensionality of the hidden layer in the MLP
            dropout - Dropout probability to use in the dropout layers
        """
        super().__init__()

        # Attention layer
        self.self_attn = MultiheadAttention(input_dim, input_dim, num_heads)

        # Two-layer MLP
        self.linear_net = nn.Sequential(
            nn.Linear(input_dim, dim_feedforward),
            nn.Dropout(dropout),
            nn.ReLU(inplace=True),
            nn.Linear(dim_feedforward, input_dim)
        )

        # Layers to apply in between the main layers
        self.norm1 = nn.LayerNorm(input_dim)
        self.norm2 = nn.LayerNorm(input_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Attention part
        attn_out = self.self_attn(x, mask=mask)
        x = x + self.dropout(attn_out)
        x = self.norm1(x)

        # MLP part
        linear_out = self.linear_net(x)
        x = x + self.dropout(linear_out)
        x = self.norm2(x)

        return x
```

Based on this block, we can implement a module for the full Transformer encoder. Additionally to a forward function that iterates through the sequence of encoder blocks, we also provide a function called `get_attention_maps`. The idea of this function is to return the attention probabilities for all Multi-Head Attention blocks in the encoder. This helps us in understanding, and in a sense, explaining the model. However, the attention probabilities should be interpreted with a grain of salt as it does not necessarily reflect the true interpretation of the model (there is a series of papers about this, including Jain and Wallace (2019) and Wiegreffe and Pinter (2019)).

```
class TransformerEncoder(nn.Module):

    def __init__(self, num_layers, **block_args):
        super().__init__()
        self.layers = nn.ModuleList(
            [EncoderBlock(**block_args) for _ in range(num_layers)])

    def forward(self, x, mask=None):
        for l in self.layers:
            x = l(x, mask=mask)
        return x

    def get_attention_maps(self, x, mask=None):
        attention_maps = []
        for l in self.layers:
            _, attn_map = l.self_attn(x, mask=mask, return_attention=True)
            attention_maps.append(attn_map)
            x = l(x)
        return attention_maps
```

54.3.8. Positional Encoding

We have discussed before that the Multi-Head Attention block is permutation-equivariant, and cannot distinguish whether an input comes before another one in the sequence or not. In tasks like language understanding, however, the position is important for interpreting the input words. The position information can therefore be added via the input features. We could learn a embedding for every possible position, but this would not generalize to a dynamical input sequence length. Hence, the better option is to use feature patterns that the network can identify from the features and potentially generalize to larger sequences. The specific pattern chosen by Vaswani et al. (2017) are sine and cosine functions of different frequencies, as follows:

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i/d_{\text{model}}}}\right) & \text{if} \quad i \bmod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d_{\text{model}}}}\right) & \text{otherwise} \end{cases}$$

$PE_{(pos,i)}$ represents the position encoding at position pos in the sequence, and hidden dimensionality i. These values, concatenated for all hidden dimensions, are added to the original input features (in the Transformer visualization above, see "Positional encoding"), and constitute the position information. We distinguish between even ($i \bmod 2 = 0$) and uneven ($i \bmod 2 = 1$) hidden dimensionalities where we apply a sine/cosine respectively. The intuition behind this encoding is that you can represent $PE_{(pos+k,:)}$ as a linear function of $PE_{(pos,:)}$, which might allow the model to easily attend to relative positions. The wavelengths in different dimensions range from 2π to $10000 \cdot 2\pi$.

The positional encoding is implemented below. The code is taken from the PyTorch tutorial https://pytorch.org/tutorials/beginner/transformer_tutorial.html#define-the-model about Transformers on NLP and adjusted for our purposes.

```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model, max_len=5000):
        """
        Inputs
            d_model - Hidden dimensionality of the input.
            max_len - Maximum length of a sequence to expect.
        """
        super().__init__()

        # Create matrix of [SeqLen, HiddenDim] representing
        # the positional encoding for max_len inputs
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0)
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)

        # register_buffer => Tensor which is not a parameter,
        # but should be part of the modules state.
        # Used for tensors that need to be on the same device as the module.
        # persistent=False tells PyTorch to not add the buffer to the
        # state dict (e.g. when we save the model)
        self.register_buffer('pe', pe, persistent=False)

    def forward(self, x):
        x = x + self.pe[:, :x.size(1)]
        return x
```

To understand the positional encoding, we can visualize it below. We will generate an

image of the positional encoding over hidden dimensionality and position in a sequence. Each pixel, therefore, represents the change of the input feature we perform to encode the specific position. Let's do it below.

```
matplotlib.rcParams['lines.linewidth'] = 2.0
plt.set_cmap('cividis')
encod_block = PositionalEncoding(d_model=48, max_len=96)
pe = encod_block.pe.squeeze().T.cpu().numpy()

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8,3))
pos = ax.imshow(pe, cmap="RdGy", extent=(1,pe.shape[1]+1,pe.shape[0]+1,1))
fig.colorbar(pos, ax=ax)
ax.set_xlabel("Position in sequence")
ax.set_ylabel("Hidden dimension")
ax.set_title("Positional encoding over hidden dimensions")
ax.set_xticks([1]+[i*10 for i in range(1,1+pe.shape[1]//10)])
ax.set_yticks([1]+[i*10 for i in range(1,1+pe.shape[0]//10)])
plt.show()
```

```
<Figure size 1650x1050 with 0 Axes>
```

You can clearly see the sine and cosine waves with different wavelengths that encode the position in the hidden dimensions. Specifically, we can look at the sine/cosine wave for each hidden dimension separately, to get a better intuition of the pattern. Below we visualize the positional encoding for the hidden dimensions 1, 2, 3 and 4.

```
sns.set_theme()
fig, ax = plt.subplots(2, 2, figsize=(12,4))
ax = [a for a_list in ax for a in a_list]
for i in range(len(ax)):
    ax[i].plot(np.arange(1,17), pe[i,:16], color=f'C{i}', marker="o",
               markersize=6, markeredgecolor="black")
    ax[i].set_title(f"Encoding in hidden dimension {i+1}")
    ax[i].set_xlabel("Position in sequence", fontsize=10)
    ax[i].set_ylabel("Positional encoding", fontsize=10)
    ax[i].set_xticks(np.arange(1,17))
    ax[i].tick_params(axis='both', which='major', labelsize=10)
    ax[i].tick_params(axis='both', which='minor', labelsize=8)
    ax[i].set_ylim(-1.2, 1.2)
fig.subplots_adjust(hspace=0.8)
sns.reset_orig()
plt.show()
```

As we can see, the patterns between the hidden dimension 1 and 2 only differ in the starting angle. The wavelength is 2π, hence the repetition after position 6. The hidden dimensions 2 and 3 have about twice the wavelength.

54.3.9. Learning Rate Warm-up

One commonly used technique for training a Transformer is learning rate warm-up. This means that we gradually increase the learning rate from 0 on to our originally specified learning rate in the first few iterations. Thus, we slowly start learning instead of taking very large steps from the beginning. In fact, training a deep Transformer without learning rate warm-up can make the model diverge and achieve a much worse performance on training and testing. Take for instance the following plot by Liu et

al. (2019) comparing Adam-vanilla (i.e. Adam without warm-up) vs Adam with a warm-up:

Figure 54.6.: Warm-up comparison. Figure taken from Liu et al. (2019)

Clearly, the warm-up is a crucial hyperparameter in the Transformer architecture. Why is it so important? There are currently two common explanations. Firstly, Adam uses the bias correction factors which however can lead to a higher variance in the adaptive learning rate during the first iterations. Improved optimizers like RAdam have been shown to overcome this issue, not requiring warm-up for training Transformers. Secondly, the iteratively applied Layer Normalization across layers can lead to very high gradients during the first iterations, which can be solved by using Pre-Layer Normalization (similar to Pre-Activation ResNet), or replacing Layer Normalization by other techniques (Adaptive Normalization, Power Normalization).

Nevertheless, many applications and papers still use the original Transformer architecture with Adam, because warm-up is a simple, yet effective way of solving the gradient problem in the first iterations. There are many different schedulers we could use. For instance, the original Transformer paper used an exponential decay scheduler with a warm-up. However, the currently most popular scheduler is the cosine warm-up scheduler, which combines warm-up with a cosine-shaped learning rate decay. We can implement it below, and visualize the learning rate factor over epochs.

```
class CosineWarmupScheduler(optim.lr_scheduler._LRScheduler):

    def __init__(self, optimizer, warmup, max_iters):
        self.warmup = warmup
        self.max_num_iters = max_iters
        super().__init__(optimizer)

    def get_lr(self):
        lr_factor = self.get_lr_factor(epoch=self.last_epoch)
        return [base_lr * lr_factor for base_lr in self.base_lrs]

    def get_lr_factor(self, epoch):
```

```
        lr_factor = 0.5 * (1 + np.cos(np.pi * epoch / self.max_num_iters))
        if epoch <= self.warmup:
            lr_factor *= epoch * 1.0 / self.warmup
        return lr_factor
```

```
# Needed for initializing the lr scheduler
p = nn.Parameter(torch.empty(4,4))
optimizer = optim.Adam([p], lr=1e-3)
lr_scheduler = CosineWarmupScheduler(optimizer=optimizer, warmup=100, max_iters=2000)

# Plotting
epochs = list(range(2000))
sns.set()
plt.figure(figsize=(8,3))
plt.plot(epochs, [lr_scheduler.get_lr_factor(e) for e in epochs])
plt.ylabel("Learning rate factor")
plt.xlabel("Iterations (in batches)")
plt.title("Cosine Warm-up Learning Rate Scheduler")
plt.show()
sns.reset_orig()
```

In the first 100 iterations, we increase the learning rate factor from 0 to 1, whereas for all later iterations, we decay it using the cosine wave. Pre-implementations of this scheduler can be found in the popular NLP Transformer library huggingface.

54.3.10. PyTorch Lightning Module

Finally, we can embed the Transformer architecture into a PyTorch lightning module. PyTorch Lightning simplifies our training and test code, as well as structures the code nicely in separate functions. We will implement a template for a classifier based on the Transformer encoder. Thereby, we have a prediction output per sequence element. If we would need a classifier over the whole sequence, the common approach is to add an additional `[CLS]` token to the sequence (`CLS` stands for classification, i.e., the first token of every sequence is always a special classification token, `CLS`). However, here we focus on tasks where we have an output per element.

Additionally to the Transformer architecture, we add a small input network (maps input dimensions to model dimensions), the positional encoding, and an output network (transforms output encodings to predictions). We also add the learning rate scheduler, which takes a step each iteration instead of once per epoch. This is needed for the warmup and the smooth cosine decay. The training, validation, and test step is left empty for now and will be filled for our task-specific models.

```
class TransformerPredictor(pl.LightningModule):

    def __init__(self, input_dim, model_dim, num_classes, num_heads, num_layers, lr, warmup, max_ite
        """
        Inputs:
            input_dim - Hidden dimensionality of the input
            model_dim - Hidden dimensionality to use inside the Transformer
            num_classes - Number of classes to predict per sequence element
            num_heads - Number of heads to use in the Multi-Head Attention blocks
            num_layers - Number of encoder blocks to use.
            lr - Learning rate in the optimizer
            warmup - Number of warmup steps. Usually between 50 and 500
            max_iters - Number of maximum iterations the model is trained for. This is needed for th
            dropout - Dropout to apply inside the model
            input_dropout - Dropout to apply on the input features
        """
        super().__init__()
        self.save_hyperparameters()
        self._create_model()

    def _create_model(self):
        # Input dim -> Model dim
        self.input_net = nn.Sequential(
            nn.Dropout(self.hparams.input_dropout),
            nn.Linear(self.hparams.input_dim, self.hparams.model_dim)
        )
        # Positional encoding for sequences
```

```
        self.positional_encoding = PositionalEncoding(d_model=self.hparams.model_dim)
        # Transformer
        self.transformer = TransformerEncoder(num_layers=self.hparams.num_layers,
                                              input_dim=self.hparams.model_dim,
                                              dim_feedforward=2*self.hparams.model_dim
                                              num_heads=self.hparams.num_heads,
                                              dropout=self.hparams.dropout)
        # Output classifier per sequence lement
        self.output_net = nn.Sequential(
            nn.Linear(self.hparams.model_dim, self.hparams.model_dim),
            nn.LayerNorm(self.hparams.model_dim),
            nn.ReLU(inplace=True),
            nn.Dropout(self.hparams.dropout),
            nn.Linear(self.hparams.model_dim, self.hparams.num_classes)
        )

    def forward(self, x, mask=None, add_positional_encoding=True):
        """
        Inputs:
            x - Input features of shape [Batch, SeqLen, input_dim]
            mask - Mask to apply on the attention outputs (optional)
            add_positional_encoding - If True, we add the positional encoding to the i
                                      Might not be desired for some tasks.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        x = self.transformer(x, mask=mask)
        x = self.output_net(x)
        return x

    @torch.no_grad()
    def get_attention_maps(self, x, mask=None, add_positional_encoding=True):
        """
        Function for extracting the attention matrices of the whole Transformer for a
        Input arguments same as the forward pass.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        attention_maps = self.transformer.get_attention_maps(x, mask=mask)
        return attention_maps

    def configure_optimizers(self):
```

```
        optimizer = optim.Adam(self.parameters(), lr=self.hparams.lr)

        # Apply lr scheduler per step
        lr_scheduler = CosineWarmupScheduler(optimizer,
                                             warmup=self.hparams.warmup,
                                             max_iters=self.hparams.max_iters)
        return [optimizer], [{'scheduler': lr_scheduler, 'interval': 'step'}]

    def training_step(self, batch, batch_idx):
        raise NotImplementedError

    def validation_step(self, batch, batch_idx):
        raise NotImplementedError

    def test_step(self, batch, batch_idx):
        raise NotImplementedError
```

54.4. Experiment: Sequence to Sequence

After having finished the implementation of the Transformer architecture, we can start experimenting and apply it to various tasks. We will focus on parallel Sequence-to-Sequence.

A Sequence-to-Sequence task represents a task where the input *and* the output is a sequence, not necessarily of the same length. Popular tasks in this domain include machine translation and summarization. For this, we usually have a Transformer encoder for interpreting the input sequence, and a decoder for generating the output in an autoregressive manner. Here, however, we will go back to a much simpler example task and use only the encoder. Given a sequence of N numbers between 0 and M, the task is to reverse the input sequence. In Numpy notation, if our input is x, the output should be x[::-1]. Although this task sounds very simple, RNNs can have issues with such because the task requires long-term dependencies. Transformers are built to support such, and hence, we expect it to perform very well.

54.4.1. Dataset and Data Loaders

First, let's create a dataset class below.

```
class ReverseDataset(data.Dataset):

    def __init__(self, num_categories, seq_len, size):
        super().__init__()
```

```
        self.num_categories = num_categories
        self.seq_len = seq_len
        self.size = size

        self.data = torch.randint(self.num_categories, size=(self.size, self.seq_len))

    def __len__(self):
        return self.size

    def __getitem__(self, idx):
        inp_data = self.data[idx]
        labels = torch.flip(inp_data, dims=(0,))
        return inp_data, labels
```

We create an arbitrary number of random sequences of numbers between 0 and `num_categories-1`. The label is simply the tensor flipped over the sequence dimension. We can create the corresponding data loaders below.

```
dataset = partial(ReverseDataset, 10, 16)
train_loader = data.DataLoader(dataset(50000),
                               batch_size=128,
                               shuffle=True,
                               drop_last=True,
                               pin_memory=True)
val_loader   = data.DataLoader(dataset(1000), batch_size=128)
test_loader  = data.DataLoader(dataset(10000), batch_size=128)
```

```
inp_data, labels = train_loader.dataset[0]
print("Input data:", inp_data)
print("Labels:    ", labels)
```

```
Input data: tensor([0, 4, 1, 2, 5, 5, 7, 6, 9, 6, 3, 1, 9, 3, 1, 9])
Labels:     tensor([9, 1, 3, 9, 1, 3, 6, 9, 6, 7, 5, 5, 2, 1, 4, 0])
```

During training, we pass the input sequence through the Transformer encoder and predict the output for each input token. We use the standard Cross-Entropy loss to perform this. Every number is represented as a one-hot vector. Remember that representing the categories as single scalars decreases the expressiveness of the model extremely as 0 and 1 are not closer related than 0 and 9 in our example. An alternative to a one-hot vector is using a learned embedding vector as it is provided by the PyTorch module `nn.Embedding`. However, using a one-hot vector with an additional linear layer as in our case has the same effect as an embedding layer (`self.input_net` maps one-hot vector to a dense vector, where each row of the weight matrix represents the embedding for a specific category).

54.4.2. The Reverse Predictor Class

To implement the training dynamic, we create a new class inheriting from `TransformerPredictor` and overwriting the training, validation and test step functions, which were left empty in the base class. We also add a `_calculate_loss` function to calculate the loss and accuracy for a batch.

```
class ReversePredictor(TransformerPredictor):

    def _calculate_loss(self, batch, mode="train"):
        # Fetch data and transform categories to one-hot vectors
        inp_data, labels = batch
        inp_data = F.one_hot(inp_data, num_classes=self.hparams.num_classes).float()

        # Perform prediction and calculate loss and accuracy
        preds = self.forward(inp_data, add_positional_encoding=True)
        loss = F.cross_entropy(preds.view(-1,preds.size(-1)), labels.view(-1))
        acc = (preds.argmax(dim=-1) == labels).float().mean()

        # Logging
        self.log(f"{mode}_loss", loss)
        self.log(f"{mode}_acc", acc)
        return loss, acc

    def training_step(self, batch, batch_idx):
        loss, _ = self._calculate_loss(batch, mode="train")
        return loss

    def validation_step(self, batch, batch_idx):
        _ = self._calculate_loss(batch, mode="val")

    def test_step(self, batch, batch_idx):
        _ = self._calculate_loss(batch, mode="test")
```

Finally, we can create a training function. We create a `pl.Trainer` object, running for N epochs, logging in TensorBoard, and saving our best model based on the validation. Afterward, we test our models on the test set.

54.4.3. Gradient Clipping

An additional parameter we pass to the trainer here is `gradient_clip_val`. This clips the norm of the gradients for all parameters before taking an optimizer step and prevents the model from diverging if we obtain very high gradients at, for instance, sharp loss

surfaces (see many good blog posts on gradient clipping, like DeepAI glossary). For Transformers, gradient clipping can help to further stabilize the training during the first few iterations, and also afterward. In plain PyTorch, you can apply gradient clipping via `torch.nn.utils.clip_grad_norm_(...)` (see documentation). The clip value is usually between 0.5 and 10, depending on how harsh you want to clip large gradients.

54.4.4. Implementation of the Lightning Trainer

The Lightning trainer can be implemented as follows:

```
def train_reverse(**kwargs):
    # Create a PyTorch Lightning trainer with the generation callback
    root_dir = os.path.join(CHECKPOINT_PATH, "ReverseTask")
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir,
                         callbacks=[ModelCheckpoint(save_weights_only=True,
                                    mode="max", monitor="val_acc")],
                         accelerator="gpu" if str(device).startswith("cuda") else "cpu
                         devices=1,
                         max_epochs=10,
                         gradient_clip_val=5)
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "ReverseTask.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = ReversePredictor.load_from_checkpoint(pretrained_filename)
    else:
        model = ReversePredictor(max_iters=trainer.max_epochs*len(train_loader), **kwa
        trainer.fit(model, train_loader, val_loader)

    # Test best model on validation and test set
    val_result = trainer.test(model, val_loader, verbose=False)
    test_result = trainer.test(model, test_loader, verbose=False)
    result = {"test_acc": test_result[0]["test_acc"], "val_acc": val_result[0]["test_a

    model = model.to(device)
    return model, result
```

54.4.5. Training the Model

Finally, we can train the model. In this setup, we will use a single encoder block and a single head in the Multi-Head Attention. This is chosen because of the simplicity of the task, and in this case, the attention can actually be interpreted as an "explanation" of the predictions (compared to the other papers above dealing with deep Transformers).

```
reverse_model, reverse_result = train_reverse(input_dim=train_loader.dataset.num_categories,
                                              model_dim=32,
                                              num_heads=1,
                                              num_classes=train_loader.dataset.num_categories,
                                              num_layers=1,
                                              dropout=0.0,
                                              lr=5e-4,
                                              warmup=50)
```

```
Found pretrained model, loading...

Testing: |          | 0/? [00:00<?, ?it/s]

Testing: |          | 0/? [00:00<?, ?it/s]
```

The warning of PyTorch Lightning regarding the number of workers can be ignored for now. As the data set is so simple and the `__getitem__` finishes a neglectable time, we don't need subprocesses to provide us the data (in fact, more workers can slow down the training as we have communication overhead among processes/threads). First, let's print the results:

```
print(f"Val accuracy:  {(100.0 * reverse_result['val_acc']):4.2f}%")
print(f"Test accuracy: {(100.0 * reverse_result['test_acc']):4.2f}%")
```

```
Val accuracy:  100.00%
Test accuracy: 100.00%
```

As we would have expected, the Transformer can correctly solve the task.

54.5. Visualizing Attention Maps

How does the attention in the Multi-Head Attention block looks like for an arbitrary input? Let's try to visualize it below.

```
data_input, labels = next(iter(val_loader))
inp_data = F.one_hot(data_input, num_classes=reverse_model.hparams.num_classes).float
inp_data = inp_data.to(device)
attention_maps = reverse_model.get_attention_maps(inp_data)
```

The object `attention_maps` is a list of length N where N is the number of layers. Each element is a tensor of shape [Batch, Heads, SeqLen, SeqLen], which we can verify below.

```
attention_maps[0].shape
```

```
torch.Size([128, 1, 16, 16])
```

Next, we will write a plotting function that takes as input the sequences, attention maps, and an index indicating for which batch element we want to visualize the attention map. We will create a plot where over rows, we have different layers, while over columns, we show the different heads. Remember that the softmax has been applied for each row separately.

```
def plot_attention_maps(input_data, attn_maps, idx=0):
    if input_data is not None:
        input_data = input_data[idx].detach().cpu().numpy()
    else:
        input_data = np.arange(attn_maps[0][idx].shape[-1])
    attn_maps = [m[idx].detach().cpu().numpy() for m in attn_maps]

    num_heads = attn_maps[0].shape[0]
    num_layers = len(attn_maps)
    seq_len = input_data.shape[0]
    fig_size = 4 if num_heads == 1 else 3
    fig, ax = plt.subplots(num_layers, num_heads, figsize=(num_heads*fig_size, num_lay
    if num_layers == 1:
        ax = [ax]
    if num_heads == 1:
        ax = [[a] for a in ax]
    for row in range(num_layers):
        for column in range(num_heads):
            ax[row][column].imshow(attn_maps[row][column], origin='lower', vmin=0)
            ax[row][column].set_xticks(list(range(seq_len)))
            ax[row][column].set_xticklabels(input_data.tolist())
            ax[row][column].set_yticks(list(range(seq_len)))
            ax[row][column].set_yticklabels(input_data.tolist())
            ax[row][column].set_title(f"Layer {row+1}, Head {column+1}")
```

```
    fig.subplots_adjust(hspace=0.5)
    cax = fig.add_axes([0.95, 0.15, 0.01, 0.7])
    cbar = fig.colorbar(ax[0][0].imshow(attn_maps[0][0], origin='lower', vmin=0), cax=cax)
    cbar.set_label('Attention')
    plt.show()
```

Finally, we can plot the attention map of our trained Transformer on the reverse task:

```
plot_attention_maps(data_input, attention_maps, idx=0)
```

The model has learned to attend to the token that is on the flipped index of itself. Hence, it actually does what we intended it to do. We see that it however also pays some attention to values close to the flipped index. This is because the model doesn't need the perfect, hard attention to solve this problem, but is fine with this approximate, noisy attention map. The close-by indices are caused by the similarity of the positional encoding, which we also intended with the positional encoding.

54.6. Conclusion

In this chapter, we took a closer look at the Multi-Head Attention layer which uses a scaled dot product between queries and keys to find correlations and similarities between input elements. The Transformer architecture is based on the Multi-Head Attention layer and applies multiple of them in a ResNet-like block. The Transformer is a very important, recent architecture that can be applied to many tasks and datasets. Although it is best known for its success in NLP, there is so much more to it. We have seen its application on sequence-to-sequence tasks. Its property of being permutation-equivariant if we do not provide any positional encodings, allows it to generalize to many settings. Hence, it is important to know the architecture, but also its possible issues such as the gradient problem during the first iterations solved by learning rate warm-up. If you are interested in continuing with the study of the Transformer architecture, please have a look at the blog posts listed in the "Further Reading" section below.

54.7. Additional Considerations

54.7.1. Complexity and Path Length

We can compare the self-attention operation with our other common layer competitors for sequence data: convolutions and recurrent neural networks. In Figure 54.7 you can find a table by Vaswani et al. (2017) on the complexity per layer, the number of sequential operations, and maximum path length. The complexity is measured by the upper bound of the number of operations to perform, while the maximum path length represents the maximum number of steps a forward or backward signal has to traverse to reach any other position. The lower this length, the better gradient signals can backpropagate for long-range dependencies. Let's take a look at the table in Figure 54.7.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Figure 54.7.: Comparison of complexity and path length of different sequence layers. Table taken from Lippe (2022)

n is the sequence length, d is the representation dimension and k is the kernel size of convolutions. In contrast to recurrent networks, the self-attention layer can parallelize all its operations making it much faster to execute for smaller sequence lengths. However,

when the sequence length exceeds the hidden dimensionality, self-attention becomes more expensive than RNNs. One way of reducing the computational cost for long sequences is by restricting the self-attention to a neighborhood of inputs to attend over, denoted by r. Nevertheless, there has been recently a lot of work on more efficient Transformer architectures that still allow long dependencies, of which you can find an overview in the paper by Tay et al. (2020) if interested.

54.8. Further Reading

There are of course many more tutorials out there about attention and Transformers. Below, we list a few that are worth exploring if you are interested in the topic and might want yet another perspective on the topic after this one:

- Transformer: A Novel Neural Network Architecture for Language Understanding (Jakob Uszkoreit, 2017) - The original Google blog post about the Transformer paper, focusing on the application in machine translation.
- The Illustrated Transformer (Jay Alammar, 2018) - A very popular and great blog post intuitively explaining the Transformer architecture with many nice visualizations. The focus is on NLP.
- Attention? Attention! (Lilian Weng, 2018) - A nice blog post summarizing attention mechanisms in many domains including vision.
- Illustrated: Self-Attention (Raimi Karim, 2019) - A nice visualization of the steps of self-attention. Recommended going through if the explanation below is too abstract for you.
- The Transformer family (Lilian Weng, 2020) - A very detailed blog post reviewing more variants of Transformers besides the original one.

55. Hyperparameter Tuning of a Transformer Network with PyTorch Lightning

55.1. Basic Setup

This section provides an overview of the hyperparameter tuning process using `spotpython` and `PyTorch` Lightning. It uses the `Diabetes` data set (see Section E.1) for a regression task.

In this section, we will show how `spotpython` can be integrated into the `PyTorch` Lightning training workflow for a regression task. It demonstrates how easy it is to use `spotpython` to tune hyperparameters for a `PyTorch` Lightning model.

After importing the necessary libraries, the `fun_control` dictionary is set up via the `fun_control_init` function. The `fun_control` dictionary contains

- `PREFIX`: a unique identifier for the experiment
- `fun_evals`: the number of function evaluations
- `max_time`: the maximum run time in minutes
- `data_set`: the data set. Here we use the `Diabetes` data set that is provided by `spotpython`.
- `core_model_name`: the class name of the neural network model. This neural network model is provided by `spotpython`.
- `hyperdict`: the hyperparameter dictionary. This dictionary is used to define the hyperparameters of the neural network model. It is also provided by `spotpython`.
- `_L_in`: the number of input features. Since the `Diabetes` data set has 10 features, `_L_in` is set to 10.
- `_L_out`: the number of output features. Since we want to predict a single value, `_L_out` is set to 1.

The method `set_hyperparameter` allows the user to modify default hyperparameter settings. Here we set the `initialization` method to `["Default"]`. No other initializations are used in this experiment. The `HyperLight` class is used to define the objective function `fun`. It connects the `PyTorch` and the `spotpython` methods and is provided by `spotpython`. Finally, a `Spot` object is created.

Note, the `divergence_threshold` is set to 25,000, which is based on some pre-experiments with the `Diabetes` data set.

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_co
from spotpython.utils.eda import print_exp_table
from spotpython.hyperparameters.values import set_hyperparameter
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename
from spotpython.utils.scaler import TorchStandardScaler

fun_control = fun_control_init(
    PREFIX="603",
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    fun_evals=inf,
    max_time=1,
    data_set = Diabetes(),
    scaler=TorchStandardScaler(),
    core_model_name="light.regression.NNTransformerRegressor",
    hyperdict=LightHyperDict,
    divergence_threshold=25_000,
    _L_in=10,
    _L_out=1)

set_hyperparameter(fun_control, "optimizer", [
                "Adadelta",
                "Adagrad",
                "Adam",
                "AdamW",
                "Adamax",
            ])
set_hyperparameter(fun_control, "epochs", [5, 7])
set_hyperparameter(fun_control, "nhead", [1, 2])
set_hyperparameter(fun_control, "dim_feedforward_mult", [1, 1])

design_control = design_control_init(init_size=5)
surrogate_control = surrogate_control_init(
    method="regression",
    min_Lambda=1e-3,
    max_Lambda=10,
)
```

```
fun = HyperLight().fun

spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control, surrogate_control=
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_11_04_21_25_04_0
Created spot_tensorboard_path: runs/spot_logs/603_maans08_2025-11-
04_21-25-04 for SummaryWriter()
module_name: light
submodule_name: regression
model_name: NNTransformerRegressor
```

We can take a look at the design table to see the initial design.

```
print_exp_table(fun_control)
```

```
| name                  | type   | default        |   lower |   upper | transform             |
|-----------------------|--------|----------------|---------|---------
|-----------------------|
| d_model_mult          | int    | 4              |    1    |     5   | transform_power_2_int |
| nhead                 | int    | 3              |    1    |     2   | transform_power_2_int |
| num_encoder_layers    | int    | 1              |    1    |     4   | transform_power_2_int |
| dim_feedforward_mult  | int    | 1              |    1    |     1   | transform_power_2_int |
| epochs                | int    | 7              |    5    |     7   | transform_power_2_int |
| batch_size            | int    | 5              |    5    |     8   | transform_power_2_int |
| optimizer             | factor | Adam           |    0    |     4   | None                  |
| dropout               | float  | 0.1            |    0.01 |     0.1 | None                  |
| lr_mult               | float  | 0.1            |    0.01 |     0.3 | None                  |
| patience              | int    | 5              |    4    |     7   | transform_power_2_int |
| initialization        | factor | xavier_uniform |    0    |     3   | None                  |
```

Calling the method `run()` starts the hyperparameter tuning process on the local machine.

```
res = spot_tuner.run()
```

```
d_model: 8, dim_feedforward: 16
```

```
     Name                  Type                  Params   Mode     FLOPs        In sizes       Out size

  0  input_proj            Linear                    88   train   20.5 K       [128, 10]        [128, 8]
```

```
  1   positional_encoding   PositionalEncoding        0   train        0        [128, 8]
  2   transformer_encoder   TransformerEncoder    4.8 K   train    134 M   [128, 128, 8]
  3   fc_out                Linear                    9   train    2.0 K        [128, 8]



Trainable params: 4.9 K
Non-trainable params: 0
Total params: 4.9 K
Total estimated model params size (MB): 0
Modules in train mode: 85
Modules in eval mode: 0
Total FLOPs: 134 M

train_model result: {'val_loss': 23955.66015625, 'hp_metric': 23955.66015625}
d_model: 128, dim_feedforward: 256



     Name                  Type                 Params   Mode     FLOPs        In sizes

  0   input_proj            Linear                1.4 K   train   81.9 K        [32, 10]
  1   positional_encoding   PositionalEncoding        0   train        0       [32, 128]
  2   transformer_encoder   TransformerEncoder    529 K   train    1.1 B   [32, 32, 128]
  3   fc_out                Linear                  129   train    8.2 K       [32, 128]



Trainable params: 531 K
Non-trainable params: 0
Total params: 531 K
Total estimated model params size (MB): 2
Modules in train mode: 45
Modules in eval mode: 0
Total FLOPs: 1.1 B

train_model result: {'val_loss': 20858.498046875, 'hp_metric': 20858.498046875}
d_model: 32, dim_feedforward: 64



     Name                  Type                 Params   Mode     FLOPs         In sizes

  0   input_proj            Linear                  352   train   81.9 K        [128, 10]
```

```
  1   positional_encoding  PositionalEncoding       0   train       0       [128, 32]   [128, 128, 3
  2   transformer_encoder  TransformerEncoder   68.4 K  train   2.1 B  [128, 128, 32]   [128, 128, 3
  3   fc_out               Linear                  33   train   8.2 K       [128, 32]         [128,


Trainable params: 68.7 K
Non-trainable params: 0
Total params: 68.7 K
Total estimated model params size (MB): 0
Modules in train mode: 85
Modules in eval mode: 0
Total FLOPs: 2.1 B

train_model result: {'val_loss': 23231.283203125, 'hp_metric': 23231.283203125}
d_model: 16, dim_feedforward: 32


     Name                 Type                Params  Mode    FLOPs      In sizes     Out sizes

  0  input_proj           Linear                 176  train  20.5 K      [64, 10]      [64, 16]
  1  positional_encoding  PositionalEncoding       0  train       0      [64, 16]  [64, 64, 16]
  2  transformer_encoder  TransformerEncoder  35.6 K  train   268 M  [64, 64, 16]  [64, 64, 16]
  3  fc_out               Linear                  17  train   2.0 K      [64, 16]       [64, 1]


Trainable params: 35.8 K
Non-trainable params: 0
Total params: 35.8 K
Total estimated model params size (MB): 0
Modules in train mode: 165
Modules in eval mode: 0
Total FLOPs: 268 M

train_model result: {'val_loss': 23903.546875, 'hp_metric': 23903.546875}
d_model: 8, dim_feedforward: 16


     Name                 Type                Params  Mode    FLOPs       In sizes      Out size

  0  input_proj           Linear                  88  train  41.0 K      [256, 10]       [256, 8]
```

```
  1   positional_encoding   PositionalEncoding        0   train        0        [256, 8]
  2   transformer_encoder   TransformerEncoder    1.2 K   train    134 M   [256, 256, 8]
  3   fc_out                Linear                    9   train    4.1 K        [256, 8]



Trainable params: 1.3 K
Non-trainable params: 0
Total params: 1.3 K
Total estimated model params size (MB): 0
Modules in train mode: 25
Modules in eval mode: 0
Total FLOPs: 134 M


train_model result: {'val_loss': 23954.796875, 'hp_metric': 23954.796875}
d_model: 128, dim_feedforward: 256



     Name                  Type                 Params   Mode     FLOPs        In sizes

  0   input_proj            Linear                1.4 K   train   81.9 K        [32, 10]
  1   positional_encoding   PositionalEncoding        0   train        0       [32, 128]
  2   transformer_encoder   TransformerEncoder    529 K   train    1.1 B   [32, 32, 128]
  3   fc_out                Linear                  129   train    8.2 K       [32, 128]



Trainable params: 531 K
Non-trainable params: 0
Total params: 531 K
Total estimated model params size (MB): 2
Modules in train mode: 45
Modules in eval mode: 0
Total FLOPs: 1.1 B


train_model result: {'val_loss': 20560.65234375, 'hp_metric': 20560.65234375}
spotpython tuning: 20560.65234375 [##--------] 18.99%. Success rate: 100.00%


d_model: 128, dim_feedforward: 256
```

```
    Name                  Type                 Params   Mode     FLOPs        In sizes       Out sizes

 0  input_proj            Linear                1.4 K   train   81.9 K        [32, 10]       [32, 128]
 1  positional_encoding   PositionalEncoding        0   train        0       [32, 128]   [32, 32, 128]
 2  transformer_encoder   TransformerEncoder    529 K   train    1.1 B   [32, 32, 128]   [32, 32, 128]
 3  fc_out                Linear                  129   train    8.2 K       [32, 128]         [32, 1]


Trainable params: 531 K
Non-trainable params: 0
Total params: 531 K
Total estimated model params size (MB): 2
Modules in train mode: 45
Modules in eval mode: 0
Total FLOPs: 1.1 B

train_model result: {'val_loss': 20688.98046875, 'hp_metric': 20688.98046875}
spotpython tuning: 20560.65234375 [####------] 38.13%. Success rate: 50.00%

d_model: 128, dim_feedforward: 256


    Name                  Type                 Params   Mode     FLOPs        In sizes       Out sizes

 0  input_proj            Linear                1.4 K   train   81.9 K        [32, 10]       [32, 128]
 1  positional_encoding   PositionalEncoding        0   train        0       [32, 128]   [32, 32, 128]
 2  transformer_encoder   TransformerEncoder    529 K   train    1.1 B   [32, 32, 128]   [32, 32, 128]
 3  fc_out                Linear                  129   train    8.2 K       [32, 128]         [32, 1]


Trainable params: 531 K
Non-trainable params: 0
Total params: 531 K
Total estimated model params size (MB): 2
Modules in train mode: 45
Modules in eval mode: 0
Total FLOPs: 1.1 B

train_model result: {'val_loss': 20638.8046875, 'hp_metric': 20638.8046875}
spotpython tuning: 20560.65234375 [######----] 57.37%. Success rate: 33.33%
```

```
d_model: 128, dim_feedforward: 256


     Name                  Type                 Params   Mode     FLOPs        In sizes

  0  input_proj            Linear                1.4 K   train   81.9 K        [32, 10]
  1  positional_encoding   PositionalEncoding        0   train        0       [32, 128]
  2  transformer_encoder   TransformerEncoder    529 K   train    1.1 B   [32, 32, 128]
  3  fc_out                Linear                  129   train    8.2 K       [32, 128]



Trainable params: 531 K
Non-trainable params: 0
Total params: 531 K
Total estimated model params size (MB): 2
Modules in train mode: 45
Modules in eval mode: 0
Total FLOPs: 1.1 B

train_model result: {'val_loss': 20515.283203125, 'hp_metric': 20515.283203125}
spotpython tuning: 20515.283203125 [########--] 77.23%. Success rate: 50.00%

d_model: 128, dim_feedforward: 256


     Name                  Type                 Params   Mode     FLOPs        In sizes

  0  input_proj            Linear                1.4 K   train   81.9 K        [32, 10]
  1  positional_encoding   PositionalEncoding        0   train        0       [32, 128]
  2  transformer_encoder   TransformerEncoder    264 K   train    536 M   [32, 32, 128]
  3  fc_out                Linear                  129   train    8.2 K       [32, 128]



Trainable params: 266 K
Non-trainable params: 0
Total params: 266 K
Total estimated model params size (MB): 1
Modules in train mode: 25
Modules in eval mode: 0
Total FLOPs: 536 M
```

```
train_model result: {'val_loss': 21880.328125, 'hp_metric': 21880.328125}
spotpython tuning: 20515.283203125 [#########-] 89.45%. Success rate: 40.00%

d_model: 128, dim_feedforward: 256



     Name                  Type                 Params   Mode     FLOPs        In sizes       Out size

  0  input_proj            Linear                1.4 K   train   81.9 K        [32, 10]       [32, 128]
  1  positional_encoding   PositionalEncoding        0   train        0       [32, 128]   [32, 32, 128]
  2  transformer_encoder   TransformerEncoder    529 K   train    1.1 B   [32, 32, 128]   [32, 32, 128]
  3  fc_out                Linear                  129   train    8.2 K       [32, 128]         [32, 1]



Trainable params: 531 K
Non-trainable params: 0
Total params: 531 K
Total estimated model params size (MB): 2
Modules in train mode: 45
Modules in eval mode: 0
Total FLOPs: 1.1 B

train_model result: {'val_loss': 20147.291015625, 'hp_metric': 20147.291015625}
spotpython tuning: 20147.291015625 [##########] 100.00%. Success rate: 50.00% Done...

Experiment saved to 603_res.pkl
```

Note that we have enabled Tensorboard-Logging, so we can visualize the results with Tensorboard. Execute the following command in the terminal to start Tensorboard.

```
tensorboard --logdir="runs/"
```

55.2. Looking at the Results

55.2.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress(log_y=True, filename=None)
```

55.2.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

```
| name                  | type    | default         |    lower |    upper | tuned
|----------------------|--------|----------------|---------|---------
|---------------------|----------------------|--------------|---------
|
| d_model_mult          | int     | 4               |      1.0 |      5.0 | 5.0
| nhead                 | int     | 3               |      1.0 |      2.0 | 2.0
| num_encoder_layers    | int     | 1               |      1.0 |      4.0 | 2.0
| dim_feedforward_mult  | int     | 1               |      1.0 |      1.0 | 1.0
| epochs                | int     | 7               |      5.0 |      7.0 | 7.0
| batch_size            | int     | 5               |      5.0 |      8.0 | 5.0
| optimizer             | factor  | Adam            |      0.0 |      4.0 | Adagrad
| dropout               | float   | 0.1             |     0.01 |      0.1 | 0.010009364943
| lr_mult               | float   | 0.1             |     0.01 |      0.3 | 0.233735795264
| patience              | int     | 5               |      4.0 |      7.0 | 4.0
| initialization        | factor  | xavier_uniform  |      0.0 |      3.0 | xavier_normal
```

55.3. Hyperparameter Considerations

1. `d_model` (or `d_embedding`):

- This is the dimension of the embedding space or the number of expected features in the input.
- All input features are projected into this dimensional space before entering the transformer encoder.
- This dimension must be divisible by `nhead` since each head in the multi-head attention mechanism will process a subset of `d_model/nhead` features.

2. `nhead`:

   - This is the number of attention heads in the multi-head attention mechanism.
   - It allows the transformer to jointly attend to information from different representation subspaces.
   - It's important that `d_model % nhead == 0` to ensure the dimensions are evenly split among the heads.

3. `num_encoder_layers`:

   - This specifies the number of transformer encoder layers stacked together.
   - Each layer contains a multi-head attention mechanism followed by position-wise feedforward layers.

4. `dim_feedforward`:

   - This is the dimension of the feedforward network model within the transformer encoder layer.
   - Typically, this dimension is larger than `d_model` (e.g., 2048 for a Transformer model with `d_model=512`).

55.3.1. Important: Constraints and Interconnections:

- `d_model` and `nhead`:
  - As mentioned, `d_model` must be divisible by `nhead`. This is critical because each attention head operates simultaneously on a part of the embedding, so `d_model/nhead` should be an integer.
- `num_encoder_layers` and `dim_feedforward`**:
  - These parameters are more flexible and can be chosen independently of `d_model` and `nhead`.
  - However, the choice of `dim_feedforward` does influence the computational cost and model capacity, as larger dimensions allow learning more complex representations.
- One hyperparameter does not strictly need to be a multiple of others except for ensuring `d_model % nhead == 0`.

55.3.2. Practical Considerations:

1. Setting `d_model`:

   - Common choices for `d_model` are powers of 2 (e.g., 256, 512, 1024).
   - Ensure that it matches the size of the input data after the linear projection layer.

2. Setting `nhead`:

   - Typically, values are 1, 2, 4, 8, etc., depending on the `d_model` value.
   - Each head works on a subset of features, so `d_model / nhead` should be large enough to be meaningful.

3. Setting `num_encoder_layers`:

   - Practical values range from 1 to 12 or more depending on the depth desired.
   - Deeper models can capture more complex patterns but are also more computationally intensive.

4. Setting `dim_feedforward`:

   - Often set to a multiple of `d_model`, such as 2048 when `d_model` is 512.
   - Ensures sufficient capacity in the intermediate layers for complex feature transformations.

> **Note:** `d_model` Calculation
>
> Since `d_model % nhead == 0` is a critical constraint to ensure that the multi-head attention mechanism can operate effectively, `spotpython` computes the value of `d_model` based on the `nhead` value provided by the user. This ensures that the hyperparameter configuration is valid. So, the final value of `d_model` is a multiple of `nhead`. `spotpython` uses the hyperparameter `d_model_mult` to determine the multiple of `nhead` to use for `d_model`, i.e., `d_model = nhead * d_model_mult`.

> **Note:** `dim_feedforward` Calculation
>
> Since this dimension is typically larger than `d_model` (e.g., 2048 for a Transformer model with `d_model=512`), `spotpython` uses the hyperparameter `dim_feedforward_mult` to determine the multiple of `d_model` to use for `dim_feedforward`, i.e., `dim_feedforward = d_model * dim_feedforward_mult`.

55.4. Summary

This section presented an introduction to the basic setup of hyperparameter tuning of a transformer with `spotpython` and `PyTorch` Lightning.

56. Saving and Loading

This tutorial shows how to save and load objects in `spotpython`. It is split into the following parts:

- Section 56.3 shows how to save and load objects in `spotpython`, if `spotpython` is used as an optimizer.
- Section 56.4 shows how to save and load hyperparameter tuning experiments.
- Section 56.5 shows how to save and load `PyTorch Lightning` models.
- Section 56.6 shows how to convert a `PyTorch Lightning` model to a plain `PyTorch` model.

56.1. Required Python Packages

```
import os
import pprint
import numpy as np
from math import inf
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from torch.utils.data import random_split
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

from spotpython.utils.file import (load_experiment, load_result, get_experiment_filename, load_and_r
from spotpython.spot import Spot
from spotpython.utils.init import (
    fun_control_init,
    design_control_init,
    surrogate_control_init,
    optimizer_control_init)
from spotpython.fun.objectivefunctions import Analytical
```

```
from spotpython.hyperparameters.values import get_tuned_hyperparameters
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.eda import print_exp_table
from spotpython.hyperparameters.values import set_hyperparameter, get_tuned_architectu
from spotpython.light.testmodel import test_model
from spotpython.light.loadmodel import load_light_from_checkpoint
from spotpython.utils.device import getDevice
from spotpython.utils.classes import get_removed_attributes_and_base_net
from spotpython.hyperparameters.optimizer import optimizer_handler
```

And we will suppress warnings in this notebook.

56.2. Quick Overview of `spotpython` Save and Load Functions

Because the most common use case is to save and load hyperparameter tuning experiments, the `save_experiment` and `load_experiment` functions are explained first. In the real- world setting, users specify an experiment on their local machine and then run the experiment on a remote machine. Then, the result file is copied back to the local machine and loaded for further analysis. So, the main steps can be summarized as follows:

1. Generate an experiment configuration on the local machine, say `42_exp.pkl`. To generate the configuration, the `fun_control` dictionary has to be initialized with `save_experiment=True`.
2. Copy the configuration file to the remote machine.
3. Run the experiment on the remote machine. The run on the remote machine is started with the command `load_and_run_spot_python_experiment(filename="42_exp.pkl")`. This generates a result file, say `42_res.pkl`.
4. Copy the result file back to the local machine. Analyze the results on the local machine as shown below.

As can be seen from these step, we will distinguish between experiment/design/configuration files (with `_exp` in the filename) and result files (with `_res` in the filename).

Example 56.1 (Save and Load Hyperparameter Tuning Experiment)**.**

1. Generate an experiment configuration on the local machine, say `42_exp.pkl` as follows:

```
PREFIX = "42"
data_set = Diabetes()
fun_control = fun_control_init(
    save_experiment=True,
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)

fun = HyperLight().fun

set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,5])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])

design_control = design_control_init(init_size=10)

print_exp_table(fun_control)

S = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
```

```
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
| name           | type   | default   |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------
|
| l1             | int    | 3         |     3   |   4     | transform_power_2_int |
| epochs         | int    | 4         |     3   |   5     | transform_power_2_int |
| batch_size     | int    | 4         |     4   |  11     | transform_power_2_int |
| act_fn         | factor | ReLU      |     0   |   5     | None                  |
| optimizer      | factor | SGD       |     0   |   2     | None                  |
| dropout_prob   | float  | 0.01      |     0   |   0.025 | None                  |
| lr_mult        | float  | 1.0       |     0.1 |  10     | None                  |
| patience       | int    | 2         |     2   |   3     | transform_power_2_int |
| batch_norm     | factor | 0         |     0   |   1     | None                  |
| initialization | factor | Default   |     0   |   4     | None                  |
```

```
Experiment saved to 42_exp.pkl
```

As the output shows, the configuration is saved as a pickle-file that contains the full information. In our example, the filename is `42_exp.pkl`.

2. Copy the configuration file to the remote machine. This can be done with the `scp` command, see below, but is omitted here for brevity.
3. Run the experiment on the remote machine. This step is simulated on the local machine for demonstration purposes. This generates a result file, say `42_res.pkl`. This can be done with the following code:

```
# S_res = S.run() is NOT used here, because we want to simulate the remote run
# Instead, we use the load_and_run_spot_python_experiment function:
S_remote = load_and_run_spot_python_experiment(filename="42_exp.pkl")
```

```
Loaded experiment from 42_exp.pkl
```

```
     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   202 K   train   826 M   [2048, 10]   [2048, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 23075.09765625, 'hp_metric': 23075.09765625}


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  51.9 K   train   6.6 M   [64, 10]     [64, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 3850.81201171875, 'hp_metric': 3850.81201171875}



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 6251.13916015625, 'hp_metric': 6251.13916015625}



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M
```

```
train_model result: {'val_loss': 24093.25390625, 'hp_metric': 24093.25390625}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 23622.310546875, 'hp_metric': 23622.310546875}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3700.771728515625, 'hp_metric': 3700.771728515625}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   52.5 M   [512, 10]    [512, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 20202.96875, 'hp_metric': 20202.96875}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 14795.3359375, 'hp_metric': 14795.3359375}



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   13.1 M   [128, 10]    [128, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 13.1 M
```

```
train_model result: {'val_loss': 21675.736328125, 'hp_metric': 21675.736328125}


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   413 M   [1024, 10]   [1024, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 413 M

train_model result: {'val_loss': 23792.47265625, 'hp_metric': 23792.47265625}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 23473.31640625, 'hp_metric': 23473.31640625}
spotpython tuning: 3700.771728515625 [----------] 3.19%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   826 M   [2048, 10]   [2048, 1]
```

```
Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 21453.419921875, 'hp_metric': 21453.419921875}
spotpython tuning: 3700.771728515625 [#---------] 5.55%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   25.8 M   [64, 10]     [64, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 16605.611328125, 'hp_metric': 16605.611328125}
spotpython tuning: 3700.771728515625 [#---------] 6.68%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   206 M   [512, 10]    [512, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M
```

```
train_model result: {'val_loss': 4418.7109375, 'hp_metric': 4418.7109375}
spotpython tuning: 3700.771728515625 [#---------] 8.15%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   826 M   [2048, 10]   [2048, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 19324.853515625, 'hp_metric': 19324.853515625}
spotpython tuning: 3700.771728515625 [#---------] 9.29%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   826 M   [2048, 10]   [2048, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 23236.640625, 'hp_metric': 23236.640625}
spotpython tuning: 3700.771728515625 [#---------] 10.61%. Success rate: 0.00%
```

```
    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   1.6 M   [16, 10]     [16, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 1.6 M

train_model result: {'val_loss': 23883.439453125, 'hp_metric': 23883.439453125}
spotpython tuning: 3700.771728515625 [#---------] 13.20%. Success rate: 0.00%



    Name     Type        Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   206 M   [512, 10]    [512, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M

train_model result: {'val_loss': 12952.9296875, 'hp_metric': 12952.9296875}
spotpython tuning: 3700.771728515625 [##--------] 15.72%. Success rate: 0.00%



    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   6.6 M   [64, 10]     [64, 1]
```

```
Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 23691.73046875, 'hp_metric': 23691.73046875}
spotpython tuning: 3700.771728515625 [##--------] 16.88%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   206 M   [512, 10]    [512, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M

train_model result: {'val_loss': 5191.41650390625, 'hp_metric': 5191.41650390625}
spotpython tuning: 3700.771728515625 [##--------] 19.57%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M
```

```
train_model result: {'val_loss': 15954.0634765625, 'hp_metric': 15954.0634765625}
spotpython tuning: 3700.771728515625 [##--------] 21.23%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 23493.109375, 'hp_metric': 23493.109375}
spotpython tuning: 3700.771728515625 [##--------] 23.41%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   6.6 M   [64, 10]     [64, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 23837.33984375, 'hp_metric': 23837.33984375}
spotpython tuning: 3700.771728515625 [##--------] 24.77%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   104 M   [1024, 10]   [1024, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': 17594.4921875, 'hp_metric': 17594.4921875}
spotpython tuning: 3700.771728515625 [###-------] 25.88%. Success rate: 0.00%


    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   6.6 M   [64, 10]     [64, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   13.1 M   [128, 10]    [128, 1]
```

```
Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.1 M

train_model result: {'val_loss': 23908.48828125, 'hp_metric': 23908.48828125}
spotpython tuning: 3700.771728515625 [###-------] 27.21%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   206 M   [512, 10]    [512, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M

train_model result: {'val_loss': 24016.99609375, 'hp_metric': 24016.99609375}
spotpython tuning: 3700.771728515625 [###-------] 28.29%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   826 M   [2048, 10]   [2048, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 826 M
```

```
train_model result: {'val_loss': 24098.27734375, 'hp_metric': 24098.27734375}
spotpython tuning: 3700.771728515625 [###-------] 29.25%. Success rate: 0.00%

Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   25.8 M   [64, 10]     [64, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 23996.900390625, 'hp_metric': 23996.900390625}
spotpython tuning: 3700.771728515625 [###-------] 30.59%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 10241.0615234375, 'hp_metric': 10241.0615234375}
spotpython tuning: 3700.771728515625 [###-------] 33.16%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 19619.40625, 'hp_metric': 19619.40625}
spotpython tuning: 3700.771728515625 [###-------] 34.51%. Success rate: 0.00%



    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 20175.65234375, 'hp_metric': 20175.65234375}
spotpython tuning: 3700.771728515625 [####------] 36.08%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 12377.2626953125, 'hp_metric': 12377.2626953125}
spotpython tuning: 3700.771728515625 [####------] 39.75%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 22732.955078125, 'hp_metric': 22732.955078125}
spotpython tuning: 3700.771728515625 [####------] 41.39%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M
```

```
train_model result: {'val_loss': 23719.169921875, 'hp_metric': 23719.169921875}
spotpython tuning: 3700.771728515625 [#####-----] 45.81%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   25.8 M   [64, 10]     [64, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 17149.6484375, 'hp_metric': 17149.6484375}
spotpython tuning: 3700.771728515625 [#####-----] 47.14%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   25.8 M   [64, 10]     [64, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 16395.931640625, 'hp_metric': 16395.931640625}
spotpython tuning: 3700.771728515625 [#####-----] 48.36%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   206 M   [512, 10]    [512, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M

train_model result: {'val_loss': 23803.443359375, 'hp_metric': 23803.443359375}
spotpython tuning: 3700.771728515625 [#####-----] 49.94%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 23311.802734375, 'hp_metric': 23311.802734375}
spotpython tuning: 3700.771728515625 [#####-----] 53.11%. Success rate: 0.00%



    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    205 K   train   25.8 M   [64, 10]     [64, 1]
```

```
Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 23695.421875, 'hp_metric': 23695.421875}
spotpython tuning: 3700.771728515625 [######----] 55.12%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   104 M   [1024, 10]   [1024, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': 23479.724609375, 'hp_metric': 23479.724609375}
spotpython tuning: 3700.771728515625 [######----] 56.94%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M
```

```
train_model result: {'val_loss': 19671.98046875, 'hp_metric': 19671.98046875}
spotpython tuning: 3700.771728515625 [######----] 58.32%. Success rate: 0.00%



     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0   layers   Sequential   51.9 K   train   6.6 M   [64, 10]     [64, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



     Name     Type        Params   Mode    FLOPs    In sizes   Out sizes

 0   layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 23461.787109375, 'hp_metric': 23461.787109375}
spotpython tuning: 3700.771728515625 [######----] 60.06%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]



Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 16165.0400390625, 'hp_metric': 16165.0400390625}
spotpython tuning: 3700.771728515625 [######----] 61.94%. Success rate: 0.00%



    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]



Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 8532.5380859375, 'hp_metric': 8532.5380859375}
spotpython tuning: 3700.771728515625 [######----] 63.31%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [######----] 64.43%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 65.43%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 66.44%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 67.44%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 68.44%. Success rate: 0.00%



     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0   layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

 0   layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 69.55%. Success rate: 0.00%
```

```
    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



    Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 70.55%. Success rate: 0.00%



    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 71.62%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 72.69%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 73.69%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 74.77%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 75.78%. Success rate: 0.00%
```

```
     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0   layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

 0   layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 76.79%. Success rate: 0.00%
```

```
    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



    Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 77.79%. Success rate: 0.00%



    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 78.82%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 79.82%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 80.81%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 81.80%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 82.86%. Success rate: 0.00%
```

	Name	Type	Params	Mode	FLOPs	In sizes	Out sizes
0	layers	Sequential	51.9 K	train	3.3 M	[32, 10]	[32, 1]

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

	Name	Type	Params	Mode	FLOPs	In sizes	Out sizes
0	layers	Sequential	202 K	train	51.6 M	[128, 10]	[128, 1]

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 83.91%. Success rate: 0.00%
```

```
    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



    Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 84.89%. Success rate: 0.00%



    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########-] 85.89%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#########-] 86.97%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########-] 87.99%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#########-] 89.00%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########-] 90.09%. Success rate: 0.00%
```

```
     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0   layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

 0   layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########-] 91.10%. Success rate: 0.00%
```

```
    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



    Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#########-] 92.12%. Success rate: 0.00%



    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########-] 93.12%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#########-] 94.20%. Success rate: 0.00%



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [###########] 95.27%. Success rate: 0.00%



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########] 96.27%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [###########] 97.26%. Success rate: 0.00%
```

```
     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0   layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

 0   layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [###########] 98.32%. Success rate: 0.00%
```

```
    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]



Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



    Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########] 99.31%. Success rate: 0.00%



    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]
```

```
Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]



Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########] 100.00%. Success rate: 0.00% Done...

Experiment saved to 42_res.pkl
```

A result file with the name `42_res.pkl` is stored in the current directory.

4. Copy the result file back to the local machine. This can be done with the `scp` command, see below, but is omitted here for brevity. The result file can be loaded with the following code:

```
S_res = load_result(PREFIX="42")
```

```
Loaded experiment from 42_res.pkl
```

```
S_res.plot_progress(log_y=True)
```

```
S_res.print_results()
```

```
min y: 3700.771728515625
l1: 4.0
epochs: 4.0
batch_size: 5.0
act_fn: 3.0
optimizer: 2.0
dropout_prob: 0.023931753071792624
lr_mult: 1.2532486761645163
patience: 3.0
batch_norm: 0.0
initialization: 4.0
```

```
[['l1', np.float64(4.0)],
 ['epochs', np.float64(4.0)],
 ['batch_size', np.float64(5.0)],
 ['act_fn', np.float64(3.0)],
 ['optimizer', np.float64(2.0)],
 ['dropout_prob', np.float64(0.023931753071792624)],
 ['lr_mult', np.float64(1.2532486761645163)],
 ['patience', np.float64(3.0)],
 ['batch_norm', np.float64(0.0)],
 ['initialization', np.float64(4.0)]]
```

If you add `fun_control=S_res.fun_control` as an argument to the `get_tuned_hyperparameters` function, the names of the hyperparameters are used as keys in the dictionary.

```
get_tuned_hyperparameters(S_res, fun_control=S_res.fun_control)
```

```
{'l1': np.float64(4.0),
 'epochs': np.float64(4.0),
 'batch_size': np.float64(5.0),
 'act_fn': 'LeakyReLU',
 'optimizer': 'Adamax',
 'dropout_prob': np.float64(0.023931753071792624),
 'lr_mult': np.float64(1.2532486761645163),
 'patience': np.float64(3.0),
 'batch_norm': 0,
 'initialization': 'xavier_normal'}
```

Get the transformed hyperparameters that can be passed to the model.

```
config = get_tuned_architecture(S_res)
pprint.pprint(config)
```

```
{'act_fn': LeakyReLU(),
 'batch_norm': False,
 'batch_size': 32,
 'dropout_prob': 0.023931753071792624,
 'epochs': 16,
 'initialization': 'xavier_normal',
 'l1': 16,
 'lr_mult': 1.2532486761645163,
 'optimizer': 'Adamax',
 'patience': 8}
```

After getting the tuned architecture, the model can be created and tested with the following code.

```
test_model(config, S_res.fun_control)
```

```
    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]
```

```
Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M
```

```
        Test metric               DataLoader 0

         hp_metric                5075.2890625
         val_loss                 5075.2890625
```

```
test_model result: {'val_loss': 5075.2890625, 'hp_metric': 5075.2890625}
```

```
(5075.2890625, 5075.2890625)
```

```
S_res.plot_important_hyperparameter_contour(max_imp=3)
```

```
l1:  0.001
epochs:  0.001
batch_size:  0.001
act_fn:  100.0
optimizer:  50.92590151644171
dropout_prob:  0.001
lr_mult:  0.12004514691831451
patience:  0.001
batch_norm:  0.001
initialization:  0.3107636463812971
```

Prediction Surface
Prediction
22000
20000
18000
16000
act_fn
optimizer
Prediction Error Surface
Error
5000
4500
4000
3500
3000
2500
2000
Prediction Contour
Error Contour
optimizer
act_fn
23600
22400
21200
20000
18800
17600
16400
15200
5400
4950
4500
4050
3600
3150
2700
2250
1800

Prediction Surface
Prediction
act_fn
initialization
Prediction Error Surface
Error
act_fn
initialization
Prediction Contour
initialization
act_fn
Error Contour
initialization
act_fn

```
S_res.plot_importance()
```

56.3. spotpython: Saving and Loading Optimization Experiments

In this section, we will show how results from `spotpython` can be saved and reloaded. Here, `spotpython` can be used as an optimizer. If `spotpython` is used as an optimizer, no dictionary of hyperparameters has be specified. The `fun_control` dictionary is sufficient.

```
PREFIX="branin"

fun = Analytical().fun_branin
fun_control = fun_control_init(
            PREFIX=PREFIX,
            lower = np.array([0, 0]),
            upper = np.array([10, 10]),
            fun_evals=8,
            fun_repeats=1,
            max_time=inf,
            noise=False,
            tolerance_x=0,
```

```
            ocba_delta=0,
            var_type=["num", "num"],
            infill_criterion="ei",
            n_points=1,
            seed=123,
            log_level=20,
            show_models=False,
            save_experiment=True,
            show_progress=True)
design_control = design_control_init(
            init_size=5,
            repeats=1)
surrogate_control = surrogate_control_init(
            model_fun_evals=10000,
            min_theta=-3,
            max_theta=3,
            theta_init_zero=True,
            n_p=1,
            optim_p=False,
            var_type=["num", "num"],
            seed=124)
optimizer_control = optimizer_control_init(
            max_iter=1000,
            seed=125)
S = Spot(fun=fun,
            fun_control=fun_control,
            design_control=design_control,
            surrogate_control=surrogate_control,
            optimizer_control=optimizer_control)
S.run()
```

```
Experiment saved to branin_exp.pkl
spotpython tuning: 4.7932399644479124 [########--] 75.00%. Success rate: 0.00%
spotpython tuning: 2.124848972105818 [#########-] 87.50%. Success rate: 50.00%
spotpython tuning: 2.124848972105818 [##########] 100.00%. Success rate: 33.33% Done.

Experiment saved to branin_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x177eea210>
```

```
S_exp = load_experiment(PREFIX=PREFIX)
```

```
Loaded experiment from branin_exp.pkl
```

```
S_res = load_result(PREFIX=PREFIX)
```

```
Loaded experiment from branin_res.pkl
```

The progress of the original experiment is shown in Figure 56.1 and the reloaded experiment in Figure 56.2.

```
S.plot_progress(log_y=True)
```

Figure 56.1.: Progress of the original experiment

```
S_res.plot_progress(log_y=True)
```

Table 56.1.

```
[['x0', np.float64(10.0)], ['x1', np.float64(2.576684029008911)]]
```

Table 56.2.

```
[['x0', np.float64(10.0)], ['x1', np.float64(2.576684029008911)]]
```

Figure 56.2.: Progress of the reloaded experiment

The results from the original experiment are shown in Table 56.1 and the reloaded experiment in Table 56.2.

```
S.print_results()
```

```
min y: 2.124848972105818
x0: 10.0
x1: 2.576684029008911
```

```
S_res.print_results()
```

```
min y: 2.124848972105818
x0: 10.0
x1: 2.576684029008911
```

56.3.1. Getting the Tuned Hyperparameters

The tuned hyperparameters can be obtained as a dictionary with the following code. Since `spotpython` is used as an optimizer, the numerical levels of the hyperparameters are identical to the optimized values of the underlying optimization problem, here: the Branin function.

```
get_tuned_hyperparameters(spot_tuner=S_res)
```

```
{'x0': np.float64(10.0), 'x1': np.float64(2.576684029008911)}
```

> Summary: Saving and Loading Optimization Experiments
>
> - If `spotpython` is used as an optimizer (without an hyperparameter dictionary), experiments can be saved and reloaded with the `save_experiment` and `load_experiment` functions.
> - The tuned hyperparameters can be obtained with the `get_tuned_hyperparameters` function.

56.4. spotpython as a Hyperparameter Tuner

If `spotpython` is used as a hyperparameter tuner, in addition to the `fun_control` dictionary a `core_model` dictionary has to be specified. Furthermore, a data set has to be selected and added to the `fun_control` dictionary. Here, we will use the `Diabetes` data set.

56.4.1. The Diabetes Data Set

The hyperparameter tuning of a `PyTorch Lightning` network on the `Diabetes` data set is used as an example. The `Diabetes` data set is a PyTorch Dataset for regression, which originates from the `scikit-learn` package, see https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_diabetes.html#sklearn.datasets.load_diabetes.

Ten baseline variables, age, sex, body mass index, average blood pressure, and six blood serum measurements were obtained for each of n = 442 diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline. The `Diabetes` data set is has the following properties:

- Samples total: 442
- Dimensionality: 10

- Features: real, $-.2 < x < .2$
- Targets: integer $25 - 346$

```
data_set = Diabetes()
```

```
PREFIX="604"
fun_control = fun_control_init(
    save_experiment=True,
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)

fun = HyperLight().fun

set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,5])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])

design_control = design_control_init(init_size=10)

print_exp_table(fun_control)
```

```
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
| name            | type    | default    |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------
|
| l1              | int     | 3          |     3   |   4     | transform_power_2_int |
| epochs          | int     | 4          |     3   |   5     | transform_power_2_int |
| batch_size      | int     | 4          |     4   |  11     | transform_power_2_int |
| act_fn          | factor  | ReLU       |     0   |   5     | None                  |
| optimizer       | factor  | SGD        |     0   |   2     | None                  |
| dropout_prob    | float   | 0.01       |     0   |   0.025 | None                  |
| lr_mult         | float   | 1.0        |     0.1 |  10     | None                  |
```

```
| patience       | int    | 2         |     2   |   3     | transform_power_2_int |
| batch_norm     | factor | 0         |     0   |   1     | None                  |
| initialization | factor | Default   |     0   |   4     | None                  |
```

In contrast to the default setting, where `save_experiment` is set to `False`, here the `fun_control` dictionary is initialized `save_experiment=True`. Alternatively, an existing `fun_control` dictionary can be updated with `{"save_experiment": True}` as shown in the following code.

```
fun_control.update({"save_experiment": True})
```

If `save_experiment` is set to `True`, the results of the hyperparameter tuning experiment are stored in a pickle file with the name `PREFIX` after the tuning is finished in the current directory.

Alternatively, the spot object and the corresponding dictionaries can be saved with the `save_experiment` method, which is part of the `spot` object. Therefore, the `spot` object has to be created as shown in the following code.

```
S_diabetes = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
S_diabetes.save_experiment(path="userExperiment", overwrite=False)
```

```
Experiment saved to 604_exp.pkl
Error: File userExperiment/604_exp.pkl already exists. Use overwrite=True to overwrite the file.
```

Here, we have added a `path` argument to specify the directory where the experiment is saved. The resulting pickle file can be copied to another directory or computer and reloaded with the `load_experiment` function. It can also be used for performing the tuning run. Here, we will execute the tuning run on the local machine, which can be done with the following code.

```
S_diabetes_res = S_diabetes.run()
```

```
     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    202 K   train   826 M   [2048, 10]   [2048, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
```

```
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 23075.09765625, 'hp_metric': 23075.09765625}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   6.6 M   [64, 10]     [64, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 3850.81201171875, 'hp_metric': 3850.81201171875}


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 6251.13916015625, 'hp_metric': 6251.13916015625}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]
```

```
Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 24093.25390625, 'hp_metric': 24093.25390625}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 23622.310546875, 'hp_metric': 23622.310546875}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 3700.771728515625, 'hp_metric': 3700.771728515625}
```

```
     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   52.5 M   [512, 10]    [512, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 20202.96875, 'hp_metric': 20202.96875}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 14795.3359375, 'hp_metric': 14795.3359375}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   13.1 M   [128, 10]    [128, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 13.1 M
```

```
train_model result: {'val_loss': 21675.736328125, 'hp_metric': 21675.736328125}


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   413 M   [1024, 10]   [1024, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 413 M

train_model result: {'val_loss': 23792.47265625, 'hp_metric': 23792.47265625}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 23473.31640625, 'hp_metric': 23473.31640625}
spotpython tuning: 3700.771728515625 [----------] 3.21%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   826 M   [2048, 10]   [2048, 1]
```

```
Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 21453.419921875, 'hp_metric': 21453.419921875}
spotpython tuning: 3700.771728515625 [#---------] 5.53%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   25.8 M   [64, 10]     [64, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 16605.611328125, 'hp_metric': 16605.611328125}
spotpython tuning: 3700.771728515625 [#---------] 6.48%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   206 M   [512, 10]    [512, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M

train_model result: {'val_loss': 4418.7109375, 'hp_metric': 4418.7109375}
spotpython tuning: 3700.771728515625 [#---------] 7.97%. Success rate: 0.00%
```

```
     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   826 M   [2048, 10]   [2048, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 19324.853515625, 'hp_metric': 19324.853515625}
spotpython tuning: 3700.771728515625 [#---------] 9.13%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   826 M   [2048, 10]   [2048, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 23236.640625, 'hp_metric': 23236.640625}
spotpython tuning: 3700.771728515625 [#---------] 10.43%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   1.6 M   [16, 10]     [16, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 1.6 M

train_model result: {'val_loss': 23883.439453125, 'hp_metric': 23883.439453125}
spotpython tuning: 3700.771728515625 [#---------] 12.98%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   206 M   [512, 10]    [512, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M

train_model result: {'val_loss': 12952.9296875, 'hp_metric': 12952.9296875}
spotpython tuning: 3700.771728515625 [##--------] 15.46%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   6.6 M   [64, 10]     [64, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 23691.73046875, 'hp_metric': 23691.73046875}
spotpython tuning: 3700.771728515625 [##--------] 16.63%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   206 M   [512, 10]    [512, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M

train_model result: {'val_loss': 5191.41650390625, 'hp_metric': 5191.41650390625}
spotpython tuning: 3700.771728515625 [##--------] 19.31%. Success rate: 0.00%


    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 15954.0634765625, 'hp_metric': 15954.0634765625}
spotpython tuning: 3700.771728515625 [##--------] 20.98%. Success rate: 0.00%


    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 23493.109375, 'hp_metric': 23493.109375}
spotpython tuning: 3700.771728515625 [##--------] 23.23%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  53.2 K   train   6.6 M   [64, 10]     [64, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': 23837.33984375, 'hp_metric': 23837.33984375}
spotpython tuning: 3700.771728515625 [##--------] 24.59%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential  53.2 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': 17594.4921875, 'hp_metric': 17594.4921875}
spotpython tuning: 3700.771728515625 [###-------] 25.67%. Success rate: 0.00%
```

```
    Name    Type        Params  Mode   FLOPs  In sizes  Out sizes

 0  layers  Sequential  51.9 K  train  6.6 M  [64, 10]    [64, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


    Name    Type        Params  Mode    FLOPs   In sizes  Out sizes

 0  layers  Sequential  53.2 K  train  13.1 M  [128, 10]   [128, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.1 M

train_model result: {'val_loss': 23908.48828125, 'hp_metric': 23908.48828125}
spotpython tuning: 3700.771728515625 [###-------] 26.98%. Success rate: 0.00%


    Name    Type        Params  Mode   FLOPs   In sizes  Out sizes

 0  layers  Sequential   205 K  train  206 M  [512, 10]   [512, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M

train_model result: {'val_loss': 24016.99609375, 'hp_metric': 24016.99609375}
spotpython tuning: 3700.771728515625 [###-------] 28.07%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    205 K   train   826 M   [2048, 10]   [2048, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 826 M

train_model result: {'val_loss': 24098.27734375, 'hp_metric': 24098.27734375}
spotpython tuning: 3700.771728515625 [###-------] 29.03%. Success rate: 0.00%

Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   25.8 M   [64, 10]     [64, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 23996.900390625, 'hp_metric': 23996.900390625}
spotpython tuning: 3700.771728515625 [###-------] 30.35%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 10241.0615234375, 'hp_metric': 10241.0615234375}
spotpython tuning: 3700.771728515625 [###-------] 32.90%. Success rate: 0.00%


    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 19619.40625, 'hp_metric': 19619.40625}
spotpython tuning: 3700.771728515625 [###-------] 34.25%. Success rate: 0.00%


    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   52.5 M   [512, 10]    [512, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 20175.65234375, 'hp_metric': 20175.65234375}
spotpython tuning: 3700.771728515625 [####------] 35.71%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 12377.2626953125, 'hp_metric': 12377.2626953125}
spotpython tuning: 3700.771728515625 [####------] 39.41%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 22732.955078125, 'hp_metric': 22732.955078125}
spotpython tuning: 3700.771728515625 [####------] 41.05%. Success rate: 0.00%
```

```
     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 23719.169921875, 'hp_metric': 23719.169921875}
spotpython tuning: 3700.771728515625 [#####-----] 45.50%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   25.8 M   [64, 10]     [64, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 17149.6484375, 'hp_metric': 17149.6484375}
spotpython tuning: 3700.771728515625 [#####-----] 46.82%. Success rate: 0.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    205 K   train   25.8 M   [64, 10]     [64, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 16395.931640625, 'hp_metric': 16395.931640625}
spotpython tuning: 3700.771728515625 [#####-----] 48.04%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   206 M   [512, 10]    [512, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 206 M

train_model result: {'val_loss': 23803.443359375, 'hp_metric': 23803.443359375}
spotpython tuning: 3700.771728515625 [#####-----] 49.64%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   53.2 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': 23311.802734375, 'hp_metric': 23311.802734375}
spotpython tuning: 3700.771728515625 [#####-----] 52.82%. Success rate: 0.00%
```

```
    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    205 K   train   25.8 M   [64, 10]     [64, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 25.8 M

train_model result: {'val_loss': 23695.421875, 'hp_metric': 23695.421875}
spotpython tuning: 3700.771728515625 [#####-----] 54.83%. Success rate: 0.00%


    Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

 0  layers   Sequential   53.2 K   train   104 M   [1024, 10]   [1024, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 104 M

train_model result: {'val_loss': 23479.724609375, 'hp_metric': 23479.724609375}
spotpython tuning: 3700.771728515625 [######----] 56.61%. Success rate: 0.00%


    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    205 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': 19671.98046875, 'hp_metric': 19671.98046875}
spotpython tuning: 3700.771728515625 [######----] 58.00%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   6.6 M   [64, 10]     [64, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 6.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    205 K   train   103 M   [256, 10]    [256, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 103 M

train_model result: {'val_loss': 23461.787109375, 'hp_metric': 23461.787109375}
spotpython tuning: 3700.771728515625 [######----] 59.71%. Success rate: 0.00%
```

```
    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   53.2 K  train   52.5 M  [512, 10]    [512, 1]


Trainable params: 53.2 K
Non-trainable params: 0
Total params: 53.2 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 52.5 M

train_model result: {'val_loss': 16165.0400390625, 'hp_metric': 16165.0400390625}
spotpython tuning: 3700.771728515625 [######----] 61.59%. Success rate: 0.00%


    Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    205 K  train   12.9 M  [32, 10]     [32, 1]


Trainable params: 205 K
Non-trainable params: 0
Total params: 205 K
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 12.9 M

train_model result: {'val_loss': 8532.5380859375, 'hp_metric': 8532.5380859375}
spotpython tuning: 3700.771728515625 [######----] 62.99%. Success rate: 0.00%


    Name     Type        Params   Mode   FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K  train   3.3 M  [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [######----] 64.06%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 65.07%. Success rate: 0.00%


    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 66.07%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 67.09%. Success rate: 0.00%
```

```
     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 68.09%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 69.20%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 70.21%. Success rate: 0.00%


    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 71.28%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 72.34%. Success rate: 0.00%
```

```
     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 73.34%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#######---] 74.40%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 75.41%. Success rate: 0.00%


    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 76.42%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 77.43%. Success rate: 0.00%
```

```
     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 78.47%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 79.48%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 80.49%. Success rate: 0.00%


    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 81.48%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 82.55%. Success rate: 0.00%
```

```
     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 83.63%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [########--] 84.62%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#########-] 85.63%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#########-] 86.71%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#########-] 87.72%. Success rate: 0.00%
```

```
     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#########-] 88.73%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########-] 89.81%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#########-] 90.81%. Success rate: 0.00%


    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#########-] 91.81%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#########-] 92.81%. Success rate: 0.00%
```

```
     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [#########-] 93.87%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########-] 94.93%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.
```

```
     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########] 95.93%. Success rate: 0.00%


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########] 96.92%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential  51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential   202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########] 97.99%. Success rate: 0.00%
```

```
     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type        Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########] 98.98%. Success rate: 0.00%


     Name     Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential   51.9 K   train   3.3 M   [32, 10]     [32, 1]


Trainable params: 51.9 K
Non-trainable params: 0
Total params: 51.9 K
```

```
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 3.3 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
Using spacefilling design as fallback.


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential    202 K   train   51.6 M   [128, 10]    [128, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 51.6 M

train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3700.771728515625 [##########] 100.00%. Success rate: 0.00% Done...

Experiment saved to 604_res.pkl
```

After the tuning run is finished, a pickle file with the name `spot_604_experiment.pickle` is stored in the local directory. This is a result of setting the `save_experiment` argument to `True` in the `fun_control` dictionary. We can load the experiment with the following code. Here, we have specified the `PREFIX` as an argument to the `load_experiment` function. Alternatively, the filename (`filename`) can be used as an argument.

```
S_diabetes_load_exp = load_experiment(PREFIX=PREFIX)
S_diabetes_load_res = load_result(PREFIX=PREFIX)
```

```
Loaded experiment from 604_exp.pkl
Loaded experiment from 604_res.pkl
```

For comparison, the tuned hyperparameters of the original experiment are shown first:

```
get_tuned_hyperparameters(S_diabetes, fun_control)
```

```
{'l1': np.float64(4.0),
 'epochs': np.float64(4.0),
 'batch_size': np.float64(5.0),
 'act_fn': 'LeakyReLU',
 'optimizer': 'Adamax',
 'dropout_prob': np.float64(0.023931753071792624),
 'lr_mult': np.float64(1.2532486761645163),
 'patience': np.float64(3.0),
 'batch_norm': 0,
 'initialization': 'xavier_normal'}
```

Second, the tuned hyperparameters of the reloaded experiment are shown:

```
get_tuned_hyperparameters(S_diabetes_load_res)
```

```
{'l1': np.float64(4.0),
 'epochs': np.float64(4.0),
 'batch_size': np.float64(5.0),
 'act_fn': np.float64(3.0),
 'optimizer': np.float64(2.0),
 'dropout_prob': np.float64(0.023931753071792624),
 'lr_mult': np.float64(1.2532486761645163),
 'patience': np.float64(3.0),
 'batch_norm': np.float64(0.0),
 'initialization': np.float64(4.0)}
```

Note: The numerical levels of the hyperparameters are used as keys in the dictionary. If the `fun_control` dictionary is used, the names of the hyperparameters are used as keys in the dictionary.

```
get_tuned_hyperparameters(S_diabetes_load_res, fun_control=S_diabetes_load_exp.fun_control)
```

```
{'l1': np.float64(4.0),
 'epochs': np.float64(4.0),
 'batch_size': np.float64(5.0),
 'act_fn': 'LeakyReLU',
 'optimizer': 'Adamax',
 'dropout_prob': np.float64(0.023931753071792624),
```

```
 'lr_mult': np.float64(1.2532486761645163),
 'patience': np.float64(3.0),
 'batch_norm': 0,
 'initialization': 'xavier_normal'}
```

Plot the progress of the original experiment are identical to the reloaded experiment.

```
S_diabetes.plot_progress()
```

Figure 56.3.

```
S_diabetes_load_res.plot_progress()
```

Figure 56.4.

Summary: Saving and Loading Hyperparameter-Tuning Experiments

- If `spotpython` is used as an hyperparameter tuner (with an hyperparameter dictionary), experiments can be saved and reloaded with the `save_experiment` and `load_experiment` functions.
- The tuned hyperparameters can be obtained with the `get_tuned_hyperparameters` function.

56.5. Saving and Loading PyTorch Lightning Models

Section 56.3 and Section 56.4 explained how to save and load optimization and hyperparameter tuning experiments and how to get the tuned hyperparameters as a dictionary. This section shows how to save and load `PyTorch Lightning` models.

56.5.1. Get the Tuned Architecture

In contrast to the function `get_tuned_hyperparameters`, the function `get_tuned_architecture` returns the tuned architecture of the model as a dictionary. Here, the transformations are already applied to the numerical levels of the hyperparameters and the encoding (and types) are the original types of the hyperparameters used by the model. Important: The `config` dictionary from `get_tuned_architecture` can be passed to the model without any modifications.

```
config = get_tuned_architecture(S_diabetes)
pprint.pprint(config)
```

```
{'act_fn': LeakyReLU(),
 'batch_norm': False,
 'batch_size': 32,
 'dropout_prob': 0.023931753071792624,
 'epochs': 16,
 'initialization': 'xavier_normal',
 'l1': 16,
 'lr_mult': 1.2532486761645163,
 'optimizer': 'Adamax',
 'patience': 8}
```

After getting the tuned architecture, the model can be created and tested with the following code.

```
test_model(config, fun_control)
```

```
     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    202 K   train   12.9 M   [32, 10]     [32, 1]


Trainable params: 202 K
Non-trainable params: 0
Total params: 202 K
Total estimated model params size (MB): 0
Modules in train mode: 17
Modules in eval mode: 0
Total FLOPs: 12.9 M


        Test metric               DataLoader 0

         hp_metric                5075.2890625
         val_loss                 5075.2890625


test_model result: {'val_loss': 5075.2890625, 'hp_metric': 5075.2890625}


(5075.2890625, 5075.2890625)
```

56.5.2. Load a Model from Checkpoint

The method `load_light_from_checkpoint` loads a model from a checkpoint file. Important: The model has to be trained before the checkpoint is loaded. As shown here, loading a model with trained weights is possible, but requires two steps:

1. The model weights have to be learned using `test_model`. The `test_model` method writes a checkpoint file.
2. The model has to be loaded from the checkpoint file.

56.5.2.1. Details About the `load_light_from_checkpoint` Method

- The `test_model` method saves the last checkpoint to a file using the following code:

```
ModelCheckpoint(
    dirpath=os.path.join(fun_control["CHECKPOINT_PATH"], config_id), save_last=True
),
```

The filename of the last checkpoint has a specific structure:

- A `config_id` is generated from the `config` dictionary. It does not use a timestamp. This differs from the config id generated in cvmodel.py and trainmodel.py, which provide time information for the TensorBoard logging.
- Furthermore, the postfix `_TEST` is added to the `config_id` to indicate that the model is tested.
- For example: `runs/saved_models/16_16_64_LeakyReLU_Adadelta_0.0014_8.5895_8_False_kaiming_uniform_`

```
model_loaded = load_light_from_checkpoint(config, fun_control)
```

```
config: {'l1': 16, 'epochs': 16, 'batch_size': 32, 'act_fn': LeakyReLU(), 'optimizer': 'Adamax', 'dr
Loading model with 16_16_32_LeakyReLU_Adamax_0.0239_1.2532_8_False_xavier_normal_TEST from runs/save
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): LeakyReLU()
    (2): Dropout(p=0.023931753071792624, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): LeakyReLU()
    (5): Dropout(p=0.023931753071792624, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): LeakyReLU()
    (8): Dropout(p=0.023931753071792624, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): LeakyReLU()
    (11): Dropout(p=0.023931753071792624, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): LeakyReLU()
    (14): Dropout(p=0.023931753071792624, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): LeakyReLU()
    (17): Dropout(p=0.023931753071792624, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): LeakyReLU()
```

```
    (20): Dropout(p=0.023931753071792624, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
```

```
vars(model_loaded)
```

```
{'training': False,
 '_parameters': {},
 '_buffers': {},
 '_non_persistent_buffers_set': set(),
 '_backward_pre_hooks': OrderedDict(),
 '_backward_hooks': OrderedDict(),
 '_is_full_backward_hook': None,
 '_forward_hooks': OrderedDict(),
 '_forward_hooks_with_kwargs': OrderedDict(),
 '_forward_hooks_always_called': OrderedDict(),
 '_forward_pre_hooks': OrderedDict(),
 '_forward_pre_hooks_with_kwargs': OrderedDict(),
 '_state_dict_hooks': OrderedDict(),
 '_state_dict_pre_hooks': OrderedDict(),
 '_load_state_dict_pre_hooks': OrderedDict(),
 '_load_state_dict_post_hooks': OrderedDict(),
 '_modules': {'layers': Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): LeakyReLU()
    (2): Dropout(p=0.023931753071792624, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): LeakyReLU()
    (5): Dropout(p=0.023931753071792624, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): LeakyReLU()
    (8): Dropout(p=0.023931753071792624, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): LeakyReLU()
    (11): Dropout(p=0.023931753071792624, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): LeakyReLU()
    (14): Dropout(p=0.023931753071792624, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): LeakyReLU()
    (17): Dropout(p=0.023931753071792624, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): LeakyReLU()
```

```
   (20): Dropout(p=0.023931753071792624, inplace=False)
   (21): Linear(in_features=80, out_features=1, bias=True)
 )},
'prepare_data_per_node': True,
'allow_zero_length_dataloader_with_multiple_devices': False,
'_log_hyperparams': True,
'_dtype': torch.float32,
'_device': device(type='mps', index=0),
'_trainer': None,
'_example_input_array': tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]),
'_automatic_optimization': True,
'_strict_loading': None,
'_current_fx_name': None,
'_param_requires_grad_state': {},
```

```
 '_metric_attributes': None,
 '_compiler_ctx': None,
 '_fabric': None,
 '_fabric_optimizers': [],
 '_device_mesh': None,
 '_L_in': 10,
 '_L_out': 1,
 '_torchmetric': 'mean_squared_error',
 'metric': <function torchmetrics.functional.regression.mse.mean_squared_error(preds:
 '_hparams_name': 'kwargs',
 '_hparams': "_L_cond":        None
 "act_fn":         LeakyReLU()
 "batch_norm":     False
 "batch_size":     32
 "dropout_prob":   0.023931753071792624
 "epochs":         16
 "initialization": xavier_normal
 "l1":             16
 "lr_mult":        1.2532486761645163
 "optimizer":      Adamax
 "patience":       8,
 '_hparams_initial': "_L_cond":        None
 "act_fn":         LeakyReLU()
 "batch_norm":     False
 "batch_size":     32
 "dropout_prob":   0.023931753071792624
 "epochs":         16
 "initialization": xavier_normal
 "l1":             16
 "lr_mult":        1.2532486761645163
 "optimizer":      Adamax
 "patience":       8}
```

```
torch.save(model_loaded, "model.pt")
```

Note the following warning: In PyTorch 2.6, the default value of the `weights_only` argument in `torch.load` was changed from `False` to `True`. Re-running `torch.load` with `weights_only` set to `False` will likely succeed, but it can result in arbitrary code execution. Do it only if you got the file from a trusted source.

```
mymodel = torch.load("model.pt", weights_only=False)
```

Show all attributes of the model:

```
vars(mymodel)
```

```
{'training': False,
 '_parameters': {},
 '_buffers': {},
 '_non_persistent_buffers_set': set(),
 '_backward_pre_hooks': OrderedDict(),
 '_backward_hooks': OrderedDict(),
 '_is_full_backward_hook': None,
 '_forward_hooks': OrderedDict(),
 '_forward_hooks_with_kwargs': OrderedDict(),
 '_forward_hooks_always_called': OrderedDict(),
 '_forward_pre_hooks': OrderedDict(),
 '_forward_pre_hooks_with_kwargs': OrderedDict(),
 '_state_dict_hooks': OrderedDict(),
 '_state_dict_pre_hooks': OrderedDict(),
 '_load_state_dict_pre_hooks': OrderedDict(),
 '_load_state_dict_post_hooks': OrderedDict(),
 '_modules': {'layers': Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): LeakyReLU()
    (2): Dropout(p=0.023931753071792624, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): LeakyReLU()
    (5): Dropout(p=0.023931753071792624, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): LeakyReLU()
    (8): Dropout(p=0.023931753071792624, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): LeakyReLU()
    (11): Dropout(p=0.023931753071792624, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): LeakyReLU()
    (14): Dropout(p=0.023931753071792624, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): LeakyReLU()
    (17): Dropout(p=0.023931753071792624, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): LeakyReLU()
    (20): Dropout(p=0.023931753071792624, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )},
 'prepare_data_per_node': True,
 'allow_zero_length_dataloader_with_multiple_devices': False,
```

```
 '_log_hyperparams': True,
 '_dtype': torch.float32,
 '_device': device(type='mps', index=0),
 '_trainer': None,
 '_example_input_array': tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]),
 '_automatic_optimization': True,
 '_strict_loading': None,
 '_current_fx_name': None,
 '_param_requires_grad_state': {},
 '_metric_attributes': None,
 '_compiler_ctx': None,
 '_fabric': None,
 '_fabric_optimizers': [],
 '_device_mesh': None,
```

```
'_L_in': 10,
'_L_out': 1,
'_torchmetric': 'mean_squared_error',
'metric': <function torchmetrics.functional.regression.mse.mean_squared_error(preds: torch.Tensor,
'_hparams_name': 'kwargs',
'_hparams': "_L_cond":        None
"act_fn":         LeakyReLU()
"batch_norm":     False
"batch_size":     32
"dropout_prob":   0.023931753071792624
"epochs":         16
"initialization": xavier_normal
"l1":             16
"lr_mult":        1.2532486761645163
"optimizer":      Adamax
"patience":       8,
'_hparams_initial': "_L_cond":        None
"act_fn":         LeakyReLU()
"batch_norm":     False
"batch_size":     32
"dropout_prob":   0.023931753071792624
"epochs":         16
"initialization": xavier_normal
"l1":             16
"lr_mult":        1.2532486761645163
"optimizer":      Adamax
"patience":       8}
```

56.6. Converting a Lightning Model to a Plain Torch Model

56.6.1. The Function `get_removed_attributes_and_base_net`

`spotpython` provides a function to covert a `PyTorch Lightning` model to a plain `PyTorch` model. The function `get_removed_attributes_and_base_net` returns a tuple with the removed attributes and the base net. The base net is a plain `PyTorch` model. The removed attributes are the attributes of the `PyTorch Lightning` model that are not part of the base net.

This conversion can be reverted.

```
removed_attributes, torch_net = get_removed_attributes_and_base_net(net=mymodel)
```

```
print(removed_attributes)
```

```
{'_metric_attributes': None, 'allow_zero_length_dataloader_with_multiple_devices': Fal
"act_fn":         LeakyReLU()
"batch_norm":     False
"batch_size":     32
"dropout_prob":   0.023931753071792624
"epochs":         16
"initialization": xavier_normal
"l1":             16
"lr_mult":        1.2532486761645163
"optimizer":      Adamax
"patience":       8, '_L_in': 10, '_strict_loading': None, '_automatic_optimization':
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]), 'metric': <function mean_squared_e
```

```
"act_fn":         LeakyReLU()
"batch_norm":     False
"batch_size":     32
"dropout_prob":   0.023931753071792624
"epochs":         16
"initialization": xavier_normal
"l1":             16
"lr_mult":        1.2532486761645163
"optimizer":      Adamax
"patience":       8, '_torchmetric': 'mean_squared_error', '_L_out': 1, '_device_mesh': None, 'prepa
```

```python
print(torch_net)
```

```
NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): LeakyReLU()
    (2): Dropout(p=0.023931753071792624, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): LeakyReLU()
    (5): Dropout(p=0.023931753071792624, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): LeakyReLU()
    (8): Dropout(p=0.023931753071792624, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): LeakyReLU()
    (11): Dropout(p=0.023931753071792624, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): LeakyReLU()
    (14): Dropout(p=0.023931753071792624, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): LeakyReLU()
    (17): Dropout(p=0.023931753071792624, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): LeakyReLU()
    (20): Dropout(p=0.023931753071792624, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
```

56.6.2. An Example how to use the Plain Torch Net

```
# Load the Diabetes dataset from sklearn
diabetes = load_diabetes()
X = diabetes.data
y = diabetes.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=

# Scale the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Convert the data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

# Create a PyTorch dataset
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

# Create a PyTorch dataloader
batch_size = 32
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size)

torch_net.to(getDevice("cpu"))

# train the net
criterion = nn.MSELoss()
optimizer = optim.Adam(torch_net.parameters(), lr=0.01)
n_epochs = 100
losses = []
for epoch in range(n_epochs):
    for inputs, targets in train_dataloader:
        targets = targets.view(-1, 1)
        optimizer.zero_grad()
        outputs = torch_net(inputs)
        loss = criterion(outputs, targets)
        losses.append(loss.item())
```

```
        loss.backward()
        optimizer.step()
# visualize the network training
plt.plot(losses)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

57. Hyperparameter Tuning with spotpython and PyTorch Lightning for the Diabetes Data Set Using a ResNet Model

In this section, we will show how spotpython can be integrated into the PyTorch Lightning training workflow for a regression task. It demonstrates how easy it is to use spotpython to tune hyperparameters for a PyTorch Lightning model.

After importing the necessary libraries, the fun_control dictionary is set up via the fun_control_init function. The fun_control dictionary contains

- PREFIX: a unique identifier for the experiment
- fun_evals: the number of function evaluations
- max_time: the maximum run time in minutes
- data_set: the data set. Here we use the Diabetes data set that is provided by spotpython.
- core_model_name: the class name of the neural network model. This neural network model is provided by spotpython.
- hyperdict: the hyperparameter dictionary. This dictionary is used to define the hyperparameters of the neural network model. It is also provided by spotpython.
- _L_in: the number of input features. Since the Diabetes data set has 10 features, _L_in is set to 10.
- _L_out: the number of output features. Since we want to predict a single value, _L_out is set to 1.

The HyperLight class is used to define the objective function fun. It connects the PyTorch and the spotpython methods and is provided by spotpython.

Note, the divergence_threshold is set to 5,000, which is based on some pre-experiments with the Diabetes data set.

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_control_init)
from spotpython.utils.eda import print_exp_table
```

```
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename

PREFIX="605"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNResNetRegressor",
    hyperdict=LightHyperDict,
    divergence_threshold=25_000,
    _L_in=10,
    _L_out=1)

fun = HyperLight().fun
```

```
module_name: light
submodule_name: regression
model_name: NNResNetRegressor
```

The method `set_hyperparameter` allows the user to modify default hyperparameter settings. Here we modify some hyperparameters to keep the model small and to decrease the tuning time.

```
from spotpython.hyperparameters.values import set_hyperparameter
set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,7])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])
set_hyperparameter(fun_control, "lr_mult", [0.1, 20.0])

design_control = design_control_init(init_size=10)

print_exp_table(fun_control)
```

```
| name           | type    | default    |    lower |    upper | transform              |
```

```
|----------------|--------|-----------|---------|---------|-----------------------
|
| l1             | int    | 3         |     3   |   4     | transform_power_2_int |
| epochs         | int    | 4         |     3   |   7     | transform_power_2_int |
| batch_size     | int    | 4         |     4   |  11     | transform_power_2_int |
| act_fn         | factor | ReLU      |     0   |   5     | None                  |
| optimizer      | factor | SGD       |     0   |   2     | None                  |
| dropout_prob   | float  | 0.01      |     0   |   0.025 | None                  |
| lr_mult        | float  | 1.0       |     0.1 |  20     | None                  |
| patience       | int    | 2         |     2   |   3     | transform_power_2_int |
| initialization | factor | Default   |     0   |   4     | None                  |
```

Finally, a `Spot` object is created. Calling the method `run()` starts the hyperparameter tuning process.

```
spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
res = spot_tuner.run()
```

```
     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   95.7 K   [128, 10]    [128, 1]
```

```
Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 95.7 K

train_model result: {'val_loss': 23488.20703125, 'hp_metric': 23488.20703125}
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   310 K   [128, 10]    [128, 1]
```

```
Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 310 K

train_model result: {'val_loss': 23382.1171875, 'hp_metric': 23382.1171875}



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 23932.60546875, 'hp_metric': 23932.60546875}



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   38.8 K   [16, 10]     [16, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 38.8 K
```

```
train_model result: {'val_loss': 23544.123046875, 'hp_metric': 23544.123046875}


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   2.5 M   [1024, 10]   [1024, 1]


Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 2.5 M

train_model result: {'val_loss': 23700.796875, 'hp_metric': 23700.796875}


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential      637   train   1.5 M   [2048, 10]   [2048, 1]


Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 1.5 M

train_model result: {'val_loss': 5643.2451171875, 'hp_metric': 5643.2451171875}


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   191 K   [256, 10]    [256, 1]
```

```
Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 191 K

train_model result: {'val_loss': 24055.76953125, 'hp_metric': 24055.76953125}



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   382 K   [512, 10]    [512, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 382 K

train_model result: {'val_loss': 22876.900390625, 'hp_metric': 22876.900390625}



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential      637   train   23.9 K   [32, 10]     [32, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 23.9 K
```

```
train_model result: {'val_loss': 23417.650390625, 'hp_metric': 23417.650390625}


     Name    Type        Params  Mode   FLOPs  In sizes  Out sizes

  0  layers  Sequential   1.8 K  train  155 K  [64, 10]    [64, 1]


Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

train_model result: {'val_loss': 23713.25390625, 'hp_metric': 23713.25390625}


     Name    Type        Params  Mode   FLOPs    In sizes  Out sizes

  0  layers  Sequential     637  train  1.5 M  [2048, 10]  [2048, 1]


Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 1.5 M

train_model result: {'val_loss': 3807.633544921875, 'hp_metric': 3807.633544921875}
spotpython tuning: 3807.633544921875 [#---------] 7.61%. Success rate: 100.00%


     Name    Type        Params  Mode   FLOPs    In sizes  Out sizes

  0  layers  Sequential     637  train  1.5 M  [2048, 10]  [2048, 1]
```

```
Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 1.5 M

train_model result: {'val_loss': 4661.5361328125, 'hp_metric': 4661.5361328125}
spotpython tuning: 3807.633544921875 [#---------] 13.10%. Success rate: 50.00%


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   77.7 K   [32, 10]     [32, 1]


Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 77.7 K

train_model result: {'val_loss': 23766.650390625, 'hp_metric': 23766.650390625}
spotpython tuning: 3807.633544921875 [##--------] 21.70%. Success rate: 33.33%


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential      637   train   1.5 M   [2048, 10]   [2048, 1]


Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 1.5 M
```

```
train_model result: {'val_loss': 23360.853515625, 'hp_metric': 23360.853515625}
spotpython tuning: 3807.633544921875 [####------] 37.63%. Success rate: 25.00%


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential      637   train   1.5 M   [2048, 10]   [2048, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 1.5 M

train_model result: {'val_loss': 21793.38671875, 'hp_metric': 21793.38671875}
spotpython tuning: 3807.633544921875 [#####-----] 45.29%. Success rate: 20.00%


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   382 K   [512, 10]    [512, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 382 K

train_model result: {'val_loss': 23223.68359375, 'hp_metric': 23223.68359375}
spotpython tuning: 3807.633544921875 [#####-----] 46.98%. Success rate: 16.67%
```

```
    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

train_model result: {'val_loss': 23547.923828125, 'hp_metric': 23547.923828125}
spotpython tuning: 3807.633544921875 [######----] 63.46%. Success rate: 14.29%



    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   2.5 M   [1024, 10]   [1024, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 2.5 M

train_model result: {'val_loss': 23432.609375, 'hp_metric': 23432.609375}
spotpython tuning: 3807.633544921875 [########--] 79.47%. Success rate: 12.50%



    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   621 K   [256, 10]    [256, 1]
```

```
Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 621 K

train_model result: {'val_loss': 8173.9130859375, 'hp_metric': 8173.9130859375}
spotpython tuning: 3807.633544921875 [#########-] 90.58%. Success rate: 11.11%


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   621 K   [256, 10]    [256, 1]


Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 621 K

train_model result: {'val_loss': 12125.3984375, 'hp_metric': 12125.3984375}
spotpython tuning: 3807.633544921875 [##########] 99.07%. Success rate: 10.00%

Using spacefilling design as fallback.


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   310 K   [128, 10]    [128, 1]


Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 310 K
```

```
train_model result: {'val_loss': 24363.330078125, 'hp_metric': 24363.330078125}
spotpython tuning: 3807.633544921875 [##########] 100.00%. Success rate: 9.09% Done..

Experiment saved to 605_res.pkl
```

57.1. Looking at the Results

57.1.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with spotpython's method plot_progress. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress()
```

57.1.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

```
| name           | type    | default    |   lower |    upper | tuned          | transform
|----------------|---------|------------|---------|---------|---------------
|-----------------------|--------------|---------|
| l1             | int     | 3          |     3.0 |     4.0 | 3.0            | transform_
```

```
| epochs         | int    | 4         |     3.0 |     7.0 | 7.0           | transform_power_2_int |
| batch_size     | int    | 4         |     4.0 |    11.0 | 11.0          | transform_power_2_int |
| act_fn         | factor | ReLU      |     0.0 |     5.0 | ReLU          | None                  |
| optimizer      | factor | SGD       |     0.0 |     2.0 | Adadelta      | None                  |
| dropout_prob   | float  | 0.01      |     0.0 |   0.025 | 0.025         | None                  |
| lr_mult        | float  | 1.0       |     0.1 |    20.0 | 20.0          | None                  |
| patience       | int    | 2         |     2.0 |     3.0 | 3.0           | transform_power_2_int |
| initialization | factor | Default   |     0.0 |     4.0 | xavier_normal | None                  |
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=1.0)
```

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
l1:  0.0016225763557771772
epochs:  0.0016225763557771772
batch_size:  0.0016225763557771772
act_fn:  0.0016225763557771772
optimizer:  4.029411510711966
dropout_prob:  0.003366578814863606
```

```
lr_mult:  12.30899443434734
patience:  0.0016225763557771772
initialization:  100.0
```

Prediction Surface
Prediction
optimizer
initialization
Prediction Error Surface
Error
optimizer
initialization
Prediction Contour
initialization
optimizer
Error Contour
initialization
optimizer

57.1.3. Get the Tuned Architecture

```
import pprint
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner)
pprint.pprint(config)
```

```
{'act_fn': ReLU(),
 'batch_size': 2048,
 'dropout_prob': 0.025,
 'epochs': 128,
 'initialization': 'xavier_normal',
 'l1': 8,
 'lr_mult': 20.0,
 'optimizer': 'Adadelta',
```

```
 'patience': 8}
```

57.1.4. Test on the full data set

```
# set the value of the key "TENSORBOARD_CLEAN" to True in the fun_control dictionary and use the upd
import os
# if the directory "./runs" exists, delete it
if os.path.exists("./runs"):
    os.system("rm -r ./runs")
fun_control.update({"tensorboard_log": True})
```

```
from spotpython.light.testmodel import test_model
from spotpython.utils.init import get_feature_names

test_model(config, fun_control)
get_feature_names(fun_control)
```

```
     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential      637   train   1.5 M   [2048, 10]   [2048, 1]


Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 1.5 M


         Test metric                DataLoader 0

          hp_metric                   25945.625
          val_loss                    25945.625


test_model result: {'val_loss': 25945.625, 'hp_metric': 25945.625}
```

```
['age',
 'sex',
 'bmi',
 'bp',
 's1_tc',
 's2_ldl',
 's3_hdl',
 's4_tch',
 's5_ltg',
 's6_glu']
```

57.1.5. Cross Validation With Lightning

- The `KFold` class from `sklearn.model_selection` is used to generate the folds for cross-validation.
- These mechanism is used to generate the folds for the final evaluation of the model.
- The `CrossValidationDataModule` class [SOURCE] is used to generate the folds for the hyperparameter tuning process.
- It is called from the `cv_model` function [SOURCE].

```
config
```

```
{'l1': 8,
 'epochs': 128,
 'batch_size': 2048,
 'act_fn': ReLU(),
 'optimizer': 'Adadelta',
 'dropout_prob': 0.025,
 'lr_mult': 20.0,
 'patience': 8,
 'initialization': 'xavier_normal'}
```

```
from spotpython.light.cvmodel import cv_model
fun_control.update({"k_folds": 2})
fun_control.update({"test_size": 0.6})
cv_model(config, fun_control)
```

```
k: 0


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes
```

```
  0   layers   Sequential      637   train   1.5 M   [2048, 10]   [2048, 1]


Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 1.5 M

train_model result: {'val_loss': 3644.15673828125, 'hp_metric': 3644.15673828125}
k: 1


     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0   layers   Sequential      637   train   1.5 M   [2048, 10]   [2048, 1]


Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 1.5 M

train_model result: {'val_loss': 6444.04931640625, 'hp_metric': 6444.04931640625}

5044.10302734375
```

57.2. Summary

This section presented an introduction to the basic setup of hyperparameter tuning with `spotpython` and `PyTorch` Lightning using a ResNet model for the Diabetes data set.

58. Hyperparameter Tuning with spotpython and PyTorch Lightning for the Diabetes Data Set Using a User Specified ResNet Model

After importing the necessary libraries, the `fun_control` dictionary is set up via the `fun_control_init` function. The `fun_control` dictionary contains

- `PREFIX`: a unique identifier for the experiment
- `fun_evals`: the number of function evaluations
- `max_time`: the maximum run time in minutes
- `data_set`: the data set. Here we use the `Diabetes` data set that is provided by `spotpython`.
- `core_model_name`: the class name of the neural network model. This neural network model is provided by `spotpython`.
- `hyperdict`: the hyperparameter dictionary. This dictionary is used to define the hyperparameters of the neural network model. It is also provided by `spotpython`.
- `_L_in`: the number of input features. Since the `Diabetes` data set has 10 features, `_L_in` is set to 10.
- `_L_out`: the number of output features. Since we want to predict a single value, `_L_out` is set to 1.

The `HyperLight` class is used to define the objective function `fun`. It connects the `PyTorch` and the `spotpython` methods and is provided by `spotpython`.

To access the user specified ResNet model, the path to the user model must be added to the Python path:

```
import sys
sys.path.insert(0, './userModel')
import my_resnet
import my_hyper_dict
```

In the following code, we do not specify the ResNet model in the `fun_control` dictionary. It will be added in a second step as the user specified model.

Note, the divergence_threshold is set to 5,000, which is based on some pre-experiments with the Diabetes data set.

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_co
from spotpython.utils.eda import print_exp_table
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename

PREFIX="606-user-resnet"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    divergence_threshold=5_000,
    _L_in=10,
    _L_out=1)

fun = HyperLight().fun
```

In a second step, we can add the user specified ResNet model to the fun_control dictionary:

```
from spotpython.hyperparameters.values import add_core_model_to_fun_control
add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=my_resnet.MyResNet,
                              hyper_dict=my_hyper_dict.MyHyperDict)
```

The method set_hyperparameter allows the user to modify default hyperparameter settings. Here we modify some hyperparameters to keep the model small and to decrease the tuning time.

```
from spotpython.hyperparameters.values import set_hyperparameter
set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,7])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
```

```
set_hyperparameter(fun_control, "patience", [2,3])
set_hyperparameter(fun_control, "lr_mult", [0.1, 20.0])

design_control = design_control_init(init_size=10)

print_exp_table(fun_control)
```

```
| name           | type   | default   |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------
|
| l1             | int    | 3         |     3   |   4     | transform_power_2_int |
| epochs         | int    | 4         |     3   |   7     | transform_power_2_int |
| batch_size     | int    | 4         |     4   |  11     | transform_power_2_int |
| act_fn         | factor | ReLU      |     0   |   5     | None                  |
| optimizer      | factor | SGD       |     0   |   2     | None                  |
| dropout_prob   | float  | 0.01      |     0   |   0.025 | None                  |
| lr_mult        | float  | 1.0       |     0.1 |  20     | None                  |
| patience       | int    | 2         |     2   |   3     | transform_power_2_int |
| initialization | factor | Default   |     0   |   4     | None                  |
```

Finally, a `Spot` object is created. Calling the method `run()` starts the hyperparameter tuning process.

```
spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
res = spot_tuner.run()
```

```
Milestones: [16, 32, 48]



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   95.7 K   [128, 10]    [128, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 95.7 K
```

```
train_model result: {'val_loss': 24029.263671875, 'hp_metric': 24029.263671875}
Milestones: [2, 4, 6]


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   310 K   [128, 10]    [128, 1]


Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 310 K

train_model result: {'val_loss': 23935.533203125, 'hp_metric': 23935.533203125}
Milestones: [16, 32, 48]


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]


Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 23620.296875, 'hp_metric': 23620.296875}
Milestones: [2, 4, 6]
```

```
    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   38.8 K   [16, 10]     [16, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 38.8 K

train_model result: {'val_loss': 23686.23828125, 'hp_metric': 23686.23828125}
Milestones: [32, 64, 96]



    Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   2.5 M   [1024, 10]   [1024, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 2.5 M

train_model result: {'val_loss': 23590.0078125, 'hp_metric': 23590.0078125}
Milestones: [32, 64, 96]



    Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

 0  layers   Sequential      637   train   1.5 M   [2048, 10]   [2048, 1]
```

```
Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 1.5 M

train_model result: {'val_loss': 24738.689453125, 'hp_metric': 24738.689453125}
Milestones: [4, 8, 12]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   191 K   [256, 10]    [256, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 191 K

train_model result: {'val_loss': 24000.283203125, 'hp_metric': 24000.283203125}
Milestones: [4, 8, 12]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   382 K   [512, 10]    [512, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 382 K
```

```
train_model result: {'val_loss': 24577.533203125, 'hp_metric': 24577.533203125}
Milestones: [8, 16, 24]



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential      637   train   23.9 K   [32, 10]     [32, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 23.9 K

train_model result: {'val_loss': 24449.7421875, 'hp_metric': 24449.7421875}
Milestones: [8, 16, 24]



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

train_model result: {'val_loss': 24279.90625, 'hp_metric': 24279.90625}
Milestones: [16, 32, 48]
```

```
     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   2.5 M   [1024, 10]   [1024, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 2.5 M

train_model result: {'val_loss': 24030.890625, 'hp_metric': 24030.890625}
spotpython tuning: 23590.0078125 [----------] 0.86%. Success rate: 0.00%

Milestones: [2, 4, 6]



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   38.8 K   [16, 10]     [16, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 38.8 K

train_model result: {'val_loss': 24261.85546875, 'hp_metric': 24261.85546875}
spotpython tuning: 23590.0078125 [----------] 1.98%. Success rate: 0.00%

Milestones: [32, 64, 96]
```

```
    Name    Type        Params  Mode   FLOPs    In sizes  Out sizes

 0  layers  Sequential   1.8 K  train  5.0 M  [2048, 10]  [2048, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 5.0 M

train_model result: {'val_loss': 24037.138671875, 'hp_metric': 24037.138671875}
spotpython tuning: 23590.0078125 [----------] 2.96%. Success rate: 0.00%

Milestones: [16, 32, 48]



    Name    Type        Params  Mode   FLOPs   In sizes  Out sizes

 0  layers  Sequential   1.8 K  train  1.2 M  [512, 10]   [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20727.541015625, 'hp_metric': 20727.541015625}
spotpython tuning: 20727.541015625 [----------] 3.73%. Success rate: 25.00%

Milestones: [16, 32, 48]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 18110.35546875, 'hp_metric': 18110.35546875}
spotpython tuning: 18110.35546875 [----------] 4.85%. Success rate: 40.00%

Milestones: [16, 32, 48]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 19513.134765625, 'hp_metric': 19513.134765625}
spotpython tuning: 18110.35546875 [#---------] 6.15%. Success rate: 33.33%

Milestones: [32, 64, 96]
```

```
    Name    Type        Params  Mode   FLOPs    In sizes  Out sizes

 0  layers  Sequential   1.8 K  train  5.0 M  [2048, 10]  [2048, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 5.0 M

train_model result: {'val_loss': 24598.349609375, 'hp_metric': 24598.349609375}
spotpython tuning: 18110.35546875 [#---------] 7.70%. Success rate: 28.57%

Milestones: [32, 64, 96]



    Name    Type        Params  Mode   FLOPs   In sizes  Out sizes

 0  layers  Sequential   1.8 K  train  1.2 M  [512, 10]   [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20484.203125, 'hp_metric': 20484.203125}
spotpython tuning: 18110.35546875 [#---------] 8.76%. Success rate: 25.00%

Milestones: [16, 32, 48]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 23700.240234375, 'hp_metric': 23700.240234375}
spotpython tuning: 18110.35546875 [#---------] 10.18%. Success rate: 22.22%

Using spacefilling design as fallback.
Milestones: [32, 64, 96]



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

train_model result: {'val_loss': 24304.251953125, 'hp_metric': 24304.251953125}
spotpython tuning: 18110.35546875 [#---------] 11.73%. Success rate: 20.00%

Milestones: [16, 32, 48]
```

```
    Name    Type        Params  Mode   FLOPs    In sizes  Out sizes

 0  layers  Sequential   1.8 K  train  2.5 M  [1024, 10]  [1024, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 2.5 M

train_model result: {'val_loss': 22364.47265625, 'hp_metric': 22364.47265625}
spotpython tuning: 18110.35546875 [#---------] 13.00%. Success rate: 18.18%

Milestones: [16, 32, 48]



    Name    Type        Params  Mode   FLOPs   In sizes  Out sizes

 0  layers  Sequential   1.8 K  train  1.2 M  [512, 10]   [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 23425.896484375, 'hp_metric': 23425.896484375}
spotpython tuning: 18110.35546875 [#---------] 14.37%. Success rate: 16.67%

Milestones: [16, 32, 48]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   621 K   [256, 10]    [256, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 621 K

train_model result: {'val_loss': 21259.501953125, 'hp_metric': 21259.501953125}
spotpython tuning: 18110.35546875 [##--------] 15.58%. Success rate: 15.38%

Milestones: [16, 32, 48]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20576.791015625, 'hp_metric': 20576.791015625}
spotpython tuning: 18110.35546875 [##--------] 17.00%. Success rate: 14.29%

Milestones: [16, 32, 48]
```

```
    Name    Type        Params  Mode   FLOPs   In sizes  Out sizes

 0  layers  Sequential   1.8 K  train  1.2 M  [512, 10]   [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 22856.45703125, 'hp_metric': 22856.45703125}
spotpython tuning: 18110.35546875 [##--------] 18.34%. Success rate: 13.33%

Milestones: [16, 32, 48]



    Name    Type        Params  Mode   FLOPs   In sizes  Out sizes

 0  layers  Sequential   1.8 K  train  1.2 M  [512, 10]   [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 21497.861328125, 'hp_metric': 21497.861328125}
spotpython tuning: 18110.35546875 [##--------] 19.68%. Success rate: 12.50%

Milestones: [16, 32, 48]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 22453.681640625, 'hp_metric': 22453.681640625}
spotpython tuning: 18110.35546875 [##--------] 20.92%. Success rate: 11.76%

Milestones: [16, 32, 48]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   382 K   [512, 10]    [512, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 382 K

train_model result: {'val_loss': 24528.6328125, 'hp_metric': 24528.6328125}
spotpython tuning: 18110.35546875 [##--------] 22.12%. Success rate: 11.11%

Milestones: [16, 32, 48]
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 23348.65625, 'hp_metric': 23348.65625}
spotpython tuning: 18110.35546875 [##--------] 23.46%. Success rate: 10.53%

Milestones: [16, 32, 48]



    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 21292.18359375, 'hp_metric': 21292.18359375}
spotpython tuning: 18110.35546875 [###-------] 25.01%. Success rate: 10.00%

Using spacefilling design as fallback.
Milestones: [8, 16, 24]
```

```
     Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

  0  layers   Sequential      637   train   765 K   [1024, 10]   [1024, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 765 K

train_model result: {'val_loss': 24059.1328125, 'hp_metric': 24059.1328125}
spotpython tuning: 18110.35546875 [###-------] 26.47%. Success rate: 9.52%

Milestones: [16, 32, 48]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20854.2578125, 'hp_metric': 20854.2578125}
spotpython tuning: 18110.35546875 [###-------] 28.07%. Success rate: 9.09%

Milestones: [16, 32, 48]
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 23338.45703125, 'hp_metric': 23338.45703125}
spotpython tuning: 18110.35546875 [###-------] 29.65%. Success rate: 8.70%

Milestones: [16, 32, 48]



    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20862.421875, 'hp_metric': 20862.421875}
spotpython tuning: 18110.35546875 [###-------] 31.07%. Success rate: 8.33%

Milestones: [16, 32, 48]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 22772.486328125, 'hp_metric': 22772.486328125}
spotpython tuning: 18110.35546875 [###-------] 32.38%. Success rate: 8.00%

Using spacefilling design as fallback.
Milestones: [4, 8, 12]



     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   95.7 K   [128, 10]    [128, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 95.7 K

train_model result: {'val_loss': 24437.765625, 'hp_metric': 24437.765625}
spotpython tuning: 18110.35546875 [###-------] 33.70%. Success rate: 7.69%

Milestones: [16, 32, 48]
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 21311.6953125, 'hp_metric': 21311.6953125}
spotpython tuning: 18110.35546875 [####------] 35.31%. Success rate: 7.41%

Milestones: [16, 32, 48]



    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 18749.962890625, 'hp_metric': 18749.962890625}
spotpython tuning: 18110.35546875 [####------] 36.89%. Success rate: 7.14%

Milestones: [16, 32, 48]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20281.53515625, 'hp_metric': 20281.53515625}
spotpython tuning: 18110.35546875 [####------] 38.46%. Success rate: 6.90%

Milestones: [16, 32, 48]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20327.638671875, 'hp_metric': 20327.638671875}
spotpython tuning: 18110.35546875 [####------] 40.02%. Success rate: 6.67%

Milestones: [16, 32, 48]
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 21904.189453125, 'hp_metric': 21904.189453125}
spotpython tuning: 18110.35546875 [####------] 41.60%. Success rate: 6.45%

Milestones: [16, 32, 48]



    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20986.640625, 'hp_metric': 20986.640625}
spotpython tuning: 18110.35546875 [####------] 43.15%. Success rate: 6.25%

Milestones: [16, 32, 48]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 23056.330078125, 'hp_metric': 23056.330078125}
spotpython tuning: 18110.35546875 [####------] 44.70%. Success rate: 6.06%

Milestones: [16, 32, 48]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 18906.537109375, 'hp_metric': 18906.537109375}
spotpython tuning: 18110.35546875 [#####-----] 46.27%. Success rate: 5.88%

Milestones: [16, 32, 48]
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 17919.68359375, 'hp_metric': 17919.68359375}
spotpython tuning: 17919.68359375 [#####-----] 47.83%. Success rate: 8.57%

Using spacefilling design as fallback.
Milestones: [32, 64, 96]



    Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   77.7 K   [32, 10]     [32, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 77.7 K

train_model result: {'val_loss': 24016.23046875, 'hp_metric': 24016.23046875}
spotpython tuning: 17919.68359375 [#####-----] 49.47%. Success rate: 8.33%

Milestones: [4, 8, 12]
```

```
     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

train_model result: {'val_loss': 23703.373046875, 'hp_metric': 23703.373046875}
spotpython tuning: 17919.68359375 [#####-----] 50.90%. Success rate: 8.11%

Milestones: [16, 32, 48]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 21716.666015625, 'hp_metric': 21716.666015625}
spotpython tuning: 17919.68359375 [#####-----] 52.22%. Success rate: 7.89%

Milestones: [16, 32, 48]
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20698.615234375, 'hp_metric': 20698.615234375}
spotpython tuning: 17919.68359375 [#####-----] 53.54%. Success rate: 7.69%

Using spacefilling design as fallback.
Milestones: [2, 4, 6]



    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   310 K   [128, 10]    [128, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 310 K

train_model result: {'val_loss': 24246.34375, 'hp_metric': 24246.34375}
spotpython tuning: 17919.68359375 [#####-----] 54.84%. Success rate: 7.50%

Milestones: [16, 32, 48]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 24258.3203125, 'hp_metric': 24258.3203125}
spotpython tuning: 17919.68359375 [######----] 56.22%. Success rate: 7.32%

Milestones: [16, 32, 48]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 24197.3515625, 'hp_metric': 24197.3515625}
spotpython tuning: 17919.68359375 [######----] 57.74%. Success rate: 7.14%

Milestones: [4, 8, 12]
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   310 K   [128, 10]    [128, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 310 K

train_model result: {'val_loss': 23864.40625, 'hp_metric': 23864.40625}
spotpython tuning: 17919.68359375 [######----] 59.22%. Success rate: 6.98%

Milestones: [16, 32, 48]



    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20694.44140625, 'hp_metric': 20694.44140625}
spotpython tuning: 17919.68359375 [######----] 60.83%. Success rate: 6.82%

Milestones: [16, 32, 48]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 22845.552734375, 'hp_metric': 22845.552734375}
spotpython tuning: 17919.68359375 [######----] 62.24%. Success rate: 6.67%

Milestones: [32, 64, 96]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 22553.849609375, 'hp_metric': 22553.849609375}
spotpython tuning: 17919.68359375 [######----] 63.77%. Success rate: 6.52%

Milestones: [8, 16, 24]
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 21646.568359375, 'hp_metric': 21646.568359375}
spotpython tuning: 17919.68359375 [#######---] 65.12%. Success rate: 6.38%

Milestones: [16, 32, 48]



    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

train_model result: {'val_loss': 24010.759765625, 'hp_metric': 24010.759765625}
spotpython tuning: 17919.68359375 [#######---] 66.54%. Success rate: 6.25%

Using spacefilling design as fallback.
Milestones: [8, 16, 24]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   191 K   [256, 10]    [256, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 191 K

train_model result: {'val_loss': 23867.04296875, 'hp_metric': 23867.04296875}
spotpython tuning: 17919.68359375 [#######---] 67.83%. Success rate: 6.12%

Milestones: [16, 32, 48]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 21343.1796875, 'hp_metric': 21343.1796875}
spotpython tuning: 17919.68359375 [#######---] 69.32%. Success rate: 6.00%

Milestones: [16, 32, 48]
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   621 K   [256, 10]    [256, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 621 K

train_model result: {'val_loss': 21577.259765625, 'hp_metric': 21577.259765625}
spotpython tuning: 17919.68359375 [#######---] 70.78%. Success rate: 5.88%

Milestones: [16, 32, 48]



    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20096.72265625, 'hp_metric': 20096.72265625}
spotpython tuning: 17919.68359375 [#######---] 72.19%. Success rate: 5.77%

Milestones: [16, 32, 48]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 22248.748046875, 'hp_metric': 22248.748046875}
spotpython tuning: 17919.68359375 [#######---] 73.61%. Success rate: 5.66%

Using spacefilling design as fallback.
Milestones: [4, 8, 12]



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential      637   train   23.9 K   [32, 10]     [32, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 23.9 K

train_model result: {'val_loss': 24082.240234375, 'hp_metric': 24082.240234375}
spotpython tuning: 17919.68359375 [#######---] 74.99%. Success rate: 5.56%

Milestones: [16, 32, 48]
```

```
    Name    Type        Params  Mode   FLOPs   In sizes  Out sizes

 0  layers  Sequential   1.8 K  train  1.2 M  [512, 10]   [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 19655.291015625, 'hp_metric': 19655.291015625}
spotpython tuning: 17919.68359375 [########--] 76.61%. Success rate: 5.45%

Milestones: [16, 32, 48]



    Name    Type        Params  Mode   FLOPs   In sizes  Out sizes

 0  layers  Sequential   1.8 K  train  1.2 M  [512, 10]   [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 19942.63671875, 'hp_metric': 19942.63671875}
spotpython tuning: 17919.68359375 [########--] 78.01%. Success rate: 5.36%

Milestones: [32, 64, 96]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 24305.08984375, 'hp_metric': 24305.08984375}
spotpython tuning: 17919.68359375 [########--] 79.42%. Success rate: 5.26%

Milestones: [8, 16, 24]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   621 K   [256, 10]    [256, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 621 K

train_model result: {'val_loss': 22457.099609375, 'hp_metric': 22457.099609375}
spotpython tuning: 17919.68359375 [########--] 81.03%. Success rate: 5.17%

Milestones: [16, 32, 48]
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20557.021484375, 'hp_metric': 20557.021484375}
spotpython tuning: 17919.68359375 [########--] 82.64%. Success rate: 5.08%

Using spacefilling design as fallback.
Milestones: [8, 16, 24]



    Name     Type         Params   Mode    FLOPs     In sizes   Out sizes

 0  layers   Sequential      637   train   1.5 M   [2048, 10]   [2048, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 1.5 M

train_model result: {'val_loss': 23803.66015625, 'hp_metric': 23803.66015625}
spotpython tuning: 17919.68359375 [########--] 83.87%. Success rate: 5.00%

Milestones: [16, 32, 48]
```

```
     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 18272.107421875, 'hp_metric': 18272.107421875}
spotpython tuning: 17919.68359375 [#########-] 85.50%. Success rate: 4.92%

Milestones: [16, 32, 48]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 21429.08203125, 'hp_metric': 21429.08203125}
spotpython tuning: 17919.68359375 [#########-] 87.12%. Success rate: 4.84%

Using spacefilling design as fallback.
Milestones: [32, 64, 96]
```

```
    Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

train_model result: {'val_loss': 24519.021484375, 'hp_metric': 24519.021484375}
spotpython tuning: 17919.68359375 [#########-] 88.84%. Success rate: 4.76%

Milestones: [16, 32, 48]



    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 20769.740234375, 'hp_metric': 20769.740234375}
spotpython tuning: 17919.68359375 [#########-] 90.43%. Success rate: 4.69%

Using spacefilling design as fallback.
Milestones: [4, 8, 12]
```

```
     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential      637   train   23.9 K   [32, 10]     [32, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 23.9 K

train_model result: {'val_loss': 25010.33203125, 'hp_metric': 25010.33203125}
spotpython tuning: 17919.68359375 [#########-] 91.79%. Success rate: 4.62%

Milestones: [8, 16, 24]



     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   621 K   [256, 10]    [256, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 621 K

train_model result: {'val_loss': 21233.41796875, 'hp_metric': 21233.41796875}
spotpython tuning: 17919.68359375 [#########-] 93.13%. Success rate: 4.55%

Using spacefilling design as fallback.
Milestones: [8, 16, 24]
```

```
    Name    Type        Params  Mode   FLOPs    In sizes  Out sizes

 0  layers  Sequential     637  train  765 K  [1024, 10]  [1024, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 765 K

train_model result: {'val_loss': 23673.33984375, 'hp_metric': 23673.33984375}
spotpython tuning: 17919.68359375 [##########] 97.59%. Success rate: 4.48%

Milestones: [16, 32, 48]



    Name    Type        Params  Mode   FLOPs   In sizes  Out sizes

 0  layers  Sequential   1.8 K  train  1.2 M  [512, 10]   [512, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 21687.8203125, 'hp_metric': 21687.8203125}
spotpython tuning: 17919.68359375 [##########] 99.17%. Success rate: 4.41%

Milestones: [16, 32, 48]
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    1.8 K   train   1.2 M   [512, 10]    [512, 1]


Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 1.2 M

train_model result: {'val_loss': 23813.521484375, 'hp_metric': 23813.521484375}
spotpython tuning: 17919.68359375 [##########] 100.00%. Success rate: 4.35% Done...

Experiment saved to 606-user-resnet_res.pkl
```

58.1. Looking at the Results

58.1.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress()
```

58.1.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

```
| name           | type   | default   |   lower |   upper | tuned              | transform
|----------------|--------|-----------|---------|---------|--------------------
|------------------------|--------------|---------|
| l1             | int    | 3         |     3.0 |     4.0 | 4.0                | transform_power_2_i
| epochs         | int    | 4         |     3.0 |     7.0 | 6.0                | transform_power_2_i
| batch_size     | int    | 4         |     4.0 |    11.0 | 9.0                | transform_power_2_i
| act_fn         | factor | ReLU      |     0.0 |     5.0 | LeakyReLU          | None
| optimizer      | factor | SGD       |     0.0 |     2.0 | Adam               | None
| dropout_prob   | float  | 0.01      |     0.0 |   0.025 | 0.025              | None
| lr_mult        | float  | 1.0       |     0.1 |    20.0 | 18.234657445315417 | None
| patience       | int    | 2         |     2.0 |     3.0 | 3.0                | transform_power_2_i
| initialization | factor | Default   |     0.0 |     4.0 | Default            | None
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=1.0)
```

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
l1:  0.001
epochs:  0.018313829029560563
batch_size:  0.01024668447731283
act_fn:  1.2403398780964416
optimizer:  0.001
dropout_prob:  0.0011315992259617268
lr_mult:  0.01621200084778313
patience:  0.144890718188217
initialization:  100.0
```

Prediction Surface
Prediction
24650
24600
24550
24500
24450
24400
24350
24300
act_fn
patience
Prediction Error Surface
Error
700
650
600
550
500
450
Prediction Contour
24645
24600
24555
24510
24465
24420
24375
24330
24285
Error Contour
720
675
630
585
540
495
450
405

58.1.3. Get the Tuned Architecture

```
import pprint
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner)
pprint.pprint(config)
```

```
{'act_fn': LeakyReLU(),
 'batch_size': 512,
 'dropout_prob': 0.025,
 'epochs': 64,
 'initialization': 'Default',
 'l1': 16,
 'lr_mult': 18.234657445315417,
```

```
 'optimizer': 'Adam',
 'patience': 8}
```

58.2. Details of the User-Specified ResNet Model

The specification of a user model requires three files:

- `my_resnet.py`: the Python file containing the user specified ResNet model
- `my_hyperdict.py`: the Python file for loading the hyperparameter dictionary `my_hyperdict.json` for the user specified ResNet model
- `my_hyperdict.json`: the JSON file containing the hyperparameter dictionary for the user specified ResNet model

58.2.1. `my_resnet.py`

```python
import lightning as L
import torch
from torch import nn
from spotpython.hyperparameters.optimizer import optimizer_handler
import torchmetrics.functional.regression
import torch.optim as optim

class ResidualBlock(nn.Module):
    def __init__(self, input_dim, output_dim, act_fn, dropout_prob):
        super(ResidualBlock, self).__init__()
        self.fc1 = nn.Linear(input_dim, output_dim)
        self.bn1 = nn.BatchNorm1d(output_dim)
        self.ln1 = nn.LayerNorm(output_dim)
        self.fc2 = nn.Linear(output_dim, output_dim)
        self.bn2 = nn.BatchNorm1d(output_dim)
        self.ln2 = nn.LayerNorm(output_dim)
        self.act_fn = act_fn
        self.dropout = nn.Dropout(dropout_prob)
        self.shortcut = nn.Sequential()

        if input_dim != output_dim:
            self.shortcut = nn.Sequential(
                nn.Linear(input_dim, output_dim),
                nn.BatchNorm1d(output_dim)
            )
```

```
    def forward(self, x):
        identity = self.shortcut(x)

        out = self.fc1(x)
        out = self.bn1(out)
        out = self.ln1(out)
        out = self.act_fn(out)
        out = self.dropout(out)
        out = self.fc2(out)
        out = self.bn2(out)
        out = self.ln2(out)
        out += identity  # Residual connection
        out = self.act_fn(out)
        return out

class MyResNet(L.LightningModule):
    def __init__(
        self,
        l1: int,
        epochs: int,
        batch_size: int,
        initialization: str,
        act_fn: nn.Module,
        optimizer: str,
        dropout_prob: float,
        lr_mult: float,
        patience: int,
        _L_in: int,
        _L_out: int,
        _torchmetric: str,
    ):
        super().__init__()
        self._L_in = _L_in
        self._L_out = _L_out
        if _torchmetric is None:
            _torchmetric = "mean_squared_error"
        self._torchmetric = _torchmetric
        self.metric = getattr(torchmetrics.functional.regression, _torchmetric)
        self.save_hyperparameters(ignore=["_L_in", "_L_out", "_torchmetric"])
        self.example_input_array = torch.zeros((batch_size, self._L_in))

        if self.hparams.l1 < 4:
            raise ValueError("l1 must be at least 4")
```

```
        # Get hidden sizes
        hidden_sizes = self._get_hidden_sizes()
        layer_sizes = [self._L_in] + hidden_sizes

        # Construct the layers with Residual Blocks and Linear Layer at the end
        layers = []
        for i in range(len(layer_sizes) - 1):
            layers.append(
                ResidualBlock(
                    layer_sizes[i],
                    layer_sizes[i + 1],
                    self.hparams.act_fn,
                    self.hparams.dropout_prob
                )
            )
        layers.append(nn.Linear(layer_sizes[-1], self._L_out))

        self.layers = nn.Sequential(*layers)

        # Initialization (Xavier, Kaiming, or Default)
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            if self.hparams.initialization == "xavier_uniform":
                nn.init.xavier_uniform_(module.weight)
            elif self.hparams.initialization == "xavier_normal":
                nn.init.xavier_normal_(module.weight)
            elif self.hparams.initialization == "kaiming_uniform":
                nn.init.kaiming_uniform_(module.weight)
            elif self.hparams.initialization == "kaiming_normal":
                nn.init.kaiming_normal_(module.weight)
            else: # "Default"
                nn.init.uniform_(module.weight)
            if module.bias is not None:
                nn.init.zeros_(module.bias)

    def _generate_div2_list(self, n, n_min) -> list:
        result = []
        current = n
        repeats = 1
        max_repeats = 4
        while current >= n_min:
            result.extend([current] * min(repeats, max_repeats))
```

```
            current = current // 2
            repeats = repeats + 1
        return result

    def _get_hidden_sizes(self):
        n_low = max(2, int(self._L_in / 4))  # Ensure minimum reasonable size
        n_high = max(self.hparams.l1, 2 * n_low)
        hidden_sizes = self._generate_div2_list(n_high, n_low)
        return hidden_sizes

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.layers(x)
        return x

    def _calculate_loss(self, batch):
        x, y = batch
        y = y.view(len(y), 1)
        y_hat = self(x)
        loss = self.metric(y_hat, y)
        return loss

    def training_step(self, batch: tuple) -> torch.Tensor:
        val_loss = self._calculate_loss(batch)
        return val_loss

    def validation_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) ->
        val_loss = self._calculate_loss(batch)
        self.log("val_loss", val_loss, prog_bar=prog_bar)
        self.log("hp_metric", val_loss, prog_bar=prog_bar)
        return val_loss

    def test_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch
        val_loss = self._calculate_loss(batch)
        self.log("val_loss", val_loss, prog_bar=prog_bar)
        self.log("hp_metric", val_loss, prog_bar=prog_bar)
        return val_loss

    def predict_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> to
        x, y = batch
        yhat = self(x)
        y = y.view(len(y), 1)
        yhat = yhat.view(len(yhat), 1)
        return (x, y, yhat)
```

```
    def configure_optimizers(self):
        optimizer = optimizer_handler(
            optimizer_name=self.hparams.optimizer,
            params=self.parameters(),
            lr_mult=self.hparams.lr_mult
        )

        # Dynamic creation of milestones based on the number of epochs.
        num_milestones = 3  # Number of milestones to divide the epochs
        milestones = [int(self.hparams.epochs / (num_milestones + 1) * (i + 1)) for i in range(num_m

        # Print milestones for debug purposes
        print(f"Milestones: {milestones}")

        # Create MultiStepLR scheduler with dynamic milestones and learning rate multiplier.
        scheduler = optim.lr_scheduler.MultiStepLR(
            optimizer,
            milestones=milestones,
            gamma=0.1  # Decay factor
        )

        # Learning rate scheduler configuration
        lr_scheduler_config = {
            "scheduler": scheduler,
            "interval": "epoch",  # Adjust learning rate per epoch
            "frequency": 1,      # Apply the scheduler at every epoch
        }

        return {"optimizer": optimizer, "lr_scheduler": lr_scheduler_config}
```

58.2.2. my_hyperdict.py

```
import json
from spotpython.data import base
import pathlib


class MyHyperDict(base.FileConfig):
    """User specified hyperparameter dictionary.

    This class extends the FileConfig class to provide a dictionary for storing hyperparameters.
```

```
    Attributes:
        filename (str):
            The name of the file where the hyperparameters are stored.
    """

    def __init__(
        self,
        filename: str = "my_hyper_dict.json",
        directory: None = None,
    ) -> None:
        super().__init__(filename=filename, directory=directory)
        self.filename = filename
        self.directory = directory
        self.hyper_dict = self.load()

    @property
    def path(self):
        if self.directory:
            return pathlib.Path(self.directory).joinpath(self.filename)
        return pathlib.Path(__file__).parent.joinpath(self.filename)

    def load(self) -> dict:
        """Load the hyperparameters from the file.

        Returns:
            dict: A dictionary containing the hyperparameters.

        Examples:
            # Assume the user specified file `my_hyper_dict.json` is in the `./hyperdi
            >>> user_lhd = MyHyperDict(filename='my_hyper_dict.json', directory='./hyp
        """
        with open(self.path, "r") as f:
            d = json.load(f)
        return d
```

58.2.3. my_hyperdict.json

```
"MyResNet": {
        "l1": {
            "type": "int",
            "default": 3,
            "transform": "transform_power_2_int",
```

```
            "lower": 3,
            "upper": 10
        },
        "epochs": {
            "type": "int",
            "default": 4,
            "transform": "transform_power_2_int",
            "lower": 4,
            "upper": 9
        },
        "batch_size": {
            "type": "int",
            "default": 4,
            "transform": "transform_power_2_int",
            "lower": 1,
            "upper": 6
        },
        "act_fn": {
            "levels": [
                "Sigmoid",
                "Tanh",
                "ReLU",
                "LeakyReLU",
                "ELU",
                "Swish"
            ],
            "type": "factor",
            "default": "ReLU",
            "transform": "None",
            "class_name": "spotpython.torch.activation",
            "core_model_parameter_type": "instance()",
            "lower": 0,
            "upper": 5
        },
        "optimizer": {
            "levels": [
                "Adadelta",
                "Adagrad",
                "Adam",
                "AdamW",
                "SparseAdam",
                "Adamax",
                "ASGD",
                "NAdam",
```

```
                "RAdam",
                "RMSprop",
                "Rprop",
                "SGD"
            ],
            "type": "factor",
            "default": "SGD",
            "transform": "None",
            "class_name": "torch.optim",
            "core_model_parameter_type": "str",
            "lower": 0,
            "upper": 11
        },
        "dropout_prob": {
            "type": "float",
            "default": 0.01,
            "transform": "None",
            "lower": 0.0,
            "upper": 0.25
        },
        "lr_mult": {
            "type": "float",
            "default": 1.0,
            "transform": "None",
            "lower": 0.1,
            "upper": 10.0
        },
        "patience": {
            "type": "int",
            "default": 2,
            "transform": "transform_power_2_int",
            "lower": 2,
            "upper": 6
        },
        "initialization": {
            "levels": [
                "Default",
                "kaiming_uniform",
                "kaiming_normal",
                "xavier_uniform",
                "xavier_normal"
            ],
            "type": "factor",
            "default": "Default",
```

```
            "transform": "None",
            "core_model_parameter_type": "str",
            "lower": 0,
            "upper": 4
        }
    }
```

58.3. Summary

This section presented an introduction to the basic setup of hyperparameter tuning with `spotpython` and `PyTorch` Lightning using a ResNet model for the Diabetes data set.

59. Hyperparameter Tuning with spotpython and PyTorch Lightning Using a CondNet Model

- We use the `Diabetes` dataset to illustrate the hyperparameter tuning process of a `CondNet` model using the `spotpython` package.
- The CondNet model is a conditional neural network that can be used to model conditional distributions [LINK].

Note, the `divergence_threshold` is set to 5,000, which is based on some pre-experiments with the `Diabetes` data set.

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_control_init)
from spotpython.utils.eda import print_exp_table
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename
from math import inf
from spotpython.hyperparameters.values import set_hyperparameter

PREFIX="CondNet_01"

data_set = Diabetes()
input_dim = 10
output_dim = 1
cond_dim = 2

fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNCondNetRegressor",
    hyperdict=LightHyperDict,
    divergence_threshold=5_000,
```

```
    _L_in=input_dim - cond_dim,
    _L_out=1,
    _L_cond=cond_dim,)

fun = HyperLight().fun


set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,7])
set_hyperparameter(fun_control, "batch_size", [4,5])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])
set_hyperparameter(fun_control, "lr_mult", [0.1, 20.0])

design_control = design_control_init(init_size=10)

print_exp_table(fun_control)
```

```
module_name: light
submodule_name: regression
model_name: NNCondNetRegressor
| name           | type    | default    |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------
|
| l1             | int     | 3          |     3   |   4     | transform_power_2_int |
| epochs         | int     | 4          |     3   |   7     | transform_power_2_int |
| batch_size     | int     | 4          |     4   |   5     | transform_power_2_int |
| act_fn         | factor  | ReLU       |     0   |   5     | None                  |
| optimizer      | factor  | SGD        |     0   |   2     | None                  |
| dropout_prob   | float   | 0.01       |     0   |   0.025 | None                  |
| lr_mult        | float   | 1.0        |     0.1 |  20     | None                  |
| patience       | int     | 2          |     2   |   3     | transform_power_2_int |
| batch_norm     | factor  | 0          |     0   |   1     | None                  |
| initialization | factor  | Default    |     0   |   4     | None                  |
```

```
spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
res = spot_tuner.run()
```

```
     Name         Type                Params   Mode     FLOPs             In sizes   Out
```

```
  0   cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32, 16]
  1   layers       Sequential            587   train   34.2 K             [32, 16]     [32, 1]



Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 24266.927734375, 'hp_metric': 24266.927734375}



     Name         Type               Params   Mode    FLOPs             In sizes   Out sizes

  0   cond_layer   ConditionalLayer       96   train   2.6 K   [[16, 8], [16, 2]]     [16, 8]
  1   layers       Sequential            153   train   4.2 K              [16, 8]     [16, 1]



Trainable params: 249
Non-trainable params: 0
Total params: 249
Total estimated model params size (MB): 0
Modules in train mode: 18
Modules in eval mode: 0
Total FLOPs: 6.7 K

train_model result: {'val_loss': 24001.298828125, 'hp_metric': 24001.298828125}



     Name         Type               Params   Mode     FLOPs             In sizes   Out sizes

  0   cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32, 16]
  1   layers       Sequential            691   train   34.2 K             [32, 16]     [32, 1]
```

```
Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 21142.806640625, 'hp_metric': 21142.806640625}



     Name         Type               Params   Mode    FLOPs             In sizes   Out s

  0  cond_layer   ConditionalLayer       96   train   5.1 K   [[32, 8], [32, 2]]     [32
  1  layers       Sequential            197   train   8.3 K              [32, 8]     [32



Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 24029.455078125, 'hp_metric': 24029.455078125}



     Name         Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer   ConditionalLayer      192   train    5.1 K   [[16, 8], [16, 2]]    [16
  1  layers       Sequential            691   train   17.1 K             [16, 16]     [1



Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 22.2 K
```

```
train_model result: {'val_loss': 23382.876953125, 'hp_metric': 23382.876953125}


     Name        Type               Params  Mode    FLOPs            In sizes   Out sizes

  0  cond_layer  ConditionalLayer      192  train   5.1 K  [[16, 8], [16, 2]]    [16, 16]
  1  layers      Sequential           587  train  17.1 K            [16, 16]     [16, 1]



Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 22.2 K

train_model result: {'val_loss': 24010.201171875, 'hp_metric': 24010.201171875}




     Name        Type               Params  Mode    FLOPs            In sizes   Out sizes

  0  cond_layer  ConditionalLayer       96  train   5.1 K  [[32, 8], [32, 2]]     [32, 8]
  1  layers      Sequential           153  train   8.3 K             [32, 8]     [32, 1]



Trainable params: 249
Non-trainable params: 0
Total params: 249
Total estimated model params size (MB): 0
Modules in train mode: 18
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 23698.623046875, 'hp_metric': 23698.623046875}
```

```
    Name        Type               Params   Mode    FLOPs            In sizes   Out s

 0  cond_layer  ConditionalLayer      96   train   2.6 K   [[16, 8], [16, 2]]     [16
 1  layers      Sequential           197   train   4.2 K              [16, 8]     [16



Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.7 K

train_model result: {'val_loss': 23921.322265625, 'hp_metric': 23921.322265625}



    Name        Type               Params   Mode    FLOPs            In sizes   Out s

 0  cond_layer  ConditionalLayer      96   train   2.6 K   [[16, 8], [16, 2]]     [16
 1  layers      Sequential           153   train   4.2 K              [16, 8]     [16



Trainable params: 249
Non-trainable params: 0
Total params: 249
Total estimated model params size (MB): 0
Modules in train mode: 18
Modules in eval mode: 0
Total FLOPs: 6.7 K

train_model result: {'val_loss': 23699.216796875, 'hp_metric': 23699.216796875}



    Name        Type               Params   Mode     FLOPs            In sizes   Out

 0  cond_layer  ConditionalLayer     192   train   10.2 K   [[32, 8], [32, 2]]    [32
 1  layers      Sequential           691   train   34.2 K             [32, 16]     [3
```

```
Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 24038.666015625, 'hp_metric': 24038.666015625}


     Name         Type               Params   Mode     FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32, 16]
  1  layers       Sequential            691   train   34.2 K             [32, 16]     [32, 1]


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 23752.283203125, 'hp_metric': 23752.283203125}
spotpython tuning: 21142.806640625 [----------] 0.57%. Success rate: 0.00%


     Name         Type               Params   Mode     FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32, 16]
  1  layers       Sequential            691   train   34.2 K             [32, 16]     [32, 1]


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K
```

```
train_model result: {'val_loss': 22542.3828125, 'hp_metric': 22542.3828125}
spotpython tuning: 21142.806640625 [----------] 1.12%. Success rate: 0.00%


     Name        Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer  ConditionalLayer      192   train    5.1 K   [[16, 8], [16, 2]]    [16
  1  layers      Sequential            691   train   17.1 K             [16, 16]     [1



Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 22.2 K

train_model result: {'val_loss': 21875.419921875, 'hp_metric': 21875.419921875}
spotpython tuning: 21142.806640625 [----------] 2.15%. Success rate: 0.00%


     Name        Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer  ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32
  1  layers      Sequential            691   train   34.2 K             [32, 16]     [3



Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 22236.15234375, 'hp_metric': 22236.15234375}
spotpython tuning: 21142.806640625 [----------] 3.14%. Success rate: 0.00%
```

```
    Name        Type               Params  Mode    FLOPs            In sizes   Out sizes

 0  cond_layer  ConditionalLayer     192  train  10.2 K  [[32, 8], [32, 2]]   [32, 16]
 1  layers      Sequential           587  train  34.2 K            [32, 16]    [32, 1]


Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 16283.1318359375, 'hp_metric': 16283.1318359375}
spotpython tuning: 16283.1318359375 [----------] 4.09%. Success rate: 20.00%


    Name        Type               Params  Mode   FLOPs            In sizes   Out sizes

 0  cond_layer  ConditionalLayer      96  train  5.1 K  [[32, 8], [32, 2]]    [32, 8]
 1  layers      Sequential           197  train  8.3 K             [32, 8]    [32, 1]


Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 22554.375, 'hp_metric': 22554.375}
spotpython tuning: 16283.1318359375 [----------] 4.75%. Success rate: 16.67%


    Name        Type               Params  Mode    FLOPs            In sizes   Out sizes

 0  cond_layer  ConditionalLayer     192  train  10.2 K  [[32, 8], [32, 2]]   [32, 16]
 1  layers      Sequential           587  train  34.2 K            [32, 16]    [32, 1]
```

```
Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 20658.12890625, 'hp_metric': 20658.12890625}
spotpython tuning: 16283.1318359375 [#---------] 5.50%. Success rate: 14.29%


     Name         Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer   ConditionalLayer      192   train    5.1 K   [[16, 8], [16, 2]]    [16
  1  layers       Sequential            587   train   17.1 K             [16, 16]     [1


Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 22.2 K

train_model result: {'val_loss': 8588.830078125, 'hp_metric': 8588.830078125}
spotpython tuning: 8588.830078125 [#---------] 6.04%. Success rate: 25.00%


     Name         Type               Params   Mode    FLOPs             In sizes   Out s

  0  cond_layer   ConditionalLayer       96   train   5.1 K   [[32, 8], [32, 2]]     [32
  1  layers       Sequential            197   train   8.3 K              [32, 8]     [32


Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K
```

```
train_model result: {'val_loss': 23619.22265625, 'hp_metric': 23619.22265625}
spotpython tuning: 8588.830078125 [#---------] 6.65%. Success rate: 22.22%


     Name         Type               Params   Mode    FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer       96   train   5.1 K   [[32, 8], [32, 2]]     [32, 8]
  1  layers       Sequential            153   train   8.3 K              [32, 8]     [32, 1]



Trainable params: 249
Non-trainable params: 0
Total params: 249
Total estimated model params size (MB): 0
Modules in train mode: 18
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 23825.615234375, 'hp_metric': 23825.615234375}
spotpython tuning: 8588.830078125 [#---------] 7.14%. Success rate: 20.00%


     Name         Type               Params   Mode    FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer       96   train   5.1 K   [[32, 8], [32, 2]]     [32, 8]
  1  layers       Sequential            197   train   8.3 K              [32, 8]     [32, 1]



Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 23993.693359375, 'hp_metric': 23993.693359375}
spotpython tuning: 8588.830078125 [#---------] 7.87%. Success rate: 18.18%
```

```
     Name         Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32
  1  layers       Sequential           691   train   34.2 K             [32, 16]     [3


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 23921.734375, 'hp_metric': 23921.734375}
spotpython tuning: 8588.830078125 [#---------] 8.66%. Success rate: 16.67%


     Name         Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32
  1  layers       Sequential           691   train   34.2 K             [32, 16]     [3


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 24051.060546875, 'hp_metric': 24051.060546875}
spotpython tuning: 8588.830078125 [#---------] 9.42%. Success rate: 15.38%


     Name         Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32
  1  layers       Sequential           691   train   34.2 K             [32, 16]     [3
```

```
Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 23882.166015625, 'hp_metric': 23882.166015625}
spotpython tuning: 8588.830078125 [#---------] 10.12%. Success rate: 14.29%


     Name         Type               Params   Mode    FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer      96   train   5.1 K   [[32, 8], [32, 2]]     [32, 8]
  1  layers       Sequential           197   train   8.3 K              [32, 8]     [32, 1]


Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 22620.6328125, 'hp_metric': 22620.6328125}
spotpython tuning: 8588.830078125 [#---------] 10.83%. Success rate: 13.33%


     Name         Type               Params   Mode     FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer     192   train   10.2 K   [[32, 8], [32, 2]]    [32, 16]
  1  layers       Sequential           691   train   34.2 K             [32, 16]     [32, 1]


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K
```

```
train_model result: {'val_loss': 24091.826171875, 'hp_metric': 24091.826171875}
spotpython tuning: 8588.830078125 [#---------] 11.49%. Success rate: 12.50%


     Name        Type               Params   Mode    FLOPs             In sizes   Out s

 0   cond_layer  ConditionalLayer      96   train   5.1 K   [[32, 8], [32, 2]]     [32
 1   layers      Sequential           197   train   8.3 K              [32, 8]     [32



Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 24000.14453125, 'hp_metric': 24000.14453125}
spotpython tuning: 8588.830078125 [#---------] 12.14%. Success rate: 11.76%


     Name        Type               Params   Mode     FLOPs             In sizes   Out

 0   cond_layer  ConditionalLayer     192   train    5.1 K   [[16, 8], [16, 2]]    [16
 1   layers      Sequential           691   train   17.1 K             [16, 16]     [1



Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 22.2 K

train_model result: {'val_loss': 23777.0234375, 'hp_metric': 23777.0234375}
spotpython tuning: 8588.830078125 [#---------] 12.97%. Success rate: 11.11%
```

```
    Name        Type               Params  Mode   FLOPs            In sizes   Out sizes

 0  cond_layer  ConditionalLayer      96  train  5.1 K   [[32, 8], [32, 2]]     [32, 8]
 1  layers      Sequential          153  train  8.3 K              [32, 8]     [32, 1]


Trainable params: 249
Non-trainable params: 0
Total params: 249
Total estimated model params size (MB): 0
Modules in train mode: 18
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 23824.72265625, 'hp_metric': 23824.72265625}
spotpython tuning: 8588.830078125 [#---------] 13.69%. Success rate: 10.53%


    Name        Type               Params  Mode    FLOPs            In sizes   Out sizes

 0  cond_layer  ConditionalLayer     192  train   5.1 K   [[16, 8], [16, 2]]    [16, 16]
 1  layers      Sequential          587  train  17.1 K             [16, 16]     [16, 1]


Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 22.2 K

train_model result: {'val_loss': 16348.5556640625, 'hp_metric': 16348.5556640625}
spotpython tuning: 8588.830078125 [##--------] 15.04%. Success rate: 10.00%


    Name        Type               Params  Mode    FLOPs            In sizes   Out sizes

 0  cond_layer  ConditionalLayer     192  train   5.1 K   [[16, 8], [16, 2]]    [16, 16]
 1  layers      Sequential          587  train  17.1 K             [16, 16]     [16, 1]
```

```
Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 22.2 K

train_model result: {'val_loss': 542253.5625, 'hp_metric': 542253.5625}
spotpython tuning: 8588.830078125 [##--------] 15.78%. Success rate: 9.52%


     Name         Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32
  1  layers       Sequential            691   train   34.2 K             [32, 16]     [3


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 21935.23046875, 'hp_metric': 21935.23046875}
spotpython tuning: 8588.830078125 [##--------] 18.34%. Success rate: 9.09%


     Name         Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32
  1  layers       Sequential            691   train   34.2 K             [32, 16]     [3


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K
```

```
train_model result: {'val_loss': 24014.890625, 'hp_metric': 24014.890625}
spotpython tuning: 8588.830078125 [##--------] 20.86%. Success rate: 8.70%


     Name         Type               Params   Mode     FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32, 16]
  1  layers       Sequential            691   train   34.2 K             [32, 16]     [32, 1]



Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 21583.421875, 'hp_metric': 21583.421875}
spotpython tuning: 8588.830078125 [##--------] 23.57%. Success rate: 8.33%


     Name         Type               Params   Mode     FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32, 16]
  1  layers       Sequential            691   train   34.2 K             [32, 16]     [32, 1]



Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 24219.591796875, 'hp_metric': 24219.591796875}
spotpython tuning: 8588.830078125 [###-------] 26.27%. Success rate: 8.00%
```

```
    Name        Type               Params   Mode    FLOPs            In sizes   Out s

 0  cond_layer  ConditionalLayer       96  train   2.6 K  [[16, 8], [16, 2]]    [16
 1  layers      Sequential            197  train   4.2 K            [16, 8]     [16


Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.7 K

train_model result: {'val_loss': 23552.76171875, 'hp_metric': 23552.76171875}
spotpython tuning: 8588.830078125 [###-------] 28.57%. Success rate: 7.69%


    Name        Type               Params   Mode    FLOPs            In sizes   Out s

 0  cond_layer  ConditionalLayer       96  train   5.1 K  [[32, 8], [32, 2]]    [32
 1  layers      Sequential            197  train   8.3 K            [32, 8]     [32


Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 23819.322265625, 'hp_metric': 23819.322265625}
spotpython tuning: 8588.830078125 [###-------] 29.19%. Success rate: 7.41%


    Name        Type               Params   Mode    FLOPs            In sizes   Out s

 0  cond_layer  ConditionalLayer       96  train   2.6 K  [[16, 8], [16, 2]]    [16
 1  layers      Sequential            153  train   4.2 K            [16, 8]     [16
```

```
Trainable params: 249
Non-trainable params: 0
Total params: 249
Total estimated model params size (MB): 0
Modules in train mode: 18
Modules in eval mode: 0
Total FLOPs: 6.7 K

train_model result: {'val_loss': 24035.205078125, 'hp_metric': 24035.205078125}
spotpython tuning: 8588.830078125 [###-------] 30.90%. Success rate: 7.14%


     Name         Type               Params   Mode     FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32, 16]
  1  layers       Sequential            691   train   34.2 K             [32, 16]     [32, 1]


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 24020.1328125, 'hp_metric': 24020.1328125}
spotpython tuning: 8588.830078125 [####------] 35.31%. Success rate: 6.90%


     Name         Type               Params   Mode    FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer       96   train   2.6 K   [[16, 8], [16, 2]]     [16, 8]
  1  layers       Sequential            197   train   4.2 K              [16, 8]     [16, 1]


Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.7 K
```

```
train_model result: {'val_loss': 23895.53125, 'hp_metric': 23895.53125}
spotpython tuning: 8588.830078125 [####------] 37.69%. Success rate: 6.67%



     Name        Type               Params   Mode    FLOPs            In sizes   Out

 0   cond_layer  ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32
 1   layers      Sequential            691   train   34.2 K            [32, 16]     [3



Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 24009.51171875, 'hp_metric': 24009.51171875}
spotpython tuning: 8588.830078125 [####------] 39.87%. Success rate: 6.45%



     Name        Type               Params   Mode    FLOPs            In sizes   Out s

 0   cond_layer  ConditionalLayer       96   train   2.6 K   [[16, 8], [16, 2]]     [16
 1   layers      Sequential            153   train   4.2 K             [16, 8]     [16



Trainable params: 249
Non-trainable params: 0
Total params: 249
Total estimated model params size (MB): 0
Modules in train mode: 18
Modules in eval mode: 0
Total FLOPs: 6.7 K

train_model result: {'val_loss': 24054.033203125, 'hp_metric': 24054.033203125}
spotpython tuning: 8588.830078125 [####------] 41.71%. Success rate: 6.25%
```

```
    Name        Type               Params  Mode   FLOPs            In sizes   Out sizes

 0  cond_layer  ConditionalLayer      96  train  5.1 K  [[32, 8], [32, 2]]    [32, 8]
 1  layers      Sequential          197  train  8.3 K             [32, 8]    [32, 1]


Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 24168.806640625, 'hp_metric': 24168.806640625}
spotpython tuning: 8588.830078125 [####------] 44.34%. Success rate: 6.06%


    Name        Type               Params  Mode   FLOPs            In sizes   Out sizes

 0  cond_layer  ConditionalLayer      96  train  2.6 K  [[16, 8], [16, 2]]    [16, 8]
 1  layers      Sequential          197  train  4.2 K             [16, 8]    [16, 1]


Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.7 K

train_model result: {'val_loss': 24035.931640625, 'hp_metric': 24035.931640625}
spotpython tuning: 8588.830078125 [#####-----] 47.22%. Success rate: 5.88%


    Name        Type               Params  Mode    FLOPs            In sizes   Out sizes

 0  cond_layer  ConditionalLayer     192  train  10.2 K  [[32, 8], [32, 2]]   [32, 16]
 1  layers      Sequential          691  train  34.2 K            [32, 16]    [32, 1]
```

```
Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 23980.1328125, 'hp_metric': 23980.1328125}
spotpython tuning: 8588.830078125 [#####-----] 48.02%. Success rate: 5.71%


     Name         Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32
  1  layers       Sequential            691   train   34.2 K             [32, 16]     [3


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 24006.662109375, 'hp_metric': 24006.662109375}
spotpython tuning: 8588.830078125 [#####-----] 52.14%. Success rate: 5.56%


     Name         Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer   ConditionalLayer      192   train    5.1 K   [[16, 8], [16, 2]]    [16
  1  layers       Sequential            691   train   17.1 K             [16, 16]     [1


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 22.2 K
```

```
train_model result: {'val_loss': 23640.08984375, 'hp_metric': 23640.08984375}
spotpython tuning: 8588.830078125 [######----] 55.91%. Success rate: 5.41%


     Name         Type               Params   Mode    FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer       96   train   2.6 K   [[16, 8], [16, 2]]     [16, 8]
  1  layers       Sequential            197   train   4.2 K              [16, 8]     [16, 1]



Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 6.7 K

train_model result: {'val_loss': 24108.263671875, 'hp_metric': 24108.263671875}
spotpython tuning: 8588.830078125 [######----] 59.47%. Success rate: 5.26%


     Name         Type               Params   Mode     FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32, 16]
  1  layers       Sequential            691   train   34.2 K             [32, 16]     [32, 1]



Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 8761.345703125, 'hp_metric': 8761.345703125}
spotpython tuning: 8588.830078125 [######----] 61.85%. Success rate: 5.13%
```

```
     Name         Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32
  1  layers       Sequential           691   train   34.2 K             [32, 16]     [3


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 23936.365234375, 'hp_metric': 23936.365234375}
spotpython tuning: 8588.830078125 [#######---] 66.56%. Success rate: 5.00%


     Name         Type               Params   Mode    FLOPs             In sizes   Out s

  0  cond_layer   ConditionalLayer       96   train   5.1 K   [[32, 8], [32, 2]]     [32
  1  layers       Sequential           197   train   8.3 K              [32, 8]     [32


Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 23658.79296875, 'hp_metric': 23658.79296875}
spotpython tuning: 8588.830078125 [#######---] 67.38%. Success rate: 4.88%


     Name         Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer   ConditionalLayer      192   train    5.1 K   [[16, 8], [16, 2]]    [16
  1  layers       Sequential           587   train   17.1 K             [16, 16]     [1
```

```
Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 22.2 K

train_model result: {'val_loss': 21879.568359375, 'hp_metric': 21879.568359375}
spotpython tuning: 8588.830078125 [#######---] 68.60%. Success rate: 4.76%


     Name         Type               Params   Mode     FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer      192   train   10.2 K   [[32, 8], [32, 2]]    [32, 16]
  1  layers       Sequential            691   train   34.2 K             [32, 16]     [32, 1]


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K

train_model result: {'val_loss': 23994.609375, 'hp_metric': 23994.609375}
spotpython tuning: 8588.830078125 [#######---] 70.56%. Success rate: 4.65%


     Name         Type               Params   Mode     FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer      192   train    5.1 K   [[16, 8], [16, 2]]    [16, 16]
  1  layers       Sequential            691   train   17.1 K             [16, 16]     [16, 1]


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 22.2 K
```

```
train_model result: {'val_loss': 22487.458984375, 'hp_metric': 22487.458984375}
spotpython tuning: 8588.830078125 [#######---] 73.29%. Success rate: 4.55%


     Name        Type               Params   Mode    FLOPs             In sizes   Out s

 0   cond_layer  ConditionalLayer      96   train   5.1 K   [[32, 8], [32, 2]]     [32
 1   layers      Sequential           197   train   8.3 K              [32, 8]     [32



Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 22477.447265625, 'hp_metric': 22477.447265625}
spotpython tuning: 8588.830078125 [#######---] 73.88%. Success rate: 4.44%


     Name        Type               Params   Mode     FLOPs             In sizes   Out

 0   cond_layer  ConditionalLayer     192   train    5.1 K   [[16, 8], [16, 2]]    [16
 1   layers      Sequential           587   train   17.1 K             [16, 16]     [1



Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 22.2 K

train_model result: {'val_loss': 13692.08984375, 'hp_metric': 13692.08984375}
spotpython tuning: 8588.830078125 [########--] 78.84%. Success rate: 4.35%
```

```
    Name        Type               Params  Mode    FLOPs            In sizes   Out sizes

 0  cond_layer  ConditionalLayer      192  train  10.2 K  [[32, 8], [32, 2]]    [32, 16]
 1  layers      Sequential           691  train  34.2 K            [32, 16]     [32, 1]



Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K


train_model result: {'val_loss': 24043.455078125, 'hp_metric': 24043.455078125}
spotpython tuning: 8588.830078125 [########--] 80.94%. Success rate: 4.26%


Using spacefilling design as fallback.



    Name        Type               Params  Mode   FLOPs            In sizes   Out sizes

 0  cond_layer  ConditionalLayer       96  train  2.6 K  [[16, 8], [16, 2]]     [16, 8]
 1  layers      Sequential            153  train  4.2 K              [16, 8]     [16, 1]



Trainable params: 249
Non-trainable params: 0
Total params: 249
Total estimated model params size (MB): 0
Modules in train mode: 18
Modules in eval mode: 0
Total FLOPs: 6.7 K


train_model result: {'val_loss': 23562.740234375, 'hp_metric': 23562.740234375}
spotpython tuning: 8588.830078125 [########--] 81.49%. Success rate: 4.17%
```

```
    Name        Type               Params   Mode    FLOPs            In sizes   Out s

 0  cond_layer  ConditionalLayer      96   train   5.1 K   [[32, 8], [32, 2]]     [32
 1  layers      Sequential           197   train   8.3 K              [32, 8]     [32


Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 23973.388671875, 'hp_metric': 23973.388671875}
spotpython tuning: 8588.830078125 [########--] 84.10%. Success rate: 4.08%


    Name        Type               Params   Mode     FLOPs            In sizes   Out

 0  cond_layer  ConditionalLayer     192   train    5.1 K   [[16, 8], [16, 2]]    [16
 1  layers      Sequential           587   train   17.1 K             [16, 16]     [1


Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 22.2 K

train_model result: {'val_loss': 23741.322265625, 'hp_metric': 23741.322265625}
spotpython tuning: 8588.830078125 [#########-] 87.88%. Success rate: 4.00%


    Name        Type               Params   Mode     FLOPs            In sizes   Out

 0  cond_layer  ConditionalLayer     192   train    5.1 K   [[16, 8], [16, 2]]    [16
 1  layers      Sequential           587   train   17.1 K             [16, 16]     [1
```

```
Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 22.2 K

train_model result: {'val_loss': 52557.8828125, 'hp_metric': 52557.8828125}
spotpython tuning: 8588.830078125 [#########-] 91.11%. Success rate: 3.92%


     Name         Type               Params   Mode    FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer      96   train   5.1 K   [[32, 8], [32, 2]]     [32, 8]
  1  layers       Sequential           197   train   8.3 K              [32, 8]     [32, 1]


Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K

train_model result: {'val_loss': 23981.197265625, 'hp_metric': 23981.197265625}
spotpython tuning: 8588.830078125 [#########-] 91.84%. Success rate: 3.85%


     Name         Type               Params   Mode     FLOPs             In sizes   Out sizes

  0  cond_layer   ConditionalLayer     192   train   10.2 K   [[32, 8], [32, 2]]    [32, 16]
  1  layers       Sequential           691   train   34.2 K             [32, 16]     [32, 1]


Trainable params: 883
Non-trainable params: 0
Total params: 883
Total estimated model params size (MB): 0
Modules in train mode: 36
Modules in eval mode: 0
Total FLOPs: 44.4 K
```

```
train_model result: {'val_loss': 24007.826171875, 'hp_metric': 24007.826171875}
spotpython tuning: 8588.830078125 [#########-] 93.63%. Success rate: 3.77%


     Name        Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer  ConditionalLayer      192   train    5.1 K   [[16, 8], [16, 2]]    [16
  1  layers      Sequential            587   train   17.1 K             [16, 16]     [1



Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 22.2 K

train_model result: {'val_loss': 24064.419921875, 'hp_metric': 24064.419921875}
spotpython tuning: 8588.830078125 [#########-] 94.95%. Success rate: 3.70%


     Name        Type               Params   Mode     FLOPs             In sizes   Out

  0  cond_layer  ConditionalLayer      192   train    5.1 K   [[16, 8], [16, 2]]    [16
  1  layers      Sequential            587   train   17.1 K             [16, 16]     [1



Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 22.2 K

train_model result: {'val_loss': 5181.326171875, 'hp_metric': 5181.326171875}
spotpython tuning: 5181.326171875 [##########] 95.69%. Success rate: 5.45%
```

```
    Name        Type               Params  Mode   FLOPs            In sizes  Out sizes

 0  cond_layer  ConditionalLayer      96  train  5.1 K  [[32, 8], [32, 2]]    [32, 8]
 1  layers      Sequential          197  train  8.3 K             [32, 8]    [32, 1]



Trainable params: 293
Non-trainable params: 0
Total params: 293
Total estimated model params size (MB): 0
Modules in train mode: 24
Modules in eval mode: 0
Total FLOPs: 13.4 K


train_model result: {'val_loss': 20914.025390625, 'hp_metric': 20914.025390625}
spotpython tuning: 5181.326171875 [##########] 98.98%. Success rate: 5.36%



    Name        Type               Params  Mode    FLOPs            In sizes  Out sizes

 0  cond_layer  ConditionalLayer     192  train   5.1 K  [[16, 8], [16, 2]]   [16, 16]
 1  layers      Sequential          587  train  17.1 K            [16, 16]    [16, 1]



Trainable params: 779
Non-trainable params: 0
Total params: 779
Total estimated model params size (MB): 0
Modules in train mode: 26
Modules in eval mode: 0
Total FLOPs: 22.2 K


train_model result: {'val_loss': 182654.625, 'hp_metric': 182654.625}
spotpython tuning: 5181.326171875 [##########] 100.00%. Success rate: 5.26% Done...

Experiment saved to CondNet_01_res.pkl
```

59.1. Looking at the Results

59.1.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with spotpython's method plot_progress. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress()
```

59.1.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

```
| name           | type    | default    |   lower |    upper | tuned    | transform
|----------------|--------|-----------|---------|---------|---------
|-----------------------|--------------|---------|
| l1             | int     | 3          |     3.0 |     4.0 | 4.0      | transform_power_
| epochs         | int     | 4          |     3.0 |     7.0 | 7.0      | transform_power_
| batch_size     | int     | 4          |     4.0 |     5.0 | 4.0      | transform_power_
| act_fn         | factor  | ReLU       |     0.0 |     5.0 | ReLU     | None
| optimizer      | factor  | SGD        |     0.0 |     2.0 | Adam     | None
| dropout_prob   | float   | 0.01       |     0.0 |   0.025 | 0.0      | None
| lr_mult        | float   | 1.0        |     0.1 |    20.0 | 20.0     | None
```

```
| patience       | int    | 2         |     2.0 |     3.0 | 2.0     | transform_power_2_int |
| batch_norm     | factor | 0         |     0.0 |     1.0 | 0       | None                  |
| initialization | factor | Default   |     0.0 |     4.0 | Default | None                  |
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=1.0)
```

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
l1:  0.005952705344705705
epochs:  0.005952705344705705
batch_size:  100.0
act_fn:  0.005952705344705705
optimizer:  12.238623055489287
dropout_prob:  0.005952705344705705
lr_mult:  0.005952705344705705
patience:  0.005952705344705705
batch_norm:  0.005952705344705705
initialization:  0.005952705344705705
```

Prediction Surface
Prediction
500000
400000
300000
200000
100000
batch_size
optimizer
Prediction Error Surface
Error
12000
10000
8000
6000
4000
Prediction Contour
Error Contour
optimizer
batch_size

Prediction Surface
Prediction
23000
22000
21000
20000
19000
18000
17000
16000
3.0
3.2
3.4
3.6
3.8
4.0
l1
4.0
4.2
4.4
4.6
4.8
5.0
batch_size
Prediction Error Surface
Error
5000
4500
4000
3500
3000
Prediction Contour
23700
22800
21900
21000
20100
19200
18300
17400
16500
15600
Error Contour
5250
4800
4350
3900
3450
3000
2550

59.1.3. Get the Tuned Architecture

```
import pprint
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner)
pprint.pprint(config)
```

```
{'act_fn': ReLU(),
 'batch_norm': False,
 'batch_size': 16,
 'dropout_prob': 0.0,
 'epochs': 128,
 'initialization': 'Default',
 'l1': 16,
 'lr_mult': 20.0,
```

```
'optimizer': 'Adam',
'patience': 4}
```

Part X.

Multi Objective Optimization

60. Introduction to Desirability Functions

The desirability function approach is a widely adopted method in industry for optimizing multiple response processes ("NIST/SEMATECH e-Handbook of Statistical Methods" 2021). It operates on the principle that the overall "quality" of a product or process with multiple quality characteristics is deemed unacceptable if any characteristic falls outside the "desired" limits. This approach identifies operating conditions that yield the most "desirable" response values, effectively balancing multiple objectives to achieve optimal outcomes.

Often, different scales are used for various objectives. When combining these objectives into a single new one, the challenge arises of how to compare the scales with each other. The fundamental idea of the desirability index is to transform the deviations of the objective value from its target value into comparable desirabilities, i.e., onto a common scale. For this, a target value as well as a lower and/or upper specification limit must be known for each objective involved. A result outside the specification limits is assigned a desirability of 0, while a result at the target value is assigned a desirability of 1. Linear or nonlinear transformation, such as a power transformation, can be chosen as the transformation between the specification limits. The desirability index according to Derringer and Suich (1980) is then the geometric mean of the desirabilities of the various objectives (Weihe et al. 1999).

The `desirability` package (Kuhn 2016), which is written in the statistical programming language `R`, contains `S3` classes for multivariate optimization using the desirability function approach of Harington (1965) with functional forms described by Derringer and Suich (1980). It is available on CRAN, see https://cran.r-project.org/package=desirability.

Hyperparameter Tuning (or Hyperparameter Optimization) is crucial for configuring machine learning algorithms, as hyperparameters significantly impact performance (Bartz et al. 2022; Bischl et al. 2023) To avoid manual, time-consuming, and irreproducible trial-and-error processes, these tuning methods can be used. They include simple techniques like grid and random search, as well as advanced approaches such as evolution strategies, surrogate optimization, Hyperband, and racing. The tuning process has to consider several objectives, such as maximizing the model's performance while minimizing the training time or model complexity. The desirability function approach is a suitable method for multi-objective optimization, as it allows for the

simultaneous optimization of multiple objectives by combining them into a single desirability score.

This paper is structured as follows: After presenting the desirability function approach in Section 60.2, we introduce the `Python` package `spotdesirability`, which is a `Python` implementation of the `R` package `desirability`. The introduction is based on several "hands-on" examples. Section 60.3 provides an overview of related work in the field of multi-objective optimization and hyperparameter tuning. Section 60.4 presents an example of a chemical reaction with two objectives: conversion and activity. The example is based on a response surface experiment described by Myers, Montgomery, and Anderson-Cook (2016) and also used by Kuhn (2016). It allows a direct comparison of the results obtained with the `R` package `desirability` and the `Python` package `spotdesirability`. Section 60.5 describes how to maximize the desirability function using the Nelder-Mead algorithm from the `scipy.optimize.minimize` function. This approach is common in RSM (G. E. P. Box and Wilson 1951; Myers, Montgomery, and Anderson-Cook 2016). The optimization process is illustrated using the chemical reaction example from Section 60.4. This example is based on the example presented in Kuhn (2016), so that, similar to the comparison in Section 60.4, a comparison of the results obtained with the `R` and `Python` packages is possible. Section 60.6 presents an example of surrogate model-based optimization (Gramacy 2020; Forrester, Sóbester, and Keane 2008) using the `spotdesirability` package. Results from the RSM optimization can be compared with the results from surrogate model-based optimization. The surrogate model is based on the `spotpython` package (Bartz-Beielstein 2023b). Section 60.7 presents an example of hyperparameter tuning of a neural network implemented in `PyTorch` using the `spotdesirability` package. The goal of this example is to demonstrate how to use the desirability function approach for hyperparameter tuning in a machine learning context. The article concludes with a summary and outlook in Section 60.8.

Citation

- If this document has been useful to you and you wish to cite it in a scientific publication, please refer to the following paper, which can be found on arXiv: https://arxiv.org/abs/2307.10262.

```bibtex
@article{bart25a,
    adsurl = {https://ui.adsabs.harvard.edu/abs/2025arXiv250323595B},
    archiveprefix = {arXiv},
    author = {{Bartz-Beielstein}, Thomas},
    doi = {10.48550/arXiv.2503.23595},
    eid = {arXiv:2503.23595},
    eprint = {2503.23595},
    journal = {arXiv e-prints},
    keywords = {Optimization and Control, Machine Learning, Applications, 90C26, I.2.6; G.1.6},
    month = mar,
    pages = {arXiv:2503.23595},
    primaryclass = {math.OC},
    title = {{Multi-Objective Optimization and Hyperparameter Tuning With Desirability Functions}}
    year = 2025,
    }
```


60.1. The Python Packages Used in This Article

The following Python packages, classes, and functions are used in this article:

```python
import os
from math import inf
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from spotpython.hyperparameters.values import set_hyperparameter
from spotpython.data.diabetes import Diabetes
from spotpython.fun.mohyperlight import MoHyperLight
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.hyperparameters.values import set_hyperparameter
from spotpython.mo.functions import fun_myer16a
from spotpython.mo.plot import plot_mo
from spotpython.plot.contour import (mo_generate_plot_grid, contour_plot,
                                     contourf_plot)
from spotpython.utils.eda import print_exp_table, print_res_table
from spotpython.utils.file import get_experiment_filename
from spotpython.spot import Spot
from spotpython.utils.init import (fun_control_init, surrogate_control_init,
                                   design_control_init)
```

```
from spotdesirability.utils.desirability import (DOverall, DMax, DCategorical, DMin,
                                                 DTarget, DArb, DBox)
from spotdesirability.plot.ccd import plotCCD
from spotdesirability.functions.rsm import rsm_opt, conversion_pred, activity_pred
warnings.filterwarnings("ignore")
```

60.2. Desirability

60.2.1. Basic Desirability Functions

The desirability function approach to simultaneously optimizing multiple equations was originally proposed by Harington (1965). The approach translates the functions to a common scale ($[0, 1]$), combines them using the geometric mean, and optimizes the overall metric. The equations can represent model predictions or other equations. Kuhn (2016) notes that desirability functions are popular in response surface methodology (RSM) (G. E. P. Box and Wilson 1951; Myers, Montgomery, and Anderson-Cook 2016) to simultaneously optimize a series of quadratic models. A response surface experiment may use measurements on a set of outcomes, where instead of optimizing each outcome separately, settings for the predictor variables are sought to satisfy all outcomes at once.

Kuhn (2016) explains that originally, Harrington used exponential functions to quantify desirability. In our `Python` implementation, which is based on the `R` package `desirablity` from Kuhn (2016), the simple discontinuous functions of Derringer and Suich (1980) are adopted. For simultaneous optimization of equations, individual "desirability" functions are constructed for each function, and Derringer and Suich (1980) proposed three forms of these functions corresponding to the optimization goal type. Kuhn (2016) describes the `R` implementation as follows:

> Suppose there are R equations or functions to simultaneously optimize, denoted $f_r(\vec{x})$ $(r = 1 \dots R)$. For each of the R functions, an individual "desirability" function is constructed that is high when $f_r(\vec{x})$ is at the desirable level (such as a maximum, minimum, or target) and low when $f_r(\vec{x})$ is at an undesirable value. Derringer and Suich (1980) proposed three forms of these functions, corresponding to the type of optimization goal, namely maximization, minimization, or target optimization. The associated desirability functions are denoted $d_r^{\max}$, $d_r^{\min}$, and d_r^{target}.

60.2.1.1. Maximization

For maximization of $f_r(\vec{x})$ ("larger-is-better"), the following function is used:

$$d_r^{\max} = \begin{cases} 0 & \text{if } f_r(\vec{x}) < A \\ \left(\frac{f_r(\vec{x})-A}{B-A}\right)^s & \text{if } A \leq f_r(\vec{x}) \leq B \\ 1 & \text{if } f_r(\vec{x}) > B, \end{cases}$$

where A, B, and s are chosen by the investigator.

60.2.1.2. Minimization

For minimization ("smaller-is-better"), the following function is proposed:

$$d_r^{\min} = \begin{cases} 0 & \text{if } f_r(\vec{x}) > B \\ \left(\frac{f_r(\vec{x})-B}{A-B}\right)^s & \text{if } A \leq f_r(\vec{x}) \leq B \\ 1 & \text{if } f_r(\vec{x}) < A \end{cases}$$

60.2.1.3. Target Optimization

In "target-is-best" situations, the following function is used:

$$d_r^{\text{target}} = \begin{cases} \left(\frac{f_r(\vec{x})-A}{t_0-A}\right)^{s_1} & \text{if } A \leq f_r(\vec{x}) \leq t_0 \\ \left(\frac{f_r(\vec{x})-B}{t_0-B}\right)^{s_2} & \text{if } t_0 \leq f_r(\vec{x}) \leq B \\ 0 & \text{otherwise.} \end{cases}$$

Kuhn (2016) explains that these functions, which are shown in Figure 60.1, share the same scale and are discontinuous at specific points A, B, and t_0. The values of s, s_1, or s_2 can be chosen so that the desirability criterion is easier or more difficult to satisfy. For example:

- If s is chosen to be less than 1 in $d_r^{\min}$, $d_r^{\min}$ is near 1 even if the model $f_r(\vec{x})$ is not low.
- As values of s move closer to 0, the desirability reflected by $d_r^{\min}$ becomes higher.
- Values of s greater than 1 will make $d_r^{\min}$ harder to satisfy in terms of desirability.

Kuhn notes that these scaling factors are useful when one equation holds more importance than others. He emphasizes that any function can reflect model desirability; Del Castillo, Montgomery, and McCarville (1996) developed alternative functions suitable for gradient-based optimizations.

```
dCategorical_obj = DCategorical(missing=0.2, values={"A": 0.8, "B": 0.6, "C": 0.4})
dCategorical_obj.plot()
```

Figure 60.1.: Examples of the three primary desirability functions. Panel (a) shows an example of a larger–is–better function, panel (b) shows a smaller–is–better desirability function and panel (c) shows a function where the optimal value corresponds to a target value. Not that increasing the scale parameter makes it more difficult to achieve higher desirability, while values smaller than 1 make it easier to achieve good results.

For each of these three desirability functions (and the others discussed in Section 60.2.3),

there are `print_class_attributes`, `plot`, and `predict` methods similar to the `R` implementation (Kuhn 2016). The `print_attributes` method prints the class attributes, the `plot` method plots the desirability function, and the `predict` method predicts the desirability for a given input.

60.2.2. Overall Desirability

Given the R desirability functions $d_1 \dots d_r$ are on the [0,1] scale, they can be combined to achieve an overall desirability function, D. One method of doing this is by the geometric mean:

$$D = \left(\prod_{r=1}^{R} d_r \right)^{1/R} .$$

The geometric mean has the property that if any one model is undesirable ($d_r = 0$), the overall desirability is also unacceptable ($D = 0$). Once D has been defined and the prediction equations for each of the R equations have been computed, it can be used to optimize or rank the predictors.

60.2.3. Non-Standard Features

The `R` package `desirability` (Kuhn 2016) offers a few non-standard features. These non-standard features are also included in the `Python` implementation and will be discussed in the following. First, we will consider the non-informative desirability and missing values, followed by zero-desirability tolerances, and finally non-standard desirability functions.

60.2.3.1. Non-Informative Desirability and Missing Values

According to Kuhn, if inputs to desirability functions are uncomputable, the package estimates a non-informative value by computing desirabilities over the possible range and taking the mean.
If an input to a desirability function is `NA`, by default, it is replaced by this non-informative value. Setting `object$missing` to `NA` (in `R`) changes the calculation to return an `NA` for the result, where `object` is the result of a call to one of the desirability functions.A similar procedure is implemented in the `Python` package. The non-informative value is plotted as a broken line in default `plot` methods.

60.2.3.2. Zero-Desirability Tolerances

Kuhn (2016) highlights that in high-dimensional outcomes, finding feasible solutions where every desirability value is acceptable can be challenging. Each desirability R function has a `tol` argument, which can be set between [0, 1] (default is `NULL`). If not null, zero desirability values are replaced by `tol`.

60.2.3.3. Non-Standard Desirability Functions

Kuhn mentions scenarios where the three discussed desirability functions are inadequate for user requirements.

60.2.3.3.1. Custom or Arbitary Desirability Functions

In this case, the `dArb` function (`Arb` stands for "Arbitary") can be used to create a custom desirability function. `dArb` accepts numeric vector inputs with matching desirabilities to approximate other functional forms. For instance, a logistic function can be used as a desirability function. The logistic function is defined as $d(\vec{x}) = \frac{1}{1+\exp(-\vec{x})}$. For inputs outside the range ± 5, desirability values remain near zero and one. The desirability function is defined using 20 computation points on this range, and these values establish the desirability function.

```
# Define the logistic function
def foo(u):
    return 1 / (1 + np.exp(-u))

# Generate input values
x_input = np.linspace(-5, 5, 20)

# Create the DArb object
logistic_d = DArb(x_input, foo(x_input))
logistic_d.print_class_attributes()
```

```
Class: DArb
x: [-5.         -4.47368421 -3.94736842 -3.42105263 -2.89473684 -
2.36842105
 -1.84210526 -1.31578947 -0.78947368 -0.26315789  0.26315789  0.78947368
  1.31578947  1.84210526  2.36842105  2.89473684  3.42105263  3.94736842
  4.47368421  5.        ]
d: [0.00669285 0.01127661 0.0189398  0.03164396 0.05241435 0.08561266
 0.1368025  0.21151967 0.31228169 0.43458759 0.56541241 0.68771831
 0.78848033 0.8631975  0.91438734 0.94758565 0.96835604 0.9810602
```

```
 0.98872339 0.99330715]
tol: None
missing: 0.5
```

Inputs in-between these grid points are linearly interpolated. Using this method, extreme values are applied outside the input range. Figure 60.2 displays a `plot` of the `logisticD` object.

```
logistic_d.plot()
```

Figure 60.2.: An example of a desirability function created using the `DArb` function. The desirability function is a logistic curve that is defined by 20 points on the range [-5, 5].

60.2.3.3.2. Desirabilty Function for Box Constraints

Kuhn also adds that there is a desirability function for implementing box constraints on an equation. For example, assigning zero desirability to values beyond ± 1.682 in the design region, instead of penalizing. Figure 60.3 demonstrates an example function.

```
box_desirability = DBox(low=-1.682, high=1.682)
box_desirability.plot(non_inform=False)
```

Figure 60.3.: An example of a box-like desirability function that assigns zero desirability to values outside of the range [-1.682, 1.682].

60.2.3.3.3. Desirability Function for Categorical Inputs

Kuhn concludes by mentioning another non-standard application involving categorical inputs. Desirabilities are assigned to each value. For example:

```
# Define desirability values for categorical inputs
values = {"value1": 0.1, "value2": 0.9, "value3": 0.2}
```

```
# Create a DCategorical object
grouped_desirabilities = DCategorical(values)

# Print the desirability values
print("Desirability values for categories:")
for category, desirability in grouped_desirabilities.values.items():
    print(f"{category}: {desirability}")

# Example usage: Predict desirability for a specific category
category = "value2"
predicted_desirability = grouped_desirabilities.predict([category])
print(f"\nPredicted desirability for '{category}': {predicted_desirability[0]}")
```

```
Desirability values for categories:
value1: 0.1
value2: 0.9
value3: 0.2

Predicted desirability for 'value2': 0.9
```

Figure 60.4 visualizes a plot of desirability profiles for this setup.

```
grouped_desirabilities.plot(non_inform=False)
```

Figure 60.4.: Desirability function for categorical values. The desirability values are assigned to three categories: 'value1', 'value2', and 'value3'.

60.3. Related Work

Multiobjective approaches are established optimization tools (Emmerich and Deutz 2018). The weighted-sum approach is a simple and widely used method for multi-objective optimization, but probably only, because its disadvantages are unknown. Compared to the weighted-sum approach, the desirability-function approach is a better choice for multi-objective optimization. The desirability function approach also allows for more flexibility in defining the objectives and their trade-offs.

Nino et al. (2015) discuss the use of Experimental Designs and RSM to optimize conflicting responses in the development of a 3D printer prototype. Specifically, they focus on an interlocking device designed to recycle polyethylene terephthalate water bottles. The optimization involves two conflicting goals: maximizing load capacity and minimizing mass. A Box Behnken Design (BBD) was used for the experimental setup, and desirability functions were applied to identify the best trade-offs.

Karl et al. (2023) describe multi-objective optimization in maschine learning. Coello et al. (2021) give an overview of Multi-Objective Evolutionary Algorithms.

60.4. An Example With Two Objectives: Chemical Reaction

Similar to the presentation in Kuhn (2016), we will use the example of a chemical reaction to illustrate the desirability function approach. The example is based on a response surface experiment described by Myers, Montgomery, and Anderson-Cook (2016). The goal is to maximize the percent conversion of a chemical reaction while keeping the thermal activity within a specified range.

The central composite design (CCD) is the most popular class of designs used for fitting second-order response surface models (Montgomery 2001). Since the location of the optimum is unknown before the RSM starts, G. E. P. Box and Hunter (1957) suggested that the design should be rotatable (it provides equal precision of estimation in all directions or stated differently, the variance of the predicted response is the same at all points that are the same distance from the center of the design space). A CCD is made rotable by using an axis distance value of $\alpha = (n_F)^{1/4}$, where n_F is the number of points (here $2^3 = 8$) (Montgomery 2001). Figure 60.5 shows the design space for the chemical reaction example. The design space is defined by three variables: reaction time, reaction temperature, and percent catalyst. This rotable CCD consists of a full factorial design with three factors, each at two levels, plus a center point and six $(2 \times k)$ axial points. The axial points are located at a distance of $\pm\alpha$ from the center point in each direction.

```
plotCCD(figsize=(8, 6), title=None)
```

Figure 60.5.: Central composite design (CCD) for the chemical reaction example.

Montgomery (2001) note that it is not important to have exact rotability. From a prediction variance point of view, the best choice is to set $\alpha = \sqrt{k}$, which results in a so-called spherical CCD.

60.4.1. The Two Objective Functions: Conversion and Activity

Myers, Montgomery, and Anderson-Cook (2016) present two equations for the fitted quadratic response surface models.

$$\begin{aligned} f_{\text{con}}(x) =& 81.09 + 1.0284 \cdot x_1 + 4.043 \cdot x_2 + 6.2037 \cdot x_3 + 1.8366 \cdot x_1^2 + 2.9382 \cdot x_2^2 \\ &+ 5.1915 \cdot x_3^2 + 2.2150 \cdot x_1 \cdot x_2 + 11.375 \cdot x_1 \cdot x_3 + 3.875 \cdot x_2 \cdot x_3 \end{aligned}$$

and

$$\begin{aligned} f_{\text{act}}(x) =& 59.85 + 3.583 \cdot x_1 + 0.2546 \cdot x_2 + 2.2298 \cdot x_3 + 0.83479 \cdot x_1^2 + 0.07484 \cdot x_2^2 \\ &+ 0.05716 \cdot x_3^2 + 0.3875 \cdot x_1 \cdot x_2 + 0.375 \cdot x_1 \cdot x_3 + 0.3125 \cdot x_2 \cdot x_3. \end{aligned}$$

They are implemented as Python functions that take a vector of three parameters (reaction time, reaction temperature, and percent catalyst) and return the predicted values

for the percent conversion and thermal activity and available in the `spotdesirability` package.

The goal of the analysis in Myers, Montgomery, and Anderson-Cook (2016) was to

- maximize conversion while
- keeping the thermal activity between 55 and 60 units. An activity target of 57.5 was used in the analysis.

Plots of the response surface models are shown in Figure 60.6 and Figure 60.7, where reaction time and percent catalyst are plotted while the reaction temperature was varied at four different levels. Both quadratic models, as pointed out by Kuhn, are saddle surfaces, and the stationary points are outside of the experimental region. To determine predictor settings for these models, a constrained optimization can be used to stay inside the experimental region. Kuhn notes:

> In practice, we would just use the `predict` method for the linear model objects to get the prediction equation. Our results are slightly different from those given by Myers and Montgomery because they used prediction equations with full floating-point precision.

60.4.2. Contour Plot Generation

60.4.2.1. Contour Plots for the Response Surface Models

We will generate contour plots for the percent conversion and thermal activity models. The contour-plot generation comprehends the following steps:

- generating a grid of points in the design space and evaluating the response surface models at these points, and
- plotting the contour plots for the response surface models

We will use the function `mo_generate_plot_grid` to generate the grid and the function `mo_contourf_plots` for creating the contour plots for the response surface models. Both functions are available in the `spotpython` package.

First we define the variables, their ranges, the resolutions for the grid, and the objective functions. The `variables` dictionary contains the variable names as keys and their ranges as values. The `resolutions` dictionary contains the variable names as keys and their resolutions as values. The `functions` dictionary contains the function names as keys and the corresponding functions as values. Next we can generate the Pandas DataFrame `plot_grid`. It has the columns `time`, `temperature`, `catalyst`, `conversionPred`, and `activityPred`.

```
variables = {
    "time": (-1.7, 1.7),
    "temperature": (-1.7, 1.7),
    "catalyst": (-1.7, 1.7)
}
resolutions = {
    "time": 50,
    "temperature": 4,
    "catalyst": 50
}
functions = {
    "conversionPred": conversion_pred,
    "activityPred": activity_pred
}
plot_grid = mo_generate_plot_grid(variables, resolutions, functions)
```

Figure 60.6 shows the response surface for the percent conversion model. To plot the model contours, the temperature variable was fixed at four diverse levels. The largest effects in the fitted model are due to the time × catalyst interaction and the linear and quadratic effects of catalyst. Figure 60.7 shows the response surface for the thermal activity model. To plot the model contours, the temperature variable was fixed at four diverse levels. The main effects of time and catalyst have the largest effect on the fitted model.

```
contourf_plot(
    plot_grid,
    x_col="time",
    y_col="catalyst",
    z_col="conversionPred",
    facet_col="temperature",
)
```

Figure 60.6.: The response surface for the percent conversion model. To plot the model contours, the temperature variable was fixed at four diverse levels.

```
contourf_plot(
    plot_grid,
    x_col="time",
    y_col="catalyst",
    z_col="activityPred",
    facet_col="temperature",
)
```

Figure 60.7.: The response surface for the thermal activity model. To plot the model contours, the temperature variable was fixed at four diverse levels.

60.4.2.2. Defining the Desirability Functions

Following the steps described in Kuhn (2016), translating the experimental goals to desirability functions, a larger-is-better function ($d_r^{\max}$) is used for percent conversion with values $A = 80$ and $B = 97$. A target-oriented desirability function (d_r^{target}) was used for thermal activity with $t_0 = 57.5$, $A = 55$, and $B = 60$.

Kuhn emphasizes that to construct the overall desirability functions, objects must be created for the individual functions. In the following, we will use classes of the `Python` package `spotdesirability` to create the desirability objects. The `spotdesirability` package is part of the `sequential parameter optimization` framework (Bartz-Beielstein 2023b). It is available on GitHub

[https://github.com/sequential-parameter-optimization/spotdesirability] and on PyPi https://pypi.org/project/spotdesirability and can be installed via `pip install spotdesirability`.

The desirability objects can be created as follows:

```
conversionD = DMax(80, 97)
activityD = DTarget(55, 57.5, 60)
```

Although the original analysis in Myers, Montgomery, and Anderson-Cook (2016) used numerous combinations of scaling parameters, following the presentation in Kuhn (2016), we will only show analyses with the default scaling factor values.

Example 60.1 (Computing Desirability at the Center Point)**.** Using these desirability objects `conversionD`and `activityD`, the following code segment shows how to predict the desirability for the center point of the experimental design. The center point is defined as [0, 0, 0].

```
pred_outcomes = [
    conversion_pred([0, 0, 0]),
    activity_pred([0, 0, 0])
]
print("Predicted Outcomes:", pred_outcomes)
```

```
Predicted Outcomes: [81.09, 59.85]
```

```
# Predict desirability for each outcome
conversion_desirability = conversionD.predict(pred_outcomes[0])
activity_desirability = activityD.predict(pred_outcomes[1])
print("Conversion Desirability:", conversion_desirability)
print("Activity Desirability:", activity_desirability)
```

```
Conversion Desirability: [0.06411765]
Activity Desirability: [0.06]
```

Similar to the implementation in Kuhn (2016), to get the overall score for these settings of the experimental factors, the `dOverall` function is used to combine the objects and `predict` is used to get the final score. The `print_class_attributes` method prints the class attributes of the `DOverall` object.

```
overallD = DOverall(conversionD, activityD)
overallD.print_class_attributes()
```

```
Class: DOverall
d_objs: [

  Class: DMax
  low: 80
  high: 97
  scale: 1
  tol: None
  missing: 0.5

  Class: DTarget
  low: 55
  target: 57.5
  high: 60
  low_scale: 1
  high_scale: 1
  tol: None
  missing: 0.4949494949494951
]
```

Note: The attribute `missing` is explained in Section 60.2.3.1.

Finally, we can print the overall desirability for the center point of the experimental design.

```
#] echo: true
overall_desirability = overallD.predict(pred_outcomes, all=True)
print("Conversion Desirability:", overall_desirability[0][0])
print("Activity Desirability:", overall_desirability[0][1])
print("Overall Desirability:", overall_desirability[1])
```

```
Conversion Desirability: [0.06411765]
Activity Desirability: [0.06]
Overall Desirability: [0.06202466]
```

60.4.2.3. Generating the Desirability DataFrame

A DataFrame `d_values_df` is created to store the individual desirability values for each outcome, and the overall desirability value is added as a new column. First, we predict desirability values and extract the individual and overall desirability values.

Note: The `all=True` argument indicates that both individual and overall desirability values should be returned.

We add the individual and overall desirability values to the `plot_grid` DataFrame, that was created earlier in Section 60.4.2.

```
d_values = overallD.predict(plot_grid.iloc[:, [3, 4]].values, all=True)
individual_desirabilities = d_values[0]
overall_desirability = d_values[1]
d_values_df = pd.DataFrame(individual_desirabilities).T
d_values_df.columns = ["D1", "D2"]
d_values_df["Overall"] = overall_desirability
plot_grid = pd.concat([plot_grid, d_values_df], axis=1)
```

60.4.2.4. Contour Plots for the Desirability Surfaces

We will use `spotpython`'s `contourf_plot` function to create the contour plots for the individual desirability surfaces and the overall desirability surface. The `plot_grid` DataFrame contains the predicted values for the conversion and activity models, which are used to create the contour plots.

Figure 60.8, Figure 60.9, and Figure 60.10 show contour plots of the individual desirability function surfaces and the overall surface. These plots are in correspondence with the figures in Kuhn (2016), but the color schemes are different. The `plot_grid` DataFrame contains the predicted values for the conversion and activity models, which are used to create the contour plots.

The individual desirability surface for the `percent conversion` outcome is shown in Figure 60.8 and the individual desirability surface for the thermal activity outcome is shown in Figure 60.9. Finally, the overall desirability surface is shown in Figure 60.10.

```
contourf_plot(
    data=plot_grid,
    x_col='time',
    y_col='catalyst',
    z_col='D1',
    facet_col='temperature',
    aspect=1,
    as_table=True,
    figsize=(3,3)
)
```

Figure 60.8.: The individual desirability surface for the percent conversion outcome using `dMax(80, 97)`

```
contourf_plot(
    data=plot_grid,
    x_col='time',
    y_col='catalyst',
    z_col='D2',
    facet_col='temperature',
    aspect=1,
    as_table=True,
    figsize=(3,3)
)
```

Figure 60.9.: The individual desirability surface for the thermal activity outcome using `dTarget(55, 57.5, 60)`

```
contourf_plot(
    data=plot_grid,
    x_col='time',
    y_col='catalyst',
    z_col='Overall',
    facet_col='temperature',
    aspect=1,
    as_table=True,
    figsize=(3,3)
)
```

Figure 60.10.: The overall desirability surface for the combined outcomes of percent conversion and thermal activity

60.5. Multi-Objective Optimization and Maximizing Desirability

Kuhn indicates that as described by Myers, Montgomery, and Anderson-Cook (2016), desirability can be maximized within a cuboidal region defined by the axial point values. The objective function (`rsmOpt`) utilizes a penalty approach: if a candidate point extends beyond the cuboidal design region, desirability is set to zero. These penalties are implemented in the `rsm_opt` function, which is used to optimize the desirability function. An α value of 1.682 ($\approx (2^k)^(1/4)$ with $k = 3$ in our case), see Montgomery (2001), is used as the limit for both circular and square spaces. After

checking the bounds, predictions for all provided functions are calculated, and the overall desirability is predicted using the `predict` method of the `DOverall` object. The negative desirability is returned to maximize the desirability function.

```
def rsm_opt(x, d_object, prediction_funcs, space="square", alpha=1.682) -> float:
    if space == "circular":
        if np.sqrt(np.sum(np.array(x) ** 2)) > alpha:
            return 0.0
    elif space == "square":
        if np.any(np.abs(np.array(x)) > alpha):
            return 0.0
    else:
        raise ValueError("space must be 'square' or 'circular'")
    predictions = [func(x) for func in prediction_funcs]
    desirability = d_object.predict(np.array([predictions]))
    return -desirability
```

Note: Instead of using the penatlty approach, alternatively the desirability function for box-constraints can be used, see Section 60.2.3.3.2. Furthermore, `scipy.optimize` provides a `bounds` argument for some optimizers to restrict the search space.

Kuhn (2016) used `R`'s `optim` function to implement the Nelder-Mead simplex method (Nelder and Mead 1965; Olsson and Nelson 1975). This direct search method relies on function evaluations without using gradient information. Although this method may converge to a local optimum, it is fast with efficient functions, allowing for multiple feasible region restarts to find the best result. Alternatively, methods like simulated annealing (Bohachevsky 1986), also available in `R`'s `optim` function, might better suit global optimum searches, though they might need parameter tuning for effective performance. We will use the `scipy.optimize.minimize` function to implement the Nelder-Mead simplex method in Python.

Putting the pieces together, the following code segment shows how to create the desirability objects and use them in the optimization process. First, a `search_grid` is created using `numpy`'s `meshgrid` function to generate a grid of restarts points in the design space. For each (restart) point in the search grid, the `rsm_opt` function is called to calculate the desirability for that point. The `conversion_pred` and `activity_pred` functions are used as prediction functions, and the `DOverall` object is created using the individual desirability objects for conversion and activity. The `overallD` (overall desirability) is passed to tne `rsm_opt` function. The `minimize` function from `scipy.optimize` is used to find the optimal parameters that minimize the negative desirability.

```
time = np.linspace(-1.5, 1.5, 5)
temperature = np.linspace(-1.5, 1.5, 5)
catalyst = np.linspace(-1.5, 1.5, 5)
```

```
search_grid = pd.DataFrame(
    np.array(np.meshgrid(time, temperature, catalyst)).T.reshape(-1, 3),
    columns=["time", "temperature", "catalyst"]
)

# List of prediction functions
prediction_funcs = [conversion_pred, activity_pred]

# Individual desirability objects
conversionD = DMax(80, 97)
activityD = DTarget(55, 57.5, 60)

# Desirability object (DOverall)
overallD = DOverall(conversionD, activityD)

# Initialize the best result
best = None

# Perform optimization for each point in the search grid
for i, row in search_grid.iterrows():
    initial_guess = row.values  # Initial guess for optimization

    # Perform optimization using scipy's minimize function
    result = minimize(
        rsm_opt,
        initial_guess,
        args=(overallD, prediction_funcs, "square"),
        method="Nelder-Mead",
        options={"maxiter": 1000, "disp": False}
    )

    # Update the best result if necessary
    # Compare based on the negative desirability
    if best is None or result.fun < best.fun:
        best = result
print("Best Parameters:", best.x)
print("Best Desirability:", -best.fun)
```

```
Best Parameters: [-0.51207663  1.68199987 -0.58609664]
Best Desirability: 0.9425092694688632
```

Using these best parameters, the predicted values for conversion and activity can be calculated as follows:

```
print(f"Conversion pred(x): {conversion_pred(best.x)}")
print(f"Activity pred(x): {activity_pred(best.x)}")
```

```
Conversion pred(x): 95.10150374903237
Activity pred(x): 57.49999992427212
```

We extract the best temperature from the best parameters and remove it from the best parameters for plotting. The `best.x` array contains the best parameters found by the optimizer, where the second element corresponds to the temperature variable.

```
best_temperature = best.x[1]
best_point = np.delete(best.x, 1)
```

Then we set the values of `temperature` to the best temperature in the `plot_grid_df` and recalculate the predicted values for `conversion` and `activity` using the `conversion_pred` and `activity_pred` functions. A copy of the `plot_grid` DataFrame is created, and the `temperature` column is updated with the best temperature value.

```
plot_grid_best = plot_grid.copy()
plot_grid_best["temperature"] = best_temperature
plot_grid_best["conversionPred"] = conversion_pred(plot_grid_best[["time",
     "temperature", "catalyst"]].values.T)
plot_grid_best["activityPred"] = activity_pred(plot_grid_best[["time",
    "temperature", "catalyst"]].values.T)
```

Now we are ready to plot the response surfaces for the best parameters found by the optimizer. The `contourf_plot` function is used to create the contour plots for the response surface models. The `highlight_point` argument is used to highlight the best point found by the optimizer in the contour plots. First, the response surface for the `percent conversion` model is plotted. The temperature variable is fixed at the best value found by the optimizer, see Figure 60.11.

```
contourf_plot(
    plot_grid_best,
    x_col="time",
    y_col="catalyst",
    z_col="conversionPred",
    facet_col="temperature",
    highlight_point=best_point,
)
```

Figure 60.11.: The response surface for the percent conversion model. To plot the model contours, the temperature variable was fixed at the best value found by the optimizer.

Second, the response surface for the `thermal activity` model is plotted. The temperature variable is fixed at the best value found by the optimizer, see Figure 60.12.

```
contourf_plot(
    plot_grid_best,
    x_col="time",
    y_col="catalyst",
    z_col="activityPred",
    facet_col="temperature",
    highlight_point=best_point,
)
```

Figure 60.12.: The response surface for the thermal activity model. To plot the model contours, the temperature variable was fixed at the best value found by the optimizer.

Analysing the Best Values From the Nelder-Mead Optimizer

- Objective function values for the best parameters found by the optimizer are:
  - `conversion` = 95.1
  - `activity` = 57.5
- The best value for the percent conversion should be maximized, as defined in `conversionD = DMax(80, 97)`. Here, we have obtained a value of 95.1, which is close to the maximum value of 97.
- Since we are using the desirabilty function `DTarget`, the values for the `thermal activity` should not be maximized, but should be close to the target. The setting `activityD = DTarget(55, 57.5, 60)`, as defined in Section 60.4.2.2, states that the best value for the `thermal activity` should be close to 57.5 as specified by the user (and not at its maximum). Here, we have obtained a value of 57.5, which is exactly the target value.

An alternative approach to the optimization process is to use a circular design region

instead of a cuboidal design region can be found in the Appendix.

60.6. Surrogate-Model Based Optimization Using Desirability

`spotpython` implements a vectorized function `fun_myer16a()` that computes the two objective functions for `conversion` and `activity`. To illustrate the vectorized evaluation, we will use two input points: the center point of the design space and the best point found by the optimizer from Section 60.5. The `fun_myer16a()` function takes a 2D array as input, where each row corresponds to a different set of parameters. The function returns a 2D array with the predicted values for `conversion` and `activity`.

```
X = np.array([[0, 0, 0], best.x])
y = fun_myer16a(X)
print("Objective function values:")
print(y)
```

```
Objective function values:
[[81.09       59.85      ]
 [95.10150375 57.49999992]]
```

Next, we define the desirability objects. This step is identical to the previous one, where we defined the desirability functions for `conversion` and `activity`. The `DMax` function is used for the `conversion` function, and the `DTarget` function is used for the `activity` function. The `DOverall` function is used to combine the two desirability functions into an overall desirability function. The `DOverall` function takes two arguments: the desirability object for `conversion` and the desirability object for `activity`.

```
from spotdesirability.utils.desirability import DOverall, DMax, DTarget
conversionD = DMax(80, 97)
activityD = DTarget(55, 57.5, 60)
overallD = DOverall(conversionD, activityD)
```

```
conversionD.plot()
```

Figure 60.13.: The desirability function for the conversion outcome.

```
activityD.plot()
```

Figure 60.14.: The desirability function for the activity outcome.

Predicting the desirability for each outcome can also be vectorized. The `predict` method of the desirability objects can take a 2D array as input, where each row corresponds to a different set of parameters. The method returns a 1D array with the predicted desirability values for each set of parameters.

```
conversion_desirability = conversionD.predict(y[:,0])
activity_desirability = activityD.predict(y[:,1])
```

```
print(f"Conversion Desirability: {conversion_desirability}")
print(f"Activity Desirability: {activity_desirability}")
```

```
Conversion Desirability: [0.06411765 0.88832375]
Activity Desirability: [0.06       0.99999997]
```

The `overall_desirability` variable contains the overall desirability values for each set of parameters. The `all=True` argument indicates that we want to return both the individual desirability values and the overall desirability value.

```
overall_desirability = overallD.predict(y, all=True)
```

```
print(f"OverallD: {overall_desirability}")
```

```
OverallD: ([array([0.06411765, 0.88832375]), array([0.06      , 0.99999997])], array(
```

During the surrogate-model based optimization, the argument `all` is set to `False`, because `spotpython` does not need the individual desirability values.

Now we have introduced all elements needed to perform surrogate-model based optimization using desirability functions and the `spotpython` package.

> **!** Maximization and Minimization
>
> - Since `spotpython` uses minimization, but desirability should be maximized, `fun_desirability` is defined to return `1 - overall_desirability`.

```
def fun_desirability(X, **kwargs):
    y = fun_myer16a(X)
    conversionD = DMax(80, 97)
    activityD = DTarget(55, 57.5, 60)
    overallD = DOverall(conversionD, activityD)
    overall_desirability = overallD.predict(y, all=False)
    return 1.0 - overall_desirability
```

We can test the function:

```
X = np.array([[0, 0, 0], best.x])
y = fun_desirability(X)
print("Objective function values:")
print(y)
```

```
Objective function values:
[0.93797534 0.05749073]
```

As expected, the output contains the two overall "1 minus desirability" function values for the center point of the design space and the best point found by the optimizer.

We are now ready to perform the surrogate-model based optimization using desirability functions. The `spotpython` package provides a class `Spot` that implements the surrogate-model based optimization algorithm. The `Spot` class takes the objective function and the control parameters as input. The control parameters define the search space and other settings for the optimization process.

```
fun_control = fun_control_init(
              lower = np.array( [-1.7, -1.7, -1.7]),
              upper = np.array([1.7, 1.7, 1.7]),
              var_name = ["time", "temperature", "catalyst"],
              fun_evals= 50
)
design_control=design_control_init(init_size=15)
S = Spot(fun=fun_desirability,
         fun_control=fun_control,
         design_control=design_control)
S.run()
```

```
spotpython tuning: 0.16080910335362375 [###-------] 32.00%. Success rate: 0.00%
spotpython tuning: 0.16080910335362375 [###-------] 34.00%. Success rate: 0.00%
spotpython tuning: 0.10021706996034485 [####------] 36.00%. Success rate: 33.33%
spotpython tuning: 0.10021706996034485 [####------] 38.00%. Success rate: 25.00%
spotpython tuning: 0.08114826746261061 [####------] 40.00%. Success rate: 40.00%
spotpython tuning: 0.08114826746261061 [####------] 42.00%. Success rate: 33.33%
spotpython tuning: 0.08114826746261061 [####------] 44.00%. Success rate: 28.57%
spotpython tuning: 0.08114826746261061 [#####-----] 46.00%. Success rate: 25.00%
spotpython tuning: 0.07335076989845213 [#####-----] 48.00%. Success rate: 33.33%
spotpython tuning: 0.07335076989845213 [#####-----] 50.00%. Success rate: 30.00%
spotpython tuning: 0.07295476428217884 [#####-----] 52.00%. Success rate: 36.36%
spotpython tuning: 0.07170395572830979 [#####-----] 54.00%. Success rate: 41.67%
spotpython tuning: 0.0715653644886115 [######----] 56.00%. Success rate: 46.15%
spotpython tuning: 0.07148937970083347 [######----] 58.00%. Success rate: 50.00%
spotpython tuning: 0.07139812994344297 [######----] 60.00%. Success rate: 53.33%
spotpython tuning: 0.07139812994344297 [######----] 62.00%. Success rate: 50.00%
spotpython tuning: 0.07139812994344297 [######----] 64.00%. Success rate: 47.06%
spotpython tuning: 0.07139812994344297 [#######---] 66.00%. Success rate: 44.44%
spotpython tuning: 0.07139812994344297 [#######---] 68.00%. Success rate: 42.11%
spotpython tuning: 0.07139812994344297 [#######---] 70.00%. Success rate: 40.00%
spotpython tuning: 0.07139812994344297 [#######---] 72.00%. Success rate: 38.10%
spotpython tuning: 0.07139812994344297 [#######---] 74.00%. Success rate: 36.36%
spotpython tuning: 0.07139812994344297 [########--] 76.00%. Success rate: 34.78%
Using spacefilling design as fallback.
spotpython tuning: 0.07139812994344297 [########--] 78.00%. Success rate: 33.33%
spotpython tuning: 0.07063092791269043 [########--] 80.00%. Success rate: 36.00%
```

```
spotpython tuning: 0.07063092791269043 [########--] 82.00%. Success rate: 34.62%
spotpython tuning: 0.07062979209339726 [########--] 84.00%. Success rate: 37.04%
spotpython tuning: 0.07062979209339726 [#########-] 86.00%. Success rate: 35.71%
spotpython tuning: 0.07062979209339726 [#########-] 88.00%. Success rate: 34.48%
spotpython tuning: 0.07062979209339726 [#########-] 90.00%. Success rate: 33.33%
Using spacefilling design as fallback.
spotpython tuning: 0.07062979209339726 [#########-] 92.00%. Success rate: 32.26%
spotpython tuning: 0.07062979209339726 [#########-] 94.00%. Success rate: 31.25%
Using spacefilling design as fallback.
spotpython tuning: 0.07062979209339726 [##########] 96.00%. Success rate: 30.30%
spotpython tuning: 0.07062979209339726 [##########] 98.00%. Success rate: 29.41%
spotpython tuning: 0.07062979209339726 [##########] 100.00%. Success rate: 28.57% Done

Experiment saved to 000_res.pkl


<spotpython.spot.spot.Spot at 0x168f29940>
```

The progress of the optimization process can be visualized using the `plot_progress` method (Figure 60.15).

```
S.plot_progress(log_y=True)
```

Figure 60.15.: The progress of the surrogate-model based optimization using desirability functions. The y-axis is on a logarithmic scale.

We can analyze the results in detail by accessing the attributes of the `Spot` object directly. The `min_X` attribute contains the best parameters found by the optimizer, and the `min_y` attribute contains the best desirability value. First, we take a look at the desirability values for the best parameters found by the optimizer. The `min_y` attribute

contains the best desirability value. Note, we have to compute 1 minus the `min_y` value, because the `fun_desirability` function returns `1 - overall_desirability`. This results in the following best desirability value:

```
Best Desirability: 0.9293702079066027
```

We can use the `min_X` attribute to calculate the predicted values for `conversion` and `activity` for the best parameters found by the optimizer. Using the `fun_myer16a` function, we can calculate these predicted values.

```
print(f"Best Parameters: {S.min_X}")
print(f"Best Conversion: {best_conversion}")
print(f"Best Activity: {best_activity}")
```

```
Best Parameters: [-0.35514269  1.7        -0.74005161]
Best Conversion: 94.95072777322638
Best Activity: 57.455297317223085
```

Analysing the Best Values from the spotpython Optimizer

- Objective function values for the best parameters found by the optimizer are very close to the values found by the Nelder-Mead optimizer.

Based on the information from the surrogate, which is by default a Kriging model in `spotpython`, we can analyze the importance of the parameters in the optimization process. The `plot_importance` method plots the importance of each parameter, see Figure 60.16.

The `plot_important_hyperparameter_contour` method plots the contour plots for the important parameters. The results are shown in Figure 60.17. The contour plots show the importance of the parameters in the optimization process, which tries to minimize the 1 minus desirability values. Regions with low values present high desirability. Note: These surface plots illustrate how the Kriging surrogate "sees the world" and decides where to sample next. The Kriging model computes the following importance values:

```
time:  100.0
temperature:  3.349151657921023
catalyst:  44.69576345018581
```

Finally, we show a comparison with the response-surface model. Similar to the procedure above, we generate the `plot_grid` DataFrame for the response surface models.

Figure 60.16.: The importance of the parameters in the optimization process

(a) The contour plots for the important parameters

(b)

```
best_x = S.min_X
best_point = np.delete(best_x, 1)
best_temperature = best_x[1]

variables = {
    "time": (-1.7, 1.7),
    "temperature": (best_temperature, best_temperature),
    "catalyst": (-1.7, 1.7)
}

resolutions = {
    "time": 50,
    "temperature": 1,
    "catalyst": 50
}

functions = {
    "conversionPred": conversion_pred,
    "activityPred": activity_pred
}

plot_grid = mo_generate_plot_grid(variables, resolutions, functions)
```

Usintg the `plot_grid` DataFrame, we generate contour plots shown in Figure 60.18 and Figure 60.19. The largest effects in the fitted model are due to the time *times* catalyst interaction and the linear and quadratic effects of catalyst. The Figure 60.19 shows the response surface for the thermal activity model. To plot the model contours, the temperature variable was fixed at four diverse levels. The main effects of time and catalyst have the largest effect on the fitted model.

```
contourf_plot(
    plot_grid,
    x_col="time",
    y_col="catalyst",
    z_col="conversionPred",
    facet_col="temperature",
    highlight_point=best_point,
)
```

Figure 60.18.: The response surface for the percent conversion model. To plot the model contours, the temperature variable was fixed at four diverse levels.

```
contourf_plot(
    plot_grid,
    x_col="time",
    y_col="catalyst",
    z_col="activityPred",
    facet_col="temperature",
    highlight_point=best_point,
)
```

Figure 60.19.: The response surface for the thermal activity model. To plot the model contours, the temperature variable was fixed at four diverse levels.

60.7. Surrogate Model Hyperparameter Tuning

This section compares three different approaches to hyperparameter tuning using the `spotpython` package. The first approach is a single-objective approach, where only the first objective function is used for hyperparameter tuning. The second approach is a weighted multi-objective approach, where a weighted mean of both objective functions is used for hyperparameter tuning. The third approach uses a desirability function to combine the two objective functions into a single objective function. The desirability function is used to maximize the desirability of the two objective functions.

The `spotpython` package provides a method for hyperparameter tuning using a surrogate model. We will extend the single-objective example "Hyperparameter Tuning with spotpython and PyTorch Lightning for the Diabetes Data Set" from the hyperparameter tuning cookbook (Bartz-Beielstein 2023b) in the follwoing way:

Instead of using a single-objective function, which returns the `validation loss` from the neural-network training, we will use a multi-objective function that returns two objectives:

- `validation loss` and

- `number of epochs`.

Clearly, both objectives should be minimized. The `validation loss` should be minimized to get the best model, and the `number of epochs` should be minimized to reduce the training time. However, if the number of training epochs is too small, the model will not be trained properly. Therefore, we will adopt the desirability function for the `number of epochs` accordingly.

```
import os
from math import inf
import warnings
warnings.filterwarnings("ignore")
```

The following process can be used for the hyperparameter tuning of the `Diabetes` data set using the `spotpython` package. The `Diabetes` data set is a regression data set that contains 10 input features and a single output feature. The goal is to predict the output feature based on the input features.

After importing the necessary libraries, the `fun_control` dictionary is set up via the `fun_control_init` function. The `fun_control` dictionary contains

- `PREFIX`: a unique identifier for the experiment
- `fun_evals`: the number of function evaluations
- `max_time`: the maximum run time in minutes
- `data_set`: the data set. Here we use the `Diabetes` data set that is provided by `spotpython`.
- `core_model_name`: the class name of the neural network model. This neural network model is provided by `spotpython`.
- `hyperdict`: the hyperparameter dictionary. This dictionary is used to define the hyperparameters of the neural network model. It is also provided by `spotpython`.
- `_L_in`: the number of input features. Since the `Diabetes` data set has 10 features, `_L_in` is set to 10.
- `_L_out`: the number of output features. Since we want to predict a single value, `_L_out` is set to 1.

The `HyperLight` class is used to define the objective function `fun`. It connects the `PyTorch` and the `spotpython` methods and is provided by `spotpython`. Details can be found in the hyperparameter tuning cookbook (Bartz-Beielstein 2023b) or online https://sequential-parameter-optimization.github.io/Hyperparameter-Tuning-Cookbook/.

60.7.1. The Single-Objective Approach

The simplest way for handling multi-objective results is to simply ignore all but the first objective function. This is done by setting the `fun_mo2so` argument in the `fun_control_init` function to `None`. The `fun_mo2so` argument is used to convert the

multi-objective function to a single-objective function. If it is set to `None`, the first objective function is used as the single-objective function. Since the `None` is also the default, no argument is needed for the single-objective approach.

```
PREFIX="0000_no_mo"
data_set = Diabetes()
fun_control = fun_control_init(
    # do not run, if a result file exists
    force_run=False,
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=10,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)
fun = MoHyperLight().fun
```

```
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
```

The method `set_hyperparameter` allows the user to modify default hyperparameter settings. Here we modify some hyperparameters to keep the model small and to decrease the tuning time.

```
set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,10])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2, 7])

design_control = design_control_init(init_size=20)
print_exp_table(fun_control)
```

```
| name           | type    | default    |   lower |    upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------
|
| l1             | int     | 3          |      3  |    4     | transform_power_2_int |
| epochs         | int     | 4          |      3  |   10     | transform_power_2_int |
| batch_size     | int     | 4          |      4  |   11     | transform_power_2_int |
```

```
| act_fn         | factor | ReLU      |     0   |   5     | None                  |
| optimizer      | factor | SGD       |     0   |   2     | None                  |
| dropout_prob   | float  | 0.01      |     0   |   0.025 | None                  |
| lr_mult        | float  | 1.0       |     0.1 |  10     | None                  |
| patience       | int    | 2         |     2   |   7     | transform_power_2_int |
| batch_norm     | factor | 0         |     0   |   1     | None                  |
| initialization | factor | Default   |     0   |   4     | None                  |
```

Finally, a `Spot` object is created. Calling the method `run()` starts the hyperparameter tuning process.

```
S = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
S.run()
```

```
Result file 0000_no_mo_res.pkl exists. Loading the result.
Loaded experiment from 0000_no_mo_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x326a94e10>
```

Figure 60.20 shows the hyperparameter tuning process. The loss and epochs are plotted versus the function evaluations. The x-axis shows the number of function evaluations, and the y-axis shows the loss and epochs. The loss is plotted in blue, and the epochs are plotted in red. The y-axis is set to a logarithmic scale for better visualization.

```
loss = S.y_mo[:, 0]
epochs = S.y_mo[:, 1]
iterations = np.arange(1, len(loss) + 1)  # Iterations (x-axis)
plt.figure(figsize=(10, 6))
plt.plot(iterations, loss, label="Loss", color="blue", marker="o")
plt.plot(iterations, epochs, label="Epochs", color="red", marker="x")
plt.xlabel("Iterations")
plt.ylabel("Values")
plt.title("Loss and Epochs vs. Iterations")
plt.yscale("log")
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.legend()
plt.show()
```

Figure 60.20.: Results of the hyperparameter tuning process. Loss and epochs are plotted versus the function evaluations.

```
_ = S.print_results()
```

```
min y: 2965.705078125
l1: 3.0
epochs: 9.0
batch_size: 6.0
act_fn: 3.0
optimizer: 1.0
dropout_prob: 0.014940301372259645
lr_mult: 8.554331694855211
patience: 3.0
batch_norm: 0.0
initialization: 3.0
```

Results from the Single-Objective Approach

- The single-objective approach reulsted in a validation loss of 2890 and 1024 ($= 2^{10}$) epochs.

60.7.2. Weighted Multi-Objective Function

The second approach is to use a weighted mean of both objective functions. This is done by setting the `fun_mo2so` argument in the `fun_control_init` function to a custom function that computes the weighted mean of both objective functions. The weights can be adjusted to give more importance to one objective function over the other. Here, we define the function `aggregate` that computes the weighted mean of both objective functions. The first objective function is weighted with 2 and the second objective function is weighted with 0.1.

```
PREFIX="0001_aggregate"

# Weight first objective with 2, second with 1/10
def aggregate(y):
    import numpy as np
    return np.sum(y*np.array([2, 0.1]), axis=1)
fun_control = fun_control_init(
    # do not run, if a result file exists
    force_run=False,
    fun_mo2so=aggregate,
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=10,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)
```

```
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
```

The remaining code is identical to the single-objective approach. The only difference is that the `fun_mo2so` argument is set to the `aggregate` function.

```
set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,10])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2, 7])

design_control = design_control_init(init_size=20)
```

```
S = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
S.run()
```

```
Result file 0001_aggregate_res.pkl exists. Loading the result.
Loaded experiment from 0001_aggregate_res.pkl

<spotpython.spot.spot.Spot at 0x317e37610>
```

```
_ = S.print_results()
```

```
min y: 5756.735546875
l1: 4.0
epochs: 7.0
batch_size: 8.0
act_fn: 2.0
optimizer: 1.0
dropout_prob: 0.015182309694564699
lr_mult: 4.398603126890015
patience: 4.0
batch_norm: 0.0
initialization: 2.0
```

> **Results from the Weighted Multi-Objective Function Approach**
>
> - The weighted multi-objective approach reulsted in a validation loss of `5824` and `64` ($= 2^6$) epochs.
> - Although the number of epochs is smaller than in the single-objective approach, the validation loss is larger.
> - This is an inherent problem of weighted multi-objective approaches, because the deteriination of "good" weights is non-trivial.

60.7.3. Multi-Objective Hyperparameter Tuning With Desirability

60.7.3.1. Setting Up the Desirability Function

The third approach is to use a desirability function to combine the two objective functions into a single objective function. The desirability function is used to maximize the desirability of the two objective functions. The desirability function is defined in the `fun_control_init` function by setting the `fun_mo2so` argument to a custom function that computes the desirability of both objective functions. The desirability function is defined in the following code segment.

```
PREFIX="0002"
data_set = Diabetes()
fun = MoHyperLight().fun
```

```
def desirability(y):
    from spotdesirability.utils.desirability import DOverall, DMin
    lossD = DMin(10, 6000)
    epochsD = DMin(32, 64)
    overallD = DOverall(lossD, epochsD)
    overall_desirability = overallD.predict(y, all=False)
    return 1.0 - overall_desirability
```

```
lossD = DMin(10, 6000)
lossD.plot(xlabel="loss", ylabel="desirability")
```

Figure 60.21.: The desirability function for the loss outcome.

```
epochsD = DMin(32, 64)
epochsD.plot(xlabel="epochs", ylabel="desirability")
```

Figure 60.22.: The desirability function for the epochs outcome.

Note: We have chosen simple desirability functions based on `DMin` for the `validation loss` and `number of epochs`. The usage of these functions might result in large plateaus where the optimizer does not find any improvement. Therefore, we will explore more sophisticated desirability functions in the future. These can easily be implemented with the approach shown in {Section 60.2.3.3.1}.

```
fun_control = fun_control_init(
    # do not run, if a result file exists
    force_run=False,
    fun_mo2so=desirability,
    device="cpu",
    PREFIX=PREFIX,
```

```
    fun_evals=inf,
    max_time=10,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)

set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,10])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2, 7])

design_control = design_control_init(init_size=20)

S = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
S.run()
```

```
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
Result file 0002_res.pkl exists. Loading the result.
Loaded experiment from 0002_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x326bfd590>
```

```
_ = S.print_results()
```

```
min y: 0.26662821493965283
l1: 4.0
epochs: 5.0
batch_size: 7.0
act_fn: 2.0
optimizer: 1.0
dropout_prob: 0.0
lr_mult: 4.341547337583857
patience: 6.0
batch_norm: 0.0
initialization: 2.0
```

```
print(f"S.y_mo.shape: {S.y_mo.shape}")
print(f"min loss: {np.nanmin(S.y_mo[:,0]):.2f}")
print(f"min epochs: {np.nanmin(S.y_mo[:,1])}")
# print unique values of S.y_mo[:,1]
print(f"unique epochs values: {np.unique(S.y_mo[:,1])}")
```

```
S.y_mo.shape: (113, 2)
min loss: 2778.37
min epochs: 8.0
unique epochs values: [   8.   16.   32.   64.  128.  256.  512. 1024.]
```

Results from the Desirability Function Approach

- The desirability multi-objective approach reulsted in a validation loss of 2960 and 32 ($= 2^5$) epochs.
- The number of epochs is much smaller than in the single-objective approach, and the validation loss is in a similar range.
- This illustrates the applicability of desirability functions for multi-objective optimization.

60.7.3.2. Pareto Front

The following two figures show the Pareto front for the multi-objective optimization problem. Figure 60.23 shows the Pareto front for the multi-objective optimization problem. The y-axis uses a logarithmic scal. Figure 60.24 shows the Pareto front for the multi-objective optimization problem. The points with loss > 1e5 are removed from the plot (due to this removal, the indices of the pointsin the plot change). The x-axis uses a double-logarithmic scale, whereas the y-axis is set to a logarithmic scale for better visualization.

The best point has the following values:

```
# generate a dataframe with S.y and S.y_mo
df = pd.DataFrame(S.y_mo, columns=["loss", "epochs"])
df["y"] = S.y
df_min = df.loc[df["y"].idxmin()]
print(f"min y: {df_min['y']}")
print(f"loss: {df_min['loss']}")
print(f"epochs: {df_min['epochs']}")
best_point = np.array([df_min["loss"], df_min["epochs"]])
print(f"best_point: {best_point}")
```

```
min y: 0.26662821493965283
loss: 2778.373291015625
epochs: 32.0
best_point: [2778.37329102   32.        ]
```

```
y_orig = S.y_mo
df_z = pd.DataFrame(y_orig, columns=["loss", "epochs"])
df_z_sel = df_z.dropna()
target_names = ["loss (log-log)", "epochs"]
combinations=[(0,1)]
plot_mo(y_orig=df_z_sel, target_names=target_names, combinations=combinations, pareto="min", pareto_
```

Figure 60.23.: Pareto front for the multi-objective optimization problem.

```
# remove loss values larger than 1e5 from the y_orig array
y_orig = S.y_mo
df_z = pd.DataFrame(y_orig, columns=["loss", "epochs"])
# remove rows with loss > 1e5
df_z = df_z[df_z["loss"] < 1e5]
df_z_sel = df_z.dropna()
target_names = ["loss", "epochs (log)"]
combinations=[(0,1)]
plot_mo(y_orig=df_z_sel, target_names=target_names, combinations=combinations, pareto="min", pareto_
```

Figure 60.24.: Pareto front for the multi-objective optimization problem. Points with loss > 1e5 removed. The Pareto points are identical to the points in the previous plot. Their indices changed due to the removal of points.

60.8. Conclusion

In this article, we have shown how to use the `spotdesitability` package to perform three different multi-objective optimization tasks using desirability functions: RSM (Myers, Montgomery, and Anderson-Cook 2016), surrogate model based optimization (Santner, Williams, and Notz 2003), and hyperparameter tuning (Bartz et al. 2022). The `spotdesirability` package is a `Python` implementation of the `R` `desirability` package (Kuhn 2016). We have demonstrated how to define desirability functions for different types of objectives, including maximization, minimization, and target objectives.

Although the desirability function approach is one of the most widely used methods in industry for the optimization of multiple response processes ("NIST/SEMATECH e-Handbook of Statistical Methods" 2021), it is rarely used in hyperparameter tuning. To fill this gap, we have shown how to use the `spotdesirability` package in combination with the `spotpython` package to perform hyperparameter tuning using desirability functions. The `spotpython` package provides a convenient way to perform surrogate model based optimization, and the `spotdesirability` package allows us to define desirability functions for different types of objectives. First results are promising, but more research is needed to evaluate the performance of the desirability function approach in hyperparameter tuning.

60.9. Appendix

60.9.1. Alternative Optimization Approach Using a Circular Design Region

Kuhn also suggests alternatively maximizing desirability such that experimental factors are constrained within a spherical design region with a radius equivalent to the axial point distance:

```
# Initialize the best result
best = None

# Perform optimization for each point in the search grid
for i, row in search_grid.iterrows():
    initial_guess = row.values  # Initial guess for optimization

    # Perform optimization using scipy's minimize function
    result = minimize(
        rsm_opt,
        initial_guess,
        args=(overallD, prediction_funcs, "circular"),
        method="Nelder-Mead",
        options={"maxiter": 1000, "disp": False}
    )

    # Update the best result if necessary
    # Compare based on the negative desirability
    if best is None or result.fun < best.fun:
        best = result
print("Best Parameters:", best.x)
print("Best Desirability:", -best.fun)
```

```
Best Parameters: [-0.50970524  1.50340746 -0.55595672]
Best Desirability: 0.8581520815997857
```

Using these best parameters, the predicted values for conversion and activity can be calculated as follows:

```
print(f"Conversion pred(x): {conversion_pred(best.x)}")
print(f"Activity pred(x): {activity_pred(best.x)}")
```

```
Conversion pred(x): 92.51922540231372
Activity pred(x): 57.499999903209876
```

```
best_temperature = best.x[1]
# remove the temperature variable from the best parameters
best_point = np.delete(best.x, 1)
# set the values of temperature to the best temperature in the df
# and recalculate the predicted values
plot_grid_best = plot_grid.copy()
plot_grid_best["temperature"] = best_temperature
# Recalculate the predicted values for conversion and activity
plot_grid_best["conversionPred"] = conversion_pred(plot_grid_best[["time",
     "temperature", "catalyst"]].values.T)
plot_grid_best["activityPred"] = activity_pred(plot_grid_best[["time",
    "temperature", "catalyst"]].values.T)
```

```
contourf_plot(
    plot_grid_best,
    x_col="time",
    y_col="catalyst",
    z_col="conversionPred",
    facet_col="temperature",
    highlight_point=best_point,
)
```

Figure 60.25.: The response surface for the percent conversion model. To plot the model contours, the temperature variable was fixed at the best value found by the optimizer.

```
contourf_plot(
    plot_grid_best,
    x_col="time",
    y_col="catalyst",
    z_col="activityPred",
    facet_col="temperature",
    highlight_point=best_point,
)
```

Figure 60.26.: The response surface for the thermal activity model. To plot the model contours, the temperature variable was fixed at the best value found by the optimizer.

Kuhn (2016) comments that the process converges to relative sub-optimual values. He suggests that using a radius of 2 achieves an overall desirability equal to one, even if the solution slightly extrapolates beyond the design region.

Part XI.

Lernmodule

61. Lernmodul: Aircraft Wing Weight Example (AWWE)

61.1. Einleitung

Die Aircraft Wing Weight Example (AWWE) Funktion dient dazu, das Gewicht eines unlackierten Leichtflugzeugflügels als Funktion verschiedener Design- und Betriebsparameter zu verstehen. Diese Sektion basiert auf Kapitel 1.3 “A ten-variable weight function” in “Engineering Design via Surrogate Modelling: A Practical Guide” von Forrester, Sóbester und Keane (2008) (Forrester, Sóbester, and Keane 2008).

Die in diesem Kontext verwendete Gleichung ist keine reale Computersimulation, wird aber zu Illustrationszwecken als solche behandelt. Sie stellt eine mathematische Gleichung dar, deren funktionale Form durch die Kalibrierung bekannter physikalischer Beziehungen an Kurven aus bestehenden Flugzeugdaten abgeleitet wurde, wie in Raymer (2006) referenziert (Raymer 2006). Im Wesentlichen fungiert sie als Ersatzmodell (*surrogate*) für tatsächliche Messungen des Flugzeuggewichts. Die Studie nimmt ein Cessna C172 Skyhawk Flugzeug als Referenzpunkt.

61.2. Die AWWE-Gleichung

Die ursprüngliche AWWE-Gleichung beschreibt das Gewicht (W) eines unlackierten Leichtflugzeugflügels basierend auf neun Design- und Betriebsparametern:

$$W = 0.036 S_W^{0.758} \times W_{fw}^{0.0035} \left(\frac{A}{\cos^2 \Lambda}\right)^{0.6} \times q^{0.006} \times \lambda^{0.04} \times \left(\frac{100 R_{tc}}{\cos \Lambda}\right)^{-0.3} \times (N_z W_{dg})^{0.49}$$

Im Rahmen einer Übung wird die Formel um das Lackgewicht (W_p) erweitert. Die aktualisierte Gleichung lautet dann:

$$W = 0.036 S_W^{0.758} \times W_{fw}^{0.0035} \times \left(\frac{A}{\cos^2 \Lambda}\right)^{0.6} \times q^{0.006} \times \lambda^{0.04} \times \left(\frac{100 R_{tc}}{\cos \Lambda}\right)^{-0.3} \times (N_z W_{dg})^{0.49} + S_w W_p$$

61.3. Eingabegrößen (Parameter)

Die AWWE-Funktion verwendet zehn Design- und Betriebsparameter.

Parameterbeschreibungen:

- S_W **(Flügelfläche):** Die projizierte Fläche des Flügels. Sie ist ein Haupttreiber für den Auftrieb, aber auch für das Gewicht und den Luftwiderstand.
- W_{fw} **(Treibstoffgewicht im Flügel):** Das Gewicht des im Flügel untergebrachten Treibstoffs.
- A **(Aspektverhältnis):** Das Verhältnis von Spannweite zum Quadrat zur Flügelfläche. Eine hohe Streckung (lange, schmale Flügel) ist aerodynamisch effizient, führt aber zu höheren Biegebelastungen und damit potenziell zu mehr Gewicht.
- Λ **(Viertel-Chord-Pfeilung):** Der Winkel, um den der Flügel nach hinten geneigt ist. Die Pfeilung ist entscheidend für das Verhalten bei hohen Geschwindigkeiten, beeinflusst aber auch die Strukturlasten.
- q **(Dynamischer Druck im Reiseflug):** Ein Maß für die aerodynamische Belastung, abhängig von Luftdichte und Fluggeschwindigkeit.
- λ **(Konizität):** Das Verhältnis der Flügeltiefe an der Spitze zur Flügeltiefe an der Wurzel. Es beeinflusst die Auftriebsverteilung und die Struktureffizienz.
- R_{tc} **(Relative Profildicke):** Das Verhältnis der maximalen Dicke des Flügelprofils zu seiner Tiefe (Profilsehne). Dickere Profile bieten mehr Platz für Struktur und Treibstoff, erzeugen aber mehr Luftwiderstand.
- N_z **(Ultimativer Lastfaktor):** Der maximale Lastvielfache (g-Kraft), dem die Flügelstruktur standhalten muss, ohne zu versagen. Ein entscheidender Parameter für die strukturelle Auslegung.
- W_{dg} **(Flugdesign-Bruttogewicht):** Das maximale Gewicht, für das das Flugzeug ausgelegt ist.
- W_p **(Lackgewicht):** Das Gewicht der Lackierung pro Flächeneinheit.

Die Parameter sind in der folgenden Tabelle mit ihren Symbolen, Beschreibungen sowie Basis-, Minimal- und Maximalwerten aufgeführt:

Symbol	Parameter	Basiswert	Minimum	Maximum
S_W	Flügelfläche (ft^2)	174	150	200
W_{fw}	Gewicht des Treibstoffs im Flügel (lb)	252	220	300
A	Aspektverhältnis	7.52	6	10
Λ	Viertel-Chord-Pfeilung (deg)	0	-10	10
q	Dynamischer Druck bei Reisegeschwindigkeit (lb/ft^2)	34	16	45
λ	Konizität	0.672	0.5	1
R_{tc}	Profil-Dicken-Sehnen-Verhältnis	0.12	0.08	0.18
N_z	Ultimativer Lastfaktor	3.8	2.5	6

Symbol	Parameter	Basiswert	Minimum	Maximum
W_{dg}	Flugdesign-Bruttogewicht (lb)	2000	1700	2500
W_p	Lackgewicht (lb/ft^2)	0.064	0.025	0.08

61.4. Ausgabegröße

Die Ausgabegröße der AWWE-Funktion ist das berechnete Gewicht des Flugzeugflügels in Pfund (lb), das Flügelgewicht (W).

61.5. Analyse von Effekten, Interaktionen und Wichtigkeit

61.5.1. Mathematische Eigenschaften und Ziele

Die mathematischen Eigenschaften der AWWE-Gleichung zeigen, dass die Reaktion des Modells in Bezug auf ihre Eingaben hochgradig nichtlinear ist. Obwohl die Anwendung des Logarithmus zur Vereinfachung von Gleichungen mit komplexen Exponenten üblich ist, bleibt die Reaktion selbst bei Modellierung des Logarithmus aufgrund des Vorhandenseins trigonometrischer Terme nichtlinear. Angesichts der Kombination aus Nichtlinearität und hoher Eingabedimension sind einfache lineare und quadratische Response-Surface-Approximationen für diese Analyse wahrscheinlich unzureichend.

Die primären Ziele der Studie umfassen das tiefgreifende Verständnis der Input-Output-Beziehungen und die Optimierung. Insbesondere besteht ein Interesse daran, das Gewicht des Flugzeugs zu minimieren, wobei bestimmte Einschränkungen, wie eine notwendige Nicht-Null-Flügelfläche für die Flugfähigkeit, berücksichtigt werden müssen. Eine globale Perspektive und der Einsatz flexibler Modellierung sind für solche (eingeschränkten) Optimierungsszenarien unerlässlich.

61.5.2. Wichtigkeit der Variablen und deren Effekte

Die Analyse des “AWWE Landscape” und spezifischer Plot-Interpretationen liefert wichtige Einblicke in die Auswirkungen und Interaktionen der einzelnen Designparameter auf das Flügelgewicht:

61.5.2.1. Dominante Variablen (wichtigste Einflussfaktoren)

Die wichtigsten Variablen, die einen signifikanten Einfluss auf das Flügelgewicht haben, sind:

- **Ultimativer Lastfaktor (N_z):** Dieser Faktor bestimmt die Größe der maximalen aerodynamischen Last auf den Flügel und ist sehr aktiv, da er stark mit anderen Variablen interagiert.
- **Flügelfläche (S_W):** Ein entscheidender Parameter für das Gesamtgewicht.
- **Flugdesign-Bruttogewicht (W_{dg}):** Hat ebenfalls einen sehr starken Einfluss auf das Flügelgewicht.

Diese Variablen zeigen in Screening-Studien sowohl große zentrale Tendenzwerte als auch hohe Standardabweichungen, was auf starke direkte Effekte und komplexe Interaktionen hindeutet.

61.5.2.2. Interaktionen zwischen Variablen

- **Interaktion von N_z (Ultimativer Lastfaktor) und A (Aspektverhältnis):** Dies ist ein klassisches Beispiel für eine Interaktion, die zu einem schweren Flügel führt, insbesondere bei hohen Aspektverhältnissen und großen G-Kräften. Diese Beobachtung steht im Einklang mit der Designphilosophie von Kampfflugzeugen, die aufgrund der Notwendigkeit, hohen G-Kräften standzuhalten, keine effizienten, segelflugzeugähnlichen Flügel aufweisen können. Die leichte Krümmung der Konturlinien im N_z vs. A-Plot deutet auf diese Interaktion hin. Der darstellte Ausgabebereich in diesem Plot (von ca. 160 bis 320) deckt fast den gesamten Bereich der Ausgaben ab, die aus verschiedenen Eingabeeinstellungen im vollen 9-dimensionalen Eingaberaum beobachtet wurden.
- **Aspektverhältnis (A) und Profil-Dicken-Sehnen-Verhältnis (R_{tc}):** Diese beiden Variablen zeigen nichtlineare Interaktionen. Ihre hohen Standardabweichungen in Screening-Studien weisen auf signifikantes nichtlineares Verhalten und Interaktionen mit anderen Variablen hin.

61.5.2.3. Variablen mit geringem Einfluss

Einige Variablen haben, insbesondere innerhalb der untersuchten Bereiche, nur einen geringen Einfluss auf das Flügelgewicht:

- **Dynamischer Druck (q):** Hat innerhalb des gewählten Bereichs nur begrenzte Auswirkungen auf das Flügelgewicht. Dies kann im Wesentlichen als Darstellung unterschiedlicher Reiseflughöhen bei gleicher Geschwindigkeit interpretiert werden.
- **Konizität (λ):** Zeigt geringen Einfluss.

- **Viertel-Chord-Pfeilung (Λ):** Hat ebenfalls geringen Einfluss, insbesondere innerhalb des engen Bereichs von -10° bis +10°, der typisch für leichte Flugzeuge ist.
- **Lackgewicht (W_p):** Wie zu erwarten, trägt das Lackgewicht nur wenig zum Gesamtgewicht des Flügels bei.

Variablen mit minimalem Einfluss gruppieren sich in Screening-Plots nahe dem Ursprung, was auf ihren geringen Einfluss auf die Zielfunktion hinweist.

61.5.2.4. Variablen mit moderatem, linearem Einfluss

- **Flügelkraftstoffgewicht (W_{fw}):** Während W_{fw} immer noch nahe dem Nullpunkt liegt, zeigt es in Screening-Studien eine leicht größere zentrale Tendenz bei sehr geringer Standardabweichung. Dies deutet auf eine moderate Bedeutung hin, aber auf minimale Beteiligung an Interaktionen mit anderen Variablen. Der Vergleich der Konizität (λ) und des Kraftstoffgewichts (W_{fw}) zeigt, dass keiner der beiden Inputs das Flügelgewicht stark beeinflusst. λ hat dabei einen geringfügig größeren Effekt, der weniger als 4 Prozent der Gewichtsspanne ausmacht, die im $A \times N_z$-Diagramm beobachtet wurde. Eine Interaktion ist zwischen λ und W_{fw} nicht erkennbar.

61.5.3. Expertenwissen und Screening-Studien

Flugzeugkonstrukteure wissen, dass das Gesamtgewicht des Flugzeugs und die Flügelfläche auf ein Minimum reduziert werden müssen. Letztere wird oft durch Beschränkungen wie die erforderliche Strömungsabrissgeschwindigkeit, die Landestrecke oder die Wendegeschwindigkeit vorgegeben. Die Anforderung an einen hohen ultimativen Lastfaktor (N_z) führt unweigerlich zu der Notwendigkeit robuster, schwerer Flügel.

Die Durchführung von Screening-Studien, oft mittels der Morris-Methode, ist von entscheidender Bedeutung, um die Anzahl der Designvariablen zu minimieren, bevor die Zielfunktion modelliert wird. Diese Methode hilft, die Dimensionalität zu reduzieren, ohne die Analyse zu beeinträchtigen. Ein großer Wert des zentralen Trends in den Ergebnissen der Morris-Methode deutet darauf hin, dass eine Variable einen signifikanten Einfluss auf die Zielfunktion über den Designbereich hinweg hat. Eine große Streuung hingegen legt nahe, dass die Variable in Interaktionen involviert ist oder zur Nichtlinearität der Funktion beiträgt. Die Visualisierung dieser Ergebnisse (Stichprobenmittelwerte und Standardabweichungen) hilft dabei, die wichtigsten Variablen zu identifizieren und zu erkennen, ob ihre Effekte linear sind oder Interaktionen beinhalten.

61.6. Zusatzmaterialien

Interaktive Webseite

- Eine interaktive Webseite zum Thema **Aircraft Wing Weight Example** ist hier zu finden: Aircraft Wing Weight Interaktiv.

Audiomaterial

- Eine Audio zum Thema **Aircraft Wing Weight Example** ist hier zu finden: Aircraft Wing Weight Audio.

Jupyter-Notebook

- Das Jupyter-Notebook für dieses Lernmodul ist auf GitHub im Hyperparameter-Tuning-Cookbook Repository verfügbar.

62. Lernmodul: Versuchspläne (Sampling-Pläne) für Computerexperimente

Dieses Dokument beschreibt die grundlegenden Ideen von Versuchsplänen, die in der Konzeption und Analyse von Computerexperimenten verwendet werden. Es stützt sich auf Kapitel 1 aus Forresters Buch "Engineering Design via Surrogate Modelling" (Forrester, Sóbester, and Keane 2008) und Kapitel 4 des "Hyperparameter Tuning Cookbook" (Bartz-Beielstein 2023b).

62.1. Einführung

Definition 62.1 (Sampling-Plan)**.** m Kontext von Computerexperimenten bezieht sich der Begriff **Sampling-Plan** auf die Menge der Eingabewerte, beispielsweise X, an denen der Computercode evaluiert wird.

Das übergeordnete Ziel eines Sampling-Plans ist es, den Eingaberaum effizient zu erkunden, um das Verhalten eines Computercodes zu verstehen und ein Surrogatmodell zu erstellen, das das Verhalten des Codes genau abbildet. Traditionell wurde die Response Surface Methodology (RSM) zur Gestaltung von Sampling-Plänen für Computerexperimente verwendet, wobei Punkte mittels eines rechteckigen Rasters oder eines faktoriellen Designs generiert wurden.

In jüngerer Zeit hat sich jedoch Design and Analysis of Computer Experiments (DACE) als flexiblerer und leistungsfähigerer Ansatz für die Gestaltung von Sampling-Plänen etabliert. Der Prozess umfasst:

- Die Abtastung diskreter Beobachtungen.
- Die Verwendung dieser Abtastungen zur Konstruktion einer Approximation $\hat{f}$.
- Die Sicherstellung, dass das Surrogatmodell wohlformuliert ist, d.h., es ist mathematisch gültig und kann Vorhersagen effektiv verallgemeinern.

Ein Sampling-Plan

$$X = \left\{x^{(i)} \in D | i = 1, \dots, n\right\}$$

bestimmt die räumliche Anordnung der Beobachtungen.

62.2. Der "Fluch der Dimensionen"

62.2.1. Das Volumen in hochdimensionalen Räume

Ein wesentliches Problem bei der Gestaltung von Sampling-Plänen, insbesondere in hochdimensionalen Räumen, ist der sogenannte **"Fluch der Dimensionen"**. Dieses Phänomen beschreibt, wie das Volumen des Eingaberaums exponentiell mit der Anzahl der Dimensionen zunimmt, was es schwierig macht, den Raum ausreichend abzutasten. Wenn eine bestimmte Vorhersagegenauigkeit durch die Abtastung eines eindimensionalen Raums an n Stellen erreicht wird, sind n^k Beobachtungen erforderlich, um die gleiche Abtastdichte in einem k-dimensionalen Raum zu erzielen.

Um diesem Problem zu begegnen, kann man entweder die Bereiche der Variablen begrenzen, sodass die zu modellierende Form ausreichend einfach ist, um aus sehr spärlichen Daten approximiert zu werden, oder viele Designwerte auf sinnvollen Werten festlegen und jeweils nur mit wenigen Variablen arbeiten.

62.2.2. Volumen einer Kugel in hohen Dimensionen

Remark 62.1 (Volumen einer Kugel in hohen Dimensionen). Unsere Intuition legt nahe, dass das Volumen einer Kugel mit steigender Dimension immer weiter zunimmt. Das Gegenteil ist der Fall: Das Volumen einer Einheitskugel (Radius = 1) wächst nur bis zur fünften Dimension und beginnt dann stetig und unaufhaltsam gegen null zu schrumpfen. Der Grund: In hohen Dimensionen befindet sich der Großteil des Volumens eines umschließenden Hyperwürfels in dessen „Ecken", sodass für die eingeschriebene Kugel kaum noch Raum bleibt.

Remark 62.1 beschreibt eines der bekanntesten Paradoxa der hochdimensionalen Geometrie. Es widerspricht unserer in drei Dimensionen geschulten Intuition, lässt sich aber sowohl mathematisch als auch geometrisch gut erklären.

Remark 62.1 lässt sich auf zwei Arten veranschaulichen: den mathematischen Grund, der sich aus der Volumenformel ergibt, und den geometrischen Grund, der das Verhältnis zwischen einer Hypersphäre und einem sie umschließenden Hyperwürfel betrifft.

62.2.3. Der mathematische Grund: Ein Wettlauf zweier Funktionen

Unsere Intuition, dass das Volumen mit der Dimension wachsen sollte, ist anfangs korrekt. Das Volumen einer Einheitskugel (Radius R=1) nimmt von Dimension 1 (eine Strecke der Länge 2) über Dimension 2 (ein Kreis mit Fläche) bis Dimension 3 (eine Kugel mit Volumen $4\pi/3$ tatsächlich zu. Dieser Trend setzt sich bis zur fünften Dimension fort.

Der Grund für den anschließenden Rückgang liegt in der Formel für das Volumen einer n-dimensionalen Kugel:

$$V_(R) = \frac{\pi^{n/2} R^n}{\Gamma(\frac{n}{2}+1)},$$

Hier kämpfen zwei Terme gegeneinander:

- Der Zähler ($\pi^{n/2} R^n$): Dieser Term wächst mit zunehmender Dimension n exponentiell an. Er ist verantwortlich für unsere Intuition und das anfängliche Wachstum des Volumens.
- Der Nenner ($\Gamma(\frac{n}{2}+1)$): DieserTermenth enthält die Gammafunktion(Γ), eine Verallgemeinerung der Fakultätsfunktion (z. B. $\Gamma(x+1) = x \cdot \Gamma(x)$). Die Gammafunktion wächst noch schneller als der Zähler. Bei niedrigen Dimensionen dominiert das Wachstum des Zählers, und das Volumen der Hypersphäre nimmt zu. Ab Dimension 5 überholt jedoch das extrem schnelle Wachstum der Gammafunktion im Nenner das Wachstum des Zählers, was dazu führt, dass das Gesamtvolumen wieder abnimmt und für $n \to \infty$ gegen null strebt. Das Maximum des Volumens einer Einheitskugel liegt bei etwa 5,26 Dimensionen.

62.2.4. Der geometrische Grund: Die Dominanz der Ecken

Eine anschaulichere Erklärung liefert die Betrachtung eines Hyperwürfels, der die Hypersphäre umschließt. Stellen Sie sich eine Einheitskugel (Radius 1) vor, die perfekt in einen Hyperwürfel mit der Kantenlänge 2 passt.

- In 2 Dimensionen (Kreis im Quadrat): Der Kreis füllt einen erheblichen Teil der Fläche des Quadrats aus (ca. 78,5 %). Die Ecken des Quadrats machen nur einen kleinen Teil der Gesamtfläche aus.
- In 3 Dimensionen (Kugel im Würfel): Die Kugel füllt bereits einen kleineren Anteil des Würfelvolumens aus (ca. 52,3 %). Die Ecken des Würfels beanspruchen relativ mehr Raum.

Mit steigender Dimension n geschehen zwei Dinge, die diesen Effekt dramatisch verstärken:

- Die Anzahl der Ecken explodiert: Ein n-dimensionaler Würfel hat 2^n Ecken. Ein 10-dimensionaler Würfel hat bereits 1024 Ecken, ein 20-dimensionaler über eine Million. Die Ecken entfernen sich vom Zentrum: Der Abstand vom Zentrum eines Würfels zu den Mittelpunkten seiner Seitenflächen (wo die eingeschriebene Kugel ihn berührt) bleibt konstant (hier: 1). Der Abstand zu den Ecken wächst jedoch mit der Dimension (er beträgt $\sqrt{n}$ für einen n-dimensionalen Würfel, z.B. $\sqrt{10} \approx 3.16$ für einen 10-dimensionalen Würfel und $\sqrt{20} \approx 4.47$ für einen 20-dimensionalen Würfel). Das bedeutet, dass in hohen Dimensionen fast das gesamte Volumen des Hyperwürfels in seine zahlreichen, weit vom Zentrum entfernten Ecken gedrängt wird. Die eingeschriebene Hypersphäre, die in der

Mitte "gefangen" ist, kann diese Ecken nicht erreichen. Ihr Volumen wird im Vergleich zum Volumen des Hyperwürfels, der "nur noch aus Ecken besteht", verschwindend gering.

Zusammenfassend lässt sich Remark 62.1 wie folgt erklärn: Einerseits gilt mathematisch, dass die Gammafunktion im Nenner der Volumenformel schneller wächst als der Potenzterm im Zähler. Andererseits lässt sich geometrisch zeigen, dass in hohen Dimensionen das Volumen eines Würfels fast vollständig in seinen Ecken konzentriert ist, wodurch für die eingeschriebene Kugel im Zentrum kaum noch Volumen übrig bleibt.

62.3. Entwurf von Vorab-Experimenten (Screening)

Bevor die Zielfunktion f modelliert wird, ist es entscheidend, die Anzahl der Designvariablen $(x_1, x_2, \dots, x_k)$ zu minimieren. Dieser Prozess wird als **Screening** bezeichnet und zielt darauf ab, die Dimensionalität zu reduzieren, ohne die Analyse zu beeinträchtigen. Der Morris-Algorithmus ist hierfür eine beliebte Methode, da er lediglich annimmt, dass die Zielfunktion deterministisch ist.

Morris' Methode zielt darauf ab, die Parameter der Verteilung von Elementareffekten zu schätzen, die mit jeder Variablen verbunden sind. Ein großes Maß an zentraler Tendenz (Mittelwert) deutet auf eine Variable mit wichtigem Einfluss auf die Zielfunktion hin, während ein großes Maß an Streuung auf Wechselwirkungen und/oder Nichtlinearität der Funktion in Bezug auf die Variable hinweist.

Definition 62.2 (Elementareffekt)**.** Für einen gegebenen Basiswert $x \in D$ bezeichne $d_i(x)$ den Elementareffekt von x_i, wobei:

$$d_i(x) = \frac{f(x_1, \dots, x_i + \Delta, \dots, x_k) - f(x_1, \dots, x_i - \Delta, \dots, x_k)}{2\Delta}, \quad i = 1, \dots, k,$$

wobei Δ die Schrittweite ist, definiert als der Abstand zwischen zwei benachbarten Levels im Raster.

Um die Effizienz zu gewährleisten, sollte der vorläufige Sampling-Plan X so gestaltet sein, dass jede Evaluierung der Zielfunktion f zur Berechnung von zwei Elementareffekten beiträgt. Zusätzlich sollte der Sampling-Plan eine bestimmte Anzahl (z.B. r) von Elementareffekten für jede Variable liefern.

Die `spotpython`-Bibliothek bietet eine Python-Implementierung zur Berechnung der Morris-Screening-Pläne. Die Funktion `screeningplan()` generiert einen Screening-Plan, indem sie die Funktion `randorient()` r-mal aufruft, um r zufällige Orientierungen zu erstellen.

Ein Beispiel aus dem `Hyperparameter-Tuning-Cookbook` demonstriert die Analyse der Variablenwichtigkeit für das `Aircraft Wing Weight Example`.

```python
import numpy as np
from spotpython.utils.effects import screeningplan

# Beispielparameter
k = 3  # Anzahl der Designvariablen (Dimensionen)
p = 3  # Anzahl der Levels im Raster für jede Variable
xi = 1 # Ein Parameter zur Berechnung der Schrittweite Delta
r = 25 # Anzahl der Elementareffekte pro Variable

# Generieren des Screening-Plans
X = screeningplan(k=k, p=p, xi=xi, r=r)
print(f"Form des generierten Screening-Plans: {X.shape}")
```

```
Form des generierten Screening-Plans: (100, 3)
```

> **Hinweis**
>
> - Der Code generiert jedes Mal einen leicht unterschiedlichen Screening-Plan, da er zufällige Orientierungen der Abtastmatrix verwendet*.

In der Praxis können durch Screening gewonnene Läufe für den eigentlichen Modellierungsschritt wiederverwendet werden, insbesondere wenn die Zielfunktion sehr teuer zu evaluieren ist. Dies ist am effektivsten, wenn sich Variablen als völlig inaktiv erweisen, doch da dies selten der Fall ist, muss ein Gleichgewicht zwischen der Wiederverwendung teurer Simulationsläufe und der Einführung potenziellen Rauschens in das Modell gefunden werden.

62.4. Entwurf eines umfassenden Sampling-Plans

Ziel ist es, einen Sampling-Plan zu entwerfen, der eine gleichmäßige Verteilung der Punkte gewährleistet, um eine einheitliche Modellgenauigkeit im gesamten Designraum zu erreichen. Ein solcher Plan wird als **raumfüllend** bezeichnet.

62.4.1. Stratifikation

Der einfachste Weg, einen Designraum gleichmäßig abzutasten, ist ein rechteckiges Gitter von Punkten, die sogenannte **vollfaktorielle Abtasttechnik**.

```python
import numpy as np
from spotpython.utils.sampling import fullfactorial
q = [3, 2]
edges = 1 # Punkte sind gleichmäßig von Rand zu Rand verteilt
X_full_factorial = fullfactorial(q, edges)
print(f"Vollfaktorieller Plan (q={q}, edges={edges}):\n{X_full_factorial}")
print(f"Form des vollfaktoriellen Plans: {X_full_factorial.shape}")
```

```
Vollfaktorieller Plan (q=[3, 2], edges=1):
[[0.  0. ]
 [0.  1. ]
 [0.5 0. ]
 [0.5 1. ]
 [1.  0. ]
 [1.  1. ]]
Form des vollfaktoriellen Plans: (6, 2)
```

Allerdings hat dieser Ansatz zwei wesentliche Einschränkungen:

- **Beschränkte Designgrößen:** Die Methode funktioniert nur für Designs, bei denen die Gesamtpunktzahl n als Produkt der Anzahl der Levels in jeder Dimension ausgedrückt werden kann ($n = q_1 \times q_2 \times \cdots \times q_k$).
- **Überlappende Projektionen:** Wenn die Abtastpunkte auf einzelne Achsen projiziert werden, können sich Punktsätze überlappen, was die Effektivität des Sampling-Plans reduziert und zu einer ungleichmäßigen Abdeckung führen kann.

Um die Gleichmäßigkeit der Projektionen für einzelne Variablen zu verbessern, kann der Bereich dieser Variablen in gleich große "Bins" unterteilt werden, und innerhalb dieser Bins können gleich große zufällige Teilstichproben generiert werden. Dies wird als **geschichtete Zufallsstichprobe** bezeichnet. Eine Erweiterung dieser Idee auf alle Dimensionen führt zu einem geschichteten Sampling-Plan, der üblicherweise mittels **Lateinischer Hyperwürfel-Abtastung (Latin Hypercube Sampling, LHS)** implementiert wird.

Definition 62.3 (Lateinische Quadrate und Hyperwürfel)**.** Im Kontext der statistischen Stichproben ist ein quadratisches Gitter, das Abtastpositionen enthält, ein Lateinisches Quadrat, wenn (und nur wenn) in jeder Zeile und jeder Spalte nur eine Abtastung vorhanden ist. Ein Lateinischer Hyperwürfel ist die Verallgemeinerung dieses Konzepts auf eine beliebige Anzahl von Dimensionen, wobei jede Abtastung die einzige in jeder achsenparallelen Hyperebene ist, die sie enthält.

Das Generieren eines Lateinischen Hyperwürfels führt zu einem randomisierten Sampling-Plan, dessen Projektionen auf die Achsen gleichmäßig verteilt sind (multidimensionale Stratifikation). Dies garantiert jedoch nicht, dass der Plan raumfüllend ist.

62.4.2. Maximin-Pläne

Eine weit verbreitete Metrik zur Beurteilung der Gleichmäßigkeit oder "Raumfüllung" eines Sampling-Plans ist die **Maximin-Metrik**.

Definition 62.4 (Maximin-Plan)**.** Ein Sampling-Plan X wird als Maximin-Plan betrachtet, wenn er unter allen Kandidatenplänen den kleinsten Zwischenpunktabstand d_1 maximiert. Unter den Plänen, die diese Bedingung erfüllen, minimiert er ferner J_1, die Anzahl der Paare, die durch diesen minimalen Abstand getrennt sind.

Um die Stratifikationseigenschaften von Lateinischen Hyperwürfeln zu bewahren, konzentriert sich die Anwendung dieser Definition auf diese Klasse von Designs. Um das Problem potenziell mehrerer äquivalenter Maximin-Designs zu lösen, wird eine umfassendere "Tie-Breaker"-Definition nach Morris und Mitchell vorgeschlagen:

Definition 62.5 (Maximin-Plan mit Tie-Breaker)**.** Ein Sampling-Plan X wird als Maximin-Plan bezeichnet, wenn er sequenziell die folgenden Bedingungen optimiert: Er maximiert d_1; unter diesen minimiert er J_1; unter diesen maximiert er d_2; unter diesen minimiert er J_2; und so weiter, bis er J_m minimiert.

Für die Berechnung von Distanzen in diesen Kontexten wird die **p-Norm** am häufigsten verwendet:

Definition 62.6 (p-Norm)**.** Die p-Norm eines Vektors $\vec{x} = (x_1, x_2, \dots, x_k)$ ist definiert als:

$$d_p(\vec{x}^{(i_1)}, \vec{x}^{(i_2)}) = \left(\sum_{j=1}^{k} |x_j^{(i_1)} - x_j^{(i_2)}|^p \right)^{1/p}.$$

Wenn $p = 1$, definiert dies die **Rechteckdistanz** (oder Manhattan-Norm), und wenn $p = 2$, die **Euklidische Norm**. Die Rechteckdistanz ist rechnerisch erheblich weniger aufwendig, was besonders bei großen Sampling-Plänen von Vorteil sein kann.

Die `spotpython` Bibliothek bietet Funktionen zur Implementierung dieser Kriterien, wie `mm()` für paarweise Vergleiche von Sampling-Plänen.

62.4.3. Das Morris-Mitchell-Kriterium (Phi_q)

Um konkurrierende Sampling-Pläne in einer kompakten Form zu bewerten, definierten Morris und Mitchell (1995) die folgende skalarwertige Kriteriumsfunktion, die **Morris-Mitchell-Kriterium** genannt wird:

Definition 62.7 (Morris-Mitchell-Kriterium)**.** Das Morris-Mitchell-Kriterium ist definiert als:

$$\Phi_q(X) = \left(\sum_{j=1}^{m} J_j d_j^{-q} \right)^{1/q},$$

wobei X der Sampling-Plan ist, d_j der Abstand zwischen den Punkten, J_j die Vielfachheit dieses Abstands und q ein benutzerdefinierter Exponent ist. Der Parameter q kann angepasst werden, um den Einfluss kleinerer Abstände auf die Gesamtmetrik zu steuern. Ein kleinerer Wert von Φ_q deutet auf bessere raumfüllende Eigenschaften des Sampling-Plans hin.

Größere Werte von q stellen sicher, dass Terme in der Summe, die kleineren Zwischenpunktabständen entsprechen, einen dominanten Einfluss haben, was dazu führt, dass Φ_q die Sampling-Pläne in einer Weise ordnet, die der ursprünglichen Maximin-Definition (Definition 4.5) sehr genau entspricht. Kleinere q-Werte hingegen erzeugen eine Φ_q-Landschaft, die der ursprünglichen Definition zwar nicht perfekt entspricht, aber im Allgemeinen optimierungsfreundlicher ist. Es wird empfohlen, Φ_q für eine Reihe von q-Werten (z.B. 1, 2, 5, 10, 20, 50 und 100) zu minimieren und dann den besten Plan aus diesen Ergebnissen anhand der tatsächlichen Maximin-Definition auszuwählen.

Die Funktionen `mmphi()` und `mmphi_intensive()` in `spotpython` berechnen das Morris-Mitchell-Kriterium, unterscheiden sich jedoch in ihrer Normalisierung. Die `mmphi_intensive()`-Funktion ist invariant gegenüber der Abtastgröße, da sie die Summe $\sum J_l d_l^{-q}$ durch M (die Gesamtzahl der Paare, $M = N(N-1)/2$) teilt, was einen durchschnittlichen Beitrag pro Paar zur $-q$-ten Potenz des Abstands vor dem Ziehen der q-ten Wurzel berechnet. Dies ermöglicht aussagekräftigere Vergleiche der Raumfüllung zwischen Designs unterschiedlicher Größe. Ein kleinerer Wert zeigt bei beiden Kriterien ein besseres (raumfüllenderes) Design an.

```
import numpy as np
from spotpython.utils.sampling import mmphi, mmphi_intensive, rlh

# Beispiel: Erstellen von zwei Latin Hypercube Designs
np.random.seed(42)
X1 = rlh(n=10, k=2) # 10 Punkte in 2 Dimensionen
X2 = rlh(n=20, k=2) # 20 Punkte in 2 Dimensionen

q_val = 2.0 # Exponent q
p_val = 2.0 # p-Norm (Euclidean distance)

# Berechne Phi_q für X1 und X2
phi_q_X1 = mmphi(X1, q_val, p_val)
phi_q_X2 = mmphi(X2, q_val, p_val)

# Berechne Phi_q_intensive für X1 und X2
```

```
phi_q_intensive_X1, _, _ = mmphi_intensive(X1, q_val, p_val)
phi_q_intensive_X2, _, _ = mmphi_intensive(X2, q_val, p_val)

print(f"Morris-Mitchell Criterion (Phi_q) für X1 (10 Pts): {phi_q_X1:.3f}")
print(f"Morris-Mitchell Criterion (Phi_q) für X2 (20 Pts): {phi_q_X2:.3f}")
print("-" * 30)
print(f"Morris-Mitchell Criterion (Phi_q_intensive) für X1 (10 Pts): {phi_q_intensive_X1:.3f}")
print(f"Morris-Mitchell Criterion (Phi_q_intensive) für X2 (20 Pts): {phi_q_intensive_X2:.3f}")
```

```
Morris-Mitchell Criterion (Phi_q) für X1 (10 Pts): 17.415
Morris-Mitchell Criterion (Phi_q) für X2 (20 Pts): 42.838
------------------------------
Morris-Mitchell Criterion (Phi_q_intensive) für X1 (10 Pts): 2.596
Morris-Mitchell Criterion (Phi_q_intensive) für X2 (20 Pts): 3.108
```

Remark 62.2 (Morris-Mitchell-Kriterium). Φ_q steigt tendenziell mit der Anzahl der Punkte (N), während Φ_q^I (`mmphi_intensive`) abtastgrößeninvariant ist und daher besser für den Vergleich von Designs unterschiedlicher Größe geeignet ist.

62.5. Alternative Sampling-Pläne: Sobol-Sequenzen

Wenn die gesamte Rechenzeit budgetiert ist, aber nicht klar ist, wie viele Kandidatendesigns in dieser Zeit evaluiert werden können, bieten sich **Sobol-Sequenzen** als Alternative an. Diese Sampling-Pläne weisen gute raumfüllende Eigenschaften auf (zumindest für große n) und besitzen die Eigenschaft, dass für jedes n und $k > 1$ die Sequenz für $n - 1$ und k eine Untermenge der Sequenz für n und k ist.

62.6. Zusammenfassung

Die Auswahl des richtigen Sampling-Plans ist ein grundlegender Schritt in Computerexperimenten und der Surrogatmodellierung. Von traditionellen RSM-Ansätzen bis hin zu modernen DACE-Methoden, die den "Fluch der Dimensionen" mildern und effiziente Screenings ermöglichen, entwickeln sich die Techniken ständig weiter. Maximin-Pläne und das Morris-Mitchell-Kriterium bieten robuste Methoden zur Quantifizierung der Raumfüllung, wobei abtastgrößeninvariante Kriterien wie `mmphi_intensive()` Vergleiche über verschiedene Designgrößen hinweg ermöglichen.

62.7. Zusatzmaterialien

Viele der in diesem Dokument beschriebenen Konzepte sind in den Jupyter Notebooks zum "Hyperparameter Tuning Cookbook" verfügbar und können dort interaktiv erkundet werden. Die Python-Bibliothek `spotpython` (GitHub Repository) bietet Implementierungen für viele dieser Probenahme- und Optimierungsstrategien.

> **Interaktive Webseite**
>
> - Eine interaktive Webseite zum Thema **Curse of Dimensionality** ist hier zu finden: Curse of Dimensionality interaktiv.

> **Jupyter-Notebook**
>
> - Das Jupyter-Notebook für dieses Lernmodul ist auf GitHub im Hyperparameter-Tuning-Cookbook Repository verfügbar.

63. Lernmodul: Eine Einführung in Kriging

63.1. Konzeptionelle Grundlagen des Kriging

63.1.1. Von einfachen Modellen zur intelligenten Interpolation

In der modernen ingenieur- und naturwissenschaftlichen Forschung werden Praktiker häufig mit „Black-Box“-Funktionen konfrontiert. Dies sind Systeme oder Simulationen, deren interne Funktionsweise entweder unbekannt oder so komplex ist, dass sie praktisch undurchschaubar ist. Ein gängiges Beispiel ist eine hochpräzise Simulation der numerischen Strömungsmechanik (CFD), bei der Eingaben wie die Flügelgeometrie oder die Strömungsgeschwindigkeit Ausgaben wie Auftrieb und Luftwiderstand erzeugen. Jede Auswertung dieser Black Box kann außerordentlich teuer sein und Stunden oder sogar Tage an Supercomputerzeit in Anspruch nehmen. Wenn das Ziel darin besteht, den Designraum zu erkunden oder ein optimales Design zu finden, ist die Durchführung von Tausenden dieser Auswertungen oft nicht durchführbar.

Diese Herausforderung führt zur Notwendigkeit von **Surrogatmodellen**, auch bekannt als Metamodelle oder Antwortflächenmodelle “Response-Surface”. Ein Surrogatmodell ist eine rechengünstige Annäherung an eine teure Black-Box-Funktion. Es wird konstruiert, indem eine kleine Anzahl sorgfältig ausgewählter Auswertungen der wahren Funktion durchgeführt und dann ein mathematisches Modell an diese beobachteten Datenpunkte angepasst wird. Dieses „Modell eines Modells“ kann dann Tausende Male zu vernachlässigbaren Kosten ausgewertet werden, was eine effiziente Optimierung, Sensitivitätsanalyse und Erkundung des Designraums ermöglicht.

63.1.1.1. Eine Brücke zum Kriging: Verständnis von Radialen Basisfunktionen (RBFs)

Eine leistungsstarke und intuitive Klasse von Surrogatmodellen ist das **Modell der Radialen Basisfunktionen (RBF)**. Die grundlegende Idee hinter einem RBF ist es, eine komplexe Funktion als gewichtete Summe einfacherer, gut verstandener Basisfunktionen darzustellen. Jede Basisfunktion ist an einem der bekannten Datenpunkte zentriert, und ihr Wert hängt nur vom Abstand zu diesem Zentrum ab.

Mathematisch hat ein RBF-Modell die Form:

$$\hat{f}(\vec{x}) = \sum_{i=1}^{n} w_i \psi(||\vec{x} - \vec{c}^{(i)}||)$$

wobei $\hat{f}(\vec{x})$ der vorhergesagte Wert an einem neuen Punkt $\vec{x}$ ist, w_i die Gewichte sind, $\vec{c}^{(i)}$ die Zentren der Basisfunktionen (typischerweise die Standorte der bekannten Datenpunkte, $\vec{x}^{(i)}$) und ψ die radiale Basisfunktion selbst ist, die auf dem euklidischen Abstand $||\vec{x} - \vec{c}^{(i)}||$ operiert.

Gängige Wahlen für ψ umfassen die linearen, kubischen, Gauß'schen oder multiquadratischen Funktionen (Forrester, Sóbester, and Keane 2008). Indem wir fordern, dass das Modell exakt durch alle bekannten Datenpunkte verläuft (ein Prozess, der als Interpolation bezeichnet wird), können wir ein System linearer Gleichungen aufstellen, um die unbekannten Gewichte w_i zu lösen. Dies wird typischerweise in Matrixform geschrieben als:

$$\Psi \vec{w} = \vec{y}$$

wobei Ψ eine Matrix der Auswertungen der Basisfunktionen ist, $\vec{w}$ der Vektor der Gewichte und $\vec{y}$ der Vektor der beobachteten Antworten ist. Das Lösen nach den Gewichten ist dann eine Frage der Matrixinversion: $\vec{w} = \Psi^{-1}\vec{y}$. Die Schönheit dieses Ansatzes liegt darin, dass er ein potenziell hochgradig nichtlineares Modellierungsproblem in ein unkompliziertes lineares Algebraproblem umwandelt (Forrester, Sóbester, and Keane 2008).

Diese Struktur weist eine bemerkenswerte Ähnlichkeit mit anderen Modellierungsparadigmen auf. Die RBF-Formulierung ist funktional identisch mit einem einschichtigen künstlichen neuronalen Netz, bei dem die Neuronen eine radiale Aktivierungsfunktion verwenden. In dieser Analogie ist die Eingabe für jedes Neuron der Abstand von einem Zentrum, die Aktivierungsfunktion des Neurons ist die Basisfunktion ψ, und die Ausgabe des Netzwerks ist die gewichtete Summe dieser Aktivierungen. Diese Verbindung bietet ein nützliches mentales Modell für diejenigen, die mit maschinellem Lernen vertraut sind, und rahmt RBFs nicht als esoterische statistische Technik, sondern als nahen Verwandten von neuronalen Netzen ein, die beide leistungsstarke universelle Funktionsapproximatoren sind.

63.1.1.2. Einordnung des Kriging

In dieser Landschaft tritt das **Kriging** als eine besonders anspruchsvolle und flexible Art eines RBF-Modells hervor. Ursprünglich aus dem Bereich der Geostatistik durch die Arbeit von Danie G. Krige und Georges Matheron stammend, wurde es entwickelt, um Erzkonzentrationen im Bergbau vorherzusagen (Forrester, Sóbester, and Keane 2008). Seine Anwendung auf deterministische Computerexperimente wurde von Sacks et al. (1989) vorangetrieben und ist seitdem zu einem Eckpfeiler des Ingenieurdesigns und der Optimierung geworden.

Im Bereich des maschinellen Lernens ist Kriging besser bekannt als **Gauß-Prozess-Regression (GPR)**. Obwohl sich die Terminologie unterscheidet, ist das zugrunde liegende mathematische Gerüst dasselbe. Kriging unterscheidet sich von einfacheren RBF-Modellen durch seine einzigartige Basisfunktion und seine statistische Grundlage, die nicht nur eine Vorhersage, sondern auch ein Maß für die Unsicherheit dieser Vorhersage liefert.

63.1.2. Die Kernphilosophie des Kriging: Eine stochastische Prozessperspektive

Um das Kriging wirklich zu verstehen, muss man einen konzeptionellen Sprung wagen, der zunächst kontraintuitiv sein kann. Selbst bei der Modellierung eines perfekt deterministischen Computercodes – bei dem dieselbe Eingabe immer genau dieselbe Ausgabe erzeugt – behandelt das Kriging die Ausgabe der Funktion als eine einzelne Realisierung eines **stochastischen (oder zufälligen) Prozesses**.

Das bedeutet nicht, dass wir annehmen, die Funktion sei zufällig. Stattdessen drücken wir unsere Unsicherheit über den Wert der Funktion an nicht beobachteten Stellen aus. Bevor wir Daten haben, könnte der Wert der Funktion an jedem Punkt alles sein. Nachdem wir einige Punkte beobachtet haben, ist unsere Unsicherheit reduziert, aber sie existiert immer noch überall sonst. Das stochastische Prozessgerüst bietet eine formale mathematische Sprache, um diese Unsicherheit zu beschreiben.

63.1.2.1. Das Prinzip der Lokalität und Korrelation

Dieser angenommene stochastische Prozess ist nicht völlig unstrukturiert. Er wird von einer **Korrelationsstruktur** bestimmt, die eine grundlegende Annahme über die Welt verkörpert: das Prinzip der Lokalität. Dieses Prinzip besagt, dass Punkte, die im Eingaberaum nahe beieinander liegen, erwartungsgemäß ähnliche Ausgabewerte haben (d. h. sie sind hoch korreliert), während Punkte, die weit voneinander entfernt sind, erwartungsgemäß unähnliche oder unzusammenhängende Ausgabewerte haben (d. h. sie sind unkorreliert). Diese Annahme gilt für die große Mehrheit der physikalischen Phänomene und glatten mathematischen Funktionen, die keine chaotischen, diskontinuierlichen Sprünge aufweisen. Die Korrelation zwischen zwei beliebigen Punkten wird durch eine **Kovarianzfunktion** oder einen **Kernel** quantifiziert, der das Herzstück des Kriging-Modells ist.

63.1.2.2. Gauß-Prozess-Prior

Speziell nimmt das Kriging an, dass dieser stochastische Prozess ein **Gauß-Prozess** ist. Ein Gauß-Prozess ist eine Sammlung von Zufallsvariablen, von denen jede endliche Anzahl eine gemeinsame multivariate Normalverteilung (MVN) hat. Dies ist eine starke Annahme, da eine multivariate Normalverteilung vollständig durch nur zwei

Komponenten definiert ist: einen **Mittelwertvektor** ($\vec{\mu}$) und eine **Kovarianzmatrix** (Σ) (Forrester, Sóbester, and Keane 2008).

Dies ist als der **Gauß-Prozess-Prior** bekannt. Es ist unsere „vorherige Überzeugung" über die Natur der Funktion, bevor wir Daten gesehen haben. Wir glauben, dass die Funktionswerte an jedem Satz von Punkten gemeinsam gaußverteilt sein werden, zentriert um einen gewissen Mittelwert, mit einer Kovarianzstruktur, die durch den Abstand zwischen den Punkten diktiert wird. Wenn wir Daten beobachten, verwenden wir Bayes'sche Inferenz, um diese vorherige Überzeugung zu aktualisieren, was zu einem **Gauß-Prozess-Posterior** führt. Dieser Posterior ist ebenfalls ein Gauß-Prozess, aber sein Mittelwert und seine Kovarianz wurden aktualisiert, um mit den beobachteten Daten konsistent zu sein. Der Mittelwert dieses posterioren Prozesses gibt uns die Kriging-Vorhersage, und seine Varianz gibt uns ein Maß für die Unsicherheit über diese Vorhersage. Diese statistische Grundlage ist es, die das Kriging auszeichnet und es ihm ermöglicht, nicht nur zu interpolieren, sondern auch seine eigene Zuverlässigkeit zu quantifizieren.

63.2. Die mathematische Architektur eines Kriging-Modells

Um vom konzeptionellen Ansatz des Kriging zur praktischen Umsetzung zu gelangen, ist es unerlässlich, seine mathematischen Komponenten zu verstehen. Dieser Abschnitt analysiert die Architektur des Modells und verbindet konsequent die abstrakte mathematische Notation aus Referenztexten mit den konkreten Variablen, die im bereitgestellten Python-Code verwendet werden.

63.2.1. Glossar der Kriging-Notation

Table 63.1 dient als Glossar, um die Notation aus (Forrester, Sóbester, and Keane 2008), dem Kochbuch (Bartz-Beielstein 2023b) und dem in diesem Dokument in Section 63.4 bereitgestellten Python-Code abzugleichen.

Table 63.1.: Glossar

Mathematisches Symbol	Konzeptionelle Bedeutung	Python-Variable
n	Anzahl der Trainings-/Stichprobenpunkte	`n` oder `X_train.shape`
k	Anzahl der Eingabedimensionen/Variablen	`X_train.shape`
m	Anzahl der Punkte für die Vorhersage	`m` oder `x_predict.shape`

Mathematisches Symbol	Konzeptionelle Bedeutung	Python-Variable
X	$n \times k$ Matrix der Trainingspunkt-Standorte	`X_train`
y	$n \times 1$ Vektor der beobachteten Antworten	`y_train`
$\vec{x}$	Ein neuer Standort für die Vorhersage	Eine Zeile in `x_predict`
Ψ (Psi)	$n \times n$ Korrelationsmatrix der Trainingsdaten	`Psi`
$\vec{\psi}$ (psi)	$n \times m$ Vorhersage-Trainings-Korrelationsmatrix	`psi`
$\vec{\theta}$ (theta)	$k \times 1$ Vektor der Aktivitäts-/Breiten-Hyperparameter	`theta`
$\vec{p}$	$k \times 1$ Vektor der Glattheits-Hyperparameter	Implizit $p = 2$ im Code
μ (mu)	Der globale Mittelwert des stochastischen Prozesses	`mu_hat`
σ^2 (sigma-quadrat)	Die Varianz des stochastischen Prozesses	Im Code nicht explizit berechnet
λ (lambda)	Der Regressions-/Nugget-Parameter	`eps`
$\hat{y}(\vec{x})$	Die Kriging-Vorhersage am Punkt $\vec{x}$	`f_predict`

63.2.2. Der Korrelationskernel: Quantifizierung von Beziehungen

Der Kern des Kriging-Modells ist seine spezialisierte Basisfunktion, auch als Kernel oder Kovarianzfunktion bekannt. Diese Funktion definiert die Korrelation zwischen zwei beliebigen Punkten im Designraum. Die gebräuchlichste Form, und die in unseren Referenztexten verwendete, ist der Gauß'sche Kernel.

Die Kriging-Basisfunktion ist definiert als:

$$\psi(\vec{x}^{(i)}, \vec{x}) = \exp\left(-\sum_{j=1}^{k} \theta_j |x_j^{(i)} - x_j|^{p_j}\right)$$

Diese Gleichung berechnet die Korrelation zwischen einem bekannten Punkt $\vec{x}^{(i)}$ und jedem anderen Punkt $\vec{x}$. Sie wird von zwei Schlüsselsätzen von Hyperparametern gesteuert: $\vec{\theta}$ und $\vec{p}$.

63.2.2.1. Hyperparameter $\vec{\theta}$ (Theta): Der Aktivitätsparameter

Der Parametervektor $\vec{\theta} = \{\theta_1, \theta_2, ..., \theta_k\}^T$ ist wohl der wichtigste Hyperparameter im Kriging-Modell. Jede Komponente θ_j steuert, wie schnell die Korrelation mit dem Abstand entlang der j-ten Dimension abfällt. Er wird oft als „Aktivitäts"- oder „Breiten"-Parameter bezeichnet.

- Ein **großes** θ_j zeigt an, dass die Funktion sehr empfindlich auf Änderungen in der j-ten Variablen reagiert. Die Korrelation wird sehr schnell abfallen, wenn sich die Punkte in dieser Dimension voneinander entfernen, was zu einer „schmalen" Basisfunktion führt. Dies impliziert, dass die zugrunde liegende Funktion entlang dieser Achse sehr „aktiv" ist oder sich schnell ändert.
- Ein **kleines** θ_j zeigt an, dass die Funktion relativ unempfindlich auf Änderungen in der j-ten Variablen reagiert. Die Korrelation wird langsam abfallen, was zu einer „breiten" Basisfunktion führt, die ihren Einfluss über einen größeren Bereich ausdehnt. Da die Korrelation auch über größere Distanzen hinweg hoch bleibt, bedeutet dies, dass das Modell davon ausgeht, dass Punkte, die in der j-ten Dimension weiter voneinander entfernt sind, immer noch stark korreliert sind. und dass der Einfluss eines Datenpunktes sich über einen großen Bereich im Eingaberaum erstreckt, bevor die Korrelation signifikant abnimmt. Das Modell geht also davon aus, dass sich die zugrunde liegende Funktion entlang dieser Achse sehr "langsam" ändert oder "inaktiv" ist.

Die Tatsache, dass $\vec{\theta}$ ein Vektor ist – mit einem separaten Wert für jede Eingabedimension – ist ein entscheidendes Merkmal, das dem Kriging immense Leistungsfähigkeit verleiht, insbesondere bei mehrdimensionalen Problemen. Dies ist als **anisotrope Modellierung** bekannt. Indem die Korrelationslänge für jede Variable unterschiedlich sein kann, kann sich das Modell an Funktionen anpassen, die sich entlang verschiedener Achsen unterschiedlich verhalten. Zum Beispiel könnte eine Funktion sehr schnell auf Temperaturänderungen, aber sehr langsam auf Druckänderungen reagieren. Ein anisotropes Kriging-Modell kann dieses Verhalten erfassen, indem es ein großes θ für die Temperatur und ein kleines θ für den Druck lernt.

Diese Fähigkeit hat eine tiefgreifende Konsequenz: **automatische Relevanzbestimmung**. Während des Modellanpassungsprozesses (den wir in Section 63.2.4 diskutieren werden) findet der Optimierungsalgorithmus die $\vec{\theta}$-Werte, die die Daten am besten erklären. Wenn eine bestimmte Eingangsvariable x_j wenig oder keinen Einfluss auf die Ausgabe y hat, wird das Modell einen sehr kleinen Wert für θ_j lernen. Ein kleines θ_j macht den Term $\theta_j|x_j^{(i)} - x_j|^{p_j}$ nahe null, was die Korrelation effektiv unempfindlich gegenüber Änderungen in dieser Dimension macht. Daher kann ein Ingenieur nach der Anpassung des Modells den optimierten $\vec{\theta}$-Vektor inspizieren, um eine Sensitivitätsanalyse durchzuführen. Die Dimensionen mit den größten θ_j-Werten sind die einflussreichsten Treiber der Systemantwort. Dies verwandelt das Surrogatmodell von einem einfachen Black-Box-Approximator in ein Werkzeug zur Generierung

wissenschaftlicher und technischer Erkenntnisse. Der im Hyperparameter Tuning Cookbook bereitgestellte und in Section 63.4 besprochene Python-Code, als eindimensionales Beispiel, vereinfacht dies durch die Verwendung eines einzelnen skalaren `theta`, aber das Verständnis seiner Rolle als Vektor ist entscheidend, um den Nutzen des Kriging in realen Anwendungen zu schätzen (Bartz-Beielstein 2025).

63.2.2.2. Hyperparameter $\vec{p}$: Der Glattheitsparameter

Der Parametervektor $\vec{p} = \{p_1, p_2, ..., p_k\}^T$ steuert die Glattheit der Funktion an den Datenpunkten. Sein Wert ist typischerweise auf das Intervall $[1, 2]$ beschränkt (Forrester, Sóbester, and Keane 2008). Die Wahl von p_j hat tiefgreifende Auswirkungen auf die Form der resultierenden Basisfunktion:

- Wenn $p_j = 2$, ist die resultierende Funktion unendlich differenzierbar, was bedeutet, dass sie sehr glatt ist. Dies ist im bereitgestellten Python-Code der Fall, was durch die Verwendung der `sqeuclidean`-Distanzmetrik (quadrierter Abstand entspricht $p = 2$) implizit ist. Die `sqeuclidean`-Metrik ist auf https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.sqeuclidean.html beschrieben.
- Wenn $p_j = 1$, ist die resultierende Funktion stetig, aber an den Datenpunkten nicht differenzierbar, was ihr ein „spitzeres“ oder „stacheligeres“ Aussehen verleiht.

Die Wahl von p spiegelt eine Annahme über die Natur der zugrunde liegenden Funktion wider. Die meisten physikalischen Prozesse sind glatt, was $p = 2$ zu einer gängigen und robusten Wahl macht. Für Funktionen mit bekannten scharfen Merkmalen kann jedoch die Optimierung von p in Richtung 1 eine bessere Anpassung ermöglichen.

63.2.3. Aufbau des Systems: Die Korrelationsmatrizen Ψ und $\vec{\psi}$

Mit dem definierten Korrelationskernel können wir nun die Matrizen konstruieren, die den Kern des Kriging-Systems bilden. Diese Matrizen quantifizieren die Beziehungen zwischen allen Punkten in unserem Problem: den bekannten Trainingspunkten und den neuen Vorhersagepunkten.

63.2.3.1. Die `Psi`-Matrix (Ψ): Korrelation der Trainingsdaten

Die Matrix Ψ ist die $n \times n$ Korrelationsmatrix der n Trainingsdaten mit sich selbst. Jedes Element Ψ_{ij} ist die Korrelation zwischen dem Trainingspunkt $\vec{x}^{(i)}$ und dem Trainingspunkt $\vec{x}^{(j)}$, berechnet mit der Basisfunktion.

$$\Psi_{ij} = \text{corr}(\vec{x}^{(i)}, \vec{x}^{(j)}) = \exp\left(-\sum_{l=1}^{k} \theta_l |x_l^{(i)} - x_l^{(j)}|^{p_l}\right).$$

Da die Korrelation eines Punktes mit sich selbst perfekt ist, sind die diagonalen Elemente Ψ_{ii} immer gleich 1. Die Matrix ist auch symmetrisch, da der Abstand von Punkt A zu B derselbe ist wie von B zu A.

Code-Analyse (`build_Psi`): Die bereitgestellte Python-Funktion `build_Psi` implementiert diese Berechnung effizient.

```
from scipy.spatial.distance import pdist, squareform, cdist
from numpy.linalg import cholesky
import numpy as np
from numpy import sqrt, spacing, exp, multiply, eye
from numpy.linalg import solve
from scipy.spatial.distance import pdist, squareform, cdist
def build_Psi(X, theta, eps=sqrt(spacing(1))):
    D = squareform(pdist(X, metric='sqeuclidean', out=None, w=theta))
    Psi = exp(-D)
    Psi += multiply(eye(X.shape[0]), eps)
    return Psi
```

1. `D = squareform(pdist(X, metric='sqeuclidean', out=None, w=theta))`: Diese Zeile ist der rechnerische Kern. Die Funktion `scipy.spatial.distance.pdist` berechnet die paarweisen Abstände zwischen allen Zeilen in der Eingabematrix `X_train`.

   - `metric='sqeuclidean'` gibt an, dass der quadrierte euklidische Abstand, $(x_i - x_j)^2$, verwendet werden soll. Dies setzt implizit den Hyperparameter $p = 2$.
   - `w=theta` wendet den Aktivitätsparameter als Gewicht auf den quadrierten Abstand jeder Dimension an und berechnet $\theta_j(x_{ij} - x_{kj})^2$ für jedes Paar von Punkten i, k und jede Dimension j. Für den 1D-Fall im Code ist dies einfach `theta * (x_i - x_j)^2`.
   - `squareform` wandelt dann den von `pdist` zurückgegebenen komprimierten Abstandsvektor in die vollständige, symmetrische $n \times n$ Abstandsmatrix um, die wir D nennen können.

2. `Psi = exp(-D)`: Dies führt eine elementweise Potenzierung des Negativen der Abstandsmatrix durch und vervollständigt die Berechnung des Gauß'schen Kernels.
3. `Psi += multiply(eye(X.shape), eps)`: Diese Zeile addiert eine kleine Konstante `eps` zur Diagonale der Ψ-Matrix. Dies ist der „Nugget"-Term, eine entscheidende Komponente sowohl für die numerische Stabilität als auch für die Rauschmodellierung, die in Section 63.3.2 behandelt wird.

63.2.3.2. Der psi-Vektor/Matrix ($\vec{\psi}$): Vorhersage-Trainings-Korrelation

Die Matrix $\vec{\psi}$ ist die $n \times m$ Matrix der Korrelationen zwischen den n bekannten Trainingspunkten und den m neuen Punkten, an denen wir eine Vorhersage machen möchten. Jedes Element ψ_{ij} ist die Korrelation zwischen dem i-ten Trainingspunkt $\vec{x}^{(i)}$ und dem j-ten Vorhersagepunkt $\vec{x}^{(j)}_{pred}$.

$$\psi_{ij} = \text{corr}(\vec{x}^{(i)}, \vec{x}^{(j)}_{pred})$$

Code-Analyse (`build_psi`): Die Funktion `build_psi` berechnet diese Matrix.

```
def build_psi(X_train, x_predict, theta):
    D = cdist(x_predict, X_train, metric='sqeuclidean', out=None, w=theta)
    psi = exp(-D)
    return psi.T
```

1. `D = cdist(x_predict, X_train, metric='sqeuclidean', out=None, w=theta)`: Hier wird `scipy.spatial.distance.cdist` verwendet, siehe https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.cdist.html, da wir Abstände zwischen Punkten aus zwei *verschiedenen* Mengen berechnen: den m Vorhersagepunkten in `x_predict` und den n Trainingspunkten in `X_train`. Dies führt zu einer $m \times n$ Matrix gewichteter quadrierter Abstände.
2. `psi = exp(-D)`: Wie zuvor vervollständigt dies die Kernel-Berechnung.
3. `return psi.T`: Die resultierende Matrix wird transponiert, um die Größe $n \times m$ zu haben. Dies ist eine Konvention, um mit der in den Vorhersageformeln in Referenztexten wie Forrester, Sóbester, and Keane (2008) dargestellten Matrixalgebra übereinzustimmen.

63.2.4. Modellkalibrierung durch Maximum-Likelihood-Schätzung (MLE)

Sobald die Struktur des Modells durch den Kernel definiert ist, müssen wir die optimalen Werte für seine Hyperparameter, nämlich $\vec{\theta}$ und $\vec{p}$, bestimmen. Der Code in Section 63.4 umgeht diesen Schritt, indem er `theta = 1.0` fest codiert, aber in jeder praktischen Anwendung müssen diese Parameter aus den Daten gelernt werden. Eine gebräuchliche Methode hierfür ist die **Maximum-Likelihood-Schätzung (MLE)**.

Die Kernidee der MLE besteht darin, die Frage zu beantworten: „Welche Werte der Hyperparameter machen bei unseren beobachteten Daten diese Daten am wahrscheinlichsten¿‘ Wir finden die Parameter, die die Wahrscheinlichkeit maximieren, die Daten beobachtet zu haben, die wir tatsächlich gesammelt haben.

63.2.4.1. Die Likelihood-Funktion

Unter der Annahme des Gauß-Prozesses wird die gemeinsame Wahrscheinlichkeit, den Antwortvektor $\vec{y}$ bei gegebenen Parametern zu beobachten, durch die multivariate normale Wahrscheinlichkeitsdichtefunktion beschrieben. Diese Funktion ist unsere Likelihood, L:

$$L(\mu, \sigma^2, \vec{\theta}, \vec{p}|\vec{y}) = \frac{1}{(2\pi\sigma^2)^{n/2}|\Psi|^{1/2}} \exp\left[-\frac{(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{2\sigma^2}\right]. \tag{63.1}$$

Hier sind μ und σ^2 der globale Mittelwert und die Varianz des Prozesses, und Ψ ist die Korrelationsmatrix, die implizit von $\vec{\theta}$ und $\vec{p}$ abhängt. Beachten Sie, dass $|\Psi|$ die Determinante (also ein reellwertiger Skalar) der Korrelationsmatrix ist.

63.2.4.2. Die konzentrierte Log-Likelihood

Die direkte Maximierung der Likelihood-Funktion (Equation 63.1) ist schwierig. Aus Gründen der rechnerischen Stabilität und mathematischen Bequemlichkeit arbeiten wir stattdessen mit ihrem natürlichen Logarithmus, der Log-Likelihood (siehe Gleichung (2.29) in Forrester, Sóbester, and Keane (2008)):

$$\ln(L) = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln(\sigma^2) - \frac{1}{2}\ln|\Psi| - \frac{(\vec{y}-\vec{1}\mu)^T\Psi^{-1}(\vec{y}-\vec{1}\mu)}{2\sigma^2}. \tag{63.2}$$

Eine wesentliche Vereinfachung ergibt sich, da wir für jedes gegebene $\vec{\theta}$ und $\vec{p}$ (und damit ein festes Ψ) die optimalen Werte für μ und σ^2 analytisch finden können, indem wir Ableitungen bilden und sie auf null setzen. Die MLE für den Mittelwert ist (siehe Gleichung (2.30) in Forrester, Sóbester, and Keane (2008)):

$$\hat{\mu} = \frac{\mathbf{1}^T\Psi^{-1}\vec{y}}{\mathbf{1}^T\Psi^{-1}\mathbf{1}}. \tag{63.3}$$

Die Berechnung in Equation 63.3 kann als Berechung eines verallgemeinerten gewichteten Durchschnitts der beobachteten Antworten interpretiert werden, wobei die Gewichtung die Korrelationsstruktur berücksichtigt.

Ein ähnlicher Ausdruck existiert für die optimale Varianz, $\hat{\sigma}^2$ (siehe Gleichung (2.31) in Forrester, Sóbester, and Keane (2008)):

$$\hat{\sigma}^2 = \frac{(\vec{y}-\mathbf{1}\hat{\mu})^T\Psi^{-1}(\vec{y}-\mathbf{1}\hat{\mu})}{n}.$$

Indem wir diese analytischen Ausdrücke für $\hat{\mu}$ und $\hat{\sigma}^2$ wieder in die Log-Likelihood-Funktion (Equation 63.2) einsetzen, erhalten wir die **konzentrierte Log-Likelihood-Funktion** (siehe Gleichung (2.32) in Forrester, Sóbester, and Keane (2008)):

$$\ln(L) \approx -\frac{n}{2}\ln(\hat{\sigma}^2) - \frac{1}{2}\ln|\Psi|.$$

Diese vereinfachte Funktion hängt nun nur noch von den Hyperparametern $\vec{\theta}$ und $\vec{p}$ ab, die in Ψ und $\hat{\sigma}^2$ eingebettet sind.

63.2.4.3. Numerische Optimierung

Wir können die optimalen $\vec{\theta}$ und $\vec{p}$ nicht analytisch bestimmen. Die konzentrierte Log-Likelihood ist jedoch eine skalare Funktion, die relativ schnell zu berechnen ist. Wir können daher einen numerischen Optimierungsalgorithmus – wie einen genetischen Algorithmus, Nelder-Mead oder Differential Evolution – verwenden, um den Hyperparameterraum zu durchsuchen und die Werte von $\vec{\theta}$ und $\vec{p}$ zu finden, die diese Funktion maximieren. Dieser Suchprozess ist die „Trainings"- oder „Anpassungs"-Phase beim Aufbau eines Kriging-Modells. Sobald die optimalen Hyperparameter gefunden sind, ist das Modell vollständig kalibriert und bereit, Vorhersagen zu treffen.

63.3. Implementierung und Vorhersage

Nachdem der theoretische und mathematische Rahmen geschaffen wurde, konzentriert sich dieser Teil auf die praktische Anwendung des Kriging-Modells: wie man es zur Erstellung von Vorhersagen verwendet und welche wesentlichen numerischen Techniken sicherstellen, dass der Prozess sowohl effizient als auch robust ist.

63.3.1. Der Kriging-Prädiktor: Generierung neuer Werte

Das ultimative Ziel beim Aufbau eines Kriging-Modells ist die Vorhersage des Funktionswerts an neuen, unbeobachteten Punkten. Die hierfür verwendete Formel ist als **Bester Linearer Unverzerrter Prädiktor (BLUP)** bekannt. Sie liefert den Mittelwert des posterioren Gauß-Prozesses am Vorhersageort, was unsere beste Schätzung des Funktionswerts ist.

Die Vorhersageformel lautet (siehe Gleichung (2.40) in Forrester, Sóbester, and Keane (2008)):

$$\hat{y}(\vec{x}) = \hat{\mu} + \vec{\psi}^T \Psi^{-1} (\vec{y} - \mathbf{1}\hat{\mu}), \tag{63.4}$$

wobei $\hat{y}(\vec{x})$ die Vorhersage an einem neuen Punkt $\vec{x}$ ist und alle anderen Terme wie im Glossar definiert sind.Im Folgenden beschreiben wir die einzelnen Komponenten dieser Gleichung und ihre Bedeutung:

1. $\hat{\mu}$: Die Vorhersage beginnt mit dem geschätzten globalen Mittelwert des Prozesses. Dies ist unsere Basisvermutung, bevor wir den Einfluss der lokalen Datenpunkte berücksichtigen.
2. $(\vec{y} - \mathbf{1}\hat{\mu})$: Dies ist der Vektor der Residuen. Er stellt die Differenz zwischen jedem beobachteten Datenpunkt und dem globalen Mittelwert dar. Dieser Vektor erfasst die spezifischen, lokalen Informationen, die unsere Trainingsdaten liefern.

3. $\Psi^{-1}(\vec{y} - \mathbf{1}\hat{\mu})$: Dieser Term kann als ein Vektor von Gewichten betrachtet werden, nennen wir ihn $\vec{w}$. Indem wir die Residuen mit der Inversen der Korrelationsmatrix multiplizieren, berechnen wir einen Satz von Gewichten, der die Interkorrelationen zwischen den Trainingspunkten berücksichtigt. Dieser Schritt sagt im Wesentlichen: „Wie viel sollte jedes Residuum beitragen, wenn man bedenkt, dass die Datenpunkte selbst nicht unabhängig sind¿‘
4. $\vec{\psi}^T\vec{w}$: Die endgültige Vorhersage wird durch eine gewichtete Summe dieser berechneten Gewichte angepasst. Die Gewichte für diese Summe sind die Korrelationen ($\vec{\psi}$) zwischen dem neuen Vorhersagepunkt $\vec{x}$ und jedem der Trainingspunkte. Im Wesentlichen besagt die Formel: „Beginne mit dem globalen Mittelwert und füge dann eine Korrektur basierend auf den beobachteten Residuen hinzu. Der Einfluss jedes Residuums wird durch die Korrelation des neuen Punktes mit dem entsprechenden Trainingspunkt bestimmt.“

Code-Analyse (`mu_hat` und `f_predict`): Der bereitgestellte Python-Code implementiert diesen Vorhersageprozess direkt.

```
Psi = build_Psi(X, theta)
U = cholesky(Psi).T
one = np.ones(n).reshape(-1, 1)
mu_hat = (one.T @ solve(U, solve(U.T, y_train))) / \
         (one.T @ solve(U, solve(U.T, one)))
f_predict = mu_hat * np.ones(m).reshape(-1, 1) \
            + psi.T @ solve(U, solve(U.T, y_train - one * mu_hat))
```

- `mu_hat = (one.T @ solve(U, solve(U.T, y_train))) / (one.T @ solve(U, solve(U.T, one)))`: Dies ist eine direkte und numerisch stabile Implementierung der Formel für $\hat{\mu}$, siehe Equation 63.3. Anstatt `Psi_inv` explizit zu berechnen, wird der auf Cholesky basierende Löser verwendet, der im nächsten Abschnitt detailliert wird. Der Ausdruck `solve(U, solve(U.T, y_train))` ist äquivalent zu `Psi_inv @ y_train`.
- `f_predict = mu_hat *... + psi.T @ solve(U, solve(U.T, y_train - one * mu_hat))`: Diese Zeile ist eine direkte Übersetzung der BLUP-Formel aus Equation 63.4. Sie berechnet die Residuen `y_train - one * mu_hat`, multipliziert sie mit `Psi_inv` unter Verwendung des Cholesky-Lösers und berechnet dann das Skalarprodukt mit `psi.T`, bevor das Ergebnis zum Basiswert `mu_hat` addiert wird.

63.3.2. Numerische Best Practices und der Nugget-Effekt

Eine direkte Implementierung der Kriging-Gleichungen unter Verwendung der Standard-Matrixinversion kann sowohl langsam als auch numerisch anfällig sein. Professionelle Implementierungen stützen sich auf spezifische numerische Techniken, um Effizienz und Robustheit zu gewährleisten.

63.3.2.1. Effiziente Inversion mit Cholesky-Zerlegung

Die Berechnung der Matrixinversen Ψ^{-1} ist der rechenintensivste Schritt sowohl im MLE-Prozess als auch bei der endgültigen Vorhersage. Eine direkte Inversion hat eine Rechenkomplexität von etwa $O(n^3)$. Wenn die Matrix schlecht konditioniert ist (fast singulär), kann die direkte Inversion außerdem zu großen numerischen Fehlern führen.

Ein überlegener Ansatz ist die Verwendung der **Cholesky-Zerlegung**. Diese Methode gilt für symmetrische, positiv definite Matrizen wie Ψ. Sie zerlegt Ψ in das Produkt einer unteren Dreiecksmatrix L und ihrer Transponierten L^T (oder einer oberen Dreiecksmatrix U und ihrer Transponierten U^T), sodass $\Psi = LL^T$. Diese Zerlegung ist schneller als die Inversion, mit einer Komplexität von ungefähr $O(n^3/3)$ (Forrester, Sóbester, and Keane 2008).

Sobald Ψ zerlegt ist, wird das Lösen eines linearen Systems wie $\Psi\vec{w} = \vec{b}$ zu einem zweistufigen Prozess der Lösung zweier viel einfacherer Dreieckssysteme, ein Verfahren, das als Vorwärts- und Rückwärtssubstitution bekannt ist:

1. Löse $L\vec{v} = \vec{b}$ nach $\vec{v}$.
2. Löse $L^T\vec{w} = \vec{v}$ nach $\vec{w}$.

Genau das macht der Python-Code. Die Zeile `U = cholesky(Psi).T` führt die Zerlegung durch (NumPys `cholesky` gibt den unteren Dreiecksfaktor L zurück, also wird er transponiert, um den oberen Dreiecksfaktor U zu erhalten). Anschließend implementieren Ausdrücke wie `solve(U, solve(U.T,...))` die effiziente zweistufige Lösung, ohne jemals die vollständige Inverse von Ψ zu bilden.

63.3.2.2. Der Nugget: Von numerischer Stabilität zur Rauschmodellierung

Die Cholesky-Zerlegung funktioniert nur, wenn die Matrix Ψ streng positiv definit ist. Ein Problem tritt auf, wenn zwei Trainingspunkte $\vec{x}^{(i)}$ und $\vec{x}^{(j)}$ sehr nahe beieinander liegen. In diesem Fall wird ihre Korrelation nahe 1 sein, was die entsprechenden Zeilen und Spalten in Ψ nahezu identisch macht. Dies führt dazu, dass die Matrix schlecht konditioniert oder fast singulär wird, was zum Scheitern der Cholesky-Zerlegung führen kann.

> **Naheliegende Punkte führen zu numerischen Problemen**
>
> Die Aussage, dass ein Problem auftritt, wenn zwei Trainingspunkte $\vec{x}^{(i)}$ und $\vec{x}^{(j)}$ sehr nahe beieinander liegen, und deren Korrelation dann nahe 1 ist, was die entsprechenden Zeilen und Spalten in der Ψ-Matrix nahezu identisch macht, ist ein wichtiger Aspekt der numerischen Stabilität und Modellierung beim Kriging.

Die Ψ-Matrix (sprich: Psi) ist die Korrelationsmatrix der Trainingsdaten mit sich selbst. Jedes Element Ψ_{ij} quantifiziert die Korrelation zwischen zwei bekannten Trainingspunkten $\vec{x}^{(i)}$ und $\vec{x}^{(j)}$. Diese Korrelation wird durch die Kriging-Basisfunktion (oder den Gauß'schen Kernel) definiert:

$$\psi(\vec{x}^{(i)}, \vec{x}^{(j)}) = \exp\left(-\sum_{l=1}^{k} \theta_l |x_l^{(i)} - x_l^{(j)}|^{p_l}\right)$$

Hierbei ist k die Anzahl der Eingabedimensionen, θ_l der Aktivitätshyperparameter für die l-te Dimension und p_l der Glattheitsparameter. Die diagonalen Elemente Ψ_{ii} sind immer 1, da die Korrelation eines Punktes mit sich selbst perfekt ist.
Das Kriging-Modell basiert auf dem Prinzip der Lokalität. Dieses besagt, dass Punkte, die im Eingaberaum nahe beieinander liegen, erwartungsgemäß ähnliche Ausgabewerte haben und somit hoch korreliert sind.
Wenn nun zwei Trainingspunkte $\vec{x}^{(i)}$ und $\vec{x}^{(j)}$ im Eingaberaum sehr nahe beieinander liegen, bedeutet dies, dass die Abstände $|x_l^{(i)} - x_l^{(j)}|$ für alle Dimensionen l sehr klein sind. Infolgedessen wird der Term in der Summe $-\sum_{l=1}^{k} \theta_l |x_l^{(i)} - x_l^{(j)}|^{p_l}$ ebenfalls sehr klein (nahe Null) sein. Da $\exp(x)$ für $x \approx 0$ gegen 1 geht, wird die Korrelation $\psi(\vec{x}^{(i)}, \vec{x}^{(j)})$ einen Wert nahe 1 annehmen.
Lassen Sie uns diesen Sachverhalt an einem hypothetischen numerischen Beispiel in einem 3-dimensionalen Raum ($k = 3$) verdeutlichen. Nehmen wir an, wir verwenden den Gauß'schen Kernel mit $p_l = 2$ für alle Dimensionen und die Aktivitätshyperparameter $\vec{\theta} = [1.0, 1.0, 1.0]$.
Betrachten wir drei Trainingspunkte:

- $\vec{x}^{(1)} = [1.0, 2.0, 3.0]$
- $\vec{x}^{(2)} = [1.0001, 2.0002, 3.0003]$ (Dieser Punkt ist extrem nahe an $\vec{x}^{(1)}$)
- $\vec{x}^{(3)} = [5.0, 6.0, 7.0]$ (Dieser Punkt ist weit entfernt von $\vec{x}^{(1)}$ und $\vec{x}^{(2)}$)

Wir berechnen die Korrelationen:

1. **Korrelation zwischen $\vec{x}^{(1)}$ und $\vec{x}^{(2)}$:** Die quadrierten Abstände sind:

   $$|x_1^{(1)} - x_1^{(2)}|^2 = |1.0 - 1.0001|^2 = (-0.0001)^2 = 0.00000001$$

   $$|x_2^{(1)} - x_2^{(2)}|^2 = |2.0 - 2.0002|^2 = (-0.0002)^2 = 0.00000004$$

   $$|x_3^{(1)} - x_3^{(2)}|^2 = |3.0 - 3.0003|^2 = (-0.0003)^2 = 0.00000009$$

   Die Summe der gewichteten quadrierten Abstände (da $\theta_l = 1.0$): $1.0 \cdot 0.00000001 + 1.0 \cdot 0.00000004 + 1.0 \cdot 0.00000009 = 0.00000014$

   Die Korrelation ist: $\psi(\vec{x}^{(1)}, \vec{x}^{(2)}) = \exp(-0.00000014) \approx$ **0.99999986**

   Wie man sieht, ist die Korrelation extrem nahe 1.

2. **Korrelation zwischen $\vec{x}^{(1)}$ und $\vec{x}^{(3)}$:** Die quadrierten Abstände sind:

$$|x_1^{(1)} - x_1^{(3)}|^2 = |1.0 - 5.0|^2 = (-4.0)^2 = 16.0$$

$$|x_2^{(1)} - x_2^{(3)}|^2 = |2.0 - 6.0|^2 = (-4.0)^2 = 16.0$$

$$|x_3^{(1)} - x_3^{(3)}|^2 = |3.0 - 7.0|^2 = (-4.0)^2 = 16.0$$

Die Summe der gewichteten quadrierten Abstände: $1.0 \cdot 16.0 + 1.0 \cdot 16.0 + 1.0 \cdot 16.0 = 48.0$

Die Korrelation ist: $\psi(\vec{x}^{(1)}, \vec{x}^{(3)}) = \exp(-48.0) \approx \mathbf{1.39 \times 10^{-21}}$

Diese Korrelation ist praktisch Null, was zeigt, dass weit entfernte Punkte unkorreliert sind.

Wenn wir eine Ψ-Matrix für diese drei Punkte aufstellen (und die Diagonalelemente $\Psi_{ii} = 1$ sind, plus ein winziges `eps` für numerische Stabilität):

$$\Psi = \begin{pmatrix} \Psi_{11} & \Psi_{12} & \Psi_{13} \\ \Psi_{21} & \Psi_{22} & \Psi_{23} \\ \Psi_{31} & \Psi_{32} & \Psi_{33} \end{pmatrix}$$

Setzen wir die berechneten Werte ein (unter Vernachlässigung des `eps`-Terms für Klarheit in diesem Schritt):

$$\Psi \approx \begin{pmatrix} 1.0 & 0.99999986 & 1.39 \times 10^{-21} \\ 0.99999986 & 1.0 & 1.39 \times 10^{-21} \\ 1.39 \times 10^{-21} & 1.39 \times 10^{-21} & 1.0 \end{pmatrix}$$

Man sieht deutlich, dass die **erste und zweite Zeile (und Spalte)** der Matrix **nahezu identisch** sind, da $\Psi_{11} \approx \Psi_{21}$ und $\Psi_{12} \approx \Psi_{22}$ und $\Psi_{13} \approx \Psi_{23}$ (wobei letztere beide nahezu Null sind).
Wenn Zeilen oder Spalten einer Matrix nahezu identisch sind, bedeutet dies, dass die Matrix **schlecht konditioniert** oder **fast singulär** ist. Eine schlecht konditionierte Matrix ist für numerische Operationen, insbesondere für die Inversion (Ψ^{-1}), problematisch. Die Inversion einer solchen Matrix ist ein rechenintensiver und numerisch anfälliger Schritt sowohl im Maximum-Likelihood-Schätzprozess zur Modellkalibrierung als auch bei der finalen Vorhersage. Dies kann zum **Scheitern der Cholesky-Zerlegung** führen, einer häufig verwendeten Methode zur effizienten und stabilen Matrixinversion im Kriging.

Dieses Problem wird gelöst, indem ein kleiner positiver Wert zur Diagonale der Korrelationsmatrix addiert wird: $\Psi_{new} = \Psi + \lambda I$, wobei I die Identitätsmatrix ist. Diese kleine Addition, oft als **Nugget** bezeichnet, stellt sicher, dass die Matrix gut konditioniert und invertierbar bleibt. Im bereitgestellten Code dient die Variable `eps` diesem Zweck.

Obwohl es als numerischer „Hack“ beginnen mag, hat dieser Nugget-Term eine tief-

greifende und starke statistische Interpretation: Er modelliert Rauschen in den Daten. Dies führt zu zwei unterschiedlichen Arten von Kriging-Modellen:

1. **Interpolierendes Kriging ($\lambda \approx 0$):** Wenn der Nugget null oder sehr klein ist (wie `eps` im Code), wird das Modell gezwungen, exakt durch jeden Trainingsdatenpunkt zu verlaufen. Dies ist für deterministische Computerexperimente geeignet, bei denen die Ausgabe rauschfrei ist.
2. **Regressives Kriging ($\lambda > 0$):** Wenn bekannt ist, dass die Daten verrauscht sind (z. B. aus physikalischen Experimenten oder stochastischen Simulationen), würde das Erzwingen der Interpolation jedes Punktes dazu führen, dass das Modell das Rauschen anpasst, was zu einer übermäßig komplexen und „zappeligen" Oberfläche führt, die schlecht generalisiert. Durch Hinzufügen eines größeren Nugget-Terms λ zur Diagonale teilen wir dem Modell explizit mit, dass es Varianz (Rauschen) in den Beobachtungen gibt. Das Modell ist nicht mehr verpflichtet, exakt durch die Datenpunkte zu verlaufen. Stattdessen wird es eine glattere Regressionskurve erstellen, die den zugrunde liegenden Trend erfasst und gleichzeitig das Rauschen herausfiltert. Die Größe von λ kann als weiterer zu optimierender Hyperparameter behandelt werden, der die Varianz des Rauschens darstellt.

Dieselbe mathematische Operation – das Hinzufügen eines Wertes zur Diagonale von Ψ – dient somit einem doppelten Zweck. Ein winziges, festes `eps` ist eine pragmatische Lösung für die numerische Stabilität. Ein größeres, potenziell optimiertes λ ist ein formaler statistischer Parameter, der das Verhalten des Modells grundlegend von einem exakten Interpolator zu einem rauschfilternden Regressor ändert.

63.4. Eine vollständige exemplarische Vorgehensweise: Kriging der Sinusfunktion

Dieser letzte Teil fasst die gesamte vorangegangene Theorie zusammen, indem er sie auf das bereitgestellte Python-Codebeispiel aus dem Hyperparameter Tuning Cookbook anwendet. Wir werden das Skript Schritt für Schritt durchgehen und die Eingaben, Prozesse und Ausgaben in jeder Phase interpretieren, um eine konkrete Veranschaulichung des Kriging in Aktion zu geben.

Zunächst zeigen wir den gesamten Code, gefolgt von einer detaillierten Erklärung jedes Schrittes.

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import (array, zeros, power, ones, exp, multiply, eye,
                   linspace, spacing, sqrt, arange, append, ravel)
from numpy.linalg import cholesky, solve
from scipy.spatial.distance import squareform, pdist, cdist
```

```
# --- 1. Kriging Basis Functions (Defining the Correlation) ---
# The core of Kriging uses a specialized basis function for correlation:
# psi(x^(i), x) = exp(- sum_{j=1}^k theta_j |x_j^(i) - x_j|^p_j)
# For this 1D example (k=1), and with p_j=2
# (squared Euclidean distance implicit from pdist usage)
# and theta_j = theta (a single value), it simplifies.

def build_Psi(X, theta, eps=sqrt(spacing(1))):
    """
    Computes the correlation matrix Psi based on pairwise squared Euclidean distances
    between input locations, scaled by theta.
    Adds a small epsilon to the diagonal for numerical stability (nugget effect).
    """
    # Calculate pairwise squared Euclidean distances (D) between points in X
    D = squareform(pdist(X, metric='sqeuclidean', out=None, w=theta))
    # Compute Psi = exp(-D)
    Psi = exp(-D)
    # Add a small value to the diagonal for numerical stability (nugget)
    # This is often done in Kriging implementations, though a regression method
    # with a 'nugget' parameter (Lambda) is explicitly mentioned for noisy data later.
    # The source code snippet for build_Psi explicitly includes
    # `multiply(eye(X.shape), eps)`.
    # FIX: Use X.shape to get the number of rows for the identity matrix
    Psi += multiply(eye(X.shape[0]), eps) # Corrected line
    return Psi

def build_psi(X_train, x_predict, theta):
    """
    Computes the correlation vector (or matrix) psi between new prediction locations
    and training data locations.
    """
    # Calculate pairwise squared Euclidean distances (D) between prediction points
    # (x_predict)
    # and training points (X_train).
    # `cdist` computes distances between each pair of the two collections of inputs.
    D = cdist(x_predict, X_train, metric='sqeuclidean', out=None, w=theta)
    # Compute psi = exp(-D)
    psi = exp(-D)
    return psi.T
    # Return transpose to be consistent with literature (n x m or n x 1)
```

```
# --- 2. Data Points for the Sinusoid Function Example ---
# The example uses a 1D sinusoid measured at eight equally spaced x-locations.
```

```
n = 8 # Number of sample locations
X_train = np.linspace(0, 2 * np.pi, n, endpoint=False).reshape(-1, 1)
y_train = np.sin(X_train) # Corresponding y-values (sine of x)
print("--- Training Data (X_train, y_train) ---")
print("x values:\n", np.round(X_train, 2))
print("y values:\n", np.round(y_train, 2))
print("-" * 40)
```

```
--- Training Data (X_train, y_train) ---
x values:
 [[0.  ]
 [0.79]
 [1.57]
 [2.36]
 [3.14]
 [3.93]
 [4.71]
 [5.5 ]]
y values:
 [[ 0.  ]
 [ 0.71]
 [ 1.  ]
 [ 0.71]
 [ 0.  ]
 [-0.71]
 [-1.  ]
 [-0.71]]
----------------------------------------
```

```
# Visualize the data points
plt.figure(figsize=(8, 5))
plt.plot(X_train, y_train, "bo", label=f"Measurements ({n} points)")
plt.title(f"Sin(x) evaluated at {n} points")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.grid(True)
plt.legend()
plt.show()
```

```
# --- 3. Calculating the Correlation Matrix (Psi) ---
# Psi is based on pairwise squared distances between input locations.
# theta is set to 1.0 for this 1D example.
theta = np.array([1.0])
Psi = build_Psi(X_train, theta)
print("\n--- Computed Correlation Matrix (Psi) ---")
print("Dimensions of Psi:", Psi.shape) # Should be (8, 8)
print("First 5x5 block of Psi:\n", np.round(Psi[:5,:5], 2))
print("-" * 40)
```

```
--- Computed Correlation Matrix (Psi) ---
Dimensions of Psi: (8, 8)
First 5x5 block of Psi:
 [[1.   0.54 0.08 0.   0.  ]
 [0.54 1.   0.54 0.08 0.  ]
 [0.08 0.54 1.   0.54 0.08]
 [0.   0.08 0.54 1.   0.54]
 [0.   0.   0.08 0.54 1.  ]]
----------------------------------------
```

```
# --- 4. Selecting New Locations (for Prediction) ---
# We want to predict at m = 100 new locations in the interval [0, 2*pi].
m = 100 # Number of new locations
x_predict = np.linspace(0, 2 * np.pi, m, endpoint=True).reshape(-1, 1)
print("\n--- New Locations for Prediction (x_predict) ---")
print(f"Number of prediction points: {m}")
print("First 5 prediction points:\n", np.round(x_predict[:5], 2).flatten())
print("-" * 40)
```

```
--- New Locations for Prediction (x_predict) ---
Number of prediction points: 100
First 5 prediction points:
 [0.   0.06 0.13 0.19 0.25]
----------------------------------------
```

```
# --- 5. Computing the psi Vector ---
# This vector contains correlations between each of the n observed data points
# and each of the m new prediction locations.
psi = build_psi(X_train, x_predict, theta)
print("\n--- Computed Prediction Correlation Matrix (psi) ---")
print("Dimensions of psi:", psi.shape) # Should be (8, 100)
print("First 5x5 block of psi:\n", np.round(psi[:5,:5], 2))
print("-" * 40)
```

```
--- Computed Prediction Correlation Matrix (psi) ---
Dimensions of psi: (8, 100)
First 5x5 block of psi:
 [[1.   1.   0.98 0.96 0.94]
 [0.54 0.59 0.65 0.7  0.75]
 [0.08 0.1  0.12 0.15 0.18]
 [0.   0.01 0.01 0.01 0.01]
 [0.   0.   0.   0.   0.  ]]
----------------------------------------
```

```
# --- 6. Predicting at New Locations (Kriging Prediction) ---
# The Maximum Likelihood Estimate (MLE) for y_hat is calculated using the formula:
# y_hat(x) = mu_hat + psi.T @ Psi_inv @ (y - 1 * mu_hat)
# Matrix inversion is efficiently performed using Cholesky factorization.
# Step 6a: Cholesky decomposition of Psi
U = cholesky(Psi).T
# Note: `cholesky` in numpy returns lower triangular L,
```

```
# we need U (upper) so transpose L.
# Step 6b: Calculate mu_hat (estimated mean)
one = np.ones(n).reshape(-1, 1) # Vector of ones
mu_hat = (one.T @ solve(U, solve(U.T, y_train))) \
         / (one.T @ solve(U, solve(U.T, one)))
mu_hat = mu_hat.item() # Extract scalar value
print("\n--- Kriging Prediction Calculation ---")
print(f"Estimated mean (mu_hat): {np.round(mu_hat, 4)}")
```

```
--- Kriging Prediction Calculation ---
Estimated mean (mu_hat): -0.0499
```

```
# Step 6c: Calculate predictions f (y_hat) at new locations
# f = mu_hat * ones(m) + psi.T @ Psi_inv @ (y - one * mu_hat)
f_predict = mu_hat * np.ones(m).reshape(-1, 1) \
            + psi.T @ solve(U, solve(U.T, y_train - one * mu_hat))
print(f"Dimensions of predicted values (f_predict): {f_predict.shape}")
# Should be (100, 1)
print("First 5 predicted f values:\n", np.round(f_predict[:5], 2).flatten())
print("-" * 40)
```

```
Dimensions of predicted values (f_predict): (100, 1)
First 5 predicted f values:
 [0.   0.05 0.1  0.15 0.21]
----------------------------------------
```

```
# --- 7. Visualization ---
# Plot the original sinusoid function, the measured points, and the Kriging predictions.
plt.figure(figsize=(10, 6))
plt.plot(x_predict, f_predict, color="orange", label="Kriging Prediction")
plt.plot(x_predict, np.sin(x_predict), color="grey", linestyle='--', \
        label="True Sinusoid Function")
plt.plot(X_train, y_train, "bo", markersize=8, label="Measurements")
plt.title(f"Kriging prediction of sin(x) with {n} points. (theta: {theta})")
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```

63.4.1. Schritt-für-Schritt-Codeausführung und Interpretation

Das Beispiel zielt darauf ab, die Funktion $y = \sin(x)$ unter Verwendung einer kleinen Anzahl von Stichprobenpunkten zu modellieren. Dies ist ein klassisches „Spielzeugproblem“, das nützlich ist, um zu visualisieren, wie sich das Modell verhält.

63.4.1.1. Schritt 1: Datengenerierung

```
n = 8
X_train = np.linspace(0, 2 * np.pi, n, endpoint=False).reshape(-1, 1)
y_train = np.sin(X_train)
```

Hier generiert das Skript die Trainingsdaten.

- `n = 8`: Wir entscheiden uns, die Funktion an acht verschiedenen Stellen abzutasten.
- `X_train`: Dies erstellt ein Array von acht gleichmäßig verteilten Punkten im Intervall $[0, 2\pi]$, die als Trainingspunkte dienen.
- `y_train = np.sin(X_train)`: Dies berechnet die Sinuswerte an den Trainingspunkten. Das Ergebnis ist ein 8×1 Vektor, der die beobachteten Antworten darstellt.

63.4.1.2. Schritt 2: Definition der Korrelationsmatrix (Ψ)

```
theta = np.array([1.0])
Psi = build_Psi(X_train, theta)
```

Dieser Schritt berechnet die 8×8 Korrelationsmatrix Ψ für die Trainingsdaten.

- `theta = np.array([1.0])`: Der Aktivitätshyperparameter θ wird auf 1.0 gesetzt. In einer realen Anwendung würde dieser Wert durch MLE gefunden, aber hier wird er zur Vereinfachung festgesetzt.
- `Psi = build_Psi(X_train, theta)`: Die Funktion `build_Psi` wird aufgerufen. Sie berechnet den gewichteten quadrierten Abstand zwischen jedem Paar von Punkten in `X_train` und wendet dann den exponentiellen Kernel an. Die resultierende `Psi`-Matrix quantifiziert die angenommene Korrelation zwischen all unseren bekannten Datenpunkten. Die diagonalen Elemente sind 1 (plus ein winziges `eps`), und die außerdiagonalen Werte nehmen ab, wenn der Abstand zwischen den entsprechenden Punkten zunimmt.

63.4.1.3. Schritt 3: Definition der Vorhersagepunkte

```
m = 100
x_predict = np.linspace(0, 2 * np.pi, m, endpoint=True).reshape(-1, 1)
```

Wir definieren nun die Orte, an denen wir neue Vorhersagen machen wollen. Wir erstellen ein dichtes Gitter von `m = 100` Punkten, das das gesamte Intervall $[0, 2\pi]$ abdeckt. Dies sind die Punkte, an denen wir unser Surrogatmodell auswerten werden, um eine glatte Kurve zu erzeugen.

63.4.1.4. Schritt 4: Berechnung der Vorhersagekorrelation ($\vec{\psi}$)

```
psi = build_psi(X_train, x_predict, theta)
```

Dieser Schritt berechnet die 8×100 Korrelationsmatrix $\vec{\psi}$. Die Funktion `build_psi` berechnet die Korrelation zwischen jedem der acht Trainingspunkte und jedem der 100 Vorhersagepunkte. Jede Spalte der resultierenden `psi`-Matrix entspricht einem Vorhersagepunkt und enthält seine acht Korrelationswerte mit dem Trainingssatz. Diese Matrix verbindet die neuen, unbekannten Orte mit unserer bestehenden Wissensbasis.

63.4.1.5. Schritt 5: Berechnung der Vorhersage

```
U = cholesky(Psi).T
one = np.ones(n).reshape(-1, 1)
mu_hat = (one.T @ solve(U, solve(U.T, y_train))) /
         (one.T @ solve(U, solve(U.T, one)))
f_predict = mu_hat * np.ones(m).reshape(-1, 1) +
            psi.T @ solve(U, solve(U.T, y_train - one * mu_hat))
```

Dies ist der entscheidende Schritt, in dem die tatsächlichen Vorhersagen gemacht werden.

1. `U = cholesky(Psi).T`: Die numerisch entscheidende Cholesky-Zerlegung von Ψ wird durchgeführt.
2. `mu_hat =...`: Die Maximum-Likelihood-Schätzung für den globalen Mittelwert μ wird unter Verwendung des numerisch stabilen Cholesky-Lösers berechnet. Für diese spezifischen symmetrischen Daten wird `mu_hat` nahe null sein.
3. `f_predict =...`: Die BLUP-Formel wird implementiert. Sie berechnet die Residuen `(y_train - one * mu_hat)`, findet die gewichteten Residuen unter Verwendung des Cholesky-Lösers (`solve(U, solve(U.T,...))`) und berechnet dann die endgültige Vorhersage durch eine gewichtete Summe basierend auf den Vorhersagekorrelationen `psi.T`. Das Ergebnis, `f_predict`, ist ein 100×1 Vektor, der die vorhergesagten Sinuswerte an jedem der `x_predict`-Orte enthält.

63.4.1.6. Schritt 6: Visualisierung

Der letzte Codeblock stellt die Ergebnisse grafisch dar.

- **Messungen (Blaue Punkte):** Die ursprünglichen 8 Datenpunkte werden dargestellt.
- **Wahre Sinusfunktion (Graue gestrichelte Linie):** Die tatsächliche $\sin(x)$-Funktion wird als Referenz dargestellt.
- **Kriging-Vorhersage (Orange Linie):** Die vorhergesagten Werte `f_predict` werden gegen `x_predict` aufgetragen.

Die Grafik zeigt die Schlüsseleigenschaften des Kriging-Modells. Die orangefarbene Linie verläuft *exakt* durch jeden der blauen Punkte und demonstriert damit ihre **interpolierende** Natur (da `eps` sehr klein war). Wichtiger noch, zwischen den Stichprobenpunkten liefert die Vorhersage eine glatte und bemerkenswert genaue Annäherung an die wahre zugrunde liegende Sinuskurve, obwohl das Modell in diesen Bereichen keine anderen Informationen über die Funktion hatte als die acht Stichprobenpunkte und die angenommene Korrelationsstruktur.

63.4.2. Fazit und Ausblick

Wir begannen damit, das Kriging als eine anspruchsvolle Form der Modellierung mit radialen Basisfunktionen einzuordnen und seine Kernphilosophie zu übernehmen, deterministische Funktionen als Realisierungen eines stochastischen Prozesses zu behandeln. Anschließend haben wir seine Architektur zerlegt: den leistungsstarken Korrelationskernel mit seinen Aktivitäts- (θ) und Glattheits- (p) Hyperparametern, die Konstruktion der Korrelationsmatrizen (Ψ und $\vec{\psi}$) und den Ansatz der Maximum-Likelihood-Schätzung zur Modellkalibrierung. Schließlich haben wir die numerischen Best Practices wie die Cholesky-Zerlegung untersucht und die doppelte Rolle des Nugget-Terms für die numerische Stabilität und die statistische Rauschmodellierung besprochen. Die schrittweise exemplarische Vorgehensweise des Sinusbeispiels lieferte eine konkrete Demonstration dieser Konzepte und zeigte, wie eine kleine Menge von Datenpunkten verwendet werden kann, um ein genaues, interpolierendes Modell einer unbekannten Funktion zu erzeugen.

Für den angehenden Praktiker ist dies nur der Anfang. Die Welt des Kriging und der Gauß-Prozess-Regression ist reich an fortgeschrittenen Techniken, die auf diesen Grundlagen aufbauen.

- **Fehlerschätzungen und sequentielles Design:** Ein wesentliches Merkmal, das im Beispielcode nicht untersucht wurde, ist, dass das Kriging nicht nur eine mittlere Vorhersage, sondern auch eine **Varianz** an jedem Punkt liefert, die die Unsicherheit des Modells quantifiziert. Diese Varianz ist in Regionen weit entfernt von Datenpunkten hoch und in deren Nähe niedrig. Diese Fehlerschätzung ist die Grundlage für **aktives Lernen** oder **sequentielles Design**, bei dem Infill-Kriterien wie die **erwartete Verbesserung (EI)** verwendet werden, um den nächsten zu beprobenden Punkt intelligent auszuwählen, wobei ein Gleichgewicht zwischen der Notwendigkeit, vielversprechende Regionen auszunutzen (niedriger vorhergesagter Wert) und der Notwendigkeit, unsichere Regionen zu erkunden (hohe Varianz), hergestellt wird.

- **Gradienten-erweitertes Kriging:** In vielen modernen Simulationsumgebungen ist es möglich, nicht nur den Funktionswert, sondern auch seine Gradienten (Ableitungen) zu geringen zusätzlichen Kosten zu erhalten. Das **Gradienten-erweiterte Kriging** integriert diese Gradienteninformationen in das Modell, was seine Genauigkeit drastisch verbessert und den Aufbau hochpräziser Modelle mit sehr wenigen Stichprobenpunkten ermöglicht.

- **Multi-Fidelity-Modellierung (Co-Kriging):** Oft haben Ingenieure Zugang zu mehreren Informationsquellen mit unterschiedlicher Genauigkeit und Kosten – zum Beispiel ein schnelles, aber ungenaues analytisches Modell und eine langsame, aber hochpräzise CFD-Simulation. **Co-Kriging** ist ein Rahmenwerk, das diese unterschiedlichen Datenquellen verschmilzt, indem es die reichlich vorhandenen billigen Daten verwendet, um einen Basistrend zu etablieren, und die spärlichen

teuren Daten, um ihn zu korrigieren, was zu einem Modell führt, das genauer ist als eines, das nur aus einer der Datenquellen allein erstellt wurde.

63.5. Zusatzmaterialien

Interaktive Webseite

- Eine interaktive Webseite zum Thema **Kriging** ist hier zu finden: Kriging Interaktiv.
- Eine interaktive Webseite zum Thema **MLE** ist hier zu finden: MLE Interaktiv.

Audiomaterial

- Ein Audio zum Thema *Kriging** ist hier zu finden: Cholesky Audio.
- Ein Audio zum Thema *Stochastische Prozesses** ist hier zu finden: Cholesky Audio.

Jupyter-Notebook

- Das Jupyter-Notebook für dieses Lernmodul ist auf GitHub im Hyperparameter-Tuning-Cookbook Repository verfügbar.

64. Lernmodul: Die Cholesky-Zerlegung

64.1. Einführung

Die Cholesky-Zerlegung ist ein grundlegendes Werkzeug in der numerischen linearen Algebra, das speziell für symmetrische, positiv definite Matrizen entwickelt wurde. Sie zerlegt eine solche Matrix in das Produkt einer unteren Dreiecksmatrix und ihrer Transponierten. Diese Zerlegung ist nicht nur rechnerisch effizient, sondern auch numerisch stabil, was sie zu einer bevorzugten Methode in vielen angewandten Bereichen macht, insbesondere im wissenschaftlichen Rechnen und bei der Modellierung von Systemen, die durch teure Computersimulationen beschrieben werden.

Im Kontext von Ersatzmodellen (*surrogate models*) und Gauß-Prozessen (Kriging) spielt die Cholesky-Zerlegung eine zentrale Rolle bei der effizienten Lösung linearer Gleichungssysteme und der Berechnung von Determinanten, die für die Modellanpassung und Vorhersage erforderlich sind.

64.2. Definition und Eigenschaften

64.2.1. Symmetrische, positiv definite Matrizen

Die Cholesky-Zerlegung ist ausschließlich für **symmetrische, positiv definite Matrizen** definiert.

- Eine Matrix A ist **symmetrisch**, wenn sie gleich ihrer Transponierten ist, d.h., $A = A^T$.
- Eine symmetrische Matrix A ist **positiv definit**, wenn alle ihre Eigenwerte positiv sind. Eine äquivalente Definition besagt, dass für jeden von Null verschiedenen Vektor $\vec{x}$ gilt: $\vec{x}^T A\vec{x} > 0$. Diese Eigenschaft ist entscheidend, da sie die Eindeutigkeit einer Lösung garantiert und numerische Stabilität gewährleistet. Wenn eine Matrix nicht positiv definit ist, kann die Cholesky-Zerlegung fehlschlagen und einen Fehler auslösen.

64.2.2. Die Zerlegung

Für eine symmetrische, positiv definite Matrix A findet die Cholesky-Zerlegung eine untere Dreiecksmatrix L (oder eine obere Dreiecksmatrix U) derart, dass:

$$A = LL^T$$

oder

$$A = U^T U.$$

Hierbei ist L^T die Transponierte von L. Wenn NumPy's `cholesky`-Funktion verwendet wird, liefert sie standardmäßig den unteren Dreiecksfaktor L; um den oberen Dreiecksfaktor U zu erhalten, muss L transponiert werden (`U = cholesky(Psi).T`).

64.2.3. Vorteile der Cholesky-Zerlegung

Die Cholesky-Zerlegung bietet erhebliche Vorteile gegenüber der direkten Matrixinversion:

- **Recheneffizienz**: Sie reduziert die rechnerische Komplexität von $O(n^3)$ für die direkte Inversion auf etwa $O(n^3/3)$.
- **Numerische Stabilität**: Die Methode ist numerisch äußerst stabil und robust gegenüber Rundungsfehlern bei Gleitkomma-Berechnungen. Dies ist besonders wichtig bei schlecht konditionierten Matrizen, wo die Determinante nahe Null liegt, was zu Instabilität führen kann.

64.3. Berechnung der Cholesky-Zerlegung

Die Cholesky-Zerlegung kann algorithmisch durchgeführt werden. Für eine Matrix $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ und $L = \begin{pmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{pmatrix}$ gilt $A = LL^T$.

Dies führt zu den Gleichungen:

$$a_{11} = l_{11}^2 \tag{64.1}$$
$$a_{21} = l_{21} l_{11} \tag{64.2}$$
$$a_{22} = l_{21}^2 + l_{22}^2 \tag{64.3}$$

Die Elemente von L können dann wie folgt bestimmt werden:

$$l_{11} = \sqrt{a_{11}} \tag{64.4}$$

$$l_{21} = \frac{a_{21}}{l_{11}} \tag{64.5}$$

$$l_{22} = \sqrt{a_{22} - l_{21}^2} \tag{64.6}$$

Dies ist der Kernalgorithmus der Cholesky-Zerlegung, der sich auf größere Matrizen erweitern lässt.

64.3.1. Beispiel mit Ψ

Betrachten wir die Korrelationsmatrix Ψ:

$$\Psi = \begin{pmatrix} 1 & e^{-1} \\ e^{-1} & 1 \end{pmatrix}$$

Um die Cholesky-Zerlegung $\Psi = LDL^T$ (oder U^TDU) zu berechnen, setzen wir:

$$LDL^T = \begin{pmatrix} 1 & 0 \\ l_{21} & 1 \end{pmatrix} \begin{pmatrix} d_{11} & 0 \\ 0 & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix}$$

Multipliziert man dies aus, erhält man:

$$\begin{pmatrix} d_{11} & d_{11}l_{21} \\ d_{11}l_{21} & d_{11}l_{21}^2 + d_{22} \end{pmatrix}$$

Durch Koeffizientenvergleich mit Ψ:

1. $d_{11} = 1$
2. $l_{21}d_{11} = e^{-1} \Rightarrow l_{21} = e^{-1}$
3. $d_{11}l_{21}^2 + d_{22} = 1 \Rightarrow d_{22} = 1 - e^{-2}$

Die Cholesky-Zerlegung von Ψ ist somit:

$$\Psi = \begin{pmatrix} 1 & 0 \\ e^{-1} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 - e^{-2} \end{pmatrix} \begin{pmatrix} 1 & e^{-1} \\ 0 & 1 \end{pmatrix} = LDL^T$$

Alternativ, ohne die explizite Diagonalmatrix D:

$$\Psi = \begin{pmatrix} 1 & 0 \\ e^{-1} & \sqrt{1 - e^{-2}} \end{pmatrix} \begin{pmatrix} 1 & e^{-1} \\ 0 & \sqrt{1 - e^{-2}} \end{pmatrix} = U^TU.$$

64.4. Anwendungen der Cholesky-Zerlegung

Die Cholesky-Zerlegung ist ein vielseitiges Werkzeug in der numerischen linearen Algebra:

64.4.1. Lösung linearer Gleichungssysteme

Anstatt die Inverse A^{-1} explizit zu berechnen (was numerisch instabil und teuer ist), kann ein lineares System

$$A\vec{x} = \vec{b}$$

mittels Cholesky-Zerlegung in zwei einfachere Dreieckssysteme zerlegt werden.

1. **Vorwärtssubstitution**: Löse $L\vec{y} = \vec{b}$ nach $\vec{y}$. Da L eine untere Dreiecksmatrix ist, lässt sich dies leicht rekursiv lösen.
2. **Rückwärtssubstitution**: Löse $L^T\vec{x} = \vec{y}$ nach $\vec{x}$. Da L^T (oder U) eine obere Dreiecksmatrix ist, lässt sich dies ebenfalls leicht rekursiv lösen.

Dieser zweistufige Prozess ist viel schneller und numerisch stabiler als die direkte Inversion. Im Python-Code wird dies oft durch Funktionen wie `scipy.linalg.cho_solve` oder `numpy.linalg.solve` nach der Cholesky-Zerlegung (`L = cholesky(Psi, lower=True)`) erledigt.

64.4.2. Berechnung von Determinanten

Für die Berechnung des Logarithmus des Absolutwerts der Determinante einer symmetrischen, positiv definiten Matrix Ψ, was in der Maximum-Likelihood-Schätzung oft vorkommt, ist die Cholesky-Zerlegung besonders nützlich. Es gilt:

$$\ln(|\Psi|) = 2\sum_{i=1}^{n} \ln(L_{ii})$$

wobei L_{ii} die Diagonalelemente der Cholesky-Faktorisierung L sind. Dieser Ansatz vermeidet die direkte Berechnung der Determinante, die bei schlecht konditionierten Matrizen sehr kleine Werte annehmen und zu numerischer Instabilität führen kann.

64.4.3. Kriging und Gauß-Prozess-Regression (GPR)

Im Bereich der Gauß-Prozess-Modellierung, auch bekannt als Kriging, ist die Cholesky-Zerlegung ein Kernbestandteil.

- **Modellanpassung (MLE)**: Bei der Schätzung der Modellparameter über die Maximum-Likelihood-Methode erfordert die Berechnung der Likelihood-Funktion (oder der konzentrierten Log-Likelihood) mehrere Matrixinversionen. Die Cholesky-Zerlegung, gefolgt von Vorwärts- und Rückwärtssubstitution, ist der schnellste und stabilste Weg, dies zu tun.
- **Vorhersage (BLUP)**: Die Vorhersage neuer Werte im Kriging-Modell, bekannt als Best Linear Unbiased Predictor (BLUP), beinhaltet ebenfalls die Lösung linearer Systeme mit der Korrelationsmatrix. Auch hier kommt die Cholesky-Zerlegung zum Einsatz, um die Vorhersage effizient und stabil zu berechnen.

- **Numerische Stabilität und Nugget-Effekt**: Wenn Trainingspunkte sehr nahe beieinander liegen, kann die Korrelationsmatrix schlecht konditioniert oder nahezu singulär werden, was die Cholesky-Zerlegung zum Scheitern bringen kann. Um dies zu verhindern, wird ein kleiner positiver Wert, der so genannte **Nugget-Effekt** (λ oder `eps`), zur Diagonale der Korrelationsmatrix addiert:

  $$\Psi_{new} = \Psi + \lambda I.$$

  Dieser Nugget stellt sicher, dass die Matrix streng positiv definit und somit die Cholesky-Zerlegung erfolgreich ist. Der Nugget kann auch als statistischer Parameter interpretiert werden, der Rauschen im Modell berücksichtigt und das Modell von einem exakten Interpolator zu einem rauschfilternden Regressor ändert.

64.4.4. Generierung von Stichproben aus multivariaten Normalverteilungen

Die Cholesky-Zerlegung kann auch verwendet werden, um Zufallsstichproben aus einer multivariaten Normalverteilung zu generieren. Wenn $\vec{u}$ ein Vektor von unabhängigen standardnormalverteilten Zufallsvariablen ist und $K = LL^T$ die Cholesky-Zerlegung der Kovarianzmatrix K ist, dann hat der Vektor $\vec{x} = \vec{\mu} + L\vec{u}$ die gewünschte multivariate Normalverteilung mit Mittelwert $\vec{\mu}$ und Kovarianzmatrix K. Auch hier kann ein kleiner "Nugget"-Term zur Kovarianzmatrix hinzugefügt werden, um numerische Stabilität zu gewährleisten, da die Eigenwerte von Kovarianzmatrizen schnell abfallen können.

64.4.5. Anwendungen in Optimierungsalgorithmen

Obwohl die Cholesky-Zerlegung selbst kein Optimierungsalgorithmus ist (im Gegensatz zu Gradientenabstieg, Newton-Verfahren oder BFGS), ist sie ein entscheidendes Werkzeug zur Lösung der linearen Systeme, die in vielen fortgeschrittenen Optimierungsverfahren auftreten. Beispielsweise erfordert das Newton-Verfahren zur Minimierung einer Funktion die Lösung eines linearen Systems mit der Hesse-Matrix. Wenn diese Hesse-Matrix symmetrisch und positiv definit ist, kann die Cholesky-Zerlegung für die effiziente und stabile Lösung dieses Systems genutzt werden.

64.5. Implementierung in Python

Python-Bibliotheken wie `numpy` bieten effiziente Funktionen zur Cholesky-Zerlegung, z.B. `np.linalg.cholesky(A)`. Wenn die Matrix nicht positiv definit ist, wird ein `LinAlgError` ausgelöst.

Example 64.1 (Cholesky-Zerlegung in Python).

```python
import numpy as np

# Definieren der symmetrischen, positiv-definiten Matrix A
A = np.array([[25., 15., -5.],
              [15., 18.,  0.],
              [-5.,  0., 11.]])

# Versuch, die Cholesky-Zerlegung durchzuführen
try:
    # Berechne die untere Dreiecksmatrix L
    L = np.linalg.cholesky(A)

    print("Matrix A:\n", A)
    print("\nUntere Dreiecksmatrix L:\n", L)

    # Überprüfung: L * L^T sollte wieder A ergeben
    # np.dot für Matrixmultiplikation, L.T für Transponierung
    A_reconstructed = np.dot(L, L.T)
    print("\nRekonstruierte Matrix L * L.T:\n", A_reconstructed)

    # Überprüfen, ob die Rekonstruktion erfolgreich war
    print("\nIst die Rekonstruktion nahe an der Originalmatrix?:",
          np.allclose(A, A_reconstructed))

except np.linalg.LinAlgError:
    print("\nFehler: Die Matrix ist nicht positiv-definit.")
```

```
Matrix A:
 [[25. 15. -5.]
 [15. 18.  0.]
 [-5.  0. 11.]]

Untere Dreiecksmatrix L:
 [[ 5.  0.  0.]
 [ 3.  3.  0.]
 [-1.  1.  3.]]

Rekonstruierte Matrix L * L.T:
 [[25. 15. -5.]
 [15. 18.  0.]
 [-5.  0. 11.]]
```

```
Ist die Rekonstruktion nahe an der Originalmatrix?: True
```

Example 64.2 (Lösung von Ax=b)**.** Die primäre Anwendung der Cholesky-Zerlegung ist die effiziente Lösung von linearen Gleichungssystemen

$$A\vec{x} = \vec{b},$$

bei denen A eine symmetrische, positiv definite Matrix ist. Anstatt die Matrix A direkt zu invertieren – ein rechenintensiver und numerisch oft instabiler Prozess – zerlegt man das Problem in zwei einfachere Schritte:

1. **Zerlegung:** Zuerst wird A in LL^T zerlegt. Das System wird zu $LL^T\vec{x} = \vec{b}$.
2. **Substitution:** Das System wird in zwei Schritten gelöst, indem ein Hilfsvektor $\vec{y}$ eingeführt wird:
   (i) **Vorwärtssubstitution:** Löse $L\vec{y} = \vec{b}$ nach $\vec{y}$. Da L eine untere Dreiecksmatrix ist, kann dies sehr effizient geschehen.
   (ii) **Rückwärtssubstitution:** Löse $L^T\vec{x} = \vec{y}$ nach $\vec{x}$. Da L^T eine obere Dreiecksmatrix ist, ist auch dieser Schritt sehr effizient.

Die Bibliothek `SciPy` bietet optimierte Funktionen für diesen Prozess.

```python
from scipy.linalg import solve_triangular, cho_solve
import numpy as np

# Verwenden der gleichen Matrix A und der berechneten Matrix L
A = np.array([[25., 15., -5.],
              [15., 18.,  0.],
              [-5.,  0., 11.]])
L = np.linalg.cholesky(A)

# Definieren des Vektors b
b = np.array([10., 5., 8.])

# --- Methode 1: Manuelle Vorwärts- und Rückwärtssubstitution ---
# Schritt 2a: Löse Ly = b für y (Vorwärtssubstitution)
y = solve_triangular(L, b, lower=True)
print("Zwischenvektor y:\n", y)

# Schritt 2b: Löse L^T x = y für x (Rückwärtssubstitution)
x1 = solve_triangular(L.T, y, lower=False)
print("\nLösung x (manuelle Substitution):\n", x1)

# --- Methode 2: Direkte Lösung mit cho_solve ---
# Diese Funktion kombiniert die Schritte für maximale Effizienz
# Sie benötigt die Cholesky-Faktorisierung (L) und b
```

```
x2 = cho_solve((L, True), b) # (L, True) bedeutet, dass L eine untere Dreiecksmatrix i
print("\nLösung x (mit cho_solve):\n", x2)

# Überprüfung der Lösung: A * x sollte wieder b ergeben
print("\nÜberprüfung A * x:\n", np.dot(A, x2))
```

```
Zwischenvektor y:
 [ 2.         -0.33333333  3.44444444]

Lösung x (manuelle Substitution):
 [ 0.92592593 -0.49382716  1.14814815]

Lösung x (mit cho_solve):
 [ 0.92592593 -0.49382716  1.14814815]

Überprüfung A * x:
 [10.  5.  8.]
```

64.6. Fazit

Die Cholesky-Zerlegung ist ein unverzichtbares Werkzeug in der numerischen linearen Algebra für symmetrische, positiv definite Matrizen. Ihre Effizienz und Robustheit machen sie zur bevorzugten Methode für Aufgaben wie die Lösung linearer Gleichungssysteme, die Berechnung von Determinanten und insbesondere in den rechenintensiven Prozessen von Gauß-Prozessen und Ersatzmodellen. Das Verständnis ihrer Funktionsweise und ihrer Anwendungsbereiche ist für jeden Ingenieur und Naturwissenschaftler, der mit komplexen Berechnungen arbeitet, von grundlegender Bedeutung.

64.7. Zusatzmaterialien

Interaktive Webseite

- Eine interaktive Webseite zum Thema **Cholesky-Zerlegung** ist hier zu finden: Cholesky Interaktiv.

Audiomaterial

- Ein Audio zum Thema **Cholesky-Zerlegung** ist hier zu finden: Cholesky Audio.

Jupyter-Notebook

- Das Jupyter-Notebook für dieses Lernmodul ist auf GitHub im Hyperparameter-Tuning-Cookbook Repository verfügbar.

65. Lernmodul: Erweiterung des Kriging-Modells: Numerische Optimierung der Hyperparameter

65.1. Einleitung

Das vorhergehende Lernmodul hat die konzeptionellen Grundlagen und die mathematische Architektur von Kriging-Modellen vorgestellt, illustriert am Beispiel der Sinusfunktion. In dieser Einführung wurde der Aktivitätsparameter θ aus Gründen der Einfachheit auf einen festen Wert (1.0) gesetzt. In realen Anwendungen ist es jedoch entscheidend, diese Parameter optimal aus den vorliegenden Daten zu bestimmen, um die bestmögliche Modellgüte zu erzielen.

Dieses Dokument baut auf dem bestehenden Wissen auf und erläutert, wie die Kriging-Hyperparameter, insbesondere der Aktivitätsparameter θ, numerisch optimiert werden können. Wir werden uns auf die Maximierung der sogenannten "konzentrierten Log-Likelihood-Funktion" konzentrieren, einem gängigen Ansatz zur Parameterschätzung in Kriging-Modellen. Die gezeigte Python-Code-Erweiterung des Sinusfunktions-Beispiels verdeutlicht die praktische Umsetzung.

65.2. Kriging-Hyperparameter: Theta ($\vec{\theta}$) und p ($\vec{p}$)

Im Kriging-Modell steuern zwei wichtige Vektoren von Hyperparametern die Form und die Eigenschaften der Korrelationsfunktion:

- **Aktivitätsparameter** $\vec{\theta} = (\theta_1, \theta_2, \dots, \theta_k)^T$: Dieser Vektor regelt, wie schnell die Korrelation zwischen zwei Punkten mit zunehmendem Abstand in jeder Dimension abfällt. Ein großer Wert für θ_j in einer Dimension j bedeutet, dass die Funktion in dieser Dimension sehr "aktiv" ist oder sich schnell ändert, und somit nur Punkte in unmittelbarer Nähe stark korrelieren. Dies ermöglicht eine automatische Relevanzbestimmung, bei der wichtige Variablen durch höhere θ-Werte identifiziert werden können.

- **Glattheitsparameter** $\vec{p} = (p_1, p_2, \dots, p_k)^T$: Dieser Vektor beeinflusst die Glattheit der Vorhersagefunktion in jeder Dimension. Üblicherweise liegen die Werte für p_j zwischen 1 und 2. Im vorherigen Lernmodul wurde implizit $p_j = 2$ verwendet (durch die "sqeuclidean"-Distanzmetrik), was zu unendlich differenzierbaren, sehr glatten Funktionen führt. Eine Optimierung von $\vec{p}$ ist möglich, wird aber in diesem Beispiel aus Gründen der Komplexität ausgeklammert, da $p_j = 2$ oft als Standard für glatte Funktionen angenommen wird.

65.3. Die Notwendigkeit der Optimierung: Die konzentrierte Log-Likelihood

Um die optimalen Werte für $\vec{\theta}$ (und $\vec{p}$) zu finden, wird häufig die Maximum-Likelihood-Schätzung (MLE) verwendet. Die Grundidee der MLE besteht darin, diejenigen Parameterwerte zu finden, die die Wahrscheinlichkeit maximieren, die tatsächlich beobachteten Daten zu erhalten.

Die zu maximierende Funktion ist die **Log-Likelihood-Funktion**. Für gegebene $\vec{\theta}$ und $\vec{p}$ (und somit eine feste Korrelationsmatrix Ψ) können die Schätzer für den globalen Mittelwert $\hat{\mu}$ und die Prozessvarianz $\hat{\sigma}^2$ analytisch abgeleitet werden. Durch Einsetzen dieser Schätzer in die Log-Likelihood-Funktion erhalten wir die sogenannte **konzentrierte Log-Likelihood-Funktion**:

$$\ln(L) \approx -\frac{n}{2}\ln(\hat{\sigma}^2) - \frac{1}{2}\ln|\vec{\Psi}|$$

Hierbei ist:

- n: Die Anzahl der Beobachtungspunkte.
- $\hat{\sigma}^2$: Der Maximum-Likelihood-Schätzer der Prozessvarianz.
- $|\vec{\Psi}|$: Die Determinante der Korrelationsmatrix $\vec{\Psi}$.

Die direkte Maximierung dieser Funktion ist mathematisch schwierig, da sie bezüglich $\vec{\theta}$ und $\vec{p}$ nicht analytisch differenzierbar ist. Daher wird eine **numerische Optimierung** eingesetzt, um die Parameter zu finden, die die konzentrierte Log-Likelihood maximieren.

65.4. Numerische Optimierungsalgorithmen

Für die numerische Optimierung der Parameter $\vec{\theta}$ und $\vec{p}$ können verschiedene Algorithmen verwendet werden, darunter:

- Nelder-Mead-Simplex-Verfahren
- Konjugierte Gradienten-Verfahren
- Simulated Annealing
- Differential Evolution

Die `scipy.optimize`-Bibliothek in Python bietet eine umfassende Sammlung solcher Optimierungsfunktionen. Da die meisten Optimierungsalgorithmen in `scipy.optimize` auf Minimierung ausgelegt sind, wird die **negative** konzentrierte Log-Likelihood-Funktion als Optimierungsziel verwendet.

Ein wichtiger numerischer Aspekt bei der Berechnung der Log-Likelihood ist die Determinante von Ψ. Für schlecht konditionierte Matrizen kann $|\Psi|$ gegen Null gehen, was zu numerischer Instabilität führen kann. Um dies zu vermeiden, wird der Logarithmus der Determinante $\ln(|\Psi|)$ stabiler berechnet, indem man die Cholesky-Zerlegung $\Psi = LL^T$ nutzt und dann $\ln(|\Psi|) = 2\sum_{i=1}^{n} \ln(L_{ii})$ berechnet.

Für die Suche nach θ ist es sinnvoll, Suchbereiche auf einer logarithmischen Skala zu definieren, typischerweise von 10^{-3} bis 10^{2}. Es ist auch ratsam, die Eingabedaten auf den Bereich zwischen Null und Eins zu skalieren, um die Konsistenz der θ-Werte über verschiedene Probleme hinweg zu gewährleisten.

65.5. Erweiterung des Sinusfunktions-Beispiels mit Hyperparameter-Optimierung

Wir erweitern nun den Beispielcode aus dem "Lernmodul: Eine Einführung in Kriging" (Kriging-Anpassung an eine Sinusfunktion mit 8 Punkten), um den Aktivitätsparameter θ numerisch zu optimieren.

Die Hauptänderung besteht in der Definition einer neuen Zielfunktion, `neg_log_likelihood`, die von `scipy.optimize.minimize` minimiert wird. Diese Funktion nimmt die zu optimierenden Parameter (hier `theta`) entgegen und berechnet die negative konzentrierte Log-Likelihood basierend auf den Trainingsdaten.

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import (exp, multiply, eye, linspace, spacing, sqrt)
from numpy.linalg import cholesky, solve
from scipy.spatial.distance import squareform, pdist, cdist
from scipy.optimize import minimize # Für die Optimierung
```

65.5.1. Kriging-Basisfunktionen (Definition der Korrelation)

Der Kernel von Kriging verwendet eine spezialisierte Basisfunktion für die Korrelation:

$$\psi(x^{(i)}, x) = \exp(-\sum_{j=1}^{k} \theta_j |x_j^{(i)} - x_j|^{p_j}).$$

Für dieses 1D-Beispiel ($k = 1$) und mit $p_j = 2$ (quadratische euklidische Distanz implizit durch `pdist`-Nutzung) und $\theta_j = \theta$ (ein einzelner Wert) vereinfacht es sich.

```
def build_Psi(X, theta, eps=sqrt(spacing(1))):
    """
    Berechnet die Korrelationsmatrix Psi basierend auf paarweisen
    quadratischen euklidischen Distanzen zwischen Eingabelokationen,
    skaliert mit theta.
    Fügt ein kleines Epsilon zur Diagonalen für numerische Stabilität
    hinzu (Nugget-Effekt).
    Hinweis: p_j ist implizit 2 aufgrund der 'sqeuclidean'-Metrik.
    """
    # Sicherstellen, dass theta ein 1D-Array für das 'w'-Argument
    # von cdist/pdist ist
    if not isinstance(theta, np.ndarray) or theta.ndim == 0:
        theta = np.array([theta])

    D = squareform(pdist(X, metric='sqeuclidean', w=theta))
    Psi = exp(-D)
    # Ein kleiner Wert wird zur Diagonalen hinzugefügt für
    # numerische Stabilität (Nugget)
    # Korrektur: X.shape für die Anzahl der Zeilen der
    # Identitätsmatrix
    Psi += multiply(eye(X.shape[0]), eps)
    return Psi

def build_psi(X_train, x_predict, theta):
    """
    Berechnet den Korrelationsvektor (oder Matrix) psi zwischen
    neuen Vorhersageorten und Trainingsdatenlokationen.
    """
    # Sicherstellen, dass theta ein 1D-Array für das 'w'-Argument
    # von cdist/pdist ist
    if not isinstance(theta, np.ndarray) or theta.ndim == 0:
        theta = np.array([theta])

    D = cdist(x_predict, X_train, metric='sqeuclidean', w=theta)
```

```
    psi = exp(-D)
    return psi.T
    # Transponieren, um konsistent mit der Literatur zu sein
    # (n x m oder n x 1)
```

65.5.2. Zielfunktion für die Hyperparameter-Optimierung (Negative Log-Likelihood)

```
def neg_log_likelihood(params, X_train, y_train):
    """
    Berechnet die negative konzentrierte Log-Likelihood für das Kriging-Modell.
    params: ein 1D-Numpy-Array, wobei params theta ist.
            (Falls auch p optimiert würde, wäre es params usw.)
    X_train: (n, k)-Matrix der Trainings-Eingabelokationen
    y_train: (n, 1)-Vektor der Trainings-Ausgabewerte
    """
    theta = params
    # Für dieses Beispiel ist p implizit auf 2 festgelegt
    # (durch 'sqeuclidean' in build_Psi).
    # Falls p optimiert würde, müsste es hier aus 'params' extrahiert
    # und an build_Psi übergeben werden
    n = X_train.shape[0]

    # 1. Korrelationsmatrix Psi aufbauen
    Psi = build_Psi(X_train, theta)

    # 2. mu_hat berechnen (MLE des Mittelwerts)
    # Verwendung der Cholesky-Zerlegung für stabile Inversion
    try:
        # numpy.cholesky gibt L (untere Dreiecksmatrix) zurück,
        # daher transponieren für U (obere)
        U = cholesky(Psi).T
    except np.linalg.LinAlgError:
        # Bei Fehlern (z.B. wenn Psi nicht positiv definit ist,
        # durch schlechte theta-Werte)
        # einen sehr großen Wert zurückgeben, um diese Parameter zu bestrafen
        return 1e15

    one = np.ones(n).reshape(-1, 1)
    # Stabile Berechnung von Psi_inv @ y und Psi_inv @ one
    Psi_inv_y = solve(U, solve(U.T, y_train))
    Psi_inv_one = solve(U, solve(U.T, one))
```

```
    # Berechnung von mu_hat
    mu_hat = (one.T @ Psi_inv_y) / (one.T @ Psi_inv_one)
    mu_hat = mu_hat.item() # Skalaren Wert extrahieren

    # 3. sigma_hat_sq berechnen (MLE der Prozessvarianz)
    y_minus_mu_one = y_train - one * mu_hat
    # Korrekte Berechnung: (y-1*mu_hat).T @ Psi_inv @ (y-1*mu_hat) / n
    sigma_hat_sq = (y_minus_mu_one.T @ \
                    solve(U, solve(U.T, y_minus_mu_one))) / n
    sigma_hat_sq = sigma_hat_sq.item()

    if sigma_hat_sq < 1e-10: # Sicherstellen, dass sigma_hat_sq
        # nicht-negativ und nicht zu klein ist
        return 1e15 # Sehr großen Wert zurückgeben zur Bestrafung

    # 4. Log-Determinante von Psi mittels Cholesky-Zerlegung für
    # Stabilität berechnen.
    # ln(|Psi|) = 2 * Summe(ln(L_ii)) wobei L die untere
    # Dreiecksmatrix der Cholesky-Zerlegung ist
    log_det_Psi = 2 * np.sum(np.log(np.diag(U.T))) # U.T ist L

    # 5. Negative konzentrierte Log-Likelihood berechnen
    # ln(L) = - (n/2) * ln(sigma_hat_sq) - (1/2) * ln(|Psi|)
    # Zu minimieren ist -ln(L)
    nll = 0.5 * n * np.log(sigma_hat_sq) + 0.5 * log_det_Psi
    return nll
```

65.5.3. Datenpunkte für das Sinusfunktions-Beispiel

Das Beispiel verwendet eine 1D-Sinusfunktion, gemessen an `n_train` gleichmäßig verteilten x-Lokationen.

```
n_train = 4 # Anzahl der Stichprobenlokationen
X_train = np.linspace(0, 2 * np.pi, n_train,\
                      endpoint=False).reshape(-1, 1)
y_train = np.sin(X_train) # Zugehörige y-Werte (Sinus von x)

# --- Originale Vorhersage-Einrichtung (festes theta=1.0) ---
theta_fixed = np.array([1.0])
Psi_fixed = build_Psi(X_train, theta_fixed)
U_fixed = cholesky(Psi_fixed).T
one_fixed = np.ones(n_train).reshape(-1, 1)
mu_hat_fixed = (one_fixed.T @ solve(U_fixed,\
```

```
                solve(U_fixed.T, y_train)))\
                / (one_fixed.T @ solve(U_fixed,\
               solve(U_fixed.T, one_fixed)))
mu_hat_fixed = mu_hat_fixed.item()

m_predict = 100 # Anzahl der neuen Lokationen für die Vorhersage
x_predict = np.linspace(0, 2 * np.pi, m_predict,\
                        endpoint=True).reshape(-1, 1)
psi_fixed = build_psi(X_train, x_predict, theta_fixed)
f_predict_fixed = mu_hat_fixed * np.ones(m_predict).reshape(-1, 1) + \
                  psi_fixed.T @ solve(U_fixed, solve(U_fixed.T,\
                  y_train - one_fixed * mu_hat_fixed))
```

65.5.4. Optimierung von Theta

```
initial_theta_guess = np.array([1.0]) # Startwert für Theta
# Suchbereiche für Theta (z.B. von 1e-3 bis 1e2 auf linearer Skala)
# SciPy minimize erwartet Suchbereiche als Tupel von (min, max)
# für jeden Parameter
bounds = [(0.001, 100.0)] # Für Theta
print("\n--- Starte Hyperparameter-Optimierung für Theta ---")
# 'L-BFGS-B' wird verwendet, da es Beschränkungen (bounds) unterstützt
# und gut für kontinuierliche Optimierung ist.
result = minimize(neg_log_likelihood, initial_theta_guess,\
                  args=(X_train, y_train),
                  method='L-BFGS-B', bounds=bounds)

opt_theta = result.x
opt_nll = result.fun

print(f"Optimierung erfolgreich: {result.success}")
# Extract the first element if it's a single value
print(f"Optimales Theta: {opt_theta[0]:.4f}")
print(f"Minimaler Negativer Log-Likelihood: {opt_nll:.4f}")
```

```
--- Starte Hyperparameter-Optimierung für Theta ---
Optimierung erfolgreich: True
Optimales Theta: 0.3157
Minimaler Negativer Log-Likelihood: -1.4767
```

65.5.5. Vorhersage mit optimiertem Theta

```
Psi_opt = build_Psi(X_train, opt_theta)
U_opt = cholesky(Psi_opt).T
one_opt = np.ones(n_train).reshape(-1, 1)
mu_hat_opt = (one_opt.T @ solve(U_opt, solve(U_opt.T, y_train))) / \
                   (one_opt.T @ solve(U_opt, solve(U_opt.T, one_opt)))
mu_hat_opt = mu_hat_opt.item()

psi_opt = build_psi(X_train, x_predict, opt_theta)
f_predict_opt = mu_hat_opt * np.ones(m_predict).reshape(-1, 1) + \
                      psi_opt.T @ solve(U_opt, solve(U_opt.T, y_train\
                      - one_opt * mu_hat_opt))
```

65.5.6. Visualisierung der Ergebnisse

```
plt.figure(figsize=(10, 6))
plt.plot(x_predict, np.sin(x_predict), color="grey", linestyle='--',\
            label="Wahre Sinusfunktion")
plt.plot(X_train, y_train, "bo", markersize=8,\
            label=f"Messpunkte ({n_train} Punkte)")
plt.plot(x_predict, f_predict_fixed, color="red", linestyle=':',\
     label=f"Kriging-Vorhersage (Fixes Theta={theta_fixed[0]:.1f})")
plt.plot(x_predict, f_predict_opt, color="orange",\
     label=f"Kriging-Vorhersage (Optimiertes Theta={opt_theta[0]:.2f})")
plt.title(f"Kriging-Vorhersage der Sinusfunktion mit {n_train} Punkten\
            \nOptimierung des Aktivitätsparameters Theta")
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```

65.5.7. 6. Ergebnisse und Diskussion

Die grafische Darstellung der Ergebnisse zeigt die Verbesserung der Kriging-Vorhersage nach der Optimierung des Aktivitätsparameters θ. Die Kurve, die mit dem optimierten θ-Wert generiert wurde, passt sich in der Regel besser an die Trainingsdaten an und bildet den wahren Funktionsverlauf präziser ab, als dies mit einem willkürlich gewählten θ-Wert der Fall wäre. Der Optimierungsalgorithmus findet den θ-Wert, der die Korrelationsstruktur der Daten am besten erklärt und somit ein "realistischeres" Modell der zugrunde liegenden Funktion liefert.

In diesem 1D-Beispiel ist der Unterschied möglicherweise subtil, aber in höherdimensionalen Problemen, wo Variablen unterschiedliche "Aktivitäten" aufweisen, ist die automatische Bestimmung von $\vec{\theta}$ entscheidend für die Modellgenauigkeit und die Identifizierung wichtiger Input-Variablen.

65.5.8. 7. Fazit und Ausblick

Dieses Lernmodul hat gezeigt, wie die Maximum-Likelihood-Schätzung in Verbindung mit numerischen Optimierungsverfahren genutzt werden kann, um die Hyperparameter eines Kriging-Modells optimal an die Daten anzupassen. Die Optimierung der konzentrierten Log-Likelihood-Funktion ist ein Standardansatz, der die Robustheit und Genauigkeit von Kriging-Modellen erheblich verbessert.

Für fortgeschrittenere Anwendungen könnten weitere Schritte unternommen werden:

- **Optimierung von $\vec{p}$**: Der Glattheitsparameter $\vec{p}$ könnte ebenfalls in den Optimierungsprozess einbezogen werden, um noch flexiblere Anpassungen zu ermöglichen.
- **Kriging-Regression für verrauschte Daten**: Falls die Trainingsdaten Rauschen enthalten (z.B. aus physikalischen Experimenten), kann ein zusätzlicher "Nugget"-Parameter λ in der Korrelationsmatrix optimiert werden. Dies transformiert das interpolierende Kriging in ein regressives Kriging, das Rauschen explizit modelliert und eine glattere Vorhersagekurve liefert.
- **Aktives Lernen und Expected Improvement (EI)**: Kriging-Modelle liefern nicht nur Vorhersagen, sondern auch Unsicherheitsschätzungen (Varianz) an jedem Punkt. Dies ermöglicht den Einsatz von "Infill-Kriterien" wie Expected Improvement (EI), um den nächsten vielversprechendsten Punkt für eine Funktionsauswertung intelligent auszuwählen, was besonders bei teuren Simulationen effizient ist.
- **Co-Kriging (Multi-Fidelity-Modellierung)**: Wenn Daten aus verschiedenen Quellen mit unterschiedlicher Genauigkeit und Kosten verfügbar sind, kann Co-Kriging (auch Multi-Fidelity-Modellierung genannt) diese Daten integrieren, um genauere Modelle zu erstellen.

Diese erweiterten Konzepte bilden die Grundlage für robuste und effiziente Optimierungsprozesse in vielen technischen und wissenschaftlichen Disziplinen.

65.6. Zusatzmaterialien

Interaktive Webseite

- Eine interaktive Webseite zum Thema **Kriging: Optimierung der Hyperparameter** ist hier zu finden: Kriging Interaktiv.

Jupyter-Notebook

- Das Jupyter-Notebook für dieses Lernmodul ist auf GitHub im Hyperparameter-Tuning-Cookbook Repository verfügbar.

66. Lernmodul: Erweiterung des Kriging-Modells zu einer Klasse (Python Code)

```python
import numpy as np
import matplotlib.pyplot as plt
from numpy import (exp, multiply, eye, linspace, spacing, sqrt)
from numpy.linalg import cholesky, solve
from scipy.spatial.distance import squareform, pdist, cdist
from scipy.optimize import minimize # Für die Optimierung


class KrigingRegressor:
    """Ein Kriging-Regressionsmodell mit Hyperparameter-Optimierung.

    Attributes:
        initial_theta (float): Startwert für den Aktivitätsparameter Theta.
        bounds (list): Liste von Tupeln für die Grenzen der Hyperparameter-Optimierung.
        opt_theta_ (float): Optimierter Theta-Wert nach dem Fitting.
        X_train_ (array): Trainings-Eingabedaten.
        y_train_ (array): Trainings-Zielwerte.
        U_ (array): Cholesky-Zerlegung der Korrelationsmatrix.
        mu_hat_ (float): Geschätzter Mittelwert.
    """


    def __init__(self, initial_theta=1.0, bounds=[(0.001, 100.0)]):
        self.initial_theta = initial_theta
        self.bounds = bounds

    def _build_Psi(self, X, theta, eps=sqrt(spacing(1))):
        """Berechnet die Korrelationsmatrix Psi."""
        if not isinstance(theta, np.ndarray) or theta.ndim == 0:
            theta = np.array([theta])
```

```
        D = squareform(pdist(X, metric='sqeuclidean', w=theta))
        Psi = exp(-D)
        Psi += multiply(eye(X.shape[0]), eps)
        return Psi

    def _build_psi(self, X_train, x_predict, theta):
        """Berechnet den Korrelationsvektor psi."""
        if not isinstance(theta, np.ndarray) or theta.ndim == 0:
            theta = np.array([theta])

        D = cdist(x_predict, X_train, metric='sqeuclidean', w=theta)
        psi = exp(-D)
        return psi.T

    def _neg_log_likelihood(self, params, X_train, y_train):
        """Berechnet die negative konzentrierte Log-Likelihood."""
        theta = params
        n = X_train.shape[0]

        try:
            Psi = self._build_Psi(X_train, theta)
            U = cholesky(Psi).T
        except np.linalg.LinAlgError:
            return 1e15

        one = np.ones(n).reshape(-1, 1)

        # Berechne mu_hat (MLE des Mittelwerts)
        Psi_inv_y = solve(U, solve(U.T, y_train))
        Psi_inv_one = solve(U, solve(U.T, one))
        mu_hat = (one.T @ Psi_inv_y) / (one.T @ Psi_inv_one)
        mu_hat = mu_hat.item()

        # Berechne sigma_hat_sq (MLE der Prozessvarianz)
        y_minus_mu_one = y_train - one * mu_hat
        sigma_hat_sq = (y_minus_mu_one.T @ Psi_inv_y) / n
        sigma_hat_sq = sigma_hat_sq.item()

        if sigma_hat_sq < 1e-10:
            return 1e15

        # Berechne negative konzentrierte Log-Likelihood
        log_det_Psi = 2 * np.sum(np.log(np.diag(U.T)))
        nll = 0.5 * (n * np.log(sigma_hat_sq) + log_det_Psi)
```

```
        return nll

    def fit(self, X, y):
        """Fit das Kriging-Modell an die Trainingsdaten.

        // ...existing docstring...
        """
        self.X_train_ = X
        self.y_train_ = y.reshape(-1, 1)

        # Optimierung der Hyperparameter
        initial_theta = np.array([self.initial_theta])
        result = minimize(self._neg_log_likelihood,
                        initial_theta,
                        args=(self.X_train_, self.y_train_),
                        method='L-BFGS-B',
                        bounds=self.bounds)

        self.opt_theta_ = result.x

        # Berechne optimale Parameter für Vorhersagen
        Psi_opt = self._build_Psi(self.X_train_, self.opt_theta_)
        self.U_ = cholesky(Psi_opt).T
        n_train = self.X_train_.shape[0]
        one = np.ones(n_train).reshape(-1, 1)

        self.mu_hat_ = (one.T @ solve(self.U_, solve(self.U_.T, self.y_train_))) / \
                       (one.T @ solve(self.U_, solve(self.U_.T, one)))
        self.mu_hat_ = self.mu_hat_.item()

        return self

    def predict(self, X):
        """Vorhersage für neue Datenpunkte.

        Args:
            X (array-like): Eingabedaten für Vorhersage der Form (n_samples, n_features)

        Returns:
            array: Vorhergesagte Werte
        """
        n_train = self.X_train_.shape[0]
        one = np.ones(n_train).reshape(-1, 1)
        # Fix: Use self._build_psi instead of build_psi
```

```
        psi = self._build_psi(self.X_train_, X, self.opt_theta_)

        return self.mu_hat_ * np.ones(X.shape[0]).reshape(-1, 1) + \
            psi.T @ solve(self.U_, solve(self.U_.T,
            self.y_train_ - one * self.mu_hat_))
```

```
def plot_kriging_results(X_train, y_train, X_test, y_test, y_pred, theta, figsize=(10
    """Visualisiert die Kriging-Vorhersage im Vergleich zur wahren Funktion.

    Args:
        X_train (array-like): Trainings-Eingabedaten
        y_train (array-like): Trainings-Zielwerte
        X_test (array-like): Test-Eingabedaten
        y_test (array-like): Wahre Werte zum Vergleich
        y_pred (array-like): Vorhersagewerte des Modells
        theta (float): Optimierter Theta-Parameter
        figsize (tuple, optional): Größe der Abbildung. Default ist (10, 6).
    """
    plt.figure(figsize=figsize)

    # Sortiere Test-Daten nach X-Werten für eine glatte Linie
    sort_idx_test = np.argsort(X_test.ravel())
    X_test_sorted = X_test[sort_idx_test]
    y_test_sorted = y_test[sort_idx_test]
    y_pred_sorted = y_pred[sort_idx_test]

    # Wahre Funktion
    plt.plot(X_test_sorted, y_test_sorted, color="grey", linestyle='--',
            label="Wahre Sinusfunktion")

    # Trainingspunkte
    n_train = len(X_train)
    plt.plot(X_train, y_train, "bo", markersize=8,
            label=f"Messpunkte ({n_train} Punkte)")

    # Vorhersage
    plt.plot(X_test_sorted, y_pred_sorted, color="orange",
            label=f"Kriging-Vorhersage (Theta={theta:.2f})")

    # Plot-Eigenschaften
    plt.title(f"Kriging-Vorhersage mit {n_train} Trainings-Punkten\n" +
             f"Optimierter Aktivitätsparameter Theta={theta:.2f}")
    plt.xlabel("x")
    plt.ylabel("y")
```

```
    plt.legend(loc='upper right')
    plt.grid(True)
    plt.show()
```

66.1. Ein erste Beispielverwendung:

```
# Generiere Trainingsdaten
n_train = 4
X_train = np.linspace(0, 2 * np.pi, n_train, endpoint=False).reshape(-1, 1)
y_train = np.sin(X_train)

# Erstelle und trainiere das Modell
model = KrigingRegressor(initial_theta=1.0)
model.fit(X_train, y_train)

# Generiere Testdaten
X_test = np.linspace(0, 2 * np.pi, 100, endpoint=True).reshape(-1, 1)
y_test = np.sin(X_test)

# Mache Vorhersagen
y_pred = model.predict(X_test)

# Visualisiere die Ergebnisse

plot_kriging_results(
    X_train=X_train,
    y_train=y_train,
    X_test=X_test,
    y_test=y_test,
    y_pred=y_pred,
    theta=model.opt_theta_[0]
)
```

66.2. Ein zweites Beispiel mit train_test_split:

```
from sklearn.model_selection import train_test_split
import numpy as np

# Generate sample data
n_samples = 100
X = np.linspace(0, 2 * np.pi, n_samples).reshape(-1, 1)
y = np.sin(X)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=96, random_state=4

# Fit model
model = KrigingRegressor(initial_theta=1)
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)
```

```
# Visualize results
plot_kriging_results(
    X_train=X_train,
    y_train=y_train,
    X_test=X_test,
    y_test=y_test,
    y_pred=y_pred,
    theta=model.opt_theta_[0]
)
```

67. Lernmodul: Kriging Projekt

Um die gestellte Aufgabe zu lösen, erstellen wir eine Python-Klasse `KrigingRegressor` für das Kriging-Modell, eine Black-Box-Funktion `f(x)`, eine Funktion zur Erstellung des initialen Stichprobenplans und implementieren dann den sequenziellen Optimierungsablauf.

1. Die `KrigingRegressor`-Klasse Diese Klasse kapselt die Logik für das Kriging-Modell, einschließlich der Berechnung der Korrelationsmatrizen, der Maximum-Likelihood-Schätzung für den globalen Mittelwert $\hat{\mu}$ und die Prozessvarianz $\hat{\sigma}^2$, der konzentrierten Log-Likelihood-Funktion zur Hyperparameter-Optimierung sowie der Vorhersagefunktion. Die Optimierung des Aktivitätsparameters $\vec{\theta}$ erfolgt über die Maximierung der konzentrierten Log-Likelihood-Funktion, wobei intern die negative Log-Likelihood minimiert wird. Der Glattheitsparameter $\vec{p}$ wird implizit auf $p_j = 2$ gesetzt, da die quadrierte euklidische Distanz verwendet wird. Es wird ein kleiner Nugget-Term (`eps`) zur Diagonalen der Korrelationsmatrix hinzugefügt, um die numerische Stabilität zu gewährleisten.

2. Die Black-Box-Funktion `f(x)` Diese Funktion simuliert ein teures oder undurchsichtiges System, dessen interne Funktionsweise dem Optimierungsalgorithmus nicht bekannt ist. Im weiteren Schritt wird ein Server zur Beantwortung der Auswertungen bereitgestellt, der die Black-Box-Funktion ausführt. In diesem Beispiel verwenden wir eine analytische Funktion, die eine gewisse Komplexität aufweist, um die Black-Box zu simulieren.

3. Initialer Stichprobenplan `X` Für einen "optimalen" Versuchsplan wird Latin Hypercube Sampling (LHS) verwendet, da dies eine raumfüllende Eigenschaft aufweist. Dies stellt sicher, dass der Eingaberaum effizient erkundet wird. Die Eingabedaten werden intern auf den Bereich $$ skaliert, um die Konsistenz der θ-Werte über verschiedene Probleme hinweg zu gewährleisten.

4. Sequenzieller Optimierungsablauf Der Prozess beginnt mit einem initialen Stichprobenplan. In jeder Iteration wird das Kriging-Modell mit den verfügbaren Daten gefittet. Anschließend wird eine Optimierung auf dem Surrogatmodell (dem gefitteten Kriging-Modell) durchgeführt, um den nächsten vielversprechenden Punkt im Designraum zu finden. Dieser Punkt wird der Black-Box-Funktion übergeben, und die erhaltene Beobachtung wird zum Trainingsdatensatz hinzugefügt. Dieser iterative Prozess wird bis zu einer maximalen Anzahl von Funktionsauswertungen fortgesetzt.

Hier ist der entsprechende Python-Code:

```python
import numpy as np
from scipy.spatial.distance import pdist, squareform, cdist
from numpy.linalg import cholesky, solve, LinAlgError
from numpy import spacing, sqrt, exp, multiply, eye, ones
from scipy.optimize import minimize
from scipy.stats import qmc
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
from numpy import (exp, multiply, eye, linspace, spacing, sqrt)
from numpy.linalg import cholesky, solve
from scipy.spatial.distance import squareform, pdist, cdist
from scipy.optimize import minimize # Für die Optimierung

# 1. Definition der KrigingRegressor-Klasse

class KrigingRegressor:
    """Ein Kriging-Regressionsmodell mit Hyperparameter-Optimierung.

    Attributes:
        initial_theta (float): Startwert für den Aktivitätsparameter Theta.
        bounds (list): Liste von Tupeln für die Grenzen der Hyperparameter-Optimierung
        opt_theta_ (float): Optimierter Theta-Wert nach dem Fitting.
        X_train_ (array): Trainings-Eingabedaten.
        y_train_ (array): Trainings-Zielwerte.
        U_ (array): Cholesky-Zerlegung der Korrelationsmatrix.
        mu_hat_ (float): Geschätzter Mittelwert.
    """


    def __init__(self, initial_theta=1.0, bounds=[(0.001, 100.0)]):
        self.initial_theta = initial_theta
        self.bounds = bounds

    def _build_Psi(self, X, log_theta, eps=sqrt(spacing(1))):
        """Berechnet die Korrelationsmatrix Psi.

        Args:
            X (np.ndarray): Eingabedaten-Matrix
            log_theta (float or np.ndarray): Log10-transformierte Theta-Werte
            eps (float, optional): Nugget-Term für numerische Stabilität

        Returns:
            np.ndarray: Korrelationsmatrix Psi
```

```
        """
        # Konvertiere log_theta zurück zu theta (immer positiv durch exp)
        theta = 10.0**log_theta

        if not isinstance(theta, np.ndarray) or theta.ndim == 0:
            theta = np.array([theta])

        D = squareform(pdist(X, metric='sqeuclidean', w=theta))
        Psi = exp(-D)
        Psi += multiply(eye(X.shape[0]), eps)
        return Psi

    def _build_psi(self, X_train, x_predict, log_theta):
        """Berechnet den Korrelationsvektor psi.

        Args:
            X_train (np.ndarray): Trainings-Eingabedaten
            x_predict (np.ndarray): Vorhersage-Eingabepunkte
            log_theta (float or np.ndarray): Log10-transformierte Theta-Werte

        Returns:
            np.ndarray: Korrelationsvektor psi
        """
        # Konvertiere log_theta zurück zu theta
        theta = 10.0**log_theta

        if not isinstance(theta, np.ndarray) or theta.ndim == 0:
            theta = np.array([theta])

        D = cdist(x_predict, X_train, metric='sqeuclidean', w=theta)
        psi = exp(-D)
        return psi.T

    def _neg_log_likelihood(self, params, X_train, y_train):
        """Berechnet die negative konzentrierte Log-Likelihood."""
        theta = params
        n = X_train.shape[0]

        try:
            Psi = self._build_Psi(X_train, theta)
            U = cholesky(Psi).T
        except np.linalg.LinAlgError:
            return 1e15
```

```
        one = np.ones(n).reshape(-1, 1)

        # Berechne mu_hat (MLE des Mittelwerts)
        Psi_inv_y = solve(U, solve(U.T, y_train))
        Psi_inv_one = solve(U, solve(U.T, one))
        mu_hat = (one.T @ Psi_inv_y) / (one.T @ Psi_inv_one)
        mu_hat = mu_hat.item()

        # Berechne sigma_hat_sq (MLE der Prozessvarianz)
        y_minus_mu_one = y_train - one * mu_hat
        sigma_hat_sq = (y_minus_mu_one.T @ Psi_inv_y) / n
        sigma_hat_sq = sigma_hat_sq.item()

        if sigma_hat_sq < 1e-10:
            return 1e15

        # Berechne negative konzentrierte Log-Likelihood
        log_det_Psi = 2 * np.sum(np.log(np.diag(U.T)))
        nll = 0.5 * (n * np.log(sigma_hat_sq) + log_det_Psi)

        return nll

    def fit(self, X, y):
        """Fit das Kriging-Modell an die Trainingsdaten.
        Args:
            X (array-like): Eingabedaten der Form (n_samples, n_features)
            y (array-like): Zielwerte der Form (n_samples,)

        Returns:
            self: Das gefittete Kriging-Modell

        Notes:
            Diese Methode optimiert die Hyperparameter Theta, indem sie die negative l

        """
        self.X_train_ = X
        self.y_train_ = y.reshape(-1, 1)

        # Optimierung der Hyperparameter
        initial_theta = np.array([self.initial_theta])
        result = minimize(self._neg_log_likelihood,
                          initial_theta,
                          args=(self.X_train_, self.y_train_),
                          method='L-BFGS-B',
```

```
                        bounds=self.bounds)

        self.opt_theta_ = result.x

        # Berechne optimale Parameter für Vorhersagen
        Psi_opt = self._build_Psi(self.X_train_, self.opt_theta_)
        self.U_ = cholesky(Psi_opt).T
        n_train = self.X_train_.shape[0]
        one = np.ones(n_train).reshape(-1, 1)

        self.mu_hat_ = (one.T @ solve(self.U_, solve(self.U_.T, self.y_train_))) / \
                       (one.T @ solve(self.U_, solve(self.U_.T, one)))
        self.mu_hat_ = self.mu_hat_.item()

        return self

    def predict(self, X):
        """Vorhersage für neue Datenpunkte.

        Args:
            X (array-like): Eingabedaten für Vorhersage der Form (n_samples, n_features)

        Returns:
            array: Vorhergesagte Werte
        """
        n_train = self.X_train_.shape[0]
        one = np.ones(n_train).reshape(-1, 1)
        # Fix: Use self._build_psi instead of build_psi
        psi = self._build_psi(self.X_train_, X, self.opt_theta_)

        return self.mu_hat_ * np.ones(X.shape[0]).reshape(-1, 1) + \
            psi.T @ solve(self.U_, solve(self.U_.T,
            self.y_train_ - one * self.mu_hat_))
```

68. 2. Die Black-Box-Funktion

```python
def f_black_box(x):
    """Analytische Black-Box-Funktion: f(x)"""
    return -x**2*np.cos(x) + 1 + x**2*np.sin(x)
```

```python
# 3. Funktion zur Erstellung des initialen Stichprobenplans (Latin Hypercube Sampling)
def create_initial_design(n_points, dimensionality, x_range):
    """Erstellt einen raumfüllenden initialen Versuchsplan mittels Latin Hypercube Sampling.

    Args:
        n_points (int): Anzahl der Designpunkte
        dimensionality (int): Dimension des Eingaberaums
        x_range (tuple): Tuple mit (unterer_grenze, oberer_grenze) für den Wertebereich

    Returns:
        np.ndarray: Matrix der Designpunkte der Form (n_points, dimensionality)
    """
    # Verwende einen Integer als Seed statt SeedSequence
    sampler = qmc.LatinHypercube(d=dimensionality, seed=1234)

    # Generiere Samples im Einheitswürfel [0,1]^d
    lhs_samples_unit_cube = sampler.random(n=n_points)

    # Extrahiere die Grenzen
    x_lower, x_upper = x_range

    # Skaliere die Samples vom Einheitswürfel auf den gewünschten Bereich
    X_initial = x_lower + (x_upper - x_lower) * lhs_samples_unit_cube

    # Stelle sicher, dass das Output ein 2D Array ist
    return X_initial.reshape(-1, dimensionality)
```

69. Hauptskript für die sequentielle Optimierung

```
np.random.seed(42) # Für Reproduzierbarkeit der Zufallszahlen

# Definition des Suchraums für die Black-Box-Funktion
x_lower_bound = -5.0
x_upper_bound = 5.0
search_range = (x_lower_bound, x_upper_bound)
dimensionality = 1 # f(x) = x^2 + 1 ist 1-dimensional

# Parameter für den initialen Stichprobenplan
n_initial_points = 7

# Maximale Gesamtzahl an Funktionsauswertungen
N_max_evaluations = 20

# --- Schritt 1: Initialen Stichprobenplan erstellen und Black-Box-Funktion auswerten ---
X_train_current = create_initial_design(n_initial_points, dimensionality, search_range)
y_train_current = f_black_box(X_train_current)

print(f"Initialer X_train (n={n_initial_points}):\n{np.round(X_train_current.flatten(), 3)}")
print(f"Zugehöriger y_train:\n{np.round(y_train_current.flatten(), 3)}")
print("-" * 70)
```

```
Initialer X_train (n=7):
[-3.538 -4.115 -2.033  4.626  3.116  1.974  0.369]
Zugehöriger y_train:
[ 17.382  24.522  -0.856 -18.479  10.956   6.114   0.922]
----------------------------------------------------------------------
```

```
# --- Schritt 2: KrigingRegressor-Modell initialisieren ---
```

```
# Bounds für log10(theta), z.B. 10^-3 bis 10^2
theta_bounds_log10 = [(-3.0, 2.0)]
```

```
M_1 = KrigingRegressor(initial_theta=1.0, bounds=theta_bounds_log10)

# --- Schritt 3: Sequenzieller Optimierungs-Loop ---
# Anzahl der Punkte, die nach dem initialen Plan hinzugefügt werden
num_infill_steps = N_max_evaluations - n_initial_points

for i in range(num_infill_steps + 1): # +1, um auch nach dem letzten Infill-Punkt zu i
    current_evals = len(X_train_current)
    print(f"\n--- Iteration {i+1} (Gesamtauswertungen: {current_evals}) ---")

    # Modell mit den aktuellen Daten fitten
    print("Kriging-Modell wird gefittet...")
    M_1.fit(X_train_current, y_train_current)
    print(f"Optimiertes Theta: {np.round(M_1.opt_theta_.item(), 4)}")
    print(f"Geschätzter globaler Mittelwert (mu_hat): {np.round(M_1.mu_hat_, 4)}")

    # Überprüfen, ob die maximale Anzahl an Auswertungen erreicht wurde
    if current_evals >= N_max_evaluations:
        print("Maximale Anzahl an Auswertungen erreicht. Optimierung wird beendet.")
        break
```

```
--- Iteration 1 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 2 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 3 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 4 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 5 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
```

```
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 6 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 7 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 8 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 9 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 10 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 11 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 12 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 13 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178

--- Iteration 14 (Gesamtauswertungen: 7) ---
Kriging-Modell wird gefittet...
```

```
Optimiertes Theta: -0.1013
Geschätzter globaler Mittelwert (mu_hat): 3.5178
```

70. — Schritt 4: Surrogat-basierte Optimierung zur Suche des nächsten Infill-Punktes —

Definiere die Zielfunktion für die innere Optimierung (Minimierung des Surrogats)

```
def surrogate_objective(x_val):
    # Stelle sicher, dass x_val im richtigen Format (2D-Array) für predict ist
    x_val_reshaped = np.atleast_2d(x_val)
    return M_1.predict(x_val_reshaped).item() # Rückgabe als Skalar
```

71. Suchgrenzen für die Optimierung auf dem Surrogatmodell

```
surrogate_search_bounds = [(x_lower_bound, x_upper_bound)] * dimensionality

print("Optimierung auf dem Surrogatmodell, um den nächsten Infill-Punkt zu finden...")
# Anfangsschätzung für die Surrogat-Optimierung (zufälliger Punkt im Suchraum)
x0_surrogate_opt = np.random.uniform(low=x_lower_bound, high=x_upper_bound, size=dimensionality)

# Verwende L-BFGS-B für die Surrogat-Optimierung
surrogate_opt_result = minimize(surrogate_objective,
                            x0=x0_surrogate_opt,
                            method='L-BFGS-B',
                            bounds=surrogate_search_bounds)

x_new_infill = np.atleast_2d(surrogate_opt_result.x) # Sicherstellen, dass 2D-Array
y_new_infill = f_black_box(x_new_infill) # Auswertung des neuen Punktes durch Black-Box

print(f"Neuer Infill-Punkt x_1: {np.round(x_new_infill.flatten(), 3)}, "
        f"Zugehöriger y_1 (Black-Box): {np.round(y_new_infill.item(), 3)}")

# Neuen Punkt zum Trainingsdatensatz hinzufügen
X_train_current = np.vstack((X_train_current, x_new_infill))
y_train_current = np.vstack((y_train_current, y_new_infill))

print(f"Aktueller X_train (total {len(X_train_current)} Punkte):\n{np.round(X_train_current.flatten(
print(f"Aktueller y_train:\n{np.round(y_train_current.flatten(), 3)}")
print("-" * 70)

print("\n--- Optimierungsprozess abgeschlossen ---")
```

```
Optimierung auf dem Surrogatmodell, um den nächsten Infill-Punkt zu finden...
Neuer Infill-Punkt x_1: [-1.93], Zugehöriger y_1 (Black-Box): -1.178
Aktueller X_train (total 8 Punkte):
[-3.538 -4.115 -2.033  4.626  3.116  1.974  0.369 -1.93 ]
Aktueller y_train:
```

```
[ 17.382  24.522  -0.856 -18.479  10.956   6.114   0.922  -1.178]
----------------------------------------------------------------------

--- Optimierungsprozess abgeschlossen ---
```

72. Fitte das finale Modell mit allen verfügbaren N_max_evaluations Punkten

```
M_1.fit(X_train_current, y_train_current)

# Visualisierung der Ergebnisse
x_plot = np.linspace(x_lower_bound, x_upper_bound, 200).reshape(-1, 1)
y_predict_final = M_1.predict(x_plot)
y_true = f_black_box(x_plot)

plt.figure(figsize=(10, 6))
plt.plot(x_plot, y_true, 'k--', linewidth=2, label='Wahre Funktion: $f(x)$')
plt.plot(X_train_current, y_train_current, 'bo', markersize=8, label=f'Ausgewertete Punkte (N={len(X
plt.plot(x_plot, y_predict_final, 'r-', linewidth=2, label='Kriging-Vorhersage')
plt.title('Kriging-Modell mit sequenzieller Optimierung der Hyperparameter')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()

print(f"\nFinal optimiertes Theta: {np.round(M_1.opt_theta_.item(), 4)}")
print(f"Final geschätzter globaler Mittelwert: {np.round(M_1.mu_hat_, 4)}")
# Der letzte optimierte Punkt auf dem Surrogat (x_new_infill) ist eine gute Schätzung des Optimums d
print(f"Bester gefundener x-Wert (aus Surrogat-Optimierung): {np.round(surrogate_opt_result.x.item()
print(f"Zugehöriger y-Wert der Black-Box an diesem x: {np.round(f_black_box(surrogate_opt_result.x).
```

72. Fitte das finale Modell mit allen verfügbaren N_max_evaluations Punkten

```
Final optimiertes Theta: -0.577
Final geschätzter globaler Mittelwert: 2.8055
Bester gefundener x-Wert (aus Surrogat-Optimierung): -1.93
Zugehöriger y-Wert der Black-Box an diesem x: -1.178
```

73. Lernmodul: Kriging Projekt mit Expected Improvement

Dies ist ein erweitertes Lernmodul, das auf dem "Lernmodul: Kriging Projekt" aufbaut und "Expected Improvement" (EI) als Infill-Kriterium verwendet.

73.1. Einleitung

Das vorhergehende "Lernmodul: Kriging Projekt" hat die Grundlagen der sequenziellen Optimierung mittels Kriging-Surrogatmodellen etabliert. Dabei wurde die nächste Evaluierungsstelle im Designraum einfach durch die Minimierung der Surrogatmodellvorhersage gewählt – ein Ansatz, der hauptsächlich auf **Exploitation** abzielt, also der Ausnutzung vielversprechender Regionen. In der Praxis ist es jedoch entscheidend, ein Gleichgewicht zwischen Exploitation und **Exploration** (Erkundung unsicherer Regionen) zu finden, insbesondere bei teuren Black-Box-Funktionen.

Dieses Lernmodul erweitert den sequenziellen Optimierungsablauf, indem es **Expected Improvement (EI)** als Infill-Kriterium integriert. EI ist eine der einflussreichsten und am weitesten verbreiteten Methoden in der bayesianischen Optimierung, da sie Exploitation und Exploration auf elegante Weise in einem einzigen Kriterium vereint.

73.2. 1. Die KrigingRegressor-Klasse (Erweiterung)

Die `KrigingRegressor`-Klasse, die bereits für die Berechnung der Korrelationsmatrizen, die Maximum-Likelihood-Schätzung der Hyperparameter ($\hat{\mu}$, $\hat{\sigma}^2$, $\vec{\theta}$) und die Vorhersagefunktion (`predict`) verwendet wurde, wird um eine Methode zur Berechnung des Expected Improvement erweitert.

EI nutzt sowohl die **Vorhersage des Modells** ($\hat{y}(\vec{x})$) als auch die **Unsicherheit der Vorhersage** (Varianz $\hat{s}^2(\vec{x})$), die Kriging-Modelle bereitstellen können. Die Fähigkeit von Kriging, ein Maß für die Unsicherheit zu liefern, ist ein entscheidender Vorteil gegenüber einfacheren Surrogatmodellen wie Polynomen.

73.3. 2. Die Black-Box-Funktion `f(x)` (Wiederholung)

Wie im vorherigen Modul simuliert die Black-Box-Funktion `f(x)` ein teures oder undurchsichtiges System. Für dieses Beispiel verwenden wir weiterhin eine analytische Funktion, um die Prinzipien zu demonstrieren.

73.4. 3. Initialer Stichprobenplan `X` (Wiederholung)

Der Prozess beginnt mit einem initialen Stichprobenplan, der typischerweise mittels **Latin Hypercube Sampling (LHS)** erstellt wird. LHS ist eine **raumfüllende** Technik, die sicherstellt, dass der Eingaberaum effizient und gleichmäßig erkundet wird, was eine gute Ausgangsbasis für das Kriging-Modell bietet.

73.5. 4. Sequenzieller Optimierungsablauf mit Expected Improvement

Der iterative Prozess der sequenziellen Optimierung wird wie folgt angepasst:

1. **Initialisierung**: Ein initialer Stichprobenplan `X` wird erstellt und die Black-Box-Funktion `f` an diesen Punkten evaluiert, um die Beobachtungen `y` zu erhalten.
2. **Kriging-Modell anpassen**: Das Kriging-Modell wird mit den gesammelten Daten (`X`, `y`) angepasst. Hierbei werden die Hyperparameter (insbesondere $\vec{\theta}$) mittels Maximum-Likelihood-Schätzung optimiert.
3. **Expected Improvement (EI) berechnen**: Für eine Vielzahl von Kandidatenpunkten im Designraum wird das Expected Improvement berechnet. EI quantifiziert den erwarteten Gewinn, wenn man die Black-Box an einem bestimmten Punkt evaluiert, im Vergleich zum besten bisher beobachteten Wert (`y_min`). Die Formel für EI lautet:

$$E[I(\vec{x})] = (\hat{y}_{min} - \hat{y}(\vec{x})) \cdot \Phi\left(\frac{\hat{y}_{min} - \hat{y}(\vec{x})}{\hat{s}(\vec{x})}\right) + \hat{s}(\vec{x}) \cdot \phi\left(\frac{\hat{y}_{min} - \hat{y}(\vec{x})}{\hat{s}(\vec{x})}\right)$$

   wobei:

   - $\hat{y}_{min}$ der beste bisher beobachtete Funktionswert ist.
   - $\hat{y}(\vec{x})$ die Kriging-Vorhersage am Punkt $\vec{x}$ ist.
   - $\hat{s}(\vec{x})$ die geschätzte Standardabweichung (Wurzel aus der Varianz) der Vorhersage am Punkt $\vec{x}$ ist.
   - Φ die kumulative Verteilungsfunktion (CDF) und ϕ die Wahrscheinlichkeitsdichtefunktion (PDF) der Standardnormalverteilung sind.
   - Ein Wert von 0 wird zurückgegeben, wenn $\hat{s}(\vec{x}) = 0$ (d.h. an einem bereits beprobten Punkt ist die Unsicherheit null).

4. **Nächsten Infill-Punkt auswählen**: Anstatt den Punkt mit der besten Vorhersage zu wählen, wird der Punkt ausgewählt, der das **Expected Improvement maximiert**. Da die meisten Optimierungsalgorithmen auf Minimierung ausgelegt sind, wird üblicherweise die **negative Expected Improvement**-Funktion minimiert.
5. **Evaluierung und Aktualisierung**: Der ausgewählte Punkt wird der teuren Black-Box-Funktion übergeben, die Beobachtung wird zum Trainingsdatensatz hinzugefügt, und der Prozess wird iteriert.

Vorteile von EI: * **Automatisches Gleichgewicht**: EI balanciert Exploration und Exploitation automatisch, ohne manuelle Gewichtungsparameter. * **Theoretische Fundierung**: Es hat eine starke theoretische Rechtfertigung aus der Entscheidungstheorie. * **Effiziente Optimierung**: Die differenzierbare Natur von EI macht es für gradientenbasierte Optimierungsalgorithmen geeignet. * **Globale Konvergenz**: Eine Maximierung von EI führt letztendlich zum globalen Optimum, da EI an beprobten Punkten auf null fällt, was die Suche in ungesampelte, unsichere oder vielversprechende Bereiche lenkt.

73.6. Der entsprechende Python-Code:

Für dieses Modul simulieren wir eine `KrigingRegressor`-Klasse, die die notwendigen `fit`, `predict` und `expected_improvement` Methoden enthält. Der Fokus des Codes liegt auf dem angepassten sequenziellen Optimierungsablauf.

```
import numpy as np
from scipy.optimize import minimize
from scipy.stats import norm
import matplotlib.pyplot as plt
from numpy.linalg import cholesky, solve
from scipy.spatial.distance import pdist, squareform, cdist

#1. Die KrigingRegressor-Klasse (erweitert um Expected Improvement)
# Für dieses Beispiel vereinfachen wir die Implementierung,
# um den Fokus auf Expected Improvement zu legen.
# Eine vollständige Implementierung wäre umfangreicher.
class KrigingRegressor:
    def __init__(self, theta=None, p=2.0, eps=np.sqrt(np.spacing(1))):
        self.theta = theta if theta is not None else np.array([1.0]) # Aktivitätshyperparameter
        self.p = p # Glattheitsparameter, hier fest auf 2.0 (sqeuclidean)
        self.eps = eps # Nugget-Effekt für numerische Stabilität
        self.X_train = None
        self.y_train = None
        self.mu_hat = None
```

```
        self.sigma_hat_sq = None
        self.Psi = None
        self.U = None # Cholesky-Faktor

    def _build_psi_matrix(self, X_data, w=None):
        """Berechnet die Korrelationsmatrix Psi mit gewichteter quadrierter euklidisch

        Args:
            X_data (np.ndarray): Eingabedaten-Matrix
            w (np.ndarray, optional): Gewichtungsparameter (theta)

        Returns:
            np.ndarray: Korrelationsmatrix Psi
        """
        if w is None:
            w = self.theta

        D = squareform(pdist(X_data, metric='sqeuclidean', w=w))
        Psi = np.exp(-D)

        # Nugget-Effekt für numerische Stabilität - verwende die Anzahl der Datenpunkt
        n_points = X_data.shape[0]  # Anzahl der Zeilen (Datenpunkte)
        Psi += np.multiply(np.eye(n_points), self.eps)

        return Psi

    def _build_psi_vector(self, X_predict, X_train, w=None):
        # Berechnet den Korrelationsvektor psi zwischen Vorhersage- und Trainingspunkt
        if w is None:
            w = self.theta
        D = cdist(X_predict, X_train, metric='sqeuclidean', w=w)
        psi = np.exp(-D)
        return psi.T # Transponieren, um n x m oder n x 1 zu erhalten

    def fit(self, X_train, y_train):
        """Trainiert das Kriging-Modell auf den gegebenen Daten.

        Args:
            X_train (np.ndarray): Trainings-Eingabedaten
            y_train (np.ndarray): Trainings-Zielwerte
        """
        self.X_train = X_train
        self.y_train = y_train
        n = self.X_train.shape[0]  # Anzahl der Datenpunkte, nicht die gesamte Shape
```

```
        # Numerisch stabile Berechnung der Psi-Matrix
        self.Psi = self._build_psi_matrix(self.X_train)

        # Cholesky-Zerlegung für effiziente Inversion
        # U ist der obere Dreiecksfaktor (Transponierte des unteren)
        try:
            self.U = cholesky(self.Psi).T
        except np.linalg.LinAlgError:
            print("Cholesky-Zerlegung fehlgeschlagen, Matrix ist nicht positiv definit.")
            # Fallback oder Fehlerbehandlung, z.B. größeren Nugget-Term verwenden
            self.Psi = self._build_psi_matrix(self.X_train, self.theta + 1e-6)
            self.U = cholesky(self.Psi).T

        # Berechnung von mu_hat (geschätzter globaler Mittelwert)
        one_vec = np.ones((n, 1))
        # solve(U, solve(U.T, vec)) ist äquivalent zu inv(Psi) @ vec unter Verwendung von Cholesky
        self.mu_hat = (one_vec.T @ solve(self.U, solve(self.U.T, self.y_train))) / \
                    (one_vec.T @ solve(self.U, solve(self.U.T, one_vec)))
        self.mu_hat = self.mu_hat.item() # Extrahiere Skalarwert

        # Berechnung von sigma_hat_sq (geschätzte Prozessvarianz)
        self.sigma_hat_sq = ((self.y_train - one_vec * self.mu_hat).T @ \
                            solve(self.U, solve(self.U.T, self.y_train - one_vec * self.mu_hat))) /
        self.sigma_hat_sq = self.sigma_hat_sq.item()

    def predict(self, X_predict):
        """Vorhersage für neue Datenpunkte.

        Args:
            X_predict (np.ndarray): Eingabedaten für Vorhersage

        Returns:
            np.ndarray: Vorhergesagte Werte
        """
        if self.X_train is None or self.y_train is None:
            raise ValueError("Modell muss zuerst mit fit() trainiert werden.")

        m = X_predict.shape[0]  # Anzahl der Vorhersagepunkte
        n = self.X_train.shape[0]  # Anzahl der Trainingspunkte

        one_vec_m = np.ones((m, 1))
        one_vec_n = np.ones((n, 1))

        psi_vec = self._build_psi_vector(X_predict, self.X_train)
```

```
        # BLUP-Formel: y_hat(x) = mu_hat + psi.T @ inv(Psi) @ (y_train - 1 * mu_hat)
        f_predict = self.mu_hat * one_vec_m + \
                    psi_vec.T @ solve(self.U, solve(self.U.T, self.y_train - one_vec_m
        return f_predict.flatten()

    def predict_variance(self, X_predict):
        if self.X_train is None or self.y_train is None:
            raise ValueError("Modell muss zuerst mit fit() trainiert werden.")

        psi_vec = self._build_psi_vector(X_predict, self.X_train)

        # Geschätzter Fehler (Varianz) s^2(x)
        # s^2(x) = sigma_hat_sq * (1 - psi.T @ inv(Psi) @ psi)
        s_sq = self.sigma_hat_sq * (1 - np.diag(psi_vec.T @ solve(self.U, solve(self.U
        # Sicherstellen, dass die Varianz nicht negativ ist (numerische Stabilität)
        s_sq[s_sq < 1e-10] = 1e-10
        return s_sq.flatten()

    def expected_improvement(self, x_cand, y_min_current):
        # x_cand sollte ein 2D-Array sein, auch für 1D-Probleme: np.array([[x]])
        mu_cand = self.predict(x_cand) # y_hat(x)
        s_cand = np.sqrt(self.predict_variance(x_cand)) # s(x)

        # Handhabung für s_cand == 0 (bereits beprobte Punkte)
        # Vermeidet Division durch Null und gibt EI = 0 zurück
        ei_values = np.zeros_like(mu_cand)

        # Indizes, wo s_cand > 0 ist
        positive_s_indices = s_cand > 1e-10

        if np.any(positive_s_indices):
            s_pos = s_cand[positive_s_indices]
            mu_pos = mu_cand[positive_s_indices]

            Z = (y_min_current - mu_pos) / s_pos

            # Formel für Expected Improvement
            # EI = (ymin - y_hat) * Phi(Z) + s * phi(Z)
            ei_values[positive_s_indices] = (y_min_current - mu_pos) * norm.cdf(Z) + s

        # Sicherstellen, dass EI nicht negativ wird (numerische Stabilität)
        ei_values[ei_values < 0] = 0
        return ei_values
```

```
    def _neg_expected_improvement(self, x_cand_flat, y_min_current):
        # Wrapper für die Optimierung, erwartet flaches x_cand
        x_cand = np.array(x_cand_flat).reshape(1, -1) # Formatiere zu 2D
        ei = self.expected_improvement(x_cand, y_min_current)
        # Minimierungsziel ist negatives EI
        return -ei # EI ist hier ein Skalar, daher
```

74. 2. Die Black-Box-Funktion f(x)

Beispiel: Sinusfunktion mit hinzugefügtem (optionalem) Rauschen

```
def f(x_val):
    # Standardisierung von x_val von zu [0, 2*pi] für die Sinusfunktion
    # Annahme, dass die Optimierung in^k durchgeführt wird und die Blackbox
    # intern die Skalierung handhabt.
    # Für dieses Beispiel behalten wir die direkte Nutzung der Sinusfunktion.
    return np.sin(x_val) # + np.random.normal(0, 0.05) # Optional: Rauschen hinzufügen
```

#3. Initialer Stichprobenplan X (Latin Hypercube Sampling)

Implementierung eines einfachen LHS für 1D für das Beispiel

```
def latin_hypercube_sampling(n_points, n_dims, lower_bound=0, upper_bound=1):
    """Erstellt Latin Hypercube Samples im gegebenen Bereich.

    Args:
        n_points (int): Anzahl der zu generierenden Punkte
        n_dims (int): Anzahl der Dimensionen
        lower_bound (float): Untere Grenze des Suchraums
        upper_bound (float): Obere Grenze des Suchraums

    Returns:
        np.ndarray: Array der Form (n_points, n_dims) mit LHS-Samples
    """
    points = np.zeros((n_points, n_dims))

    for i in range(n_dims):
        # Erstelle n_points gleichmäßige Intervalle
        bins = np.linspace(lower_bound, upper_bound, n_points + 1)

        # Berechne die Intervallbreite
        interval_width = (upper_bound - lower_bound) / n_points

        # Generiere zufällige Offsets innerhalb jedes Intervalls
        random_offsets = np.random.rand(n_points) * interval_width
```

```
        # Setze Punkte in jedes Intervall
        points[:, i] = bins[:-1] + random_offsets

        # Permutiere die Punkte für jede Dimension (wichtig für LHS)
        np.random.shuffle(points[:, i])

    return points
```

74.1. Hauptskript für die sequentielle Optimierung

```
n_initial_points = 5 # Anzahl der initialen Stichprobenpunkte
n_infill_points = 10 # Anzahl der hinzuzufügenden Infill-Punkte
n_dimensions = 1     # Problem-Dimension (für sin(x) ist k=1)

# 1. Initialisierung: Initialer Stichprobenplan und Evaluierung
# x-Werte im Bereich für die Optimierung
X_initial = latin_hypercube_sampling(n_initial_points, n_dimensions)
# y-Werte im Bereich [0, 2*pi] für die Sinusfunktion
# Wir skalieren X_initial für die f-Funktion, wenn f in einem anderen Bereich definier
# Für Sin(x) verwenden wir direkt X_initial skaliert auf [0, 2*pi]
X_train_scaled = X_initial * (2 * np.pi) # Skalierung für sin(x)
y_train = np.array([f(x_val) for x_val in X_train_scaled]).reshape(-1, 1)
X_train = X_initial # Behalte X_train in für KrigingRegressor

# Initialisiere KrigingRegressor
kriging_model = KrigingRegressor()
kriging_model.fit(X_train, y_train)

# Speicher für die Visualisierung
all_X = X_train.copy()
all_y = y_train.copy()

# Range für die Vorhersage/Visualisierung
x_plot = np.linspace(0, 1, 100).reshape(-1, 1) # Im standardisierten Bereich
x_plot_scaled = x_plot * (2 * np.pi) # Skalieren für die wahre Funktion

plt.figure(figsize=(12, 8))
plt.plot(x_plot_scaled, f(x_plot_scaled), 'grey', linestyle='--', label='Wahre Sinusfu
plt.plot(all_X * (2 * np.pi), all_y, 'bo', markersize=8, label='Messungen (Initial)')
plt.title('Sequenzielle Optimierung mit Expected Improvement')
plt.xlabel('x')
```

```
plt.ylabel('y')
plt.grid(True)
plt.legend()
plt.show()


# Sequenzieller Optimierungsablauf
for i in range(n_infill_points):
    print(f"\n--- Iteration {i+1}: Suche Infill-Punkt mittels Expected Improvement ---")

    # Besten bisher beobachteten Wert finden (für Minimierungsproblem)
    y_min_current = np.min(all_y)
    print(f"Bester bisheriger Wert (y_min): {np.round(y_min_current, 4)}")

    # Suchgrenzen für die Optimierung auf dem Surrogatmodell (im Einheitshyperwürfel)
    # Für 1D ist es einfach
    bounds = [(0, 1)] * n_dimensions

    # Optimierung des Surrogatmodells zur Suche des nächsten Infill-Punktes
    # Wir minimieren die negative Expected Improvement
    # Starte die Suche von mehreren zufälligen Punkten, um lokale Minima zu vermeiden
    num_restarts = 5
    best_ei_val = -np.inf
    next_x_infill = None

    for _ in range(num_restarts):
        # Zufälliger Startpunkt innerhalb der Grenzen
        x0 = np.random.rand(n_dimensions)

        res = minimize(kriging_model._neg_expected_improvement, x0,
                       args=(y_min_current,),
                       bounds=bounds,
                       method='L-BFGS-B') # Oder andere geeignete Methode

        # EI ist der negative Wert des Ergebnisses der Minimierung
        current_ei_val = -res.fun

        if current_ei_val > best_ei_val:
            best_ei_val = current_ei_val
            next_x_infill = res.x

    if next_x_infill is None:
        # Fallback, falls Optimierung fehlschlägt, z.B. zufälligen Punkt wählen
        next_x_infill = np.random.rand(n_dimensions)
```

```
        print("Warnung: EI-Optimierung fehlgeschlagen, wähle zufälligen Punkt.")

    # Stelle sicher, dass next_x_infill 2D ist für die predict-Methoden
    next_x_infill_2d = next_x_infill.reshape(1, -1)

    print(f"Gewählter Infill-Punkt (standardisiert): {np.round(next_x_infill, 4)}")
    print(f"Geschätztes EI am Infill-Punkt: {np.round(best_ei_val, 4)}")

    # Evaluierung des Infill-Punktes auf der realen Black-Box-Funktion
    # Skaliere den Infill-Punkt für die f-Funktion
    next_x_infill_scaled = next_x_infill * (2 * np.pi)
    y_new = np.array([f(next_x_infill_scaled)]).reshape(-1, 1)
    print(f"Tatsächlicher Wert am Infill-Punkt: {np.round(y_new.item(), 4)}")

    # Daten zum Trainingssatz hinzufügen
    all_X = np.vstack((all_X, next_x_infill_2d))
    all_y = np.vstack((all_y, y_new))

    # Kriging-Modell neu anpassen mit den aktualisierten Daten
    kriging_model.fit(all_X, all_y)

    # Visualisierung des aktuellen Zustands
    plt.figure(figsize=(10, 8))

    # Plot auf der ersten Y-Achse (linke Seite)
    ax1 = plt.gca()
    ax1.plot(x_plot_scaled, f(x_plot_scaled), 'grey', linestyle='--', label='Wahre Si
    ax1.plot(x_plot_scaled, kriging_model.predict(x_plot), 'orange', label='Kriging V
    ax1.plot(all_X[:-1] * (2 * np.pi), all_y[:-1], 'bo', markersize=8, label='Messunge
    ax1.plot(all_X[-1] * (2 * np.pi), all_y[-1], 'ro', markersize=10, label='Neuer Ini
    ax1.set_xlabel('x')
    ax1.set_ylabel('y')
    ax1.grid(True)

    # Plot Expected Improvement auf der zweiten Y-Achse (rechte Seite)
    ax2 = ax1.twinx()
    ei_values_plot = kriging_model.expected_improvement(x_plot, y_min_current)
    ax2.plot(x_plot_scaled, ei_values_plot, 'g:', label='Expected Improvement (EI)')
    ax2.set_ylabel('Expected Improvement', color='g')
    ax2.tick_params(axis='y', labelcolor='g')

    plt.title(f'Iteration {i+1}: Kriging Vorhersage und EI (y_min={np.round(y_min_curr

    # Korrekte Behandlung der Legenden von beiden Achsen
```

```
    lines1, labels1 = ax1.get_legend_handles_labels()
    lines2, labels2 = ax2.get_legend_handles_labels()
    ax1.legend(lines1 + lines2, labels1 + labels2, loc='upper right')

    plt.show()

print("\n--- Optimierung abgeschlossen ---")
print(f"Finaler bester Wert gefunden: {np.round(np.min(all_y).item(), 4)}")
print(f"Gesamtzahl der Evaluierungen: {len(all_y)}")
```

```
--- Iteration 1: Suche Infill-Punkt mittels Expected Improvement ---
Bester bisheriger Wert (y_min): -0.6694
Gewählter Infill-Punkt (standardisiert): [0.7608]
Geschätztes EI am Infill-Punkt: 0.3817
Tatsächlicher Wert am Infill-Punkt: -0.9977
```

```
--- Iteration 2: Suche Infill-Punkt mittels Expected Improvement ---
Bester bisheriger Wert (y_min): -0.9977
Gewählter Infill-Punkt (standardisiert): [0.8524]
Geschätztes EI am Infill-Punkt: 0.0
Tatsächlicher Wert am Infill-Punkt: -0.8001
```

```
--- Iteration 3: Suche Infill-Punkt mittels Expected Improvement ---
Bester bisheriger Wert (y_min): -0.9977
Gewählter Infill-Punkt (standardisiert): [0.7178]
Geschätztes EI am Infill-Punkt: 0.0
Tatsächlicher Wert am Infill-Punkt: -0.9796
```

```
--- Iteration 4: Suche Infill-Punkt mittels Expected Improvement ---
Bester bisheriger Wert (y_min): -0.9977
Gewählter Infill-Punkt (standardisiert): [0.0181]
Geschätztes EI am Infill-Punkt: 0.0
Tatsächlicher Wert am Infill-Punkt: 0.1137
```

```
--- Iteration 5: Suche Infill-Punkt mittels Expected Improvement ---
Bester bisheriger Wert (y_min): -0.9977
Gewählter Infill-Punkt (standardisiert): [0.8074]
Geschätztes EI am Infill-Punkt: 0.0
Tatsächlicher Wert am Infill-Punkt: -0.9358
```

```
--- Iteration 6: Suche Infill-Punkt mittels Expected Improvement ---
Bester bisheriger Wert (y_min): -0.9977
Gewählter Infill-Punkt (standardisiert): [0.3061]
Geschätztes EI am Infill-Punkt: 0.0
Tatsächlicher Wert am Infill-Punkt: 0.9384
```

```
--- Iteration 7: Suche Infill-Punkt mittels Expected Improvement ---
Bester bisheriger Wert (y_min): -0.9977
Gewählter Infill-Punkt (standardisiert): [0.846]
Geschätztes EI am Infill-Punkt: 0.0
Tatsächlicher Wert am Infill-Punkt: -0.8237
```

```
--- Iteration 8: Suche Infill-Punkt mittels Expected Improvement ---
Bester bisheriger Wert (y_min): -0.9977
Gewählter Infill-Punkt (standardisiert): [0.7503]
Geschätztes EI am Infill-Punkt: 0.0025
Tatsächlicher Wert am Infill-Punkt: -1.0
```

```
--- Iteration 9: Suche Infill-Punkt mittels Expected Improvement ---
Bester bisheriger Wert (y_min): -1.0
Gewählter Infill-Punkt (standardisiert): [0.5428]
Geschätztes EI am Infill-Punkt: 0.0
Tatsächlicher Wert am Infill-Punkt: -0.2659
```

```
--- Iteration 10: Suche Infill-Punkt mittels Expected Improvement ---
Bester bisheriger Wert (y_min): -1.0
Gewählter Infill-Punkt (standardisiert): [0.7708]
Geschätztes EI am Infill-Punkt: 0.0
Tatsächlicher Wert am Infill-Punkt: -0.9914
```

```
--- Optimierung abgeschlossen ---
Finaler bester Wert gefunden: -1.0
Gesamtzahl der Evaluierungen: 15
```

74.2. Ergebnisse und Diskussion

Der angepasste Code demonstriert die Funktionsweise von Expected Improvement. In jeder Iteration wird nicht nur die Kriging-Vorhersage aktualisiert, sondern auch das Expected Improvement über den gesamten Designraum berechnet und visualisiert.

- **Visuelle Darstellung**: Sie werden sehen, dass die "Expected Improvement"-Kurve (grün gestrichelt) in Bereichen mit niedriger Vorhersage (gut für Exploitation) und/oder hoher Unsicherheit (gut für Exploration) hohe Werte aufweist. Der neue Infill-Punkt (roter Kreis) wird typischerweise an der Spitze eines solchen EI-Peaks platziert.
- **Gleichgewicht**: Im Gegensatz zur reinen Minimierung der Vorhersage, die dazu neigen könnte, in lokalen Minima stecken zu bleiben, fördert EI die Erkundung von Regionen, in denen das Modell unsicher ist, auch wenn die aktuelle Vorhersage

dort nicht optimal ist. Dies ist besonders vorteilhaft bei multimodalen Funktionen oder wenn die initialen Stichproben den Designraum nicht vollständig abdecken.

- **Konvergenz**: Durch die kontinuierliche Hinzufügung von Punkten mit hohem Expected Improvement wird das Modell schrittweise genauer, und die Suche wird effizienter auf das globale Optimum hingeführt. Die EI-Werte nehmen typischerweise ab, wenn das Modell sicherer wird und das Optimum gefunden wird.

Dieses Modul zeigt, wie die Integration von Expected Improvement die Effizienz und Robustheit der sequenziellen Optimierung mit Kriging-Modellen erheblich verbessert, indem es eine intelligente Strategie zur Auswahl des nächsten Funktionsauswertungspunkts bereitstellt.

A. Introduction to Jupyter Notebook

Jupyter Notebook is a widely used tool in the Data Science community. It is easy to use and the produced code can be run per cell. This has a huge advantage, because with other tools e.g. (pycharm, vscode, etc.) the whole script is executed. This can be a time consuming process, especially when working with huge data sets.

A.1. Different Notebook cells

There are different cells that the notebook is currently supporting:

- code cells
- markdown cells
- raw cells

As a default, every cells in jupyter is set to "code"

A.1.1. Code cells

The code cells are used to execute the code. They are following the logic of the choosen kernel. Therefore, it is important to keep in mind which programming language is currently used. Otherwise one might yield an error because of the wrong syntax.

The code cells are executed my be **Run** button (can be found in the header of the notebook).

A.1.2. Markdown cells

The markdown cells are a usefull tool to comment the written code. Especially with the help of headers can the code be brought in a more readable format. If you are not familiar with the markdown syntax, you can find a usefull cheat sheet here: Markdown Cheat Sheeet

A.1.3. Raw cells

The “Raw NBConvert” cell type can be used to render different code formats into HTML or LaTeX by Sphinx. This information is stored in the notebook metadata and converted appropriately.

A.1.3.1. Usage

To select a desired format from within Jupyter, select the cell containing your special code and choose options from the following dropdown menus:

1. Select “Raw NBConvert”
2. Switch the Cell Toolbar to “Raw Cell Format” (The cell toolbar can be found under View)
3. Chose the appropriate “Raw NBConvert Format” within the cell

Data Science is fun

A.2. Install Packages

Because python is a heavily used programming language, there are many different packags that can make your life easier. Sadly, there are only a few standard packages that are already included in your python enviroment. If you have the need to install a new package in your enviroment, you can simply do that by exectuing the following code snippet in a **code cell**

```
!pip install numpy
```

- The *!* is used to run the cell as a shell command
- *pip* is package manager for python packages.
- *numpy* is the the package you want to install

Hint: It is often usefull to restart the kernel after installing a package, otherwise loading the package could lead to an error.

A.3. Load Packages

After successfully installing the package it is necessary to import them before you can work with them. The import of the packages is done in the following way:

```
import numpy as np
```

The imported packages are often abbreviated. This is because you need to specify where the function is coming from.

The most common abbreviations for data science packages are:

Table A.1.: Abbreviations for data science packages

Abbreviation	Package	Import
np	numpy	import numpy as np
pd	pandas	import pandas as pd
plt	matplotlib	import matplotlib.pyplot as plt
px	plotly	import plotly.exprss as px
tf	tensorflow	import tensorflow as tf
sns	seaborn	import seaborn as sns
dt	datetime	import datetime as dt
pkl	pickle	import pickle as pkl

A.4. Functions in Python

Because python is not using Semicolon's it is import to keep track of indentation in your code. The indentation works as a placeholder for the semicolons. This is especially important if your are defining loops, functions, etc. …

Example: We are defining a function that calculates the squared sum of its input parameters

```
def squared_sum(x,y):
    z = x**2 + y**2
    return z
```

If you are working with something that needs indentation, it will be already done by the notebook.

Hint: Keep in mind that is good practice to use the *return* parameter. If you are not using *return* and a function has multiple paramaters that you would like to return, it will only return the last one defined.

A.5. List of Useful Jupyter Notebook Shortcuts

Table A.2.: List of useful Jupyter Notebook Shortcuts

Function	Keyboard Shortcut	Menu Tools
Save notebook	Esc + s	File → Save and Checkpoint
Create new Cell	Esc + a (above), Esc + b (below)	Insert → Cell above; Insert → Cell below
Run Cell	Ctrl + enter	Cell → Run Cell
Copy Cell	c	Copy Key
Paste Cell	v	Paste Key
Interrupt Kernel	Esc + i i	Kernel → Interrupt
Restart Kernel	Esc + 0 0	Kernel → Restart

If you combine everything you can create beautiful graphics

```
import matplotlib.pyplot as plt
import numpy as np

# Generate 100 random data points along 3 dimensions
x, y, scale = np.random.randn(3, 100)
fig, ax = plt.subplots()

# Map each onto a scatterplot we'll create with Matplotlib
ax.scatter(x=x, y=y, c=scale, s=np.abs(scale)*500)
ax.set(title="Some random data, created with the Jupyter Notebook!")
plt.show()
```

Some random data, created with the Jupyter Notebook!
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

B. Git Introduction

B.1. Learning Objectives

In this learning unit, you will learn how to set up Git as a version control system for a project. The most important Git commands will be explained. You will learn how to track and manage changes to your projects with Git. Specifically:

- Initializing a repository: `git init`
- Ignoring files: `.gitignore`
- Adding files to the staging area: `git add`
- Checking status changes: `git status`
- Reviewing history: `git log`
- Creating a new branch: `git branch`
- Switching to the current branch: `git switch` and `git checkout`
- Merging two branches: `git merge`
- Resolving conflicts
- Reverting changes: `git revert`
- Uploading changes to GitLab: `git push`
- Downloading changes from GitLab: `git pull`
- Advanced: `git rebase`

B.2. Basics of Git

B.2.1. Initializing a Repository: `git init`

To set up Git as a version control system for your project, you need to initialize a new Git repository at the top-level folder, which is the working directory of your project. This is done using the `git init` command.

All files in this folder and its subfolders will automatically become part of the repository. Creating a Git repository is similar to adding an all-powerful passive observer of all things to your project. Git sits there, observes, and takes note of even the smallest changes, such as a single character in a file within a repository with hundreds of files. And it will tell you where these changes occurred if you forget. Once Git is initialized, it monitors all changes made within the working directory, and it tracks the history of events from that point forward. For this purpose, a historical timeline is created

for your project, referred to as a "branch," and the initial branch is named `main`. So, when someone says they are on the `main branch` or working on the `main branch`, it means they are in the historical main timeline of the project. The Git repository, often abbreviated as `repo`, is a virtual representation of your project, including its history and branches, a book, if you will, where you can look up and retrieve the entire history of the project: you work in your working directory, and the Git repository tracks and stores your work.

B.2.2. Ignoring Files: `.gitignore`

It's useful that Git watches and keeps an eye on everything in your project. However, in most projects, there are files and folders that you don't need or want to keep an eye on. These may include system files, local project settings, libraries with dependencies, and so on.

You can exclude any file or folder from your Git repository by including them in the `.gitignore` file. In the `.gitignore` file, you create a list of file names, folder names, and other items that Git should not track, and Git will ignore these items. Hence the name "gitignore." Do you want to track a file that you previously ignored? Simply remove the mention of the file in the gitignore file, and Git will start tracking it again.

B.2.3. Adding Changes to the Staging Area: `git add`

The interesting thing about Git as an all-powerful, passive observer of all things is that it's very passive. As long as you don't tell Git what to remember, it will passively observe the changes in the project folder but do nothing.

When you make a change to your project that you want Git to include in the project's history to take a snapshot of so you can refer back to it later, your personal checkpoint, if you will, you need to first stage the changes in the staging area. What is the staging area? The staging area is where you collect changes to files that you want to include in the project's history.

This is done using the `git add` command. You can specify which files you want to add by naming them, or you can add all of them using `-A`. By doing this, you're telling Git that you've made changes and want it to remember these particular changes so you can recall them later if needed. This is important because you can choose which changes you want to stage, and those are the changes that will eventually be transferred to the history.

Note: When you run `git add`, the changes are not transferred to the project's history. They are only transferred to the staging area.

Example B.1 (Example of git add from the beginning)**.**

```
# Create a new directory for your
# repository and navigate to that directory:

mkdir my-repo
cd my-repo

# Initialize the repository with git init:

git init

# Create a .gitignore file for Python code.
# You can use a template from GitHub:

curl https://raw.githubusercontent.com/github/gitignore/master/Python.gitignore -o .gitignore

# Add your files to the repository using git add:

git add .
```

This adds all files in the current directory to the repository, except for the files listed in the .gitignore file.

B.2.4. Transferring Changes to Memory: `git commit`

The power of Git becomes evident when you start transferring changes to the project history. This is done using the `git commit` command. When you run `git commit`, you inform Git that the changes in the staging area should be added to the history of the project so that they can be referenced or retrieved later.

Additionally, you can add a commit message with the `-m` option to explain what changes were made. So when you look back at the project history, you can see that you added a new feature.

`git commit` creates a snapshot, an image of the current state of your project at that specific time, and adds it to the branch you are currently working on.

As you work on your project and transfer more snapshots, the branch grows and forms a timeline of events. This means you can now look back at every transfer in the branch and see what your code looked like at that time.

You can compare any phase of your code with any other phase of your code to find errors, restore deleted code, or do things that would otherwise not be possible, such as resetting the project to a previous state or creating a new timeline from any point.

So how often should you add these commits? My rule of thumb is not to commit too often. It's better to have a Git repository with too many commits than one with too few commits.

Example B.2 (Continuing the example from above:)**.** After adding your files with `git add`, you can create a commit to save your changes. Use the `git commit` command with the `-m` option to specify your commit message:

```
git commit -m "My first commit message"
```

This creates a new commit with the added files and the specified commit message.

B.2.5. Check the Status of Your Repository: `git status`

If you're wondering what you've changed in your project since the last commit snapshot, you can always check the Git status. Git will list every modified file and the current status of each file.

This status can be either:

- Unchanged (`unmodified`), meaning nothing has changed since you last transferred it, or
- It's been changed (`changed`) but not staged (`staged`) to be transferred into the history, or
- Something has been added to staging (`staged`) and is ready to be transferred into the history.

When you run `git status`, you get an overview of the current state of your project.

Example B.3 (Continuing the example from above:)**.** The `git status` command displays the status of your working directory and the staging area. It shows you which files have been modified, which files are staged for commit, and which files are not yet being tracked:

```
git status
```

`git status` is a useful tool to keep track of your changes and ensure that you have added all the desired files for commit.

B.2.6. Review Your Repository's History: `git log`

Example B.4 (Continuing the example from above:)**.** You can view the history of your commits with the `git log` command. This command displays a list of all the commits in the current branch, along with information such as the author, date, and commit message:

```
git log
```

There are many options to customize the output of `git log`. For example, you can use the `--pretty` option to change the format of the output:

```
git log --pretty=oneline
```

This displays each commit in a single line.

B.3. Branches (Timelines)

B.3.1. Creating an Alternative Timeline: `git branch`

In the course of developing a project, you often reach a point where you want to add a new feature, but doing so might require changing the existing code in a way that could be challenging to undo later.

Or maybe you just want to experiment and be able to discard your work if the experiment fails. In such cases, Git allows you to create an alternative timeline called a `branch` to work in.

This new `branch` has its own name and exists in parallel with the `main branch` and all other branches in your project.

During development, you can switch between branches and work on different versions of your code concurrently. This way, you can have a stable codebase in the `main branch` while developing an experimental feature in a separate `branch`. When you switch from one `branch` to another, the code you're working on is automatically reset to the latest commit of the branch you're currently in.

If you're working in a team, different team members can work on their own branches, creating an entire universe of alternative timelines for your project. When features are completed, they can be seamlessly merged back into the `main branch`.

Example B.5 (Continuing the example from above:)**.** To create a new `branch`, you can use the `git branch` command with the name of the new `branch` as an argument:

```
git branch my-tests
```

B.3.2. The Pointer to the Current Branch: HEAD

How does Git know where you are on the timeline, and how can you keep track of your position?

You're always working at the tip (HEAD) of the currently active branch. The HEAD pointer points there quite literally. In a new project archive with just a single main branch and only new commits being added, HEAD always points to the latest commit in the main branch. That's where you are.

However, if you're in a repository with multiple branches, meaning multiple alternative timelines, HEAD will point to the latest commit in the branch you're currently working on.

B.3.3. Switching to an Alternative Timeline: git switch

As your project grows, and you have multiple branches, you need to be able to switch between these branches. This is where the switch command comes into play.

At any time, you can use the git switch command with the name of the branch you want to switch to, and HEAD moves from your current branch to the one you specified.

If you've made changes to your code before switching, Git will attempt to carry those changes over to the branch you're switching to. However, if these changes conflict with the target branch, the switch will be canceled.

To resolve this issue without losing your changes, return to the original branch, add and commit your recent changes, and then perform the switch.

B.3.4. Switching to an Alternative Timeline and Making Changes: git checkout

To switch between branches, you can also use the git checkout command. It works similarly to git switch for this purpose: you pass the name of the branch you want to switch to, and HEAD moves to the beginning of that branch.

But checkout can do more than just switch to another timeline. With git checkout, you can also move to any commit point in any timeline. In other words, you can travel back in time and work on code from the past.

To do this, use git checkout and provide the commit ID. This is an automatically generated, random combination of letters and numbers that identifies each commit.

You can retrieve the commit ID using `git log`. When you run `git log`, you get a list of all the commits in your repository, starting with the most recent ones.

When you use `git checkout` with an older commit ID, you check out a commit in the middle of a branch. This disrupts the timeline, as you're actively attempting to change history. Git doesn't want you to do that because, much like in a science fiction movie, altering the past might also alter the future. In our case, it would break the version control branch's coherence.

To prevent you from accidentally disrupting time and altering history, checking out an earlier commit in any branch results in the warning "Detached Head," which sounds rather ominous. The "Detached Head" warning is appropriate because it accurately describes what's happening. Git literally detaches the head from the branch and sets it aside.

Now, you're working outside of time in a space unbound to any timeline, which again sounds rather threatening but is perfectly fine in reality.

To continue working on this past code, all you need to do is reattach it to the timeline. You can use `git branch` to create a new branch, and the detached head will automatically attach to this new branch.

Instead of breaking the history, you've now created a new alternative timeline that starts in the past, allowing you to work safely. You can continue working on the branch as usual.

Example B.6 (Continuing the example from above:)**.** To switch to a new branch, you can use the `git checkout` command:

```
git checkout meine-tests
```

Now you're using the new branch and can make changes independently from the original branch.

B.3.5. The Difference Between `checkout` and `switch`

What is the difference between `git switch` and `git checkout`? `git switch` and `git checkout` are two different commands that both serve the purpose of switching between branches. You can use both to switch between branches, but they have an important distinction. `git switch` is a new command introduced with Git 2.23. `git checkout` is an older command that has existed since Git 1.6.0. So, `git switch` and `git checkout` have different origins. `git switch` was introduced to separate the purposes of `git checkout`. `git checkout` has two different purposes: 1. It can be used to switch between branches, and 2. It can be used to reset files to the state of the last commit.

Here's an example: In my project, I made a change since the last commit, but I haven't staged it yet. Then, I realized that I actually don't want this change. I want to reset the file to the state before the last commit. As long as I haven't committed my changes, I can do this with `git checkout` by targeting the specific file. So, if that file is named `main.js`, I can say: `git checkout main.js`. And the file will be reset to the state of the last commit, which makes sense. I'm checking out the file from the last commit.

But that's quite different from switching between the beginning of one branch to another. `git switch` and `git restore` were introduced to separate these two operations. `git switch` is for switching between branches, and `git restore` is for resetting the specified file to the state of the last commit. If you try to restore a file with `git switch`, it simply won't work. It's not intended for that. As I mentioned earlier, it's about separating concerns.

Example B.7 (Difference between `git switch` and `git checkout`)**.** Here's an example demonstrating how to initialize a repository and switch between branches:

```
# Create a new directory for your repository
# and navigate to that directory:
mkdir my-repo
cd my-repo

# Initialize the repository with git init:
git init

# Create a new branch with git branch:
git branch my-new-branch

# Switch to the new branch using git switch:
git switch my-new-branch

# Alternatively, you can also use git checkout
# to switch to the new branch:

git checkout my-new-branch
```

Both commands lead to the same result: You are now on the new branch.

B.4. Merging Branches and Resolving Conflicts

B.4.1. git `merge`: Merging Two Timelines

Git allows you to split your development work into as many branches or alternative timelines as you like, enabling you to work on many different versions of your code

simultaneously without losing or overwriting any of your work.

This is all well and good, but at some point, you need to bring those various versions of your code back together into one branch. That's where `git merge` comes in.

Consider an example where you have two branches, a `main branch` and an experimental branch called `experimental-branch`. In the experimental branch, there is a new feature. To merge these two branches, you set `HEAD` to the branch where you want to incorporate the code and execute `git merge` followed by the name of the branch you want to merge. `HEAD` is a special pointer that points to the current branch. When you run `git merge`, it combines the code from the branch associated with `HEAD` with the code from the branch specified by the branch name you provide.

```
# Initialize the repository
git init

# Create a new branch called "experimental-branch"
git branch experimental-branch

# Switch to the "experimental-branch"
git checkout experimental-branch

# Add the new feature here and
# make a commit
# ...

# Switch back to the "main" branch
git checkout main

# Perform the merge
git merge experimental-branch
```

During the merge, matching pieces of code in the branches overlap, and any new code from the branch being merged is added to the project. So now, the main branch also contains the code from the experimental branch, and the events of the two separate timelines have been merged into a single one. What's interesting is that even though the experimental branch was merged with the main branch, the last commit of the experimental branch remains intact, allowing you to continue working on the experimental branch separately if you wish.

B.4.2. Resolving Conflicts When Merging

Merging branches where there are no code changes at the same place in both branches is a straightforward process. It's also a rare process. In most cases, there will be some

form of conflict between the branches – the same code or the same code area has been modified differently in the different branches. Merging two branches with such conflicts will not work, at least not automatically.

In this case, Git doesn't know how to merge this code. So, when such a situation occurs, it's marked as a conflict, and the merging process is halted. This might sound more dramatic than it is. When you get a conflict warning, Git is saying there are two different versions here, and Git needs to know which one you want to keep. To help you figure out the conflict, Git combines all the code into a single file and automatically marks the conflicting code as the current change, which is the original code from the branch you're working on, or as the incoming change, which is the code from the file you're trying to merge.

To resolve this conflict, you'll edit the file to literally resolve the code conflict. This might mean accepting either the current or incoming change and discarding the other. It could mean combining both changes or something else entirely. It's up to you. So, you edit the code to resolve the conflict. Once you've resolved the conflict by editing the code, you add the new conflict-free version to the staging area with `git add` and then commit the merged code with `git commit`. That's how the conflict is resolved.

A merge conflict occurs when Git struggles to automatically merge changes from two different branches. This usually happens when changes were made to the same line in the same file in both branches. To resolve a merge conflict, you must manually edit the affected files and choose the desired changes. Git marks the conflict areas in the file with special markings like `<<<<<<<`, `=======`, and `>>>>>>>`. You can search for these markings and manually select the desired changes. After resolving the conflicts, you can add the changes with `git add` and create a new commit with `git commit` to complete the merge.

Example B.8.

```
# Perform the merge (this will cause a conflict)
git merge experimenteller-branch

# Open the affected file in an editor and manually resolve the conflicts
# ...

# Add the modified file
git add <filename>

# Create a new commit
git commit -m "Resolved conflicts"
```

B.4.3. git `revert`: Undoing Something

One of the most powerful features of any software tool is the "Undo" button. Make a mistake, press "Undo," and it's as if it never happened. However, that's not quite as simple when an all-powerful, passive observer is watching and recording your project's history. How do you undo something that you've added to the history without rewriting the history?

The answer is that you can overwrite the history with the `git reset` command, but that's quite risky and not a good practice.

A better solution is to work with the historical timeline and simply place an older version of your code at the top of the branch. This is done with `git revert`. To make this work, you need to know the commit ID of the commit you want to go back to.

The commit ID is a machine-generated set of random numbers and letters, also known as a hash. To get a list of all the commits in the repository, including the commit ID and commit message, you can run `git log`.

```
# Show the list of all operations in the repository
git log
```

By the way, it's a good idea to leave clear and informative commit messages for this reason. This way, you know what happened in your previous commits. Once you've found the commit you want to revert to, call that commit ID with `git revert`, and then the ID. This will create a new commit at the top of the branch with the code from the reference commit. To transfer the code to the branch, add a commit message and save it. Now, the last commit in your branch matches the commit you're reverting to, and your project's history remains intact.

Example B.9 (An example with `git revert`)**.**

```
# Initialize a new repository
git init

# Create a new file
echo "Hello, World" > file.txt

# Add the file to the repository
git add file.txt

# Create a new commit
git commit -m "First commit"

# Modify the file
```

```
echo "Goodbye, World" > file.txt

# Add the modified file
git add file.txt

# Create a new commit
git commit -m "Second commit"

# Use git log to find the commit ID of the second commit
git log

# Use git revert to undo the changes from the second commit
git revert <commit-id>
```

To download the `students` branch from the repository `git@git-ce.rwth-aachen.de:spotseven-lab/numerische-mathematik-sommersemester2023.git` to your local machine, add a file, and upload the changes, you can follow these steps:

Example B.10 (An example with `git clone`, `git checkout`, `git add`, `git commit`, `git push`)**.**

```
# Clone the repository to your local machine:
git clone git@git-ce.rwth-aachen.de:spotseven-lab/numerische-mathematik-sommersemester

# Change to the cloned repository:
cd numerische-mathematik-sommersemester2023

# Switch to the students branch:
git checkout students

# Create the Test folder if it doesn't exist:
mkdir Test

# Create the Testdatei.txt file in the Test folder:
touch Test/Testdatei.txt

# Add the file with git add:
git add Test/Testdatei.txt

# Commit the changes with git commit:
git commit -m "Added Testdatei.txt"

# Push the changes with git push:
git push origin students
```

This will upload the changes to the server and update the students branch in the repository.

B.5. Downloading from GitLab

To download changes from a GitLab repository to your local machine, you can use the `git pull` command. This command downloads the latest changes from the specified remote repository and merges them with your local repository.

Here is an example:

Example B.11 (An example with `git pull`).

```
# Navigate to the local repository
# linked to the GitHub repository:
cd my-local-repository

# Make sure you are in the correct branch:
git checkout main

# Download the latest changes from GitHub:
git pull origin main
```

This downloads the latest changes from the main branch of the remote repository named "origin" and merges them with your local repository.

If there are conflicts between the downloaded changes and your local changes, you will need to resolve them manually before proceeding.

B.6. Advanced

B.6.1. git `rebase`: Moving the Base of a Branch

In some cases, you may need to "rewrite history." A common scenario is that you've been working on a new feature in a feature branch, and you realize that the work should have actually happened in the `main branch`.

To resolve this issue and make it appear as if the work occurred in the `main branch`, you can reset the experimental branch. "Rebase" literally means detaching the base of the experimental branch and moving it to the beginning of another branch, giving the branch a new base, thus "rebasing."

This operation is performed from the branch you want to "rebase." You use `git rebase` and specify the branch you want to use as the new base. If there are no conflicts between the experimental branch and the branch you want to rebase onto, this process happens automatically.

If there are conflicts, Git will guide you through the conflict resolution process for each commit from the rebase branch.

This may sound like a lot, but there's a good reason for it. You are literally rewriting history by transferring commits from one branch to another. To maintain the coherence of the new version history, there should be no conflicts within the commits. So, you need to resolve them one by one until the history is clean. It goes without saying that this can be a fairly labor-intensive process. Therefore, you should not use `git rebase` frequently.

Example B.12 (An example with `git rebase`)**.** `git rebase` is a command used to change the base of a branch. This means that commits from the branch are applied to a new base, which is usually another branch. It can be used to clean up the repository history and avoid merge conflicts.

Here is an example showing how to use `git rebase`:

- In this example, we initialize a new Git repository and create a new file. We add the file to the repository and make an initial commit. Then, we create a new branch called "feature" and switch to that branch. We make changes to the file in the feature branch and create a new commit.
- Then, we switch back to the main branch and make changes to the file again. We add the modified file and make another commit.
- To rebase the feature branch onto the main branch, we first switch to the feature branch and then use the `git rebase` command with the name of the main branch as an argument. This applies the commits from the feature branch to the main branch and changes the base of the feature branch.

```
# Initialize a new repository
git init
# Create a new file
echo "Hello World" > file.txt
# Add the file to the repository
git add file.txt
# Create an initial commit
git commit -m "Initial commit"
# Create a new branch called "feature"
git branch feature
# Switch to the "feature" branch
git checkout feature
```

```
# Make changes to the file in the "feature" branch
echo "Hello Feature World" > file.txt
# Add the modified file
git add file.txt
# Create a new commit in the "feature" branch
git commit -m "Feature commit"
# Switch back to the "main" branch
git checkout main
# Make changes to the file in the "main" branch
echo "Hello Main World" > file.txt
# Add the modified file
git add file.txt
# Create a new commit in the "main" branch
git commit -m "Main commit"
# Use git rebase to rebase the "feature" branch
# onto the "main" branch
git checkout feature
git rebase main
```

B.7. Exercises

In order to be able to carry out this exercise, we provide you with a functional working environment. This can be accessed here. You can log in using your GMID. If you do not have one, you can generate one here. Once you have successfully logged in to the server, you must open a terminal instance. You are now in a position to carry out the exercise.

Alternatively, you can also carry out the exercise locally on your computer, but then you will need to install git.

B.7.1. Create project folder

First create the `test-repo` folder via the command line and then navigate to this folder using the corresponding command.

B.8. Initialize repo

Now initialize the repository so that the future project, which will be saved in the `test-repo` folder, and all associated files are versioned.

B.8.1. Do not upload / ignore certain file types

In order to carry out this exercise, you must first download a file which you then have git ignore. To do this, download the current examination regulations for the Bachelor's degree program in Electrical Engineering using the following command `curl -o pruefungsordnung.pdf https://www.th-koeln.de/mam/downloads/deutsch/studium/studiengaenge/f07/ordnungen_plaene/f07_bpo_ba_e`

The PDF file has been stored in the root directory of your repo and you must now exclude it from being uploaded so that no changes to this file are tracked. Please note that not only this one PDF file should be ignored, but all PDF files in the repo.

B.8.2. Create file and stage it

In order to be able to commit a change later and thus make it traceable, it must first be staged. However, as we only have a PDF file so far, which is to be ignored by git, we cannot stage anything. Therefore, in this task, a file `test.txt` with some string as content is to be created and then staged.

B.8.3. Create another file and check status

To understand the status function, you should create the file `test2.txt` and then call the status function of git.

B.8.4. Commit changes

After the changes to the `test.txt` file have been staged and these are now to be transferred to the project process, they must be committed. Therefore, in this step you should perform a corresponding commit in the current branch with the message `test-commit`. Finally, you should also display the history of the commits.

B.8.5. Create a new branch and switch to it

In this task, you are to create a new branch with the name `change-text` in which you will later make changes. You should then switch to this branch.

B.8.6. Commit changes in the new branch

To be able to merge the new branch into the main branch later, you must first make changes to the `test.txt` file. To do this, open the file and simply change the character string in this file before saving the changes and closing the file. Before you now commit the file, you should reset the file to the status of the last commit for practice purposes and thus undo the change. After you have done this, open the file `test.txt` again and change the character string again before saving and closing the file. This time you should commit the file `test.txt` and then commit it with the message `test-commit2`.

B.8.7. Merge branch into main

After you have committed the change to the `test.txt` file, you should merge the `change-text` branch including the change into the main branch so that it is also available there.

B.8.8. Resolve merge conflict

To simulate a merge conflict, you must first change the content of the `test.txt` file before you commit the change. Then switch to the branch `change-text` and change the file `test.txt` there as well before you commit the change. Now you should try to merge the branch `change-text` into the main branch and solve the problems that occur in order to be able to perform the merge successfully.

C. Python

C.1. Data Types and Precision in Python

The float16 data type in numpy represents a half-precision floating point number. It uses 16 bits of memory, which gives it a precision of about 3 decimal digits.

The float32 data type in numpy represents a single-precision floating point number. It uses 32 bits of memory, which gives it a precision of about 7 decimal digits. On the other hand, float64 represents a double-precision floating point number. It uses 64 bits of memory, which gives it a precision of about 15 decimal digits.

The reason float16 and float32 show fewer digits is because it has less precision due to using less memory. The bits of memory are used to store the sign, exponent, and fraction parts of the floating point number, and with fewer bits, you can represent fewer digits accurately.

Example C.1 (16 versus 32 versus 64 bit)**.**

```python
import numpy as np

# Define a number
num = 0.123456789123456789

num_float16 = np.float16(num)
num_float32 = np.float32(num)
num_float64 = np.float64(num)

print("float16: ", num_float16)
print("float32: ", num_float32)
print("float64: ", num_float64)
```

```
float16:  0.1235
float32:  0.12345679
float64:  0.12345678912345678
```

C.2. Recommendations

Beginner's Guide to Python

D. Gaussian Processes—Some Background Information

The concept of GP (Gaussian Process) regression can be understood as a simple extension of linear modeling. It is worth noting that this approach goes by various names and acronyms, including "kriging," a term derived from geostatistics, as introduced by Matheron in 1963. Additionally, it is referred to as Gaussian spatial modeling or a Gaussian stochastic process, and machine learning (ML) researchers often use the term Gaussian process regression (GPR). In all of these instances, the central focus is on regression. This involves training on both inputs and outputs, with the ultimate objective of making predictions and quantifying uncertainty (referred to as uncertainty quantification or UQ).

However, it's important to emphasize that GPs are not a universal solution for every problem. Specialized tools may outperform GPs in specific, non-generic contexts, and GPs have their own set of limitations that need to be considered.

D.1. Gaussian Process Prior

In the context of GP, any finite collection of realizations, which is represented by n observations, is modeled as having a multivariate normal (MVN) distribution. The characteristics of these realizations can be fully described by two key parameters:

1. Their mean, denoted as an n-vector μ.
2. The covariance matrix, denoted as an $n \times n$ matrix Σ. This covariance matrix encapsulates the relationships and variability between the individual realizations within the collection.

D.2. Covariance Function

The covariance function is defined by inverse exponentiated squared Euclidean distance:

$$\Sigma(\vec{x}, \vec{x}') = \exp\{-||\vec{x} - \vec{x}'||^2\},$$

where $\vec{x}$ and $\vec{x}'$ are two points in the k-dimensional input space and $\|\cdot\|$ denotes the Euclidean distance, i.e.,

$$||\vec{x} - \vec{x}'||^2 = \sum_{i=1}^{k} (x_i - x_i')^2.$$

An 1-d example is shown in Figure D.1.

Figure D.1.: One-dim inverse exponentiated squared Euclidean distance

The covariance function is also referred to as the kernel function. The *Gaussian* kernel uses an additional parameter, σ^2, to control the rate of decay. This parameter is referred to as the length scale or the characteristic length scale. The covariance function is then defined as

$$\Sigma(\vec{x}, \vec{x}') = \exp\{-||\vec{x} - \vec{x}'||^2/(2\sigma^2)\}. \tag{D.1}$$

The covariance decays exponentially fast as $\vec{x}$ and $\vec{x}'$ become farther apart. Observe that

$$\Sigma(\vec{x}, \vec{x}) = 1$$

and

$$\Sigma(\vec{x}, \vec{x}') < 1$$

for $\vec{x} \neq \vec{x}'$. The function $\Sigma(\vec{x}, \vec{x}')$ must be positive definite.

Remark D.1 (Kriging and Gaussian Basis Functions). The Kriging basis function (Equation 9.1) is related to the 1-dim Gaussian basis function (Equation D.1), which is defined as

$$\Sigma(\vec{x}^{(i)}, \vec{x}^{(j)}) = \exp\{-||\vec{x}^{(i)} - \vec{x}^{(j)}||^2/(2\sigma^2)\}. \tag{D.2}$$

There are some differences between Gaussian basis functions and Kriging basis functions:

- Where the Gaussian basis function has $1/(2\sigma^2)$, the Kriging basis has a vector $\theta = [\theta_1, \theta_2, \dots, \theta_k]^T$.
- The θ vector allows the width of the basis function to vary from dimension to dimension.
- In the Gaussian basis function, the exponent is fixed at 2, Kriging allows this exponent p_l to vary (typically from 1 to 2).

D.2.1. Positive Definiteness

Positive definiteness in the context of the covariance matrix Σ_n is a fundamental requirement. It is determined by evaluating $\Sigma(x_i, x_j)$ at pairs of n $\vec{x}$-values, denoted as $\vec{x}_1, \vec{x}_2, \dots, \vec{x}_n$. The condition for positive definiteness is that for all $\vec{x}$ vectors that are not equal to zero, the expression $\vec{x}^\top \Sigma_n \vec{x}$ must be greater than zero. This property is essential when intending to use Σ_n as a covariance matrix in multivariate normal (MVN) analysis. It is analogous to the requirement in univariate Gaussian distributions where the variance parameter, σ^2, must be positive.

Gaussian Processes (GPs) can be effectively utilized to generate random data that follows a smooth functional relationship. The process involves the following steps:

1. Select a set of $\vec{x}$-values, denoted as $\vec{x}_1, \vec{x}_2, \dots, \vec{x}_n$.
2. Define the covariance matrix Σ_n by evaluating $\Sigma_n^{ij} = \Sigma(\vec{x}_i, \vec{x}_j)$ for $i, j = 1, 2, \dots, n$.
3. Generate an n-variate realization Y that follows a multivariate normal distribution with a mean of zero and a covariance matrix Σ_n, expressed as $Y \sim \mathcal{N}_n(0, \Sigma_n)$.
4. Visualize the result by plotting it in the x-y plane.

D.3. Construction of the Covariance Matrix

Here is an one-dimensional example. The process begins by creating an input grid using $\vec{x}$-values. This grid consists of 100 elements, providing the basis for further analysis and visualization.

```
import numpy as np
n = 100
X = np.linspace(0, 10, n, endpoint=False).reshape(-1,1)
```

In the context of this discussion, the construction of the covariance matrix, denoted as Σ_n, relies on the concept of inverse exponentiated squared Euclidean distances. However, it's important to note that a modification is introduced later in the process. Specifically, the diagonal of the covariance matrix is augmented with a small value, represented as "eps" or ϵ.

The reason for this augmentation is that while inverse exponentiated distances theoretically ensure the covariance matrix's positive definiteness, in practical applications, the matrix can sometimes become numerically ill-conditioned. By adding a small value to the diagonal, such as ϵ, this ill-conditioning issue is mitigated. In this context, ϵ is often referred to as "jitter."

```
import numpy as np
from numpy import array, zeros, power, ones, exp, multiply, eye, linspace, spacing, so
from numpy.linalg import cholesky, solve
from numpy.random import multivariate_normal
def build_Sigma(X, sigma2):
    n = X.shape[0]
    k = X.shape[1]
    D = zeros((k, n, n))
    for l in range(k):
        for i in range(n):
            for j in range(i, n):
                D[l, i, j] = 1/(2*sigma2[l])*(X[i,l] - X[j,l])**2
    D = sum(D)
    D = D + D.T
    return exp(-D)
```

```
sigma2 = np.array([1.0])
Sigma = build_Sigma(X, sigma2)
np.round(Sigma[:3,:], 3)
```

```
array([[1.   , 0.995, 0.98 , 0.956, 0.923, 0.882, 0.835, 0.783, 0.726,
        0.667, 0.607, 0.546, 0.487, 0.43 , 0.375, 0.325, 0.278, 0.236,
        0.198, 0.164, 0.135, 0.11 , 0.089, 0.071, 0.056, 0.044, 0.034,
        0.026, 0.02 , 0.015, 0.011, 0.008, 0.006, 0.004, 0.003, 0.002,
        0.002, 0.001, 0.001, 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
```

```
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   ],
[0.995, 1.   , 0.995, 0.98 , 0.956, 0.923, 0.882, 0.835, 0.783,
 0.726, 0.667, 0.607, 0.546, 0.487, 0.43 , 0.375, 0.325, 0.278,
 0.236, 0.198, 0.164, 0.135, 0.11 , 0.089, 0.071, 0.056, 0.044,
 0.034, 0.026, 0.02 , 0.015, 0.011, 0.008, 0.006, 0.004, 0.003,
 0.002, 0.002, 0.001, 0.001, 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   ],
[0.98 , 0.995, 1.   , 0.995, 0.98 , 0.956, 0.923, 0.882, 0.835,
 0.783, 0.726, 0.667, 0.607, 0.546, 0.487, 0.43 , 0.375, 0.325,
 0.278, 0.236, 0.198, 0.164, 0.135, 0.11 , 0.089, 0.071, 0.056,
 0.044, 0.034, 0.026, 0.02 , 0.015, 0.011, 0.008, 0.006, 0.004,
 0.003, 0.002, 0.002, 0.001, 0.001, 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   ]])
```

```python
import matplotlib.pyplot as plt
plt.imshow(Sigma, cmap='hot', interpolation='nearest')
plt.colorbar()
plt.show()
```

D.4. Generation of Random Samples and Plotting the Realizations of the Random Function

In the context of the multivariate normal distribution, the next step is to utilize the previously constructed covariance matrix denoted as `Sigma`. It is used as an essential component in generating random samples from the multivariate normal distribution.

The function `multivariate_normal` is employed for this purpose. It serves as a random number generator specifically designed for the multivariate normal distribution. In this case, the mean of the distribution is set equal to `mean`, and the covariance matrix is provided as `Psi`. The argument `size` specifies the number of realizations, which, in this specific scenario, is set to one.

By default, the mean vector is initialized to zero. To match the number of samples, which is equivalent to the number of rows in the `X` and `Sigma` matrices, the argument `zeros(n)` is used, where `n` represents the number of samples (here taken from the size of the matrix, e.g.,: `Sigma.shape[0]`).

```
rng = np.random.default_rng(seed=12345)
Y = rng.multivariate_normal(zeros(Sigma.shape[0]), Sigma, size = 1, check_valid="raise
Y.shape
```

```
(100, 1)
```

Now we can plot the results, i.e., a finite realization of the random function $Y()$ under a GP prior with a particular covariance structure. We will plot those `X` and `Y` pairs as connected points on an x-y plane.

```
import matplotlib.pyplot as plt
plt.plot(X, Y)
plt.title("Realization of Random Functions under a GP prior.\n sigma2: {}".format(sigma2[0]))
plt.show()
```

Figure D.2.: Realization of one random function under a GP prior. sigma2: 1.0

```
rng = np.random.default_rng(seed=12345)
Y = rng.multivariate_normal(zeros(Sigma.shape[0]), Sigma, size = 3, check_valid="raise")
plt.plot(X, Y.T)
plt.title("Realization of Three Random Functions under a GP prior.\n sigma2: {}".format(sigma2[0]))
plt.show()
```

Figure D.3.: Realization of three random functions under a GP prior. sigma2: 1.0

D.5. Properties of the 1d Example

D.5.1. Several Bumps:

In this analysis, we observe several bumps in the x-range of $[0, 10]$. These bumps in the function occur because shorter distances exhibit high correlation, while longer distances tend to be essentially uncorrelated. This leads to variations in the function's behavior:

- When x and x' are one σ unit apart, the correlation is $\exp\left(-\sigma^2/(2\sigma^2)\right) = \exp(-1/2) \approx 0.61$, i.e., a relative high correlation.
- 2σ apart means correlation $\exp(-2^2/2) \approx 0.14$, i.e., only small correlation.
- 4σ apart means correlation $\exp(-4^2/2) \approx 0.0003$, i.e., nearly no correlation—variables are considered independent for almost all practical application.

D.5.2. Smoothness:

The function plotted in Figure D.2 represents only a finite realization, which means that we have data for a limited number of pairs, specifically 100 points. These points appear smooth in a tactile sense because they are closely spaced, and the plot function connects the dots with lines to create the appearance of smoothness. The complete surface, which can be conceptually extended to an infinite realization over a compact domain, is exceptionally smooth in a calculus sense due to the covariance function's property of being infinitely differentiable.

D.5.3. Scale of Two:

Regarding the scale of the Y values, they have a range of approximately $[-2, 2]$, with a 95% probability of falling within this range. In standard statistical terms, 95% of the data points typically fall within two standard deviations of the mean, which is a common measure of the spread or range of data.

```
import numpy as np
from numpy import array, zeros, power, ones, exp, multiply, eye, linspace, spacing, sqrt, arange, ap
from numpy.random import multivariate_normal

def build_Sigma(X, sigma2):
    n = X.shape[0]
    k = X.shape[1]
    D = zeros((k, n, n))
    for l in range(k):
        for i in range(n):
            for j in range(i, n):
                D[l, i, j] = 1/(2*sigma2[l])*(X[i,l] - X[j,l])**2
    D = sum(D)
    D = D + D.T
    return exp(-D)

def plot_mvn( a=0, b=10, sigma2=1.0, size=1, n=100, show=True):
    X = np.linspace(a, b, n, endpoint=False).reshape(-1,1)
    sigma2 = np.array([sigma2])
    Sigma = build_Sigma(X, sigma2)
    rng = np.random.default_rng(seed=12345)
    Y = rng.multivariate_normal(zeros(Sigma.shape[0]), Sigma, size = size, check_valid="raise")
    plt.plot(X, Y.T)
    plt.title("Realization of Random Functions under a GP prior.\n sigma2: {}".format(sigma2[0]))
    if show:
        plt.show()
```

```
plot_mvn(a=0, b=10, sigma2=10.0, size=3, n=250)
```

Figure D.4.: Realization of Random Functions under a GP prior. sigma2: 10

```
plot_mvn(a=0, b=10, sigma2=0.1, size=3, n=250)
```

Figure D.5.: Realization of Random Functions under a GP prior. sigma2: 0.1

D.6. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

E. Datasets

E.1. The Diabetes Data Set

This section describes the `Diabetes` data set. This is a PyTorch Dataset for regression, which is derived from the `Diabetes` data set from `scikit-learn` (sklearn). Ten baseline variables, age, sex, body mass index, average blood pressure, and six blood serum measurements were obtained for each of n = 442 diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline.

E.1.1. Data Exploration of the sklearn Diabetes Data Set

```
from sklearn.datasets import load_diabetes
from spotpython.plot.xy import plot_y_vs_X
data = load_diabetes()
X, y = data.data, data.target
plot_y_vs_X(X, y, nrows=5, ncols=2, figsize=(20, 15))
```

- Each of these 10 feature variables have been mean centered and scaled by the standard deviation times the square root of n_samples (i.e., the sum of squares of each column totals 1).
- `s3_hdl` shows a different behavior than the other features. It has a negative slope. `HDL` (high-density lipoprotein) cholesterol, sometimes called "good" cholesterol, absorbs cholesterol in the blood and carries it back to the liver. The liver then flushes it from the body. High levels of HDL cholesterol can lower your risk for heart disease and stroke.

E.1.2. Generating the PyTorch Data Set

`spotpython` provides a `Diabetes` class to load the diabetes data set. The `Diabetes` class is a subclass of `torch.utils.data.Dataset`. It loads the diabetes data set from `sklearn` and returns the data set as a `torch.utils.data.Dataset` object, so that features and targets can be accessed as `torch.tensors`. [CODE REFERENCE].

```
from spotpython.data.diabetes import Diabetes
data_set = Diabetes()
```

```
print(len(data_set))
print(data_set.names)
```

```
442
['age', 'sex', 'bmi', 'bp', 's1_tc', 's2_ldl', 's3_hdl', 's4_tch', 's5_ltg', 's6_glu']
```

E.2. The Friedman Drift Dataset

E.2.1. The Friedman Drift Dataset as Implemented in `river`

We will describe the Friedman synthetic dataset with concept drifts [SOURCE], see also Friedman (1991) and Ikonomovska, Gama, and Džeroski (2011). Each observation is composed of ten features. Each feature value is sampled uniformly in [0, 1]. Only the first five features are relevant. The target is defined by different functions depending on the type of the drift. Global Recurring Abrupt drift will be used, i.e., the concept drift appears over the whole instance space.

The target is defined by the following function:

$$y = 10\sin(\pi x_0 x_1) + 20(x_2 - 0.5)^2 + 10x_3 + 5x_4 + \epsilon,$$

where $\epsilon \sim \mathcal{N}(0, 1)$ is normally distributed noise.

If the Global Recurring Abrupt drift variant of the Friedman Drift dataset is used, the target function changes at two points in time, namely p_1 and p_2. At the first point, the concept changes to:

$$y = 10\sin(\pi x_3 x_5) + 20(x_1 - 0.5)^2 + 10x_0 + 5x_2 + \epsilon,$$

At the second point of drift the old concept reoccurs. This can be implemented as follows, see https://riverml.xyz/latest/api/datasets/synth/FriedmanDrift/:

```
def __iter__(self):
    rng = random.Random(self.seed)

    i = 0
    while True:
        x = {i: rng.uniform(a=0, b=1) for i in range(10)}
        y = self._global_recurring_abrupt_gen(x, i) + rng.gauss(mu=0, sigma=1)

        yield x, y
        i += 1
```

```
def _global_recurring_abrupt_gen(self, x, index: int):
    if index < self._change_point1 or index >= self._change_point2:
        # The initial concept is recurring
        return (
            10 * math.sin(math.pi * x[0] * x[1]) + 20 * (x[2] - 0.5) ** 2 + 10 * x[3]
        )
    else:
        # Drift: the positions of the features are swapped
        return (
            10 * math.sin(math.pi * x[3] * x[5]) + 20 * (x[1] - 0.5) ** 2 + 10 * x[0]
        )
```

spotpython requires the specification of a **train** and **test** data set. These data sets can be generated as follows:

```
from river.datasets import synth
import pandas as pd
import numpy as np
from spotriver.utils.data_conversion import convert_to_df

seed = 123
shuffle = True
n_train = 6_000
n_test = 4_000
n_samples = n_train + n_test
target_column = "y"

dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=(n_train/4, n_train/2),
   seed=123
)

train = convert_to_df(dataset, n_total=n_train)
train.columns = [f"x{i}" for i in range(1, 11)] + [target_column]


dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=(n_test/4, n_test/2),
   seed=123
)
test = convert_to_df(dataset, n_total=n_test)
test.columns = [f"x{i}" for i in range(1, 11)] + [target_column]
```

```
def plot_data_with_drift_points(data, target_column, n_train, title=""):
    indices = range(len(data))
    y_values = data[target_column]

    plt.figure(figsize=(10, 6))
    plt.plot(indices, y_values, label="y Value", color='blue')

    drift_points = [n_train / 4, n_train / 2]
    for dp in drift_points:
        plt.axvline(x=dp, color='red', linestyle='--', label=f'Drift Point at {int(dp)}')

    handles, labels = plt.gca().get_legend_handles_labels()
    by_label = dict(zip(labels, handles))
    plt.legend(by_label.values(), by_label.keys())

    plt.xlabel('Index')
    plt.ylabel('Target Value (y)')
    plt.title(title)
    plt.grid(True)
    plt.show()
```

```
plot_data_with_drift_points(train, target_column, n_train, title="Training Data with Drift Points")
```

```
plot_data_with_drift_points(test, target_column, n_train, title="Testing Data with Dri
```

E.2.2. The Friedman Drift Data Set from `spotpython`

A data generator for the Friedman Drift dataset is implemented in the `spotpython` package, see friedman.py. The `spotpython` version is a simplified version of the `river` implementation. The `spotPyton` version allows the generation of constant input values for the features. This is useful for visualizing the concept drifts. For the productive use the `river version` should be used.

Plotting the first 100 samples of the Friedman Drift dataset, we can not see the concept drifts at p_1 and p_2. Drift can be visualized by plotting the target values over time for constant features, e,g, if x_0 is set to 1 and all other features are set to 0. This is illustrated in the following plot.

```
from spotpython.data.friedman import FriedmanDriftDataset

def plot_friedman_drift_data(n_samples, seed, change_point1, change_point2, constant=1
    data_generator = FriedmanDriftDataset(n_samples=n_samples, seed=seed, change_point
    data = [data for data in data_generator]
    indices = [i for _, _, i in data]
    values = {f"x{i}": [] for i in range(5)}
```

```
    values["y"] = []
    for x, y, _ in data:
        for i in range(5):
            values[f"x{i}"].append(x[i])
        values["y"].append(y)

    plt.figure(figsize=(10, 6))
    for label, series in values.items():
        plt.plot(indices, series, label=label)
    plt.xlabel('Index')
    plt.ylabel('Value')
    plt.axvline(x=change_point1, color='k', linestyle='--', label='Drift Point 1')
    plt.axvline(x=change_point2, color='r', linestyle='--', label='Drift Point 2')
    plt.legend()
    plt.grid(True)
    plt.show()

plot_friedman_drift_data(n_samples=100, seed=42, change_point1=50, change_point2=75, constant=False)
plot_friedman_drift_data(n_samples=100, seed=42, change_point1=50, change_point2=75, constant=True)
```

E. Datasets

17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Value
0
20
40
60
80
100
Index
x0
x1
x2
x3
x4
y
Drift Point 1
Drift Point 2

F. Using Slurm

F.1. Introduction

This chapter describes how to generate a `spotpython` configuration on a local machine and run the `spotpython` code on a remote machine using Slurm. We recommend using a jupyter notebook (`*.ipynb`) or a Quarto document (`*.qmd`) on the local machine to generate the configuration and analyze the results.

F.2. Packages important for this Chapter

```
import argparse
import pickle
from math import inf
import torch
from spotpython.utils.file import load_result, load_and_run_spot_python_experiment
from spotpython.data.manydataset import ManyToManyDataset
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_control_init)
from spotpython.spot import Spot
from spotpython.hyperparameters.values import set_hyperparameter, get_tuned_architecture
from torch.utils.data import TensorDataset
from spotpython.utils.eda import print_res_table
```

F.3. Prepare the Slurm Scripts for Runs on the Remote Machine

Two scripts are required to run the `spotpython` code on the remote machine:

- `startSlurm.sh` and
- `startPython.py`.

They should be saved in the same directory on the remote machine as the pickle-configuration (`pkl`) file. These two scripts must be generated only once and can be reused for different configurations. For convenience, the scripts are available as templates here:

- startSlurm.sh
- startPython.py

The `startSlurm.sh` script is a shell script that contains the following code:

```
#!/bin/bash

### Vergabe von Ressourcen
#SBATCH --job-name=Test
#SBATCH --account=Accountname/Projektname  # Hier den gewünschten Account angeben
#SBATCH --cpus-per-task=20
#SBATCH --gres=gpu:1
#SBATCH --time=48:00:00
#SBATCH --error=job.%J.err
#SBATCH --output=job.%J.out
#----
#SBATCH --partition=gpu

if [ -z "$1" ]; then
    echo "Usage: $0 <path_to_spot.pkl>"
    exit 1
fi

SPOT_PKL=$1

module load conda

### change to your conda environment with spotpython installed via
### pip install spotpython
conda activate spot312

python startPython.py "$SPOT_PKL"

exit
```

Save the code in a file named `startSlurm.sh` and copy the file to the remote machine via `scp`, i.e.,

```
scp startSlurm.sh user@144.33.22.1:
```

The startPython.py script is a Python script that contains the following code:

```
import argparse
import pickle
from spotpython.utils.file import load_and_run_spot_python_experiment
from spotpython.data.manydataset import ManyToManyDataset

# Uncomment the following if you want to use a custom model (python source code)
# import sys
# sys.path.insert(0, './userModel')
# import my_regressor
# import my_hyper_dict


def main(pickle_file):
    spot_tuner = load_and_run_spot_python_experiment(filename=pickle_file)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Process a pickle file.')
    parser.add_argument('pickle_file', type=str, help='The path to the pickle file to be processed.'

    args = parser.parse_args()
    main(args.pickle_file)
```

Save the code in a file named startPython.py and copy the file to the remote machine via scp, i.e.,

```
scp startPython.py user@144.33.22.1:
```

F.4. Generate a spotpython Configuration

The configuration can be generated on a local machine using the following command:

```
# generate data
num_samples = 10_000
input_dim = 10
X = torch.randn(num_samples, input_dim)  # random data for example
Y = torch.randn(num_samples, 1)  # random target for example
data_set = TensorDataset(X, Y)

PREFIX="a06"
```

```
fun_control = fun_control_init(
    accelerator="gpu",
    devices="auto",
    num_nodes=1,
    num_workers=19,
    precision="32",
    strategy="auto",
    save_experiment=True,
    PREFIX=PREFIX,
    fun_evals=10,
    max_time=inf,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=input_dim,
    _L_out=1)

fun = HyperLight().fun

set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [2,3])
set_hyperparameter(fun_control, "epochs", [2,3])
set_hyperparameter(fun_control, "batch_size", [2,5])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,4])

design_control = design_control_init(init_size=5)

S = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
```

The configuration is saved as a pickle-file that contains the full information. In our example, the filename is `a06_exp.pkl`.

> ⚠ Note on `save_experiment`
>
> The `fun_control` dictionary must be initialized with `save_experiment=True` to save the experiment/design configuration.

F.5. Copy the Configuration to the Remote Machine

You can copy the configuration to the remote machine using the `scp` command. The following command copies the configuration to the remote machine `144.33.22.1`:

```
scp a06_exp.pkl user@144.33.22.1:
```

F.6. Run the `spotpython` Code on the Remote Machine

Login on the remote machine and run the following command to start the `spotpython` code:

```
ssh user@144.33.22.1
# change this to your conda environment!
conda activate spot312
sbatch ./startSlurm.sh a06_exp.pkl
```

F.7. Copy the Results to the Local Machine

After the `spotpython` code has finished, you can copy the results back to the local machine using the `scp` command. The following command copies the results to the local machine:

```
scp user@144.33.22.1:a06_res.pkl .
```

> **Experiment and Result Files**
>
> - `spotpython` generates two files:
> - `PREFIX_exp.pkl` (experiment file), which stores the information about running the experiment, and
> - `PREFIX_res.pkl` (result file), which stores the results of the experiment.

F.8. Analyze the Results on the Local Machine

The file `a06_res.pkl` contains the results of the `spotpython` code. You can analyze the results on the local machine using the following code. Note: `PREFIX` is the same as in the previous steps, i.e., `"a06"`.

```
spot_tuner = load_result(PREFIX)
```

F.8.1. Visualizing the Tuning Progress

Now the `spot_tuner` object is loaded and you can analyze the results interactively.

```
spot_tuner.plot_progress(log_y=True, filename=None)
```

F.8.2. Design Table with Default and Tuned Hyperparameters

```
print_res_table(spot_tuner)
```

F.8.3. Plotting Important Hyperparameters

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

F.8.4. The Tuned Hyperparameters

```
get_tuned_architecture(spot_tuner)
```

F.9. Slurm Command Reference

Table F.1 summarizes commands used to manage jobs on a remote machine using Slurm.

Table F.1.: Slurm and related commands

Command	Description
`sbatch startSlurm.sh a06_exp.pkl`	Submit a job to the Slurm scheduler. The job will run the `startSlurm.sh` script with the argument `a06_exp.pkl`.
`squeue -u username`	Check the status of your jobs in the queue. Replace `username` with your actual username.

Command	Description
`scancel job_id`	Cancel a job. Replace `job_id` with the actual job ID you want to cancel.
`ssh user@remote_host`	Log in to a remote machine. Replace `user` with your username and `remote_host` with the hostname or IP address of the remote machine.
`scp source_file user@remote_host:destination_path`	Copy a file to a remote machine. Replace `source_file` with the path to the file you want to copy, `user` with your username, `remote_host` with the hostname or IP address of the remote machine, and `destination_path` with the path where you want to copy the file on the remote machine.
`module load conda`	Load the Conda module on the remote machine. This command may vary depending on the system configuration.
`conda activate env_name`	Activate a Conda environment. Replace `env_name` with the name of your Conda environment.

F.10. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

G. Python Package Building

Introduction

This notebook will guide you through the process of creating a Python package.

- All examples can be found in the `userPackage` directory, see: userPackage

G.1. Create a Conda Environment

- `conda create -n userpackage python=3.12`
- `conda activate userpackage`
- Install the following packages:

```
python -m pip install build flake8 black mkdocs mkdocs-gen-files mkdocs-literate-nav mkdocs-section-
```

G.2. Download the User Package

The user package can be found in the `userPackage` directory, see: userPackage

G.3. Build the User Package

- cd into the `userPackage` directory and run the following command:
  - `./makefile.sh`
  - Alternatively, you can run the following commands:
    - `rm -f dist/userpackage*; python -m build; python -m pip install dist/userpackage*.tar.gz`
    - `python -m mkdocs build`

G.4. Open the Documentation of the User Package

- `mkdocs serve` to view the documentation

H. Parallelism in Initial Design

In spotpython, we provide a wrapper function, that encapsulates the objective function to enable its parallel execution via multiprocessing or joblib, allowing multiple configurations to be evaluated at the same time.

H.1. Setup

To demonstrate the performance gain enabled by parallelization, we use a similar example to that in Section 47, where we perform hyperparameter tuning with spotpythonand PyTorch Lightning on the Diabetes dataset using a ResNet model. We compare the time required with and without parallelization. First, we import the necessary libraries, including the wrapper function make_parallel. We then define the fun_control and design_control settings. For design_control, we deliberately choose an initial design size of 10 for demonstration purposes.

```
import time
from math import inf
import multiprocessing
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, design_control_init)
from spotpython.hyperparameters.values import set_hyperparameter
from spotpython.spot import Spot
from spotpython.utils.parallel import make_parallel
```

```
dataset = Diabetes()
fun_control = fun_control_init(
    fun_evals=10,
    max_time=inf,
    data_set=dataset,
    core_model_name="light.regression.NNResNetRegressor",
```

```
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1,
    seed=125,
    tensorboard_log=False,
    TENSORBOARD_CLEAN=False,
)
set_hyperparameter(fun_control, "optimizer", ["Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [2, 5])
set_hyperparameter(fun_control, "epochs", [5, 8])
set_hyperparameter(fun_control, "batch_size", [5, 8])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.5])
set_hyperparameter(fun_control, "patience", [2, 3])
set_hyperparameter(fun_control, "lr_mult", [0.1, 10.0])

design_control = design_control_init(
    init_size=10
)

fun = HyperLight().fun
```

```
module_name: light
submodule_name: regression
model_name: NNResNetRegressor
```

H.2. Experiments

We now measure the time required for sequential and parallel evaluation, beginning with the sequential approach.

H.2.1. Sequential Execution

```
tic = time.perf_counter()
spot_tuner = Spot(fun=fun, fun_control=fun_control, design_control=design_control)
res = spot_tuner.run()
toc = time.perf_counter()
time_seq = toc - tic
print(f"Time taken for sequential execution: {time_seq:.2f} seconds")
```

```
    Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

 0  layers   Sequential    5.6 K   train   2.2 M   [256, 10]    [256, 1]



Trainable params: 5.6 K
Non-trainable params: 0
Total params: 5.6 K
Total estimated model params size (MB): 0
Modules in train mode: 139
Modules in eval mode: 0
Total FLOPs: 2.2 M

train_model result: {'val_loss': 23673.822265625, 'hp_metric': 23673.822265625}



    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential      637   train   95.7 K   [128, 10]    [128, 1]



Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 95.7 K

train_model result: {'val_loss': 21004.400390625, 'hp_metric': 21004.400390625}



    Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

 0  layers   Sequential      225   train   32.3 K   [128, 10]    [128, 1]
```

```
Trainable params: 225
Non-trainable params: 0
Total params: 225
Total estimated model params size (MB): 0
Modules in train mode: 34
Modules in eval mode: 0
Total FLOPs: 32.3 K

train_model result: {'val_loss': 22744.083984375, 'hp_metric': 22744.083984375}



     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   77.7 K   [32, 10]     [32, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 77.7 K

train_model result: {'val_loss': 21952.509765625, 'hp_metric': 21952.509765625}



     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]



Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K
```

```
train_model result: {'val_loss': 24014.548828125, 'hp_metric': 24014.548828125}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential      637   train   47.9 K   [64, 10]     [64, 1]


Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 47.9 K

train_model result: {'val_loss': 18914.611328125, 'hp_metric': 18914.611328125}


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      225   train   32.3 K   [128, 10]    [128, 1]


Trainable params: 225
Non-trainable params: 0
Total params: 225
Total estimated model params size (MB): 0
Modules in train mode: 34
Modules in eval mode: 0
Total FLOPs: 32.3 K

train_model result: {'val_loss': 21966.66015625, 'hp_metric': 21966.66015625}


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential      637   train   47.9 K   [64, 10]     [64, 1]
```

```
Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 47.9 K

train_model result: {'val_loss': 23201.72265625, 'hp_metric': 23201.72265625}


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    5.6 K   train   1.1 M   [128, 10]    [128, 1]


Trainable params: 5.6 K
Non-trainable params: 0
Total params: 5.6 K
Total estimated model params size (MB): 0
Modules in train mode: 139
Modules in eval mode: 0
Total FLOPs: 1.1 M

train_model result: {'val_loss': 21864.96875, 'hp_metric': 21864.96875}


     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]


Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

train_model result: {'val_loss': 23692.11328125, 'hp_metric': 23692.11328125}
Experiment saved to 000_res.pkl
Time taken for sequential execution: 86.14 seconds
```

H.2.2. Parallel Execution

To use `make_parallel`, the number of cores must be specified via the `num_cores` parameter. By default, the function utilizes `multiprocessing`, but other parallelization methods can be selected using the `method` argument. The following two lines of code demonstrate how to set up the parallel function and run the `Spot` tuner with it.

- `parallel_fun = make_parallel(fun, num_cores=num_cores)`
- `spot_parallel_tuner = Spot(fun=parallel_fun, fun_control=fun_control, design_control=design_control)`

We consider parallel efficiency, a metric that measures how effectively additional computational resources (cores/processors) are being utilized in a parallel computation. It's calculated as:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Processors}},$$

where:

- Speedup = Time(Sequential) / Time(Parallel)
- Number of Processors = Number of cores used

It can be interpreted as follows:

- 1.0 (100%): Perfect linear scaling - doubling cores halves execution time
- 0.8-0.9 (80-90%): Excellent scaling - minimal parallelization overhead
- 0.5-0.7 (50-70%): Good scaling - reasonable utilization of additional cores
- <0.5 (<50%): Poor scaling - diminishing returns from adding more cores

When efficiency drops significantly as you add cores, it indicates:

- Communication overhead increasing
- Synchronization bottlenecks
- Load imbalance between cores
- Portions of code that remain sequential (Amdahl's Law limitation)

```
# Get available cores
available_cores = multiprocessing.cpu_count()
print(f"Available cores: {available_cores}")

# Generate list of cores to test (powers of 2 up to available cores)
cores_to_test = []
power = 0
while 2**(power+1) < available_cores:
    cores_to_test.append(2**power)
    power += 1

# If the number of available cores is not a power of 2, add it to the list
```

```
if available_cores not in cores_to_test:
    cores_to_test.append(available_cores)

# Prepare DataFrame to store results
results_df = pd.DataFrame(columns=["number_of_cores", "time"])

# Run the experiment for each core count
for num_cores in cores_to_test:
    print(f"\nTesting with {num_cores} cores...")
    tic = time.perf_counter()
    parallel_fun = make_parallel(fun, num_cores=num_cores)
    spot_parallel_tuner = Spot(fun=parallel_fun, fun_control=fun_control, design_contr
    res = spot_parallel_tuner.run()
    toc = time.perf_counter()
    time_taken = toc - tic

    # Add result to DataFrame
    results_df = pd.concat([results_df, pd.DataFrame({
        "number_of_cores": [num_cores],
        "time": [time_taken]
    })], ignore_index=True)

    print(f"Time taken with {num_cores} cores: {time_taken:.2f} seconds")
```

```
Available cores: 24

Testing with 1 cores...


     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    5.6 K   train   2.2 M   [256, 10]    [256, 1]

Trainable params: 5.6 K
Non-trainable params: 0
Total params: 5.6 K
Total estimated model params size (MB): 0
Modules in train mode: 139
Modules in eval mode: 0
Total FLOPs: 2.2 M


train_model result: {'val_loss': 23673.822265625, 'hp_metric': 23673.822265625}
```

```
     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   95.7 K   [128, 10]    [128, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 95.7 K

train_model result: {'val_loss': 22347.912109375, 'hp_metric': 22347.912109375}

     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      225   train   32.3 K   [128, 10]    [128, 1]

Trainable params: 225
Non-trainable params: 0
Total params: 225
Total estimated model params size (MB): 0
Modules in train mode: 34
Modules in eval mode: 0
Total FLOPs: 32.3 K

train_model result: {'val_loss': 21897.943359375, 'hp_metric': 21897.943359375}

     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   77.7 K   [32, 10]     [32, 1]

Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 77.7 K

train_model result: {'val_loss': 21803.751953125, 'hp_metric': 21803.751953125}

     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes
```

```
  0   layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]

Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

train_model result: {'val_loss': 23591.19140625, 'hp_metric': 23591.19140625}

     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0   layers   Sequential      637   train   47.9 K   [64, 10]     [64, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 47.9 K

train_model result: {'val_loss': 17341.458984375, 'hp_metric': 17341.458984375}

     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0   layers   Sequential      225   train   32.3 K   [128, 10]    [128, 1]

Trainable params: 225
Non-trainable params: 0
Total params: 225
Total estimated model params size (MB): 0
Modules in train mode: 34
Modules in eval mode: 0
Total FLOPs: 32.3 K

train_model result: {'val_loss': 22432.20703125, 'hp_metric': 22432.20703125}

     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0   layers   Sequential      637   train   47.9 K   [64, 10]     [64, 1]
```

```
Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 47.9 K

train_model result: {'val_loss': 23983.16015625, 'hp_metric': 23983.16015625}

     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    5.6 K   train   1.1 M   [128, 10]    [128, 1]

Trainable params: 5.6 K
Non-trainable params: 0
Total params: 5.6 K
Total estimated model params size (MB): 0
Modules in train mode: 139
Modules in eval mode: 0
Total FLOPs: 1.1 M

train_model result: {'val_loss': 23051.548828125, 'hp_metric': 23051.548828125}

     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]

Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K
train_model result: {'val_loss': 23849.2734375, 'hp_metric': 23849.2734375}
Experiment saved to 000_res.pkl
Time taken with 1 cores: 93.36 seconds

Testing with 2 cores...


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes
```

```
  0   layers   Sequential     225   train   32.3 K   [128, 10]    [128, 1]

Trainable params: 225
Non-trainable params: 0
Total params: 225
Total estimated model params size (MB): 0
Modules in train mode: 34
Modules in eval mode: 0
Total FLOPs: 32.3 K

     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0   layers   Sequential    5.6 K   train   2.2 M   [256, 10]    [256, 1]

Trainable params: 5.6 K
Non-trainable params: 0
Total params: 5.6 K
Total estimated model params size (MB): 0
Modules in train mode: 139
Modules in eval mode: 0
Total FLOPs: 2.2 M


train_model result: {'val_loss': 23673.822265625, 'hp_metric': 23673.822265625}

     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0   layers   Sequential     637   train   95.7 K   [128, 10]    [128, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 95.7 K


train_model result: {'val_loss': 21897.943359375, 'hp_metric': 21897.943359375}

     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0   layers   Sequential    1.8 K   train   77.7 K   [32, 10]     [32, 1]

Trainable params: 1.8 K
Non-trainable params: 0
```

```
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 77.7 K

train_model result: {'val_loss': 22347.912109375, 'hp_metric': 22347.912109375}

     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]

Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

train_model result: {'val_loss': 23591.19140625, 'hp_metric': 23591.19140625}

     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential      637   train   47.9 K   [64, 10]     [64, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 47.9 K

train_model result: {'val_loss': 17341.458984375, 'hp_metric': 17341.458984375}

     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      225   train   32.3 K   [128, 10]    [128, 1]

Trainable params: 225
Non-trainable params: 0
Total params: 225
Total estimated model params size (MB): 0
```

```
Modules in train mode: 34
Modules in eval mode: 0
Total FLOPs: 32.3 K

train_model result: {'val_loss': 22432.20703125, 'hp_metric': 22432.20703125}

     Name    Type        Params   Mode    FLOPs   In sizes   Out sizes

  0  layers  Sequential     637   train  47.9 K   [64, 10]    [64, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 47.9 K

train_model result: {'val_loss': 23983.16015625, 'hp_metric': 23983.16015625}

     Name    Type        Params   Mode   FLOPs    In sizes   Out sizes

  0  layers  Sequential   5.6 K   train  1.1 M   [128, 10]   [128, 1]

Trainable params: 5.6 K
Non-trainable params: 0
Total params: 5.6 K
Total estimated model params size (MB): 0
Modules in train mode: 139
Modules in eval mode: 0
Total FLOPs: 1.1 M

train_model result: {'val_loss': 23051.548828125, 'hp_metric': 23051.548828125}

     Name    Type        Params   Mode   FLOPs   In sizes   Out sizes

  0  layers  Sequential   1.8 K   train  155 K   [64, 10]    [64, 1]

Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K
```

```
train_model result: {'val_loss': 21803.751953125, 'hp_metric': 21803.751953125}
train_model result: {'val_loss': 23849.2734375, 'hp_metric': 23849.2734375}
Experiment saved to 000_res.pkl
Time taken with 2 cores: 61.07 seconds

Testing with 4 cores...


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      225   train   32.3 K   [128, 10]    [128, 1]

Trainable params: 225
Non-trainable params: 0
Total params: 225
Total estimated model params size (MB): 0
Modules in train mode: 34
Modules in eval mode: 0
Total FLOPs: 32.3 K

     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   95.7 K   [128, 10]    [128, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 95.7 K

     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   77.7 K   [32, 10]     [32, 1]

Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 77.7 K

     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes
```

```
  0   layers   Sequential    5.6 K   train   2.2 M   [256, 10]    [256, 1]

Trainable params: 5.6 K
Non-trainable params: 0
Total params: 5.6 K
Total estimated model params size (MB): 0
Modules in train mode: 139
Modules in eval mode: 0
Total FLOPs: 2.2 M

train_model result: {'val_loss': 23673.822265625, 'hp_metric': 23673.822265625}

     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0   layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]

Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

train_model result: {'val_loss': 21897.943359375, 'hp_metric': 21897.943359375}

     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0   layers   Sequential     637   train   47.9 K   [64, 10]     [64, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 47.9 K

train_model result: {'val_loss': 22347.912109375, 'hp_metric': 22347.912109375}

     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0   layers   Sequential     225   train   32.3 K   [128, 10]    [128, 1]
```

```
Trainable params: 225
Non-trainable params: 0
Total params: 225
Total estimated model params size (MB): 0
Modules in train mode: 34
Modules in eval mode: 0
Total FLOPs: 32.3 K

train_model result: {'val_loss': 23591.19140625, 'hp_metric': 23591.19140625}

     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential      637   train   47.9 K   [64, 10]     [64, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 47.9 K

train_model result: {'val_loss': 22432.20703125, 'hp_metric': 22432.20703125}

     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    5.6 K   train   1.1 M   [128, 10]    [128, 1]

Trainable params: 5.6 K
Non-trainable params: 0
Total params: 5.6 K
Total estimated model params size (MB): 0
Modules in train mode: 139
Modules in eval mode: 0
Total FLOPs: 1.1 M

train_model result: {'val_loss': 17341.458984375, 'hp_metric': 17341.458984375}

     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]

Trainable params: 1.8 K
```

```
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K


train_model result: {'val_loss': 23849.2734375, 'hp_metric': 23849.2734375}
train_model result: {'val_loss': 23051.548828125, 'hp_metric': 23051.548828125}
train_model result: {'val_loss': 23983.16015625, 'hp_metric': 23983.16015625}
train_model result: {'val_loss': 21803.751953125, 'hp_metric': 21803.751953125}
Experiment saved to 000_res.pkl
Time taken with 4 cores: 55.63 seconds

Testing with 8 cores...


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      225   train   32.3 K   [128, 10]    [128, 1]

Trainable params: 225
Non-trainable params: 0
Total params: 225
Total estimated model params size (MB): 0
Modules in train mode: 34
Modules in eval mode: 0
Total FLOPs: 32.3 K

     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential      637   train   47.9 K   [64, 10]     [64, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 47.9 K

     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   77.7 K   [32, 10]     [32, 1]
```

```
Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 77.7 K


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      225   train   32.3 K   [128, 10]    [128, 1]

Trainable params: 225
Non-trainable params: 0
Total params: 225
Total estimated model params size (MB): 0
Modules in train mode: 34
Modules in eval mode: 0
Total FLOPs: 32.3 K

     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   95.7 K   [128, 10]    [128, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 95.7 K

     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    5.6 K   train   2.2 M   [256, 10]    [256, 1]


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential      637   train   47.9 K   [64, 10]     [64, 1]

Trainable params: 5.6 K
Non-trainable params: 0
```

```
Total params: 5.6 K
Total estimated model params size (MB): 0
Modules in train mode: 139
Modules in eval mode: 0
Total FLOPs: 2.2 M
Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 47.9 K

     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]

Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K


train_model result: {'val_loss': 22432.20703125, 'hp_metric': 22432.20703125}

     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    5.6 K   train   1.1 M   [128, 10]    [128, 1]

Trainable params: 5.6 K
Non-trainable params: 0
Total params: 5.6 K
Total estimated model params size (MB): 0
Modules in train mode: 139
Modules in eval mode: 0
Total FLOPs: 1.1 M


train_model result: {'val_loss': 23673.822265625, 'hp_metric': 23673.822265625}

     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]
```

```
Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

train_model result: {'val_loss': 21897.943359375, 'hp_metric': 21897.943359375}
train_model result: {'val_loss': 23591.19140625, 'hp_metric': 23591.19140625}
train_model result: {'val_loss': 23849.2734375, 'hp_metric': 23849.2734375}
train_model result: {'val_loss': 22347.912109375, 'hp_metric': 22347.912109375}
train_model result: {'val_loss': 17341.458984375, 'hp_metric': 17341.458984375}
train_model result: {'val_loss': 23051.548828125, 'hp_metric': 23051.548828125}
train_model result: {'val_loss': 23983.16015625, 'hp_metric': 23983.16015625}
train_model result: {'val_loss': 21803.751953125, 'hp_metric': 21803.751953125}
Experiment saved to 000_res.pkl
Time taken with 8 cores: 55.96 seconds

Testing with 24 cores...


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      225   train   32.3 K   [128, 10]    [128, 1]

Trainable params: 225
Non-trainable params: 0
Total params: 225
Total estimated model params size (MB): 0
Modules in train mode: 34
Modules in eval mode: 0
Total FLOPs: 32.3 K

     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential      637   train   47.9 K   [64, 10]     [64, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
```

```
Modules in eval mode: 0
Total FLOPs: 47.9 K


     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      637   train   95.7 K   [128, 10]    [128, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 95.7 K

     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]

Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K

     Name     Type         Params   Mode     FLOPs    In sizes   Out sizes

  0  layers   Sequential      225   train   32.3 K   [128, 10]    [128, 1]

Trainable params: 225
Non-trainable params: 0
Total params: 225
Total estimated model params size (MB): 0
Modules in train mode: 34
Modules in eval mode: 0
Total FLOPs: 32.3 K

     Name     Type         Params   Mode    FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   155 K   [64, 10]     [64, 1]

Trainable params: 1.8 K
```

```
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 155 K


     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential    1.8 K   train   77.7 K   [32, 10]     [32, 1]

Trainable params: 1.8 K
Non-trainable params: 0
Total params: 1.8 K
Total estimated model params size (MB): 0
Modules in train mode: 101
Modules in eval mode: 0
Total FLOPs: 77.7 K

     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes

  0  layers   Sequential    5.6 K   train   2.2 M   [256, 10]    [256, 1]

Trainable params: 5.6 K
Non-trainable params: 0
Total params: 5.6 K
Total estimated model params size (MB): 0
Modules in train mode: 139
Modules in eval mode: 0
Total FLOPs: 2.2 M

     Name     Type         Params   Mode     FLOPs   In sizes   Out sizes

  0  layers   Sequential      637   train   47.9 K   [64, 10]     [64, 1]

Trainable params: 637
Non-trainable params: 0
Total params: 637
Total estimated model params size (MB): 0
Modules in train mode: 63
Modules in eval mode: 0
Total FLOPs: 47.9 K

     Name     Type         Params   Mode    FLOPs    In sizes   Out sizes
```

```
  0   layers   Sequential    5.6 K   train   1.1 M   [128, 10]    [128, 1]

Trainable params: 5.6 K
Non-trainable params: 0
Total params: 5.6 K
Total estimated model params size (MB): 0
Modules in train mode: 139
Modules in eval mode: 0
Total FLOPs: 1.1 M


train_model result: {'val_loss': 23849.2734375, 'hp_metric': 23849.2734375}
train_model result: {'val_loss': 22432.20703125, 'hp_metric': 22432.20703125}
train_model result: {'val_loss': 21897.943359375, 'hp_metric': 21897.943359375}
train_model result: {'val_loss': 23673.822265625, 'hp_metric': 23673.822265625}
train_model result: {'val_loss': 23591.19140625, 'hp_metric': 23591.19140625}
train_model result: {'val_loss': 23051.548828125, 'hp_metric': 23051.548828125}
train_model result: {'val_loss': 17341.458984375, 'hp_metric': 17341.458984375}
train_model result: {'val_loss': 22347.912109375, 'hp_metric': 22347.912109375}
train_model result: {'val_loss': 23983.16015625, 'hp_metric': 23983.16015625}
train_model result: {'val_loss': 21803.751953125, 'hp_metric': 21803.751953125}
Experiment saved to 000_res.pkl
Time taken with 24 cores: 57.17 seconds
```

H.2.3. Results

```
print("\nPerformance comparison across different numbers of cores:")
results_df["speedup_vs_sequential"] = time_seq / results_df["time"]
results_df["efficiency"] = results_df["speedup_vs_sequential"] / results_df["number_of
print(results_df)
```

```
Performance comparison across different numbers of cores:
  number_of_cores       time  speedup_vs_sequential efficiency
0               1  93.360117               0.922717   0.922717
1               2  61.069113               1.410614   0.705307
2               4  55.633875               1.548426   0.387107
3               8  55.961298               1.539366   0.192421
4              24  57.170689               1.506803   0.062783
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Execution time vs number of cores
ax1.plot(results_df["number_of_cores"], results_df["time"], marker='o', linestyle='-')
ax1.set_xlabel("Number of cores")
ax1.set_ylabel("Execution time (seconds)")
ax1.set_title("Execution Time vs Number of Cores")
ax1.grid(True)

# Speedup vs number of cores
ax1.axhline(y=time_seq, color='r', linestyle='--', label=f'Sequential ({time_seq:.2f}s)')
ax1.legend()

# Parallel efficiency
ax2.plot(results_df["number_of_cores"], results_df["efficiency"], marker='o', linestyle='-')
ax2.set_xlabel("Number of cores")
ax2.set_ylabel("Parallel efficiency")
ax2.set_title("Parallel Efficiency vs Number of Cores")
ax2.set_ylim(0, 1.1)
ax2.grid(True)

plt.tight_layout()
plt.show()
```

Operating-system Differences in Parallelization Methods

Linux uses the `fork` method by default to start new processes, whereas macOS and Windows use the `spawn` method. This leads to differences in how processes are handled across operating systems. We use the functionality of `set_all_seeds`

to ensure that the evaluation remains reproducible across all operating systems.

I. Solutions to Selected Exercises

> ⚠ Warning
>
> - Solutions are incomplete and need to be corrected!
> - They serve as a starting point for the final solution.

I.1. Data-Driven Modeling and Optimization

I.1.1. Histograms

Solution I.1 (Density Curve).

- We can calculate propabilities.
- We only need two parameters (the mean and the sd) to form the curve -> Store data more efficently
- Blanks can be filled

I.1.2. The Normal Distribution

Solution I.2 (TwoSDAnswer). 95%

Solution I.3 (OneSDAnswer). 68%

Solution I.4 (ThreeSDAnswer). 99,7%

Solution I.5 (DataRangeAnswer). 80 - 120

Solution I.6 (PeakHeightAnswer). low

I.1.3. The mean, the media, and the mode

I.1.4. The exponential distribution

I.1.5. Population and Estimated Parameters

Solution I.7 (ProbabilityAnswer). 50%

I.1.6. Calculating the Mean, Variance and Standard Deviation

Solution I.8 (MeanDifferenceAnswer). If we have all the data, μ is the population mean and x-bar is the sample mean. We don't have the full information.

Solution I.9 (EstimateMeanAnswer). Sum of the values divided by n.

Solution I.10 (SigmaSquaredAnswer). Variance

Solution I.11 (EstimatedSDAnswer). The same as the normal standard deviation, but using n-1.

Solution I.12 (VarianceDifferenceAnswer). n and $n-1$

Solution I.13 (ModelBenefitsAnswer).

- Approximation
- Prediction
- Understanding

Solution I.14 (SampleDefinitionAnswer). It's a subset of the data.

I.1.7. Hypothesis Testing and the Null-Hypothesis

Solution I.15 (RejectHypothesisAnswer). It means the evidence supports the alternative hypothesis, indicating that the null hypothesis is unlikely to be true.

Solution I.16 (NullHypothesisAnswer). It's a statement that there is no effect or no difference, and it serves as the default or starting assumption in hypothesis testing.

Solution I.17 (BetterDrugAnswer). By conducting experiments and statistical tests to compare the new drug's effectiveness against the current standard and demonstrating a significant improvement.

I.1.8. Alternative Hypotheses, Main Ideas

I.1.9. p-values: What they are and how to interpret them

Solution I.18 (PValueIntroductionAnswer). We can reject the null hypothesis. We can make a decision.

Solution I.19 (PValueRangeAnswer). It can only be between 0 and 1.

Solution I.20 (PValueRangeAnswer). It can only be between 0 and 1.

Solution I.21 (TypicalPValueAnswer). The chance that we wrongly reject the null hypothesis.

Solution I.22 (FalsePositiveAnswer). If we have a false-positive, we succeed in rejecting the null hypothesis. But in fact/reality, this is false -> False positive.

I.1.10. How to calculate p-values

Solution I.23 (CalculatePValueAnswer). Probability of specific result, probability of outcome with the same probability, and probability of events with smaller probability.

Solution I.24 (SDCalculationAnswer). 7 is the SD.

Solution I.25 (SidedPValueAnswer). If we are not interested in the direction of the change, we use the two-sided. If we want to know about the direction, the one-sided.

Solution I.26 (CoinTestAnswer). TBD

Solution I.27 (BorderPValueAnswer). TBD

Solution I.28 (OneSidedPValueCautionAnswer). If you look in the wrong direction, there is no change.

Solution I.29 (BinomialDistributionAnswer). TBD

I.1.11. p-hacking: What it is and how to avoid it

Solution I.30 (PHackingWaysAnswer).

- Performing repeats until you find one result with a small p-value -> false positive result.
- Increasing the sample size within one experiment when it is close to the threshold.

Solution I.31 (AvoidPHackingAnswer). Specify the number of repeats and the sample sizes at the beginning.

Solution I.32 (MultipleTestingProblemAnswer). TBD

I.1.12. Covariance

Solution I.33 (CovarianceDefinitionAnswer). Formula

Solution I.34 (CovarianceMeaningAnswer). Large values in the first variable result in large values in the second variable.

Solution I.35 (CovarianceVarianceRelationshipAnswer). Formula

Solution I.36 (HighCovarianceAnswer). No, size doesn't matter.

Solution I.37 (ZeroCovarianceAnswer). No relationship

Solution I.38 (NegativeCovarianceAnswer). Yes

Solution I.39 (NegativeVarianceAnswer). No

I.1.13. Pearson's Correlation

Solution I.40 (CorrelationValueAnswer). Recalculate

Solution I.41 (CorrelationRangeAnswer). From -1 to 1

Solution I.42 (CorrelationFormulaAnswer). Formula

I.1.14. Boxplots

Solution I.43 (UnderstandingStatisticalPower). It is the probability of correctly rejecting the null hypothesis.

Solution I.44 (DistributionEffectOnPower). Power analysis is not applicable.

Solution I.45 (IncreasingPower). By taking more samples.

Solution I.46 (PreventingPHacking). TBD

Solution I.47 (SampleSizeAndPower). The power will be low.

I.1.15. Power Analysis

Solution I.48 (MainFactorsAffectingPower). The overlap (distance of the two means) and sample sizes.

Solution I.49 (PowerAnalysisOutcome). The sample size needed.

Solution I.50 (RisksInExperiments). Few experiments lead to very low power, and many experiments might result in p-hacking.

Solution I.51 (StepsToPerformPowerAnalysis).

1. Select power
2. Select threshold for significance (alpha)
3. Estimate the overlap (done by the effect size)

I.1.16. The Central Limit Theorem

Solution I.52 (CentralLimitTheoremAnswer). TBD

I.1.17. Boxplots

Solution I.53 (MedianAnswer). The median.

Solution I.54 (BoxContentAnswer). 50% of the data.

I.1.18. R-squared

Solution I.55 (RSquaredFormulaAnswer). TBD

Solution I.56 (NegativeRSquaredAnswer). If you fit a line, no, but there are cases where it could be negative. However, these are usually considered useless.

Solution I.57 (RSquaredCalculationAnswer). TBD

I.1.18.1. The main ideas of fitting a line to data (The main ideas of least squares and linear regression.)

Solution I.58 (LeastSquaresAnswer). It is the calculation of the smallest sum of residuals when you fit a model to data.

I.1.19. Linear Regression

I.1.20. Multiple Regression

I.1.21. A Gentle Introduction to Machine Learning

Solution I.59 (RegressionVsClassificationAnswer). Regression involves predicting continuous values (e.g., temperature, size), while classification involves predicting discrete values (e.g., categories like cat, dog).

I.1.22. Maximum Likelihood

Solution I.60 (LikelihoodConceptAnswer). The distribution that fits the data best.

I.1.23. Probability is not Likelihood

Solution I.61 (ProbabilityVsLikelihoodAnswer). Likelihood: Finding the curve that best fits the data. Probability: Calculating the probability of an event given a specific curve.

I.1.24. Cross Validation

Solution I.62 (TrainVsTestDataAnswer). Training data is used to fit the model, while testing data is used to evaluate how well the model fits.

Solution I.63 (SingleValidationIssueAnswer). The performance might not be representative because the data may not be equally distributed between training and testing sets.

Solution I.64 (FoldDefinitionAnswer). TBD

Solution I.65 (LeaveOneOutValidationAnswer). Only one data point is used as the test set, and the rest are used as the training set.

I.1.25. The Confusion Matrix

Solution I.66 (ConfusionMatrixAnswer). TBD

I.1.26. Sensitivity and Specificity

Solution I.67 (SensitivitySpecificityAnswer1). TBD

Solution I.68 (SensitivitySpecificityAnswer2). TBD

I.1.27. Bias and Variance

Solution I.69 (BiasAndVarianceAnswer). TBD

I.1.28. Mutual Information

Solution I.70 (MutualInformationExampleAnswer). TBD

I.1.29. Principal Component Analysis (PCA)

Solution I.71 (WhatIsPCAAnswer). A dimension reduction technique that helps discover important variables.

Solution I.72 (screePlotAnswer). It shows how much variation is defined by the data.

Solution I.73 (LeastSquaresInPCAAnswer). No, in the first step it tries to maximize distances.

Solution I.74 (PCAStepsAnswer).

1. Calculate mean
2. Shift the data to the center of the coordinate system
3. Fit a line by maximizing the distances
4. Calculate the sum of squared distances
5. Calculate the slope
6. Rotate

Solution I.75 (EigenvaluePC1Answer). Formula (to be specified).

Solution I.76 (DifferencesBetweenPointsAnswer). No, because the first difference is measured on the PC1 scale and it is more important.

Solution I.77 (ScalingInPCAAnswer). Scaling by dividing by the standard deviation (SD).

Solution I.78 (DetermineNumberOfComponentsAnswer). TBD

Solution I.79 (LimitingNumberOfComponentsAnswer).

1. The dimension of the problem
2. Number of samples

I.1.30. t-SNE

Solution I.80 (WhyUseTSNEAnswer). For dimension reduction and picking out the relevant clusters.

Solution I.81 (MainIdeaOfTSNEAnswer). To reduce the dimensions of the data by reconstructing the relationships in a lower-dimensional space.

Solution I.82 (BasicConceptOfTSNEAnswer).

1. First, randomly arrange the points in a lower dimension

2. Decide whether to move points left or right, depending on distances in the original dimension
3. Finally, arrange points in the lower dimension similarly to the original dimension

Solution I.83 (TSNEStepsAnswer).

1. Project data to get random points
2. Set up a matrix of distances
3. Calculate the inner variances of the clusters and the Gaussian distribution
4. Do the same with the projected points
5. Move projected points so the second matrix gets more similar to the first matrix

I.1.31. K-means clustering

Solution I.84 (HowKMeansWorksAnswer).

1. Select the number of clusters
2. Randomly select distinct data points as initial cluster centers
3. Measure the distance between each point and the cluster centers
4. Assign each point to the nearest cluster
5. Repeat the process

Solution I.85 (QualityOfClustersAnswer). Calculate the within-cluster variation.

Solution I.86 (IncreasingKAnswer). If k is too high, each point would be its own cluster. If k is too low, you cannot see the structures.

I.1.32. DBSCAN

Solution I.87 (CorePointInDBSCANAnswer). A point that is close to at least k other points.

Solution I.88 (AddingVsExtendingAnswer). Adding means we add a point and then stop. Extending means we add a point and then look for other neighbors from that point.

Solution I.89 (OutliersInDBSCANAnswer). Points that are not core points and do not belong to existing clusters.

I.1.33. K-nearest neighbors

Solution I.90 (AdvantagesAndDisadvantagesOfKAnswer).

- k = 1: Noise can disturb the process because of possibly incorrect measurements of points.
- k = 100: The majority can be wrong for some groups. It is smoother, but there is less chance to discover the structure of the data.

I.1.34. Naive Bayes

Solution I.91 (NaiveBayesFormulaAnswer). TBD

Solution I.92 (CalculateProbabilitiesAnswer). TBD

I.1.35. Gaussian Naive Bayes

Solution I.93 (UnderflowProblemAnswer). Small values multiplied together can become smaller than the limits of computer memory, resulting in zero. Using logarithms (e.g., log(1/2) -> -1, log(1/4) -> -2) helps prevent underflow.

I.1.36. Trees

Solution I.94 (Tree Usage). Classication, Regression, Clustering

Solution I.95 (Tree Usage). TBD

Solution I.96 (Tree Feature Importance). The most important feature.

Solution I.97 (Regression Tree Limitations). High dimensions

Solution I.98 (Regression Tree Score). SSR + alpha * T

Solution I.99 (Regression Tree Alpha Value Small). The tree is more complex.

Solution I.100 (Regression Tree Increase Alpha Value). We get smaller trees

Solution I.101 (Regression Tree Pruning). Decreases the complexity of the tree to enhance performance and reduce overfitting

I.2. Machine Learning and Artificial Intelligence

I.2.1. Backpropagation

Solution I.102 (ChainRuleAndGradientDescentAnswer). Combination of the chain rule and gradient descent.

Solution I.103 (BackpropagationNamingAnswer). Because you start at the end and go backwards.

I.2.2. Gradient Descent

Solution I.104 (GradDescStepSize). learning rate x slope

Solution I.105 (GradDescIntercept). Old intercept - step size

Solution I.106 (GradDescIntercept). When the step size is small or after a certain number of steps

I.2.3. ReLU

Solution I.107 (Graph ReLU). Graph of ReLU function: f(x) = max(0, x)

I.2.4. CNNs

Solution I.108 (CNNImageRecognitionAnswer).

- too many features for input layer -> high memory consumption
- always shift in data
- it learns local informations and local correlations

Solution I.109 (CNNFiltersInitializationAnswer). The filter values in CNNs are randomly initialized and then trained and optimized through the process of backpropagation.

Solution I.110 (CNNFilterInitializationAnswer). The filter values in CNNs are initially set by random initialization. These filters undergo training via backpropagation, where gradients are computed and used to adjust the filter values to optimize performance.

Solution I.111 (GenNNStockPredictionAnswer). A limitation of using classical neural networks for stock market prediction is their reliance on fixed inputs. Stock market data is dynamic and requires models that can adapt to changing conditions over time.

I.2.5. RNN

Solution I.112 (RNNUnrollingAnswer). In the unrolling process of RNNs, the network is copied and the output from the inner loop is fed into the second layer of the copied network.

Solution I.113 (RNNReliabilityAnswer). RNNs sometimes fail to work reliably due to the vanishing gradient problem (where gradients are less than 1) and the exploding gradient problem (where gradients are greater than 1). Additionally, reliability issues arise because the network and the weights are copied during the unrolling process.

I.2.6. LSTM

Solution I.114 (LSTMSigmoidTanhAnswer). The sigmoid activation function outputs values between 0 and 1, making it suitable for probability determination, whereas the tanh activation function outputs values between -1 and 1.

Solution I.115 (LSTMSigmoidTanhAnswer). State how much of the long term memory should be used.

Solution I.116 (LSTMGatesAnswer). An LSTM network has three types of gates: the forget gate, the input gate, and the output gate. The forget gate decides what information to discard from the cell state, the input gate updates the cell state with new information, and the output gate determines what part of the cell state should be output.

Solution I.117 (LSTMLongTermInfoAnswer). Long-term information is used in the output gate of an LSTM network.

Solution I.118 (LSTMUpdateGatesAnswer). In the input and forget gates.

I.2.7. Pytorch/Lightning

Solution I.119 (PyTorchRequiresGradAnswer). In PyTorch, `requires_grad` indicates whether a tensor should be trained. If set to False, the tensor will not be trained.

I.2.8. Embeddings

Solution I.120 (NN STrings). No, they process numerical values.

Solution I.121 (Embedding Definition). Representation of a word as a vector.

Solution I.122 (Embedding Dimensions). We can model similarities.

I.2.9. Sequence to Sequence Models

Solution I.123 (LSTM). Because they are able to consider "far away" information.

Solution I.124 (Teacher Forcing). We need to force the correct words for the training.

Solution I.125 (Attention). Attention scores compute similarities for one input to the others.

I.2.10. Transformers

Solution I.126 (ChatGPT). Decoder only.

Solution I.127 (Translation). Encoder-Decoder structure.

Solution I.128 (Difference Encoder-Decoder and Decoder Only.).

- Encoder-Decoder: self-attention.
- Decoder only: masked self-attention.

Solution I.129 (Weights).

- a: Randomly
- b: Backpropagation

Solution I.130 (Order of Words). Positional Encoding

Solution I.131 (Relationship Between Words). Masked self-attention which looks at the previous tokens.

Solution I.132 (Masked Self Attention). It works by investigating how similar each word is to itself and all of the proceeding words in the sentence.

Solution I.133 (Softmax). Transformation to values between 0 and 1.

Solution I.134 (Softmax Output). We create two new numbers: Values – like K and Q with different weights. We scale these values by the percentage. -> we get the scaled V´s

Solution I.135 (V´s). Lastly, we sum these values together, which combine separate encodings for both words relative to their similarities to "is", are the masked-self-attention values for "is".

Solution I.136 (Residual Connections). They are bypasses, which combine the position encoded values with masked-self-attention values.

Solution I.137 (Generate Known Word in Sequence).

- Training
- Because it is a Decoder-Only transformer used for prediction and the calculations that you need.

Solution I.138 (Masked-Self-Attention Values and Bypass). We use a simple neural network with two inputs and five outputs for the vocabulary.

References

Abadi, Martin, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2016. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems." *arXiv e-Prints*, March, arXiv:1603.04467.

Aggarwal, Charu, ed. 2007. *Data Streams – Models and Algorithms*. Springer-Verlag.

Arlot, Sylvain, Alain Celisse, et al. 2010. "A Survey of Cross-Validation Procedures for Model Selection." *Statistics Surveys* 4: 40–79.

Bartz, Eva, Thomas Bartz-Beielstein, Martin Zaefferer, and Olaf Mersmann, eds. 2022. *Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide*. Springer.

Bartz-Beielstein, Thomas. 2023a. "PyTorch Hyperparameter Tuning with SPOT: Comparison with Ray Tuner and Default Hyperparameters on CIFAR10." https://github.com/sequential-parameter-optimization/spotpython/blob/main/notebooks/14_spot_ray_hpt_torch_cifar10.ipynb.

———. 2023b. "Hyperparameter Tuning Cookbook: A guide for scikit-learn, PyTorch, river, and spotpython." *arXiv e-Prints*, July. https://doi.org/10.48550/arXiv.2307.10262.

———. 2024a. "Evaluation and Performance Measurement." In, edited by Eva Bartz and Thomas Bartz-Beielstein, 47–62. Singapore: Springer Nature Singapore.

———. 2024b. "Hyperparameter Tuning." In, edited by Eva Bartz and Thomas Bartz-Beielstein, 125–40. Singapore: Springer Nature Singapore.

———. 2024c. "Introduction: From Batch to Online Machine Learning." In *Online Machine Learning: A Practical Guide with Examples in Python*, edited by Eva Bartz and Thomas Bartz-Beielstein, 1–11. Singapore: Springer Nature Singapore. https://doi.org/10.1007/978-981-99-7007-0_1.

———. 2025. "Kriging (Gaussian Process Regression): The Complete Python Code for the Example." https://sequential-parameter-optimization.github.io/Hyperparameter-Tuning-Cookbook/006_num_gp.html.

Bartz-Beielstein, Thomas, Jürgen Branke, Jörn Mehnen, and Olaf Mersmann. 2014. "Evolutionary Algorithms." *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 4 (3): 178–95.

Bartz-Beielstein, Thomas, and Martina Friese. 2011. "Sequential Parameter Optimization and Optimal Computational Budget Allocation for Noisy Optimization Problems." Cologne University of Applied Science, Faculty of Computer Science; Engineering Science.

Bartz-Beielstein, Thomas, Martina Friese, Martin Zaefferer, Boris Naujoks, Oliver Flasch, Wolfgang Konen, and Patrick Koch. 2011. "Noisy optimization with sequential parameter optimization and optimal computational budget allocation." In

Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation, 119–20. New York, NY, USA: ACM.

Bartz-Beielstein, Thomas, and Lukas Hans. 2024. "Drift Detection and Handling." In *Online Machine Learning: A Practical Guide with Examples in Python*, edited by Eva Bartz and Thomas Bartz-Beielstein, 23–39. Singapore: Springer Nature Singapore. https://doi.org/10.1007/978-981-99-7007-0_3.

Bartz-Beielstein, Thomas, Christian Lasarczyk, and Mike Preuss. 2005. "Sequential Parameter Optimization." In *Proceedings 2005 Congress on Evolutionary Computation (CEC'05), Edinburgh, Scotland*, edited by B McKay et al., 773–80. Piscataway NJ: IEEE Press.

Bartz-Beielstein, Thomas, and Martin Zaefferer. 2022. "Hyperparameter Tuning Approaches." In *Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide*, edited by Eva Bartz, Thomas Bartz-Beielstein, Martin Zaefferer, and Olaf Mersmann, 67–114. Springer.

Bifet, Albert. 2010. *Adaptive Stream Mining: Pattern Learning and Mining from Evolving Data Streams.* Vol. 207. Frontiers in Artificial Intelligence and Applications. IOS Press.

Bifet, Albert, and Ricard Gavaldà. 2007. "Learning from Time-Changing Data with Adaptive Windowing." In *Proceedings of the 2007 SIAM International Conference on Data Mining (SDM)*, 443–48.

———. 2009. "Adaptive Learning from Evolving Data Streams." In *Proceedings of the 8th International Symposium on Intelligent Data Analysis: Advances in Intelligent Data Analysis VIII*, 249–60. IDA '09. Berlin, Heidelberg: Springer-Verlag.

Bifet, Albert, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. 2010a. "MOA: Massive Online Analysis." *Journal of Machine Learning Research* 99: 1601–4.

———. 2010b. "MOA: Massive Online Analysis." *Journal of Machine Learning Research* 11: 1601–4.

Bischl, Bernd, Martin Binder, Michel Lang, Tobias Pielok, Jakob Richter, Stefan Coors, Janek Thomas, et al. 2023. "Hyperparameter Optimization: Foundations, Algorithms, Best Practices, and Open Challenges." *WIREs Data Mining and Knowledge Discovery* 13 (2): e1484.

Bohachevsky, I O. 1986. "Generalized Simulated Annealing for Function Optimization." *Technometrics* 28 (3): 209–17.

Box, G E P. 1957. "Evolutionary operation: A method for increasing industrial productivity." *Applied Statistics* 6: 81–101.

Box, G. E. P., and J. S. Hunter. 1957. "Multi-Factor Experimental Designs for Exploring Response Surfaces." *The Annals of Mathematical Statistics* 28 (1): 195–241.

Box, G. E. P., and K. B. Wilson. 1951. "On the Experimental Attainment of Optimum Conditions." *Journal of the Royal Statistical Society. Series B (Methodological)* 13 (1): 1–45.

Chen, Chun Hung. 2010. *Stochastic simulation optimization: an optimal computing budget allocation.* World Scientific.

Chen, Ricky T. Q., Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. 2018. "Neural Ordinary Differential Equations." *arXiv e-Prints*, June, arXiv:1806.07366.

Coello, Carlos A. Coello, Silvia González Brambila, Josué Figueroa Gamboa, and Ma.

Guadalupe Castillo Tapia. 2021. "Multi-Objective Evolutionary Algorithms: Past, Present, and Future." In, edited by Panos M. Pardalos, Varvara Rasskazova, and Michael N. Vrahatis, 137–62. Cham: Springer International Publishing.

Del Castillo, E., D. C. Montgomery, and D. R. McCarville. 1996. "Modified Desirability Functions for Multiple Response Optimization." *Journal of Quality Technology* 28: 337–45.

Derringer, G., and R. Suich. 1980. "Simultaneous Optimization of Several Response Variables." *Journal of Quality Technology* 12: 214–19.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." *arXiv e-Prints*, October, arXiv:1810.04805.

Domingos, Pedro M., and Geoff Hulten. 2000. "Mining High-Speed Data Streams." In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Boston, MA, USA, August 20-23, 2000*, edited by Raghu Ramakrishnan, Salvatore J. Stolfo, Roberto J. Bayardo, and Ismail Parsa, 71–80. ACM.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, et al. 2020. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale." *arXiv e-Prints*, October, arXiv:2010.11929.

Dredze, Mark, Tim Oates, and Christine Piatko. 2010. "We're Not in Kansas Anymore: Detecting Domain Changes in Streams." In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, 585–95.

Emmerich, Michael T. M., and AndréH. Deutz. 2018. "A Tutorial on Multiobjective Optimization: Fundamentals and Evolutionary Methods." *Natural Computing* 17 (3): 585–609.

Forrester, Alexander, András Sóbester, and Andy Keane. 2008. *Engineering Design via Surrogate Modelling.* Wiley.

Friedman, Jerome H. 1991. "Multivariate Adaptive Regression Splines." *The Annals of Statistics* 19 (1): 1–67.

Gaber, Mohamed Medhat, Arkady Zaslavsky, and Shonali Krishnaswamy. 2005. "Mining Data Streams: A Review." *SIGMOD Rec.* 34: 18–26.

Gama, João, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. 2004. "Learning with Drift Detection." In *Advances in Artificial Intelligence – SBIA 2004*, edited by Ana L. C. Bazzan and Sofiane Labidi, 286–95. Berlin, Heidelberg: Springer Berlin Heidelberg.

Gama, João, Raquel Sebastião, and Pedro Pereira Rodrigues. 2013. "On Evaluating Stream Learning Algorithms." *Machine Learning* 90 (3): 317–46.

Gramacy, Robert B. 2020. *Surrogates.* CRC press.

Harington, J. 1965. "The Desirability Function." *Industrial Quality Control* 21: 494–98.

Hartung, Joachim, Bärbel Elpert, and Karl-Heinz Klösener. 1995. *Statistik.* Oldenbourg.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2017. *The Elements of Statistical Learning.* Second. Springer.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Deep Residual

Learning for Image Recognition."

———. 2016. "Identity Mappings in Deep Residual Networks." *arXiv e-Prints*, March, arXiv:1603.05027.

Hoeglinger, Stefan, and Russel Pears. 2007. "Use of Hoeffding Trees in Concept Based Data Stream Mining." *2007 Third International Conference on Information and Automation for Sustainability*, 57–62.

Ikonomovska, Elena. 2012. "Algorithms for Learning Regression Trees and Ensembles on Evolving Data Streams." PhD thesis, Jozef Stefan International Postgraduate School.

Ikonomovska, Elena, João Gama, and Sašo Džeroski. 2011. "Learning Model Trees from Evolving Data Streams." *Data Mining and Knowledge Discovery* 23 (1): 128–68.

Jain, Sarthak, and Byron C. Wallace. 2019. "Attention is not Explanation." *arXiv e-Prints*, February, arXiv:1902.10186.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2014. *An Introduction to Statistical Learning with Applications in R.* 7th ed. Springer.

Johnson, M. E., L. M. Moore, and D. Ylvisaker. 1990. "Minimax and Maximin Distance Designs." *Journal of Statistical Planning and Inference* 26 (2): 131–48.

Jones, Donald R., Matthias Schonlau, and William J. Welch. 1998. "Efficient Global Optimization of Expensive Black-Box Functions." *Journal of Global Optimization* 13 (4): 455–92.

Karl, Florian, Tobias Pielok, Julia Moosbauer, Florian Pfisterer, Stefan Coors, Martin Binder, Lennart Schneider, et al. 2023. "Multi-Objective Hyperparameter Optimization in Machine Learning—an Overview." *ACM Trans. Evol. Learn. Optim.* 3 (4).

Keane, Andrew J, and Prasanth B Nair. 2005. *Computational Approaches for Aerospace Design: The Pursuit of Excellence.* Wiley.

Keller-McNulty, Sallie, ed. 2004. *Statistical Analysis of Massive Data Streams: Proceedings of a Workshop.* Washington, DC: Committee on Applied; Theoretical Statistics, National Research Council; National Academies Press.

Kidger, Patrick. 2022. "On Neural Differential Equations." *arXiv e-Prints*, February, arXiv:2202.02435.

Kohavi, Ron. 1995. "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection." In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, 1137–43. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Kuhn, Max. 2016. "Desirability: Function Optimization and Ranking via Desirability Functions."

Lewis, R M, V Torczon, and M W Trosset. 2000. "Direct search methods: Then and now." *Journal of Computational and Applied Mathematics* 124 (1–2): 191–207.

Li, Lisha, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization." *arXiv e-Prints*, March, arXiv:1603.06560.

Lippe, Phillip. 2022. "UvA Deep Learning Tutorials." https://github.com/phlippe/uvadlc_notebooks/tree/master.

Liu, Liyuan, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng

Gao, and Jiawei Han. 2019. "On the Variance of the Adaptive Learning Rate and Beyond." *arXiv e-Prints*, August, arXiv:1908.03265.

Manapragada, Chaitanya, Geoffrey I. Webb, and Mahsa Salehi. 2018. "Extremely Fast Decision Tree." In *KDD' 2018 - Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, edited by Chih-Jen Lin and Hui Xiong, 1953–62. United States of America: Association for Computing Machinery (ACM). https://doi.org/10.1145/3219819.3220005.

Masud, Mohammad, Jing Gao, Latifur Khan, Jiawei Han, and Bhavani M Thuraisingham. 2011. "Classification and Novel Class Detection in Concept-Drifting Data Streams Under Time Constraints." *IEEE Transactions on Knowledge and Data Engineering* 23 (6): 859–74.

Meignan, David, Sigrid Knust, Jean-Marc Frayet, Gilles Pesant, and Nicolas Gaud. 2015. "A Review and Taxonomy of Interactive Optimization Methods in Operations Research." *ACM Transactions on Interactive Intelligent Systems*, September.

Micchelli, Charles A. 1986. "Interpolation of Scattered Data: Distance Matrices and Conditionally Positive Definite Functions." *Constructive Approximation* 2 (1): 11–22. https://doi.org/10.1007/BF01893414.

Močkus, J. 1974. "On Bayesian Methods for Seeking the Extremum." In *Optimization Techniques IFIP Technical Conference*, 400–404.

Montgomery, D C. 2001. *Design and Analysis of Experiments*. 5th ed. New York NY: Wiley.

Montiel, Jacob, Max Halford, Saulo Martiello Mastelini, Geoffrey Bolmier, Raphael Sourty, Robin Vaysse, Adil Zouitine, et al. 2021. "River: Machine Learning for Streaming Data in Python."

Morris, Max D., and Toby J. Mitchell. 1995. "Exploratory Designs for Computational Experiments." *Journal of Statistical Planning and Inference* 43 (3): 381–402. https://doi.org/https://doi.org/10.1016/0378-3758(94)00035-T.

Mourtada, Jaouad, Stephane Gaiffas, and Erwan Scornet. 2019. "AMF: Aggregated Mondrian Forests for Online Learning." *arXiv e-Prints*, June, arXiv:1906.10529. https://doi.org/10.48550/arXiv.1906.10529.

Myers, Raymond H, Douglas C Montgomery, and Christine M Anderson-Cook. 2016. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. John Wiley & Sons.

Nelder, J. A., and R. Mead. 1965. "A Simplex Method for Function Minimization." *The Computer Journal* 7 (4): 308–13.

Nino, Esmeralda, Juan Rosas Rubio, Samuel Bonet, Nazario Ramirez-Beltran, and Mauricio Cabrera-Rios. 2015. "Multiple Objective Optimization Using Desirability Functions for the Design of a 3D Printer Prototype." In.

"NIST/SEMATECH e-Handbook of Statistical Methods." 2021.

Olsson, Donald M, and Lloyd S Nelson. 1975. "The Nelder-Mead Simplex Procedure for Function Minimization." *Technometrics* 17 (1): 45–51.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al. 2011. "Scikit-Learn: Machine Learning in Python." *Journal of Machine Learning Research* 12: 2825–30.

Poggio, T, and F Girosi. 1990. "Regularization Algorithms for Learning That Are

Equivalent to Multilayer Networks." *Science* 247 (4945): 978–82. https://doi.org/10.1126/science.247.4945.978.

Pontryagin. 1987. *Mathematical Theory of Optimal Processes.* Routledge.

Putatunda, Sayan. 2021. *Practical Machine Learning for Streaming Data with Python.* Springer.

Raymer, Daniel P. 2006. *Aircraft Design: A Conceptual Approach.* AIAA.

Rummel, R. J. 1976. "Understanding Correlation." https://www.hawaii.edu/powerkills/UC.HTM.

Sacks, J, W J Welch, T J Mitchell, and H P Wynn. 1989. "Design and analysis of computer experiments." *Statistical Science* 4 (4): 409–35.

Santner, T J, B J Williams, and W I Notz. 2003. *The Design and Analysis of Computer Experiments.* Berlin, Heidelberg, New York: Springer.

Street, W. Nick, and YongSeog Kim. 2001. "A Streaming Ensemble Algorithm (SEA) for Large-Scale Classification." In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 377–82. KDD '01. New York, NY, USA: Association for Computing Machinery.

Tay, Yi, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. "Efficient Transformers: A Survey." *arXiv e-Prints*, September, arXiv:2009.06732.

Vapnik, V N. 1998. *Statistical learning theory.* Wiley; Wiley.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. "Attention Is All You Need." *arXiv e-Prints*, June, 1–15.

Wang, Zhiqiang. 2007. "Two Postestimation Commands for Assessing Confounding Effects in Epidemiological Studies." *The Stata Journal* 7 (2): 183–96.

Weihe, Karsten, Ulrik Brandes, Annegret Liebers, Matthias Mi ller-Hannemann, Dorothea Wagner, and Thomas Willhalm. 1999. "Empirical Design of Geometric Algorithms." In *SCG '99: Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, 86–94. New York NY: Association for Computing Machinery.

Wiegreffe, Sarah, and Yuval Pinter. 2019. "Attention is not not Explanation." *arXiv e-Prints*, August, arXiv:1908.04626.

Wikipedia contributors. 2024. "Partial Correlation — Wikipedia, the Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Partial_correlation&oldid=1253637419.

Index

batch_size, 845

correlation coefficient, 566

DataLoaders, 845

fun_sphere, 299

load_from_checkpoint(), 839
log(), 834

Maximum Likelihood Estimation, 578

on_train_epoch_end(), 835
on_validation_epoch_end(), 836

Pearson correlation coefficient, 566
predict_step(), 837

R-squared, 651

sample standard deviation, 580
sample variance, 580
save_hyperparameters(), 838
Sphere function, 299

test_step(), 837
training_step(), 834

validate(), 836
validation_step(), 835