Hyperparameter Tuning Cookbook

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

Thomas Bartz-Beielstein

Feb 24, 2024

Table of contents

Preface 3
Book Structure . . . 3
Software Used in this Book . . . 4
Citation . . . 4

I Optimization 6

1 Introduction: Optimization 7
1.1 Optimization, Simulation, and Surrogate Modeling . . . 7
1.2 Surrogates . . . 7
1.2.1 Costs of Simulation . . . 8
1.2.2 Mathematical Models and Meta-Models . . . 8
1.2.3 Surrogates = Trained Meta-models . . . 8
1.2.4 Computer Experiments . . . 8
1.2.5 Limits of Mathematical Modeling . . . 9
1.2.6 Example: Why Computer Simulations are Necessary . . . 9
1.2.7 Simulation Requirements . . . 9
1.3 Jupyter Notebook . . . 10

2 Aircraft Wing Weight Example 11
2.1 AWWE Equation . . . 11
2.2 AWWE Parameters and Equations (Part 1) . . . 11
2.3 Goals: Understanding and Optimization . . . 12
2.4 Properties of the Python "Solver" . . . 13
2.5 Plot 1: Load Factor (N_z) and Aspect Ratio (A) . . . 14
2.6 Plot 2: Taper Ratio and Fuel Weight . . . 16
2.7 The Big Picture: Combining all Variables . . . 17
2.8 AWWE Landscape . . . 21
2.9 Summary of the First Experiments . . . 22
2.10 Exercise . . . 22
2.10.1 Adding Paint Weight . . . 22
2.11 Jupyter Notebook . . . 23

3 Introduction to `scipy.optimize` **24**
3.1 Derivative-free Optimization Algorithms . . . 25
3.1.1 Nelder-Mead Simplex Algorithm . . . 25
3.1.2 Powell's Method . . . 26
3.2 Gradient-based optimization algorithms . . . 27
3.2.1 An Introductory Example: Broyden-Fletcher-Goldfarb-Shanno Algorithm (BFGS) . . . 27
3.2.2 Background and Basics for Gradient-based Optimization . . . 29
3.2.3 Gradient . . . 29
3.2.4 Jacobian Matrix . . . 29
3.2.5 Hessian Matrix . . . 30
3.2.6 Gradient for Optimization . . . 31
3.2.7 Newton Method . . . 33
3.2.8 BFGS-Algorithm . . . 36
3.2.9 Procedure: . . . 36
3.2.10 Visualization BFGS for Rosenbrock . . . 37
3.3 Gradient- and Hessian-based optimization algorithms . . . 38
3.3.1 Newton-Conjugate-Gradient Algorithm . . . 38
3.3.2 Trust-Region Newton-Conjugate-Gradient Algorithm . . . 38
3.3.3 Trust-Region Truncated Generalized Lanczos / Conjugate Gradient Algorithm . . . 38
3.4 Global Optimization . . . 38
3.4.1 Dual Annealing Optimization . . . 42
3.4.2 Differential Evolution . . . 42
3.4.3 DIRECT . . . 42
3.4.4 SHGO . . . 43
3.4.5 Basin-hopping . . . 43
3.5 Jupyter Notebook . . . 43

4 Sequential Parameter Optimization: Using `scipy` Optimizers **44**
4.1 The Objective Function Branin . . . 44
4.2 The Optimizer . . . 45
4.2.1 TensorBoard . . . 46
4.3 Print the Results . . . 47
4.4 Show the Progress . . . 48
4.5 Exercises . . . 49
4.5.1 `dual_annealing` . . . 49
4.5.2 `direct` . . . 50
4.5.3 `shgo` . . . 52
4.5.4 `basinhopping` . . . 54
4.5.5 Performance Comparison . . . 56
4.6 Jupyter Notebook . . . 57

II Numerical Methods 58

5 Introduction: Numerical Methods 59
5.1 Response Surface Methods: What is RSM? 59
5.1.1 Visualization: Problems in Practice 62
5.1.2 RSM: Strategies 62
5.1.3 RSM: Noise in the Empirical Model 63
5.1.4 RSM: Natural and Coded Variables 63
5.1.5 RSM Low-order Polynomials 64
5.2 First-Order Models (Main Effects Model) 64
5.2.1 First-Order Model Properties 65
5.2.2 First-order Model with Interactions in python 66
5.2.3 Observations: First-Order Model with Interactions 67
5.3 Second-Order Models 67
5.3.1 Second-Order Models: Properties 68
5.3.2 Example: Stationary Ridge 68
5.3.3 Observations: Second-Order Model (Ridge) 69
5.3.4 Example: Rising Ridge 70
5.3.5 Summary: Rising Ridge 71
5.3.6 Falling Ridge 71
5.3.7 Saddle Point 71
5.3.8 Interpretation: Saddle Points 72
5.3.9 Summary: Ridge Analysis 72
5.4 General RSM Models 73
5.4.1 Ordinary Least Squares 73
5.5 Designs 73
5.5.1 Different Designs 73
5.6 RSM Experimentation 74
5.6.1 First Step 74
5.6.2 Second Step 74
5.6.3 Third Step 74
5.7 RSM: Review and General Considerations 74
5.7.1 Historical Considerations about RSM 75
5.7.2 Status Quo 75
5.7.3 The Role of Statistics 76
5.7.4 New RSM is needed: DACE 76
5.8 Exercises 77
5.9 Jupyter Notebook 77

6 Kriging (Gaussian Process Regression) 78
6.1 DACE and RSM 78
6.2 Background: Expectation, Mean, Standard Deviation 79
6.2.1 Sample Mean 79

6.2.2 Variance and Standard Deviation . . . 79
6.2.3 Standard Deviation . . . 80
6.2.4 Calculation of the Standard Deviation with Python . . . 80
6.2.5 The Empirical Standard Deviation . . . 80
6.2.6 The Argument "axis" . . . 81
6.3 Data Types and Precision in Python . . . 81
6.4 Distributions and Random Numbers in Python . . . 83
6.4.1 The Uniform Distribution . . . 84
6.4.2 The Normal Distribution . . . 85
6.4.3 Visualization of the Standard Deviation . . . 87
6.4.4 Standardization of Random Variables . . . 87
6.4.5 Realizations of a Normal Distribution . . . 87
6.4.6 The Multivariate Normal Distribution . . . 88
6.4.7 The Bivariate Normal Distribution with Mean Zero and Zero Covariances $\sigma_{12} = \sigma_{21} = 0$ . . . 92
6.4.8 The Bivariate Normal Distribution with Mean Zero and Negative Covariances $\sigma_{12} = \sigma_{21} = -4$ . . . 92
6.5 Cholesky Decomposition and Positive Definite Matrices . . . 92
6.6 Maximum Likelihood Estimation: Multivariate Normal Distribution . . . 95
6.7 Introduction to Gaussian Processes . . . 95
6.7.1 Gaussian Process Prior . . . 96
6.7.2 Covariance Function . . . 96
6.7.3 Construction of the Covariance Matrix . . . 98
6.7.4 Generation of Random Samples and Plotting the Realizations of the Random Function . . . 100
6.7.5 Properties of the 1d Example . . . 102
6.8 Kriging: Modeling Basics . . . 105
6.8.1 The Kriging Idea in a Nutshell . . . 105
6.8.2 The Kriging Basis Function . . . 105
6.8.3 The Correlation Coefficient . . . 106
6.8.4 Covariance Matrix and Correlation Matrix . . . 107
6.8.5 The Kriging Model . . . 108
6.8.6 Correlations . . . 109
6.8.7 The Condition Number . . . 113
6.8.8 MLE to estimate θ and p . . . 114
6.8.9 Tuning θ and p . . . 115
6.9 Kriging Prediction . . . 115
6.9.1 The Augmented Correlation Matrix . . . 115
6.9.2 Properties of the Predictor . . . 116
6.10 Kriging Example: Sinusoid Function . . . 116
6.10.1 Calculating the Correlation Matrix Ψ . . . 116
6.10.2 Computing the ψ Vector . . . 117
6.10.3 Predicting at New Locations . . . 118

6.10.4 Visualization . . . 118
6.11 Cholesky Example With Two Points . . . 119
6.11.1 Cholesky Decomposition . . . 119
6.11.2 Computation of the Inverse Matrix . . . 120
6.12 Jupyter Notebook . . . 121

7 Introduction to spotPython 122
7.1 Example: `Spot` and the Sphere Function . . . 122
7.1.1 The Objective Function: Sphere . . . 123
7.1.2 The `Spot` Method as an Optimization Algorithm Using a Surrogate Model 124
7.2 `Spot` Parameters: `fun_evals`, `init_size` and `show_models` . . . 126
7.3 Print the Results . . . 127
7.4 Show the Progress . . . 127
7.5 Visualizing the Optimization and Hyperparameter Tuning Process with TensorBoard . . . 128
7.6 Jupyter Notebook . . . 131

8 Multi-dimensional Functions 132
8.1 Example: `Spot` and the 3-dim Sphere Function . . . 132
8.1.1 The Objective Function: 3-dim Sphere . . . 132
8.1.2 Results . . . 133
8.1.3 A Contour Plot . . . 134
8.1.4 TensorBoard . . . 136
8.2 Conclusion . . . 137
8.3 Exercises . . . 138
8.3.1 1. The Three Dimensional `fun_cubed` . . . 138
8.3.2 2. The Ten Dimensional `fun_wing_wt` . . . 138
8.3.3 3. The Three Dimensional `fun_runge` . . . 138
8.3.4 4. The Three Dimensional `fun_linear` . . . 139
8.3.5 5. The Two Dimensional Rosenbrock Function `fun_rosen` . . . 139
8.4 Selected Solutions . . . 139
8.4.1 Solution to Exercise Section 8.3.5: The Two-dimensional Rosenbrock Function `fun_rosen` . . . 139
8.5 Jupyter Notebook . . . 143

9 Isotropic and Anisotropic Kriging 144
9.1 Example: Isotropic `Spot` Surrogate and the 2-dim Sphere Function . . . 144
9.1.1 The Objective Function: 2-dim Sphere . . . 144
9.1.2 Results . . . 145
9.2 Example With Anisotropic Kriging . . . 146
9.2.1 Taking a Look at the `theta` Values . . . 148
9.3 Exercises . . . 149
9.3.1 1. The Branin Function `fun_branin` . . . 149

9.3.2 2. The Two-dimensional Sin-Cos Function `fun_sin_cos` 150
9.3.3 3. The Two-dimensional Runge Function `fun_runge` 150
9.3.4 4. The Ten-dimensional Wing-Weight Function `fun_wingwt` 150
9.3.5 5. The Two-dimensional Rosenbrock Function `fun_rosen` 151
9.4 Selected Solutions 151
9.4.1 Solution to Exercise Section 9.3.5: The Two-dimensional Rosenbrock Function `fun_rosen` 151
9.5 Jupyter Notebook 157

10 Using `sklearn` Surrogates in `spotPython` 158
10.1 Example: Branin Function with `spotPython`'s Internal Kriging Surrogate . . . 158
10.1.1 The Objective Function Branin 158
10.1.2 Running the surrogate model based optimizer `Spot`: 159
10.1.3 TensorBoard 160
10.1.4 Print the Results 161
10.1.5 Show the Progress and the Surrogate 161
10.2 Example: Using Surrogates From scikit-learn 162
10.2.1 GaussianProcessRegressor as a Surrogate 163
10.3 Example: One-dimensional Sphere Function With `spotPython`'s Kriging 164
10.3.1 Results 170
10.4 Example: `Sklearn` Model GaussianProcess 171
10.5 Exercises 177
10.5.1 1. A decision tree regressor: `DecisionTreeRegressor` 177
10.5.2 2. A random forest regressor: `RandomForestRegressor` 177
10.5.3 3. Ordinary least squares Linear Regression: `LinearRegression` 177
10.5.4 4. Linear least squares with l2 regularization: `Ridge` 178
10.5.5 5. Gradient Boosting: `HistGradientBoostingRegressor` 178
10.5.6 6. Comparison of Surrogates 178
10.6 Selected Solutions 179
10.6.1 Solution to Exercise Section 10.5.5: Gradient Boosting 179
10.7 Jupyter Notebook 195

11 Sequential Parameter Optimization: Gaussian Process Models 196
11.1 Gaussian Processes Regression: Basic Introductory `scikit-learn` Example . . 196
11.1.1 Train and Test Data 196
11.1.2 Building the Surrogate With `Sklearn` 197
11.1.3 Plotting the `Sklearn`Model 197
11.1.4 The `spotPython` Version 198
11.1.5 Visualizing the Differences Between the `spotPython` and the `sklearn` Model Fits 199
11.2 Exercises 200
11.2.1 `Schonlau Example Function` 200
11.2.2 `Forrester Example Function` 200

11.2.3 `fun_runge Function (1-dim)` . . . 201
11.2.4 `fun_cubed (1-dim)` . . . 202
11.2.5 The Effect of Noise . . . 202

12 Expected Improvement **203**
12.1 Example: `Spot` and the 1-dim Sphere Function . . . 203
12.1.1 The Objective Function: 1-dim Sphere . . . 203
12.1.2 Results . . . 204
12.2 Same, but with EI as infill_criterion . . . 205
12.3 Non-isotropic Kriging . . . 207
12.4 Using `sklearn` Surrogates . . . 210
12.4.1 The spot Loop . . . 210
12.4.2 spot: The Initial Model . . . 211
12.4.3 Init: Build Initial Design . . . 212
12.4.4 Evaluate . . . 215
12.4.5 Build Surrogate . . . 215
12.4.6 A Simple Predictor . . . 215
12.5 Gaussian Processes regression: basic introductory example . . . 215
12.6 The Surrogate: Using scikit-learn models . . . 218
12.7 Additional Examples . . . 221
12.7.1 Optimize on Surrogate . . . 223
12.7.2 Evaluate on Real Objective . . . 223
12.7.3 Impute / Infill new Points . . . 223
12.8 Tests . . . 223
12.9 EI: The Famous Schonlau Example . . . 225
12.10 EI: The Forrester Example . . . 227
12.11 Noise . . . 230
12.12 Cubic Function . . . 233
12.13 Modifying Lambda Search Space . . . 239
12.14 Factors . . . 240

13 Handling Noise **242**
13.1 Example: `Spot` and the Noisy Sphere Function . . . 242
13.1.1 The Objective Function: Noisy Sphere . . . 242
13.1.2 Reproducibility: Noise Generation and Seed Handling . . . 244
13.2 spotPython's Noise Handling Approaches . . . 246
13.3 Print the Results . . . 252
13.4 Noise and Surrogates: The Nugget Effect . . . 252
13.4.1 The Noisy Sphere . . . 252
13.5 Exercises . . . 255
13.5.1 Noisy `fun_cubed` . . . 255
13.5.2 `fun_runge` . . . 256
13.5.3 `fun_forrester` . . . 256

13.5.4 fun_xsin . . . 256

14 Optimal Computational Budget Allocation in Spot **257**
14.1 Example: Spot, OCBA, and the Noisy Sphere Function . . . 257
14.1.1 The Objective Function: Noisy Sphere . . . 257
14.2 Print the Results . . . 264
14.3 Noise and Surrogates: The Nugget Effect . . . 264
14.3.1 The Noisy Sphere . . . 264
14.4 Exercises . . . 267
14.4.1 Noisy fun_cubed . . . 267
14.4.2 fun_runge . . . 268
14.4.3 fun_forrester . . . 268
14.4.4 fun_xsin . . . 268

15 Kriging with Varying Correlation-p **269**
15.1 Example: Spot Surrogate and the 2-dim Sphere Function . . . 269
15.1.1 The Objective Function: 2-dim Sphere . . . 269
15.1.2 Results . . . 270
15.2 Example With Modified p . . . 271
15.2.1 Taking a Look at the p Values . . . 273
15.3 Optimization of the p Values . . . 274
15.4 Optimization of Multiple p Values . . . 275
15.5 Exercises . . . 277
15.5.1 fun_branin . . . 277
15.5.2 fun_sin_cos . . . 278
15.5.3 fun_runge . . . 278
15.5.4 fun_wingwt . . . 278
15.6 Jupyter Notebook . . . 278

III Hyperparameter Tuning with Sklearn **279**

16 HPT: sklearn **280**
16.1 Introduction to sklearn . . . 280

17 HPT: sklearn SVC on Moons Data **281**
17.1 Step 1: Setup . . . 281
17.2 Step 2: Initialization of the Empty fun_control Dictionary . . . 281
17.3 Step 3: SKlearn Load Data (Classification) . . . 282
17.4 Step 4: Specification of the Preprocessing Model . . . 284
17.5 Step 5: Select Model (algorithm) and core_model_hyper_dict . . . 285

17.6 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` . . . 287
17.6.1 Modify hyperparameter of type numeric and integer (boolean) . . . 288
17.6.2 Modify hyperparameter of type factor . . . 288
17.6.3 Optimizers . . . 289
17.7 Step 7: Selection of the Objective (Loss) Function . . . 289
17.7.1 Predict Classes or Class Probabilities . . . 289
17.8 Step 8: Calling the SPOT Function . . . 290
17.8.1 The Objective Function . . . 290
17.8.2 Run the `Spot` Optimizer . . . 290
17.8.3 TensorBoard . . . 292
17.9 Step 9: Results . . . 293
17.10 Get Default Hyperparameters . . . 295
17.11 Get SPOT Results . . . 295
17.11.1 Plot: Compare Predictions . . . 296
17.11.2 Detailed Hyperparameter Plots . . . 298
17.11.3 Parallel Coordinates Plot . . . 302
17.11.4 Plot all Combinations of Hyperparameters . . . 302

IV Hyperparameter Tuning with River 303

18 HPT: River 304
18.1 Introduction to River . . . 304

19 Simplifying Hyperparameter Tuning in Online Machine Learning—The spotRiver-GUI 305
19.1 Introduction . . . 305
19.2 Installation and Starting . . . 306
19.2.1 Installation . . . 306
19.2.2 Starting the GUI . . . 307
19.3 Binary Classification . . . 307
19.3.1 Binary Classification Options . . . 307
19.3.2 Experiment Options . . . 310
19.3.3 Evaluation Options . . . 311
19.3.4 Online Machine Learning Model Options . . . 313
19.4 Regression . . . 315
19.5 Showing the Data . . . 315
19.6 Saving and Loading . . . 320
19.6.1 Saving the Experiment . . . 320
19.6.2 Loading an Experiment . . . 320
19.7 Running a New Experiment . . . 321
19.7.1 Starting and Stopping Tensorboard . . . 321

19.8 Performing the Analysis . . . 322
19.9 Summary and Outlook . . . 326

20 `river` Hyperparameter Tuning: Hoeffding Adaptive Tree Regressor with Friedman Drift Data 328
20.1 Setup . . . 328
20.2 Initialization of the `fun_control` Dictionary . . . 329
20.3 Load Data: The Friedman Drift Data . . . 330
20.4 Specification of the Preprocessing Model . . . 331
20.5 SelectSelect Model (`algorithm`) and `core_model_hyper_dict` . . . 331
20.6 Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` 332
20.7 Selection of the Objective (Loss) Function . . . 333
20.8 Calling the SPOT Function . . . 335
20.8.1 The Objective Function . . . 335
20.8.2 Run the `Spot` Optimizer . . . 335
20.8.3 TensorBoard . . . 336
20.8.4 Results . . . 337
20.9 The Larger Data Set . . . 339
20.10 Get Default Hyperparameters . . . 340
20.10.1 Show Predictions . . . 342
20.11 Get SPOT Results . . . 343
20.12 Visualize Regression Trees . . . 346
20.12.1 Spot Model . . . 347
20.13 Detailed Hyperparameter Plots . . . 348
20.14 Parallel Coordinates Plots . . . 384
20.15 Plot all Combinations of Hyperparameters . . . 384

21 `river` Hyperparameter Tuning: Mondrian Tree Regressor with Friedman Drift Data 385
21.1 Setup . . . 385
21.2 Initialization of the `fun_control` Dictionary . . . 386
21.3 Load Data: The Friedman Drift Data . . . 387
21.4 Specification of the Preprocessing Model . . . 388
21.5 SelectSelect Model (`algorithm`) and `core_model_hyper_dict` . . . 388
21.6 Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` 389
21.7 Selection of the Objective (Loss) Function . . . 390
21.8 Calling the SPOT Function . . . 391
21.8.1 The Objective Function . . . 391
21.8.2 Run the `Spot` Optimizer . . . 391
21.8.3 TensorBoard . . . 392
21.8.4 Results . . . 394
21.9 The Larger Data Set . . . 396
21.10 Get Default Hyperparameters . . . 397
21.10.1 Show Predictions . . . 398

21.11 Get SPOT Results . . . 399
21.12 Detailed Hyperparameter Plots . . . 402
21.13 Parallel Coordinates Plots . . . 403
21.14 Plot all Combinations of Hyperparameters . . . 403

V Hyperparameter Tuning with PyTorch Lightning 404

22 HPT PyTorch Lightning: Diabetes 405
22.1 Step 1: Setup . . . 405
22.2 Step 2: Initialization of the `fun_control` Dictionary . . . 406
22.3 Step 3: Loading the Diabetes Data Set . . . 407
22.4 Step 4: Preprocessing . . . 408
22.5 Step 5: Select the Core Model (`algorithm`) and `core_model_hyper_dict` . . . 408
22.6 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` . . . 409
22.7 Step 7: Data Splitting, the Objective (Loss) Function and the Metric . . . 410
22.7.1 Evaluation . . . 410
22.7.2 Loss Function . . . 411
22.7.3 Metric . . . 411
22.8 Step 8: Calling the SPOT Function . . . 411
22.8.1 Preparing the SPOT Call . . . 411
22.8.2 The Objective Function `fun` . . . 412
22.8.3 Showing the fun_control Dictionary . . . 412
22.8.4 Starting the Hyperparameter Tuning . . . 416
22.9 Step 9: Tensorboard . . . 423
22.10 Step 10: Results . . . 423
22.10.1 Get the Tuned Architecture . . . 424
22.10.2 Parallel Coordinates Plot . . . 430
22.10.3 Cross Validation With Lightning . . . 431
22.10.4 Plot all Combinations of Hyperparameters . . . 432
22.10.5 Visualizing the Activation Distribution (Under Development) . . . 432

23 HPT PyTorch Lightning: Diabetes Using a Recurrent Neural Network 434
23.1 Step 1: Setup . . . 434
23.2 Step 2: Initialization of the `fun_control` Dictionary . . . 435
23.3 Step 3: Loading the Diabetes Data Set . . . 436
23.4 Step 4: Preprocessing . . . 437
23.5 Step 5: Select the Core Model (`algorithm`) and `core_model_hyper_dict` . . . 438
23.6 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` . . . 439
23.7 Step 7: Data Splitting, the Objective (Loss) Function and the Metric . . . 440
23.7.1 Evaluation . . . 440

23.7.2 Loss Function . . . 440
23.7.3 Metric . . . 441
23.8 Step 8: Calling the SPOT Function . . . 441
23.8.1 Preparing the SPOT Call . . . 441
23.8.2 The Objective Function `fun` . . . 441
23.8.3 Showing the fun_control Dictionary . . . 442
23.8.4 Starting the Hyperparameter Tuning . . . 446
23.9 Step 9: Tensorboard . . . 453
23.10 Step 10: Results . . . 453
23.10.1 Get the Tuned Architecture . . . 455
23.10.2 Parallel Coordinates Plot . . . 457
23.10.3 Cross Validation With Lightning . . . 458
23.10.4 Plot all Combinations of Hyperparameters . . . 459
23.10.5 Visualizing the Activation Distribution (Under Development) . . . 459

24 HPT PyTorch Lightning: User Specified Data Set and Regression Model 461
24.1 Step 1: Setup . . . 461
24.2 Step 2: Initialization of the `fun_control` Dictionary . . . 462
24.3 Step 3: Loading the User Specified Data Set . . . 463
24.4 Step 4: Preprocessing . . . 465
24.5 Step 5: Select the Core Model (`algorithm`) and `core_model_hyper_dict` . . . 466
24.6 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` . . . 466
24.7 Step 7: Data Splitting, the Objective (Loss) Function and the Metric . . . 469
24.7.1 Evaluation . . . 469
24.7.2 Loss Function . . . 469
24.7.3 Metric . . . 469
24.8 Step 8: Calling the SPOT Function . . . 469
24.8.1 Preparing the SPOT Call . . . 469
24.8.2 The Objective Function `fun` . . . 470
24.8.3 Showing the fun_control Dictionary . . . 470
24.8.4 Starting the Hyperparameter Tuning . . . 474
24.9 Step 9: Tensorboard . . . 479
24.10 Step 10: Results . . . 479
24.10.1 Get the Tuned Architecture . . . 481
24.10.2 Parallel Coordinates Plot . . . 484
24.10.3 Cross Validation With Lightning . . . 485
24.10.4 Plot all Combinations of Hyperparameters . . . 486
24.10.5 Visualizing the Activation Distribution (Under Development) . . . 486

25 Explainable AI with SpotPython and Pytorch 488

26 HPT PyTorch Lightning Transformer: Introduction **520**
26.1 Decoder-Only Transformer . . . 520
26.1.1 Embedding . . . 520
26.1.2 Masked Self-Attention . . . 521
26.1.3 Generation of outputs . . . 521
26.1.4 End-Of-Sequence-Token . . . 522

27 Example: Word Predictor **523**

28 HPT PyTorch Lightning Transformer: Diabetes **527**
28.1 Step 1: Setup . . . 527
28.2 Step 2: Initialization of the `fun_control` Dictionary . . . 528
28.3 Step 3: Loading the Diabetes Data Set . . . 529
28.4 Step 4: Preprocessing . . . 529
28.5 Step 5: Select the Core Model (`algorithm`) and `core_model_hyper_dict` . . . 530
28.6 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` . . . 530
28.7 Step 7: Data Splitting, the Objective (Loss) Function and the Metric . . . 532
28.7.1 Evaluation . . . 532
28.7.2 Loss Function . . . 532
28.7.3 Metric . . . 532
28.8 Step 8: Calling the SPOT Function . . . 532
28.8.1 Preparing the SPOT Call . . . 532
28.8.2 The Objective Function `fun` . . . 533
28.8.3 Showing the fun_control Dictionary . . . 533
28.8.4 Starting the Hyperparameter Tuning . . . 533
28.9 Step 9: Tensorboard . . . 534
28.10 Step 10: Results . . . 534
28.10.1 Get the Tuned Architecture . . . 534
28.10.2 Parallel Coordinates Plot . . . 535
28.10.3 Cross Validation With Lightning . . . 535
28.10.4 Plot all Combinations of Hyperparameters . . . 535
28.10.5 Visualizing the Activation Distribution (Under Development) . . . 536

Appendices **537**

A Introduction to Jupyter Notebook **537**
A.1 Different Notebook cells . . . 537
A.1.1 Code cells . . . 537
A.1.2 Markdown cells . . . 537
A.1.3 Raw cells . . . 538
A.2 Install Packages . . . 538

A.3 Load Packages . . . 539
A.4 Functions in Python . . . 539
A.5 List of Useful Jupyter Notebook Shortcuts . . . 540

B Git Introduction **542**
B.1 Learning Objectives . . . 542
B.2 Basics of Git . . . 542
B.2.1 Initializing a Repository: `git init` . . . 542
B.2.2 Ignoring Files: `.gitignore` . . . 543
B.2.3 Adding Changes to the Staging Area: `git add` . . . 543
B.2.4 Transferring Changes to Memory: `git commit` . . . 544
B.2.5 Check the Status of Your Repository: `git status` . . . 545
B.2.6 Review Your Repository's History: `git log` . . . 546
B.3 Branches (Timelines) . . . 546
B.3.1 Creating an Alternative Timeline: `git branch` . . . 546
B.3.2 The Pointer to the Current Branch: `HEAD` . . . 547
B.3.3 Switching to an Alternative Timeline: `git switch` . . . 547
B.3.4 Switching to an Alternative Timeline and Making Changes: `git checkout` 547
B.3.5 The Difference Between `checkout` and `switch` . . . 548
B.4 Merging Branches and Resolving Conflicts . . . 550
B.4.1 git `merge`: Merging Two Timelines . . . 550
B.4.2 Resolving Conflicts When Merging . . . 551
B.4.3 git `revert`: Undoing Something . . . 552
B.5 Downloading from GitLab . . . 554
B.6 Advanced . . . 555
B.6.1 git `rebase`: Moving the Base of a Branch . . . 555
B.7 Exercises . . . 557
B.7.1 Create project folder . . . 558
B.8 Initialize repo . . . 558
B.8.1 Do not upload / ignore certain file types . . . 558
B.8.2 Create file and stage it . . . 558
B.8.3 Create another file and check status . . . 558
B.8.4 Commit changes . . . 558
B.8.5 Create a new branch and switch to it . . . 559
B.8.6 Commit changes in the new branch . . . 559
B.8.7 Merge branch into main . . . 559
B.8.8 Resolve merge conflict . . . 559

C Python Introduction **560**
C.1 Recommendations . . . 560

D Documentation of the Sequential Parameter Optimization **561**
D.1 An Initial Example . . . 561

D.2 Organization . . . 563
D.3 The Spot Object . . . 564
D.4 Run . . . 564
D.5 Print the Results . . . 564
D.6 Show the Progress . . . 565
D.7 Visualize the Surrogate . . . 565
D.8 Run With a Specific Start Design . . . 566
D.9 Init: Build Initial Design . . . 567
D.10 Replicability . . . 568
D.11 Surrogates . . . 569
D.11.1 A Simple Predictor . . . 569
D.12 Demo/Test: Objective Function Fails . . . 569

References **572**

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 2023) 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 Introduction: Optimization

1.1 Optimization, 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

1.2 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

1.2.1 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.

1.2.2 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

1.2.3 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**

1.2.4 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

1.2.5 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

1.2.6 Example: 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

1.2.7 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

! Goals

- How to choose models and optimizers for solving real-world problems
- How to use simulation to understand and improve processes

1.3 Jupyter Notebook

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

2 Aircraft Wing Weight Example

2.1 AWWE Equation

- Example from Forrester et al.
- Understand the **weight** of an unpainted light aircraft wing as a function of nine design and operational parameters:

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

2.2 AWWE Parameters and Equations (Part 1)

Table 2.1: 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
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 2012. 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.

2.3 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 2.1. To map values from the interval $[a, b]$ to the interval $[0, 1]$, the following formula can be employed:

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

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

$$g(y) = a + (b - a)y$$

can be used.

```
import numpy as np

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)
```

2.4 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.

```
import numpy as np
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)
```

```
[(0.0, 0.0),
 (0.5, 0.0),
 (1.0, 0.0),
 (0.0, 0.5),
 (0.5, 0.5),
 (1.0, 0.5),
 (0.0, 1.0),
 (0.5, 1.0),
 (1.0, 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
import matplotlib.pyplot as plt
# plt.style.use('seaborn-white')
import numpy as np
x = np.linspace(0, 1, 100)
y = np.linspace(0, 1, 100)
X, Y = np.meshgrid(x, y)
```

2.5 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.

2.6 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 $\lambda \times W_{fw}$

2.7 The Big Picture: Combining all Variables

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

```
import math

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.

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

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="0",
                 )
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()
```

2.8 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.

2.9 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

2.10 Exercise

2.10.1 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.

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

2.11 Jupyter Notebook

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

3 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 3.1.

Table 3.1: 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.

3.1 Derivative-free Optimization Algorithms

Section 3.1.1 and Section 3.1.2 present two approaches that do not need gradient information to find the minimum. They use function evaluations to find the minimum.

3.1.1 Nelder-Mead Simplex Algorithm

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

```
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.

3.1.2 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.

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 $b = 1$:

```python
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]
```

3.2 Gradient-based optimization algorithms

3.2.1 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: 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:

$$\frac{\partial f}{\partial x_j} = \sum_{i=1}^{N} 200(x_i - x_{i-1}^2)(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}) - 2(1 - x_{i-1})\delta_{i-1,j} \tag{3.1}$$

$$= 200(x_j - x_{j-1}^2) - 400x_j(x_{j+1} - x_j^2) - 2(1 - x_j) \tag{3.2}$$

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]
```

3.2.2 Background and Basics for Gradient-based Optimization

3.2.3 Gradient

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

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

3.2.4 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: Jacobian 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.

3.2.5 Hessian Matrix

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

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

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

Example: 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:

$$\frac{\partial^2 f}{\partial x_1^2} = 2 + \cos(x_1 x_2) x_2^2 \tag{3.3}$$

$$\frac{\partial^2 f}{\partial x_1 \partial x_2} = 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) \tag{3.4}$$

$$\frac{\partial^2 f}{\partial x_2 \partial x_1} = 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) \tag{3.5}$$

$$\frac{\partial^2 f}{\partial x_2^2} = 4x_2^2 + \cos(x_1 x_2) x_1^2 \tag{3.6}$$

So, the Hessian matrix is:

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

3.2.6 Gradient for Optimization

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: Gradient Descent**
>
> Let's 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.

```python
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='Opt

    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
```



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

    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()
```

3.2.7 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 with respect to the variable x, evaluated at the current solution x_k.
- $\nabla^2 f(x_k)$: 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: 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.

```python
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
```

3.2.8 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.

3.2.9 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: BFGS for Rosenbrock

```python
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: 2.843987518235081e-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
```

3.2.10 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

Tasks

- In which situations is it possible to use algorithms like BFGS, but not the classical Newton method?
- Investigate the Newton-CG method
- Use an objective function of your choice and apply Newton-CG
- Compare the Newton-CG method with the BFGS. What are the similarities and differences between the two algorithms?

3.3 Gradient- and Hessian-based optimization algorithms

Section 3.3.1 presents an optimization algorithm that uses gradient and Hessian information to find the minimum. Section 3.3.2 presents an optimization algorithm that uses gradient and Hessian information to find the minimum. Section 3.3.3 presents an optimization algorithm that uses gradient and Hessian information to find the minimum.

The methods Newton-CG, trust-ncg and trust-krylov are suitable for dealing with large-scale problems (problems with thousands of variables). That is because the conjugate gradient algorithm approximately solve the trust-region subproblem (or invert the Hessian) by iterations without the explicit Hessian factorization. Since only the product of the Hessian with an arbitrary vector is needed, the algorithm is specially suited for dealing with sparse Hessians, allowing low storage requirements and significant time savings for those sparse problems.

3.3.1 Newton-Conjugate-Gradient Algorithm

Newton-Conjugate Gradient algorithm is a modified Newton's method and uses a conjugate gradient algorithm to (approximately) invert the local Hessian.

3.3.2 Trust-Region Newton-Conjugate-Gradient Algorithm

3.3.3 Trust-Region Truncated Generalized Lanczos / Conjugate Gradient Algorithm

3.4 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. 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: -935.3379515605789
    funl: [-9.353e+02]
       x: [ 4.395e+02  4.540e+02]
      xl: [[ 4.395e+02  4.540e+02]]
     nit: 1
    nfev: 45
   nlfev: 40
   nljev: 10
   nlhev: 0
```

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

```
 message: ['Maximum number of iteration reached']
 success: True
  status: 0
     fun: -935.3379515597601
       x: [ 4.395e+02  4.540e+02]
     nit: 1000
    nfev: 4109
    njev: 36
    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: -894.578900390515
       x: [-4.657e+02  3.857e+02]
     nit: 22
    nfev: 711
     jac: [-1.137e-05 -2.274e-05]
```

`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: -959.640662720831
    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: 3529
   nlfev: 2327
   nljev: 634
   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()
```

3.4.1 Dual Annealing Optimization

This function implements the Dual Annealing optimization.

3.4.2 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.

3.4.3 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

3.4.4 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).

3.4.5 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

3.5 Jupyter Notebook

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

4 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_in
```

4.1 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

```
$$y = a * (x2 - b * x1**2 + c * x1 - r) ** 2 + s * (1 - t) * cos(x1) + s,$$
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 $(9.42478, 2.475)$.

- Input Domain: This function is usually evaluated on the square x1 in [-5, 10] x x2 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
```

4.2 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:

`surrogate_control = "model_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 7.5, 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_"
>  )
> surrogate_control=surrogate_control_init(
>  n_theta=len(lower))
> ```

```
Created spot_tensorboard_path: runs/spot_logs/04_DE__p040025_2024-02-24_00-18-20 for Summar
```

```
spot_de = spot.Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate_control=surrogate_control)
spot_de.run()
```

```
spotPython tuning: 3.8004644561334935 [######----] 55.00%
spotPython tuning: 3.8004644561334935 [######----] 60.00%
spotPython tuning: 3.1590379739505225 [######----] 65.00%
spotPython tuning: 3.1345599589760926 [#######---] 70.00%
spotPython tuning: 2.8987595919440583 [########--] 75.00%
spotPython tuning: 0.4124604824941809 [########--] 80.00%
spotPython tuning: 0.40391426855740775 [########--] 85.00%
spotPython tuning: 0.3990718447916741 [#########-] 90.00%
spotPython tuning: 0.3990718447916741 [##########] 95.00%
spotPython tuning: 0.3990718447916741 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x10552e990>
```

4.2.1 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 4.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

4.3 Print the Results

```
spot_de.print_results()
```

```
min y: 0.3990718447916741
x0: 3.149600915888656
x1: 2.2983701643039107

[['x0', 3.149600915888656], ['x1', 2.2983701643039107]]
```

4.4 Show the Progress

```
spot_de.plot_progress(log_y=True)
```

```
spot_de.surrogate.plot()
```

4.5 Exercises

4.5.1 `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.Spot(fun=fun,
                    fun_control=fun_control,
                    optimizer=dual_annealing,
                    surrogate_control=surrogate_control)
spot_da.run()
spot_da.print_results()
spot_da.plot_progress(log_y=True)
spot_da.surrogate.plot()
```

```
spotPython tuning: 3.800452934057194 [######----] 55.00%
spotPython tuning: 3.800452934057194 [######----] 60.00%
spotPython tuning: 3.1590242778566413 [######----] 65.00%
spotPython tuning: 3.1341475332648105 [#######---] 70.00%
spotPython tuning: 2.8915909597236436 [########--] 75.00%
spotPython tuning: 0.4195069442130439 [########--] 80.00%
spotPython tuning: 0.401848680281649 [########--] 85.00%
spotPython tuning: 0.3992571870039132 [#########-] 90.00%
spotPython tuning: 0.3992571870039132 [##########] 95.00%
spotPython tuning: 0.3992571870039132 [##########] 100.00% Done...

min y: 0.3992571870039132
x0: 3.150936988317143
x1: 2.2985561477641263
```

4.5.2 `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.Spot(fun=fun,
                    fun_control=fun_control,
                    optimizer=direct,
                    surrogate_control=surrogate_control)
spot_di.run()
spot_di.print_results()
spot_di.plot_progress(log_y=True)
spot_di.surrogate.plot()
```

```
spotPython tuning: 3.812970247994418 [######----] 55.00%
spotPython tuning: 3.812970247994418 [######----] 60.00%
spotPython tuning: 3.162514679816068 [######----] 65.00%
spotPython tuning: 3.1189615135325983 [#######---] 70.00%
spotPython tuning: 2.6597698275013 [########--] 75.00%
spotPython tuning: 0.3984917773445744 [########--] 80.00%
spotPython tuning: 0.3984917773445744 [########--] 85.00%
spotPython tuning: 0.3984917773445744 [#########-] 90.00%
spotPython tuning: 0.3984917773445744 [##########] 95.00%
spotPython tuning: 0.3984917773445744 [##########] 100.00% Done...

min y: 0.3984917773445744
x0: 3.1378600823045257
x1: 2.3010973936899863
```

4.5.3 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.Spot(fun=fun,
                    fun_control=fun_control,
                    optimizer=shgo,
                    surrogate_control=surrogate_control)
spot_sh.run()
spot_sh.print_results()
spot_sh.plot_progress(log_y=True)
spot_sh.surrogate.plot()
```

```
spotPython tuning: 3.8004552384813834 [######----] 55.00%
spotPython tuning: 3.8004552384813834 [######----] 60.00%
spotPython tuning: 3.1590504084857294 [######----] 65.00%
spotPython tuning: 3.1341080537914 [#######---] 70.00%
spotPython tuning: 2.8853849830561646 [########--] 75.00%
spotPython tuning: 0.4239413355798014 [########--] 80.00%
spotPython tuning: 0.4016765366794104 [########--] 85.00%
spotPython tuning: 0.3993233052368623 [#########-] 90.00%
spotPython tuning: 0.3993233052368623 [##########] 95.00%
spotPython tuning: 0.3993233052368623 [##########] 100.00% Done...

min y: 0.3993233052368623
x0: 3.1514683455618786
x1: 2.2984189502295536
```

4.5.4 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.Spot(fun=fun,
                    fun_control=fun_control,
                    optimizer=basinhopping,
                    surrogate_control=surrogate_control)
spot_bh.run()
spot_bh.print_results()
spot_bh.plot_progress(log_y=True)
spot_bh.surrogate.plot()
```

```
spotPython tuning: 3.80045375093536 [######----] 55.00%
spotPython tuning: 3.80045375093536 [######----] 60.00%
spotPython tuning: 3.159009257538889 [######----] 65.00%
spotPython tuning: 3.1341512916720102 [#######---] 70.00%
spotPython tuning: 2.8796407604155867 [########--] 75.00%
spotPython tuning: 0.414633458827506 [########--] 80.00%
spotPython tuning: 0.40117926479755717 [########--] 85.00%
spotPython tuning: 0.3993792812618935 [#########-] 90.00%
spotPython tuning: 0.3993792812618935 [##########] 95.00%
spotPython tuning: 0.3993792812618935 [##########] 100.00% Done...

min y: 0.3993792812618935
x0: 3.150848989866496
x1: 2.3006645011798197
```

4.5.5 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?

4.6 Jupyter Notebook

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

Part II

Numerical Methods

5 Introduction: Numerical 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. 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

5.1 Response Surface Methods: 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 $(\xi_1.\xi_2)$

5.1.1 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

5.1.2 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

5.1.3 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

5.1.4 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 $\eta = f(x_1, x_2, \dots, x_m)$

5.1.5 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**

5.2 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: $50 + 8x_1 + 3x_2$')
```

5.2.1 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 $\eta = 50 + 8x_1 + 3x_2 - 4x_1 x_2$

5.2.2 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')
```

5.2.3 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

5.3 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 $(0.6,0.2)$')
```

5.3.1 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

5.3.2 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')
```

5.3.3 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

5.3.4 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: $\\eta = 80 + 4x_1 + 8x_2 - 3x_1^2 - 12x_2^2 - 12x_1x_2$')
```

5.3.5 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.

5.3.6 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.

5.3.7 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: $\\eta = 80 + 4x_1 + 8x_2 - 2x_2^2 - 12x_1x_2$')
```

5.3.8 Interpretation: Saddle Points

- Likely further data collection, and/or outside expertise, is needed before determining a course of action in this situation

5.3.9 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

5.4 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

$$\eta = \beta_0 + \sum_{j=1}^{m} + \sum_{j=1}^{m} x_j^2 + \sum_{j=2}^{m} \sum_{k=1}^{j} \beta_{kj} x_k x_j.$$

5.4.1 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

5.5 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

5.5.1 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

5.6 RSM Experimentation

5.6.1 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

5.6.2 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

5.6.3 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

5.7 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

5.7.1 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

5.7.2 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

5.7.3 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

5.7.4 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

5.8 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.

5.9 Jupyter Notebook

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

6 Kriging (Gaussian Process Regression)

6.1 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).

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.

6.2 Background: Expectation, Mean, Standard Deviation

The distribution of a random vector is characterized by some indexes. One of them is the expected value, which is defined as

$$E[X] = \sum_{x \in D_X} x p_X(x) \qquad \text{if } X \text{ is discrete}$$

$$E[X] = \int_{x \in D_X} x f_X(x) \mathrm{d}x \quad \text{if } X \text{ is continuous.}$$

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: Expectation**
>
> Let X denote the number produced by rolling a fair die. Then
>
> $$E(X) = 1 \times 1/6 + 2 \times 1/6 + 3 \times 1/6 + 4 \times 1/6 + 5 \times 1/6 + 6 \times 1/6 = 3.5.$$

6.2.1 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

$$\overline{x} = \frac{1}{n} \sum_i x_i.$$

6.2.2 Variance and Standard Deviation

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:

The variance of a random variable X is the mean squared deviation of X from its expected value $\mu = E(X)$.

$$Var(X) = E[(X - \mu)^2]. \tag{6.1}$$

6.2.3 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.

$$sd(X) = \sqrt{Var(X)}. \tag{6.2}$$

6.2.4 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

$$\sqrt{\frac{1}{N}\sum_i (x_i - \bar{x})^2} \qquad \text{with} \qquad \bar{x} = \sum_{i=1}^{N} x_i/N. \tag{6.3}$$

Example: 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)
```

```
0.816496580927726
```

6.2.5 The Empirical Standard Deviation

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 as follows:

```
np.std(a, ddof=1)
```

```
1.0
```

6.2.6 The Argument "axis"

> **Axes along which the standard deviation is computed**
>
> - 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.

```
A = np.array([[1, 2], [3, 4]])
A
```

```
array([[1, 2],
       [3, 4]])
```

```
np.std(A)
```

```
1.118033988749895
```

```
np.std(A, axis=0)
```

```
array([1., 1.])
```

```
np.std(A, axis=1)
```

```
array([0.5, 0.5])
```

6.3 Data Types and Precision in Python

We consider single versus double precision in Python. In single precision, `std()` can be inaccurate:

```python
a = np.zeros((2, 4*4), dtype=np.float32)
a[0, :] = 1.0
a[1, :] = 0.1
a
```

```
array([[1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. ,
        1. , 1. , 1. ],
       [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,
        0.1, 0.1, 0.1]], dtype=float32)
```

```python
np.std(a, axis=0)
```

```
array([0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45,
       0.45, 0.45, 0.45, 0.45, 0.45], dtype=float32)
```

```python
np.std(a, axis=1)
```

```
array([0., 0.], dtype=float32)
```

```python
abs(0.45 - np.std(a))
```

```
1.7881393421514957e-08
```

> **Float data types**
>
> - float32 and float64 are data types in numpy that specify the precision of floating point numbers.
> - float32 is a single-precision floating point number that occupies 32 bits of memory. It has a precision of about 7 decimal digits.
> - float64 is a double-precision floating point number that occupies 64 bits of memory. It has a precision of about 15 decimal digits.
> - The main difference between float32 and float64 is the precision and memory usage. float64 provides a higher precision but uses more memory, while float32 uses less memory but has a lower precision.

Computing the standard deviation in float64 is more accurate (result may vary), see https://numpy.org/devdocs/reference/generated/numpy.std.html.

```
abs(0.45 - np.std(a, dtype=np.float64))
```

```
7.450580707946131e-10
```

Example: 32 versus 64 bit

```
import numpy as np

# Define a number
num = 0.123456789123456789

# Convert to float32 and float64
num_float32 = np.float32(num)
num_float64 = np.float64(num)

# Print the number in both formats
print("float32: ", num_float32)
print("float64: ", num_float64)
```

```
float32:  0.12345679
float64:  0.12345678912345678
```

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 float32 shows 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.

6.4 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.

6.4.1 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()
```

6.4.2 The Normal Distribution

The probability density function of the normal distribution is defined as:

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right), \tag{6.4}$$

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.

```
# generate 10 random numbers between from a normal distribution
import numpy as np
rng = np.random.default_rng()
n = 10
mu, sigma = 2, 0.1
x = rng.normal(mu, sigma, n)
x
```

```
array([2.07329406, 1.86483175, 1.95888477, 1.8291571 , 2.15938261,
       2.11163324, 2.03246283, 2.13081809, 2.11669928, 2.17015707])
```

Verify the mean:

```
abs(mu - np.mean(x))
```

```
0.04473207903238707
```

Note: To verify the standard deviation, we use `ddof = 1` (empirical standard deviation):

```
abs(sigma - np.std(x, ddof=1))
```

```
0.021616335622827346
```

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.

```
plot_normal_distribution(mu=0, sigma=1, num_samples=10000)
```

6.4.3 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.

6.4.4 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. 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 $sd(X^*) = 1$.

6.4.5 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.

For example, 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's a Python example:

```
import numpy as np

# Define the parameters of the normal distribution
mu = 0
sigma = 1

# Draw 10 samples (realizations) from the normal distribution
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 10 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.

6.4.6 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 k-dimensional random vectors $X = (X_1, X_2, \dots, X_k)$. When drawing samples from this distribution, it results in a set of values represented as $\{x_1, x_2, \dots, x_k\}$. 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 6.4.

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.

> **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:

$$\operatorname{cov}(X, Y) = \mathrm{E}\left[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])\right]$$

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. 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 $k \times 1$ 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. For discrete random variables, covariance can be written as:

$$\operatorname{cov}(X, Y) = \frac{1}{n} \sum_{i=1}^{n} (x_i - E(X))(y_i - E(Y)).$$

Figure 6.1 shows draws from a bivariate normal distribution with $\mu = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and $\Sigma = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$.

```
import numpy as np
rng = np.random.default_rng()
import matplotlib.pyplot as plt
mean = [0, 0]
cov = [[9, 4], [4, 9]]  # diagonal covariance
x, y = rng.multivariate_normal(mean, cov, 1000).T
# Create a scatter plot of the numbers
plt.scatter(x, y, s=2)
plt.axis('equal')
plt.grid()
plt.title(f"Bivariate Normal. Mean zero and positive covariance: {cov}")
plt.show()
```

Figure 6.1: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$

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 6.1, the variances are identical and the variables are correlated (covariance is 4), so the distribution is stretched along the line $y = x$.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

# Parameters
```

```
mu = np.array([0, 0])
cov = np.array([[9, 4], [4, 9]])

# Create grid and multivariate normal
x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
X, Y = np.meshgrid(x,y)
pos = np.empty(X.shape + (2,))
pos[:, :, 0] = X; pos[:, :, 1] = Y
rv = multivariate_normal(mu, cov)

fig = plt.figure()
ax = plt.axes(projection='3d')
surf=ax.plot_surface(X, Y, rv.pdf(pos),cmap='viridis',linewidth=0)
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
ax.set_title('Bivariate Normal Distribution')
fig.colorbar(surf, shrink=0.5, aspect=10)
plt.show()
```

Figure 6.2: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$

6.4.7 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 6.3: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & 0 \\ 0 & 9 \end{pmatrix}$

6.4.8 The Bivariate Normal Distribution with Mean Zero and Negative Covariances $\sigma_{12} = \sigma_{21} = -4$

$$\Sigma = \begin{pmatrix} 9 & -4 \\ -4 & 9 \end{pmatrix}$$

6.5 Cholesky Decomposition and Positive Definite Matrices

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

Figure 6.4: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & -4 \\ -4 & 9 \end{pmatrix}$

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.

```
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
```

More effficent (and check if symmetric) is based on Cholesky decomposition.

```
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
```

Example: Cholesky decomposition.

`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.]]
```

6.6 Maximum Likelihood Estimation: 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., $X^{(j)} \sim N(\mu, \Sigma)$, $j = 1, 2, \ldots$. 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 $x_1, \ldots x_n$, i.e., the realizations of the first n random vectors in the sequence, to estimate the two unknown parameters μ and Σ.

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 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:

$$\begin{aligned} L(\mu, \Sigma) &= \prod_{j=1}^{n} f_X(x_j) = \prod_{j=1}^{n} \frac{1}{\sqrt{(2\pi)^k \det(\Sigma)}} \exp\left(-\frac{1}{2}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu)\right) \\ &= \frac{1}{(2\pi)^{nk/2} \det(\Sigma)^{n/2}} \exp\left(-\frac{1}{2}\sum_{j=1}^{n}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu)\right). \end{aligned}$$

The log-likelihood function is

$$\ell(\mu, \Sigma) = -\frac{nk}{2}\ln(2\pi) - \frac{n}{2}\ln(\det(\Sigma)) - \frac{1}{2}\sum_{j=1}^{n}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu).$$

The likelihood function is well-defined only if $\det(\Sigma) > 0$.

6.7 Introduction to Gaussian Processes

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.

6.7.1 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.

6.7.2 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 6.5.

```
visualize_inverse_exp_squared_distance(5, 0.0, [0.5, 1, 2.0])
```

Figure 6.5: 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

$$\Sigma(\vec{x}, \vec{x}') = \exp\{-||\vec{x} - \vec{x}'||^2/(2\sigma^2)\}. \tag{6.5}$$

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.

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 $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n$. 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.

6.7.3 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, mat, spacing, sqr
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.   ],
       [0.995, 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.   ],
       [0.98 , 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()
```

6.7.4 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").resh
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 6.6: 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(sigma
plt.show()
```

Figure 6.7: Realization of three random functions under a GP prior. sigma2: 1.0

6.7.5 Properties of the 1d Example

6.7.5.1 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.

6.7.5.2 Smoothness:

The function plotted in Figure 6.6 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.

6.7.5.3 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, mat, spacing, sqrt
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
    if show:
        plt.show()
```

```
plot_mvn(a=0, b=10, sigma2=10.0, size=3, n=250)
```

Figure 6.8: Realization of Random Functions under a GP prior. sigma2: 10

```
plot_mvn(a=0, b=10, sigma2=0.1, size=3, n=250)
```

Figure 6.9: Realization of Random Functions under a GP prior. sigma2: 0.1

6.8 Kriging: Modeling Basics

6.8.1 The Kriging Idea in a Nutshell

We consider observed data of an unknown function f at n points $x_1, \dots, x_n$, see Figure 6.10. These measurements a considered as realizations of MVN random variables $Y_1, \dots, Y_n$ with mean μ and covariance matrix Σ_n as shown in Figure 6.7, Figure 6.8 or Figure 6.9. In Kriging, a more general covariance matrix (or equivalently, a correlation matrix Ψ) is used, see Equation 6.6. Using a maximum likelihood approach, we can estimate the unknown parameters μ and Σ_n from the data so that the likelihood function is maximized.

6.8.2 The Kriging Basis Function

k-dimensional basis functions of the form

$$\psi(\vec{x}^{(i)}, \vec{x}^{(j)}) = \exp\left(-\sum_{l=1}^{k} \theta_l |x_l^{(i)} - x_l^{(j)}|^{p_l}\right) \tag{6.6}$$

Figure 6.10: Eight measurements of an unknown function

are used in a method known as Kriging. Note, $\vec{x}^{(i)}$ denotes the k-dim vector $\vec{x}^{(i)} = (x_1^{(i)}, \dots, x_k^{(i)})^T$.

The Kriging basis function is related to the 1-dim Gaussian basis function (Equation 6.5), which is defined as

$$\Sigma(\vec{x}^{(i)}, \vec{x}^{(j)}) = \exp\{-||\vec{x}^{(i)} - \vec{x}^{(j)}||^2/(2\sigma^2)\}. \tag{6.7}$$

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).

6.8.3 The Correlation Coefficient

In a bivariate normal distribution, the covariance matrix and the correlation coefficient are closely related. The covariance matrix Σ for a bivariate normal distribution is a 2×2 matrix that looks like this:

$$\Sigma = \begin{pmatrix} \sigma_1^2 & \sigma_{12} \\ \sigma_{21} & \sigma_2^2 \end{pmatrix},$$

where σ_1^2 and σ_2^2 are the variances of X_1 and X_2, and $\sigma_{12} = \sigma_{21}$ is the covariance between X_1 and X_2.

The correlation coefficient, often denoted as ρ, is a normalized measure of the linear relationship between two variables. It is calculated from the covariance and the standard deviations σ_1 and σ_2 (or the square roots of the variances) of X_1 and X_2 as follows:

$$\rho = \sigma_{12}/(\sqrt{\sigma_1^2} \times \sqrt{\sigma_2^2}) = \sigma_{12}/(\sigma_1 \times \sigma_2).$$

So we can express the correlation coefficient ρ in terms of the elements of the covariance matrix Σ. It can be interpreted as follows: The correlation coefficient ranges from -1 to 1. A value of 1 means that X_1 and X_2 are perfectly positively correlated, a value of -1 means they are perfectly negatively correlated, and a value of 0 means they are uncorrelated. This gives the same information as the covariance, but on a standardized scale that does not depend on the units of X_1 and X_2.

6.8.4 Covariance Matrix and Correlation Matrix

Covariance and Correlation (taken from @Forr08a)

Covariance is a measure of the correlation between two or more sets of random variables.

$$\mathrm{Cov}(X, Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y]$$

From the covariance, we can derive the correlation

$$\mathrm{Corr}(X, Y) = \frac{\mathrm{Cov}(X, Y)}{\sqrt{\mathrm{Var}(X)\mathrm{Var}(Y)}} = \frac{\mathrm{Cov}(X, Y)}{\sigma_X \sigma_Y}. \tag{6.8}$$

For a vector of random variables

$$Y = \left((Y^{(l)}, \dots, Y^{(n)})\right)^T$$

the covariance matrix is a matrix of covariances between the random variables

$$\Sigma = \mathrm{Cov}(Y, Y) = \begin{pmatrix} \mathrm{Cov}(Y^{(1)}, Y^{(1)}) & \dots & \mathrm{Cov}(Y^{(1)}, Y^{(n)}) \\ \vdots & \ddots & \vdots \\ \mathrm{Cov}(Y^{(n)}, Y^{(1)}) & \dots & \mathrm{Cov}(Y^{(n)}, Y^{(n)}) \end{pmatrix},$$

and from Equation 6.8

$$\mathrm{Cov}(Y, Y) = \sigma_Y^2 \mathrm{Cor}(Y, Y).$$

You can compute the correlation matrix Ψ from a covariance matrix Σ in Python using the numpy library. The correlation matrix is computed by dividing each element of the covariance matrix by the product of the standard deviations of the corresponding variables.

The function `covariance_to_correlation` first computes the standard deviations of the variables with `np.sqrt(np.diag(cov))`. It then computes the correlation matrix by dividing each element of the covariance matrix by the product of the standard deviations of the corresponding variables with `cov / np.outer(std_devs, std_devs)`.

```
import numpy as np

def covariance_to_correlation(cov):
    # Compute standard deviations
    std_devs = np.sqrt(np.diag(cov))

    # Compute correlation matrix
    corr = cov / np.outer(std_devs, std_devs)

    return corr

cov = np.array([[9, -4], [-4, 9]])
print(covariance_to_correlation(cov))
```

```
[[ 1.         -0.44444444]
 [-0.44444444  1.        ]]
```

6.8.5 The Kriging Model

Consider sample data $\vec{X}$ and $\vec{y}$ from n locations that are available in matrix form: $\vec{X}$ is a $(n \times k)$ matrix, where k denotes the problem dimension and $\vec{y}$ is a $(n \times 1)$ vector.

The observed responses $\vec{y}$ are considered as if they are from a stochastic process, which will be denoted as

$$\begin{pmatrix} \vec{Y}(\vec{x}^{(1)}) \\ \vdots \\ \vec{Y}(\vec{x}^{(n)}) \end{pmatrix}.$$

The set of random vectors (also referred to as a *random field*) has a mean of $\vec{1}\mu$, which is a $(n \times 1)$ vector.

6.8.6 Correlations

The random vectors are correlated with each other using the basis function expression from Equation 6.6:

$$\text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) = \exp\left\{-\sum_{j=1}^{k} \theta_j |x_j^{(i)} - x_j^{(l)}|^{p_j}\right\}.$$

The $(n \times n)$ correlation matrix of the observed sample data is

$$\vec{\Psi} = \begin{pmatrix} \text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) & \dots & \text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) \\ \vdots & \vdots & \vdots \\ \text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) & \dots & \text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) \end{pmatrix}.$$

Note: correlations depend on the absolute distances between sample points $|x_j^{(n)} - x_j^{(n)}|$ and the parameters p_j and θ_j.

Correlation is intuitive, because when two points move close together, then $|x_l^{(i)} - x_l| \to 0$ and $\exp(-|x_l^{(i)} - x_l| \to 1$, points show very close correlation and $Y(x_l^{(i)}) = Y(x_l)$.

θ can be seen as a width parameter:

- 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 6.11.

```
visualize_inverse_exp_squared_distance(5, 0, theta_values=[0.5, 1, 2.0])
```

Figure 6.11: Theta set to 1/2, 1, and 2

Example: 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:

$$D_1 = \begin{pmatrix} x_{11} - x_{11} & x_{11} - x_{21} & x_{11} - x_{31} & x_{11} - x_{41} \\ x_{21} - x_{11} & x_{21} - x_{21} & x_{21} - x_{31} & x_{21} - x_{41} \\ x_{31} - x_{11} & x_{31} - x_{21} & x_{31} - x_{31} & x_{31} - x_{41} \\ x_{41} - x_{11} & x_{41} - x_{21} & x_{41} - x_{31} & x_{41} - x_{41} \end{pmatrix}$$

$$D_2 = \begin{pmatrix} x_{12} - x_{12} & x_{12} - x_{22} & x_{12} - x_{32} & x_{12} - x_{42} \\ x_{22} - x_{12} & x_{22} - x_{22} & x_{22} - x_{32} & x_{22} - x_{42} \\ x_{32} - x_{12} & x_{32} - x_{22} & x_{32} - x_{32} & x_{32} - x_{42} \\ x_{42} - x_{12} & x_{42} - x_{22} & x_{42} - x_{32} & x_{42} - x_{42} \end{pmatrix}$$

$$D_3 = \begin{pmatrix} x_{13} - x_{13} & x_{13} - x_{23} & x_{13} - x_{33} & x_{13} - x_{43} \\ x_{23} - x_{13} & x_{23} - x_{23} & x_{23} - x_{33} & x_{23} - x_{43} \\ x_{33} - x_{13} & x_{33} - x_{23} & x_{33} - x_{33} & x_{33} - x_{43} \\ x_{43} - x_{13} & x_{43} - x_{23} & x_{43} - x_{33} & x_{43} - x_{43} \end{pmatrix}$$

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:

$$\begin{pmatrix} 0 & \theta_1(x_{11} - x_{21})^2 + \theta_2(x_{12} - x_{22})^2 + \theta_3(x_{13} - x_{23})^2 & \theta_1(x_{11} - x_{31})^2 + \theta_2(x_{12} - x_{32})^2 + \theta_3(x_{13} - x_{33})^2 & \theta_1(\\ 0 & 0 & \theta_1(x_{21} - x_{31})^2 + \theta_2(x_{22} - x_{32})^2 + \theta_3(x_{23} - x_{33})^2 & \theta_1 \\ 0 & 0 & 0 & \theta_1(\\ 0 & 0 & 0 & \end{pmatrix}$$

Finally,

$$\Psi = \exp(-D)$$

is computed.
Next, we will demonstrate how this computation can be implemented in Python.

```
from numpy import (array, zeros, power, ones, exp, multiply,
                   eye, linspace, mat, spacing, sqrt, arange,
                   append, ravel)
from numpy.linalg import cholesky, solve
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: The Correlation Matrix (Using Existing Functions)

The same result as computed in the previous example can be obtained with existing python functions, e.g., from the package `scipy`.

```
from scipy.spatial.distance import squareform
from scipy.spatial.distance import pdist

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.        ]])
```

6.8.7 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:  200000000.53159264
```

```
np.linalg.cond(Psi)
```

```
2.163953413738652
```

6.8.8 MLE to estimate θ and p

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 $\vec{y}$, which can be expressed in terms of the sample data

$$L\left(\vec{Y}(\vec{x}^{(1)}), \ldots, \vec{Y}(\vec{x}^{(n)})|\mu, \sigma\right) = \frac{1}{(2\pi\sigma)^{n/2}|\vec{\Psi}|^{1/2}} \exp\left\{\frac{-(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{2\sigma^2}\right\},$$

and formulated as the log-likelihood:

$$\ln(L) = -\frac{n}{2}\ln(2\pi\sigma) - \frac{1}{2}\ln|\vec{\Psi}|\frac{-(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{2\sigma^2}.$$

Optimization of the log-likelihood by taking derivatives with respect to μ and σ results in

$$\hat{\mu} = \frac{\vec{1}^T\vec{\Psi}^{-1}\vec{y}^T}{\vec{1}^T\vec{\Psi}^{-1}\vec{1}^T}$$

and

$$\hat{\sigma} = \frac{(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{n}.$$

Combining the equations leads to the concentrated log-likelihood:

$$\ln(L) = -\frac{n}{2}\ln(\hat{\sigma}) - \frac{1}{2}\ln|\vec{\Psi}|. \tag{6.9}$$

> **Note: The Concentrated Log-Likelihood**
>
> - The first term in Equation 6.9 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 equation is not differentiable.

6.8.9 Tuning 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.

6.9 Kriging Prediction

6.9.1 The Augmented Correlation Matrix

We will use the Kriging correlation Ψ to predict new values based on the observed data. The matrix algebra involved for calculating the likelihood is the most computationally intensive part of the Kriging process. Care must be taken that the computer code is as efficient as possible.

Basic elements of the Kriging based surrogate optimization such as interpolation, expected improvement, and regression are presented. 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 $\vec{X}$ 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

$$\vec{\psi} = \begin{pmatrix} \operatorname{cor}\left(\vec{Y}(\vec{x}^{(1)}), \vec{Y}(\vec{x})\right) \\ \vdots \\ \operatorname{cor}\left(\vec{Y}(\vec{x}^{(n)}), \vec{Y}(\vec{x})\right) \end{pmatrix} = \begin{pmatrix} \vec{\psi}^{(1)} \\ \vdots \\ \vec{\psi}^{(n)} \end{pmatrix}.$$

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

$$\ln(L) = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln(\hat{\sigma}^2) - \frac{1}{2}\ln|\vec{\hat{\Psi}}| - \frac{(\vec{\tilde{y}} - \vec{1}\hat{\mu})^T \vec{\tilde{\Psi}}^{-1} (\vec{\tilde{y}} - \vec{1}\hat{\mu})}{2\hat{\sigma}^2}.$$

The MLE for $\hat{y}$ can be calculated as

$$\hat{y}(\vec{x}) = \hat{\mu} + \vec{\psi}^T \vec{\tilde{\Psi}}^{-1} (\vec{y} - \vec{1}\hat{\mu}). \tag{6.10}$$

6.9.2 Properties of the Predictor

Equation 6.10 reveals two important properties of the Kriging predictor:

1. 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 $\vec{w} = \vec{\tilde{\Psi}}^{(-1)}(\vec{y} - \vec{1}\hat{\mu})$.
2. 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, $\hat{y}(\vec{x}^{(i)}) = y^{(i)}$.

6.10 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.

6.10.1 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)
# theta should be an array (of one value, for the moment, will be changed later)
theta = np.array([1.0])
Psi = build_Psi(X, theta)
```

Evaluate at sample points

```
y = np.sin(X)
```

```
import matplotlib.pyplot as plt
plt.plot(X, y, "bo")
plt.title(f"Sin(x) evaluated at {n} points")
plt.show()
```

6.10.2 Computing the ψ Vector

Distances between testing locations x and training data locations X.

```
from scipy.spatial.distance import cdist

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)
    print(D.shape)
    psi = exp(-D)
    # return psi transpose to be consistent with the literature
    return(psi.T)
```

6.10.3 Predicting at New Locations

We would like to predict at $m = 100$ new 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)
psi = build_psi(X, x, theta)
```

```
(100, 8)
```

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)))
```

To compute f, Equation 6.10 is used.

6.10.4 Visualization

```
import matplotlib.pyplot as plt
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()
```

6.11 Cholesky Example With Two Points

6.11.1 Cholesky Decomposition

We consider $k = 1$ 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 6.6, we can compute the correlation matrix Ψ:

$$\Psi = \begin{pmatrix} 1 & e^{-1} \\ e^{-1} & 1 \end{pmatrix}.$$

To determine MLE as in Equation 6.10, 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:

$$LDL^T = \begin{pmatrix} 1 & 0 \\ l_{21} & 1 \end{pmatrix} \begin{pmatrix} d_{11} & 0 \\ 0 & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix} =$$

$$\begin{pmatrix} d_{11} & 0 \\ d_{11}l_{21} & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} d_{11} & d_{11}l_{21} \\ d_{11}l_{21} & d_{11}l_{21}^2 + d_{22} \end{pmatrix}. \tag{6.11}$$

Using Equation 6.11, 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

$$\Psi = LDL^T = U^T DU.$$

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

$$\Psi = \begin{pmatrix} 1 & 0 \\ e^{-1} & \sqrt{1 - e^{-2}} \end{pmatrix} \begin{pmatrix} 1 & e^{-1} \\ 0 & \sqrt{1 - e^{-2}} \end{pmatrix} = U^T U.$$

6.11.2 Computation of the Inverse Matrix

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]]
```

6.12 Jupyter Notebook

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

7 Introduction to spotPython

Surrogate model based optimization methods are common approaches in simulation and optimization. SPOT was developed because there is a great need for sound statistical analysis of simulation and optimization algorithms. 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
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. Goto 3.

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.

7.1 Example: `Spot` and the Sphere Function

```
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
```

7.1.1 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()
```

7.1.2 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.Spot(fun=fun,
                   fun_control=fun_control)
spot_0.run()
```

```
spotPython tuning: 1.2026789271012512e-09 [#######---] 73.33%
spotPython tuning: 1.2026789271012512e-09 [########--] 80.00%
spotPython tuning: 1.2026789271012512e-09 [#########-] 86.67%
spotPython tuning: 1.2026789271012512e-09 [#########-] 93.33%
spotPython tuning: 3.7010904275056666e-10 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2ccd03f90>
```

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: 3.7010904275056666e-10
x0: 1.9238218284201025e-05
```

```
[['x0', 1.9238218284201025e-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 7.1: 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 7.2: Visualization of the model and the underlying objective function values.

7.2 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.Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
spot_1.run()
```

```
spotPython tuning: 1.2031167009156832e-09 [##########] 100.00% Done...
```

7.3 Print the Results

```
spot_1.print_results()
```

```
min y: 1.2031167009156832e-09
x0: -3.468597268227724e-05

[['x0', -3.468597268227724e-05]]
```

7.4 Show the Progress

```
spot_1.plot_progress()
```

7.5 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]))
design_control = design_control_init(init_size=5)
```

```
Created spot_tensorboard_path: runs/spot_logs/01_p040025_2024-02-24_00-20-41 for SummaryWrite
```

Since the `PREFIX` is not None, `spotPython` will log the optimization process in the TensorBoard files.

```
spot_tuner = spot.Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
spot_tuner.run()
spot_tuner.print_results()
```

```
spotPython tuning: 2.7705278094872058e-05 [####------] 40.00%
spotPython tuning: 8.061545220547415e-07 [#####-----] 46.67%
spotPython tuning: 7.385022589686283e-07 [#####-----] 53.33%
spotPython tuning: 3.677917685242894e-07 [######----] 60.00%
spotPython tuning: 4.911502304103013e-09 [#######---] 66.67%
spotPython tuning: 4.911502304103013e-09 [#######---] 73.33%
spotPython tuning: 4.911502304103013e-09 [########--] 80.00%
spotPython tuning: 4.911502304103013e-09 [#########-] 86.67%
spotPython tuning: 4.911502304103013e-09 [#########-] 93.33%
spotPython tuning: 4.911502304103013e-09 [##########] 100.00% Done...

min y: 4.911502304103013e-09
x0: -7.008211115615035e-05

[['x0', -7.008211115615035e-05]]
```

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 7.3: TensorBoard visualization of the spotPython process.

7.6 Jupyter Notebook

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

8 Multi-dimensional Functions

This chapter illustrates how high-dimensional functions can be optimized and analyzed.

8.1 Example: Spot and the 3-dim Sphere Function

```
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
```

8.1.1 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:

$$f(x) = \sum_{i}^{k} x_i^2.$$

It is avaliable 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 7.5), where only one `theta` value was used, we will use three different `theta` values (one for each dimension), i.e., we set `n_theta=3` in the `surrogate_control`. 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 interpreable 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"],
              show_progress=True)
surrogate_control = surrogate_control_init(n_theta=3)
spot_3 = spot.Spot(fun=fun,
                  fun_control=fun_control,
                  surrogate_control=surrogate_control)
spot_3.run()
```

```
Created spot_tensorboard_path: runs/spot_logs/03_p040025_2024-02-24_00-21-01 for SummaryWrite
spotPython tuning: 0.03443324167631616 [#######---] 73.33%
spotPython tuning: 0.03134655155643102 [########--] 80.00%
spotPython tuning: 0.0009630181526749273 [#########-] 86.67%
spotPython tuning: 8.570154459856623e-05 [#########-] 93.33%
spotPython tuning: 6.496172516667557e-05 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2b92c8c90>
```

> **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/
> ```

8.1.2 Results

```
_ = spot_3.print_results()
```

```
min y: 6.496172516667557e-05
Pressure: 0.005280070995399376
Temp: 0.0019490323308060742
Lambda: 0.005769215581315232
```

```
spot_3.plot_progress()
```

8.1.3 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:  95.79368533570627
Temp:  99.99999999999999
Lambda:  87.19542775477797
```

```
spot_3.plot_importance()
```

8.1.4 TensorBoard

Figure 8.1: TensorBoard visualization of the spotPython process. Objective function values plotted against wall time.

The second TensorBoard visualization shows the input values, i.e., $x_0, \ldots, x_2$, 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 8.2: TensorBoard visualization of the spotPython surrogate model.

8.2 Conclusion

Based on this quick analysis, we can conclude that all three dimensions are equally important (as expected, because the analytical function is known).

8.3 Exercises

8.3.1 1. The Three Dimensional `fun_cubed`

- The input dimension is `3`. The search range is $-1 \leq x \leq 1$ 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?

8.3.2 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?

8.3.3 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?

8.3.4 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?

8.3.5 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?

8.4 Selected Solutions

8.4.1 Solution to Exercise Section 8.3.5: 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
```

8.4.1.1 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), i.e., we set `n_theta=3` in the `surrogate_control`. 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),
              show_progress=True,
              fun_evals=25)
surrogate_control = surrogate_control_init(n_theta=2)
spot_rosen = spot.Spot(fun=fun_rosen,
                  fun_control=fun_control,
                  surrogate_control=surrogate_control)
spot_rosen.run()
```

```
Created spot_tensorboard_path: runs/spot_logs/ROSEN_p040025_2024-02-24_00-21-03 for SummaryWr
spotPython tuning: 90.7801015955818 [####------] 44.00%
spotPython tuning: 1.0172832635943474 [#####-----] 48.00%
spotPython tuning: 1.0172832635943474 [#####-----] 52.00%
spotPython tuning: 1.0172832635943474 [######----] 56.00%
spotPython tuning: 1.0172832635943474 [######----] 60.00%
spotPython tuning: 1.0172832635943474 [######----] 64.00%
spotPython tuning: 1.0172832635943474 [#######---] 68.00%
spotPython tuning: 1.0172832635943474 [#######---] 72.00%
spotPython tuning: 1.0172832635943474 [########--] 76.00%
spotPython tuning: 1.0172832635943474 [########--] 80.00%
spotPython tuning: 0.9921822630967522 [########--] 84.00%
spotPython tuning: 0.7147779101762312 [#########-] 88.00%
spotPython tuning: 0.7147779101762312 [#########-] 92.00%
spotPython tuning: 0.7147779101762312 [##########] 96.00%
spotPython tuning: 0.7147779101762312 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2babc3950>
```

> **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/
> ```

8.4.1.2 Results

```
_ = spot_rosen.print_results()
```

```
min y: 0.7147779101762312
x0: 0.19951670458655138
x1: 0.1258327277797004
```

```
spot_rosen.plot_progress(log_y=True)
```

8.4.1.3 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:  100.0
x1:  1.2641431841859785
```

```
spot_rosen.plot_importance()
```

8.4.1.4 TensorBoard

TBD

8.5 Jupyter Notebook

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

9 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.

9.1 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"
```

9.1.1 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]))
```

```
Created spot_tensorboard_path: runs/spot_logs/003_p040025_2024-02-24_00-21-26 for SummaryWrit
```

Although the default `spot` surrogate model is an isotropic Kriging model, we will explicitly set the `n_theta` parameter to a value of `1`, 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(n_theta=1)
```

```
spot_2 = spot.Spot(fun=fun,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)

spot_2.run()
```

```
spotPython tuning: 1.801603872454505e-05 [#######---] 73.33%
spotPython tuning: 1.801603872454505e-05 [########--] 80.00%
spotPython tuning: 1.801603872454505e-05 [#########-] 86.67%
spotPython tuning: 1.801603872454505e-05 [#########-] 93.33%
spotPython tuning: 1.801603872454505e-05 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2c54d6010>
```

9.1.2 Results

```
spot_2.print_results()
```

```
min y: 1.801603872454505e-05
x0: 0.0019077911677074135
x1: 0.003791618596979743

[['x0', 0.0019077911677074135], ['x1', 0.003791618596979743]]
```

```
spot_2.plot_progress(log_y=True)
```

9.2 Example With Anisotropic Kriging

As described in Section 9.1, the default parameter setting of `spotPython`'s Kriging surrogate uses the same `theta` value for every dimension. This is referred to as "using an isotropic kernel". If different `theta` values are used for each dimension, then an anisotropic kernel is used. To enable anisotropic models in `spotPython`, the number of `theta` values should be larger than one. We can use `surrogate_control=surrogate_control_init(n_theta=2)` to enable this behavior (2 is the problem dimension).

```
surrogate_control = surrogate_control_init(n_theta=2)
spot_2_anisotropic = spot.Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate_control=surrogate_control)
spot_2_anisotropic.run()
```

```
spotPython tuning: 1.783225688095949e-05 [#######---] 73.33%
spotPython tuning: 1.783225688095949e-05 [########--] 80.00%
spotPython tuning: 1.783225688095949e-05 [#########-] 86.67%
spotPython tuning: 3.0185289245739795e-06 [#########-] 93.33%
spotPython tuning: 3.0185289245739795e-06 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2c569c350>
```

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: 3.0185289245739795e-06
x0: -0.0001531610695200253
x1: -0.0017306272306182697

[['x0', -0.0001531610695200253], ['x1', -0.0017306272306182697]]
```

```
spot_2_anisotropic.surrogate.plot()
```

9.2.1 Taking a Look at the `theta` Values

9.2.1.1 `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([-0.29237522, -0.13253124])
```

- 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([-0.04189656])
```

9.2.1.2 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 9.1: TensorBoard visualization of the spotPython surrogate model.

9.3 Exercises

9.3.1 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)
```

9.3.2 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`.

9.3.3 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`.

9.3.4 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`.

9.3.5 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`.

9.4 Selected Solutions

9.4.1 Solution to Exercise Section 9.3.5: The Two-dimensional Rosenbrock Function `fun_rosen`

9.4.1.1 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(n_theta=1)
```

```
spot_rosen = spot.Spot(fun=fun_rosen,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)
spot_rosen.run()
```

```
Created spot_tensorboard_path: runs/spot_logs/ROSEN_p040025_2024-02-24_00-21-29 for SummaryW
spotPython tuning: 52.87631878551649 [#######---] 73.33%
spotPython tuning: 52.361867783356715 [########--] 80.00%
spotPython tuning: 52.361867783356715 [#########-] 86.67%
spotPython tuning: 43.44273941029301 [#########-] 93.33%
spotPython tuning: 12.275684138292505 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2ce60c5d0>
```

> **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/
> ```

9.4.1.1.1 Results

```
_ = spot_rosen.print_results()
```

```
min y: 12.275684138292505
x0: -2.3708459318321333
x1: 5.923082873674319
```

```
spot_rosen.plot_progress()
```

9.4.1.1.2 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.

9.4.1.1.3 TensorBoard

TBD

9.4.1.2 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.

We can also add interpreable 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)
surrogate_control = surrogate_control_init(n_theta=2)
spot_rosen = spot.Spot(fun=fun_rosen,
                  fun_control=fun_control,
                  surrogate_control=surrogate_control)
spot_rosen.run()
```

```
Created spot_tensorboard_path: runs/spot_logs/ROSEN_p040025_2024-02-24_00-21-30 for SummaryW
spotPython tuning: 90.7801015955818 [#######---] 73.33%
spotPython tuning: 1.0172832635943474 [########--] 80.00%
spotPython tuning: 1.0172832635943474 [#########-] 86.67%
spotPython tuning: 1.0172832635943474 [#########-] 93.33%
spotPython tuning: 1.0172832635943474 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2c5cd1ed0>
```

> **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/
> ```

9.4.1.2.1 Results

```
_ = spot_rosen.print_results()
```

```
min y: 1.0172832635943474
x0: 0.0028122200003174065
x1: -0.04784582169505708
```

```
spot_rosen.plot_progress()
```

9.4.1.2.2 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:  2.2276773313197755
```

```
spot_rosen.plot_importance()
```

9.4.1.2.3 TensorBoard

TBD

9.5 Jupyter Notebook

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

10 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
```

10.1 Example: Branin Function with spotPython's Internal Kriging Surrogate

10.1.1 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 7.5, 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)
```

```
Created spot_tensorboard_path: runs/spot_logs/04_p040025_2024-02-24_00-22-07 for SummaryWri
```

10.1.2 Running the surrogate model based optimizer `Spot`:

```
spot_2 = spot.Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
```

```
spot_2.run()
```

```
spotPython tuning: 3.146824136952164 [######----] 55.00%
spotPython tuning: 3.146824136952164 [######----] 60.00%
spotPython tuning: 3.146824136952164 [######----] 65.00%
spotPython tuning: 3.146824136952164 [#######---] 70.00%
spotPython tuning: 1.1487233101571483 [########--] 75.00%
spotPython tuning: 1.0236891516766402 [########--] 80.00%
spotPython tuning: 0.41994270072214057 [########--] 85.00%
spotPython tuning: 0.40193544341108023 [#########-] 90.00%
spotPython tuning: 0.3991519598268951 [##########] 95.00%
spotPython tuning: 0.3991519598268951 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2cc1d6890>
```

10.1.3 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 10.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

10.1.4 Print the Results

```
spot_2.print_results()
```

```
min y: 0.3991519598268951
x0: 3.1546575195040987
x1: 2.285931113926263

[['x0', 3.1546575195040987], ['x1', 2.285931113926263]]
```

10.1.5 Show the Progress and the Surrogate

```
spot_2.plot_progress(log_y=True)
```

```
spot_2.surrogate.plot()
```

10.2 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.build.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)
```

10.2.1 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.Spot(fun=fun,
                      fun_control=fun_control,
                      design_control=design_control,
                      surrogate = S_GP)
spot_2_GP.run()
```

```
spotPython tuning: 18.865121449825782 [######----] 55.00%
spotPython tuning: 4.06700305855078 [######----] 60.00%
spotPython tuning: 3.461906927549384 [######----] 65.00%
spotPython tuning: 3.461906927549384 [#######---] 70.00%
spotPython tuning: 1.3280944252046556 [########--] 75.00%
spotPython tuning: 0.9548334920645392 [########--] 80.00%
spotPython tuning: 0.9344485781421579 [########--] 85.00%
spotPython tuning: 0.39916716809341857 [#########-] 90.00%
spotPython tuning: 0.3982254000779708 [##########] 95.00%
spotPython tuning: 0.3982254000779708 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2ccd979d0>
```

```
spot_2_GP.plot_progress()
```

```
spot_2_GP.print_results()
```

```
min y: 0.3982254000779708
x0: 3.1499822680266343
x1: 2.268811272474469

[['x0', 3.1499822680266343], ['x1', 2.268811272474469]]
```

10.3 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.Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
spot_1.run()
```

```
spotPython tuning: 0.03475493366922229 [####------] 40.00%
spotPython tuning: 0.03475483461229862 [#####-----] 50.00%
spotPython tuning: 0.03475338954992179 [######----] 60.00%
spotPython tuning: 0.03437475313644103 [#######---] 70.00%
spotPython tuning: 0.015290217643803946 [########--] 80.00%
spotPython tuning: 0.0017932523576966073 [#########-] 90.00%
spotPython tuning: 8.771851669068651e-06 [##########] 100.00% Done...
```

fun_evals: 5. min_y: 0.034755
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 6. min_y: 0.034753
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 7. min_y: 0.034375
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 8. min_y: 0.01529
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 9. min_y: 0.001793
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 10. min_y: 9e-06
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

10.3.1 Results

```
spot_1.print_results()
```

```
min y: 8.771851669068651e-06
x0: 0.002961731194600322

[['x0', 0.002961731194600322]]
```

```
spot_1.plot_progress(log_y=True)
```

- The method `plot_model` plots the final surrogate:

```
spot_1.plot_model()
```

10.4 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.Spot(fun=fun,
                      fun_control=fun_control,
                      design_control=design_control,
                      surrogate = S_GP)
spot_1_GP.run()
```

```
spotPython tuning: 0.004925671374769521 [####------] 40.00%
spotPython tuning: 0.002612062924748803 [#####-----] 50.00%
spotPython tuning: 3.6666409852957783e-07 [######----] 60.00%
```

```
spotPython tuning: 4.638244203084832e-08 [#######---] 70.00%
spotPython tuning: 3.2711094860544125e-09 [########--] 80.00%
spotPython tuning: 2.2493573831304313e-10 [#########-] 90.00%
spotPython tuning: 2.2493573831304313e-10 [##########] 100.00% Done...
```

fun_evals: 6. min_y: 0.0
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 7. min_y: 0.0
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 8. min_y: 0.0
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 9. min_y: 0.0
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

```
spot_1_GP.print_results()
```

```
min y: 2.2493573831304313e-10
x0: 1.499785779079943e-05

[['x0', 1.499785779079943e-05]]
```

```
spot_1_GP.plot_progress(log_y=True)
```

```
spot_1_GP.plot_model()
```

10.5 Exercises

10.5.1 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.

10.5.2 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.

10.5.3 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.

10.5.4 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.

10.5.5 5. Gradient Boosting: `HistGradientBoostingRegressor`

- Describe the surrogate model. Use the information from the scikit-learn documentation.
- Use the surrogate as the model for optimization.

10.5.6 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 10.1.

Table 10.1: 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?

10.6 Selected Solutions

10.6.1 Solution to Exercise Section 10.5.5: Gradient Boosting

10.6.1.1 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)
```

```
Created spot_tensorboard_path: runs/spot_logs/BRANIN_p040025_2024-02-24_00-22-19 for SummaryW
```

- Running the surrogate model based optimizer `Spot`:

```
spot_2 = spot.Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
spot_2.run()
```

```
spotPython tuning: 3.146824136952164 [######----] 55.00%
spotPython tuning: 3.146824136952164 [######----] 60.00%
spotPython tuning: 3.146824136952164 [######----] 65.00%
spotPython tuning: 3.146824136952164 [#######---] 70.00%
spotPython tuning: 1.1487233101571483 [########--] 75.00%
```

```
spotPython tuning: 1.0236891516766402 [########--] 80.00%
spotPython tuning: 0.41994270072214057 [########--] 85.00%
spotPython tuning: 0.40193544341108023 [#########-] 90.00%
spotPython tuning: 0.3991519598268951 [##########] 95.00%
spotPython tuning: 0.3991519598268951 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2d3bed2d0>
```

- Print the results

```
spot_2.print_results()
```

```
min y: 0.3991519598268951
x0: 3.1546575195040987
x1: 2.285931113926263
```

```
[['x0', 3.1546575195040987], ['x1', 2.285931113926263]]
```

- Show the optimization progress:

```
spot_2.plot_progress(log_y=True)
```

- Generate a surrogate model plot:

```
spot_2.surrogate.plot()
```

10.6.1.2 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.Spot(fun=fun,
                       fun_control=fun_control,
                       design_control=design_control,
                       surrogate = S_XGB)
spot_2_XGB.run()
```

```
spotPython tuning: 30.69410528614059 [######----] 55.00%
spotPython tuning: 30.69410528614059 [######----] 60.00%
spotPython tuning: 30.69410528614059 [######----] 65.00%
spotPython tuning: 30.69410528614059 [#######---] 70.00%
spotPython tuning: 1.3263745845108854 [########--] 75.00%
spotPython tuning: 1.3263745845108854 [########--] 80.00%
spotPython tuning: 1.3263745845108854 [########--] 85.00%
spotPython tuning: 1.3263745845108854 [#########-] 90.00%
spotPython tuning: 1.3263745845108854 [##########] 95.00%
spotPython tuning: 1.3263745845108854 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2d7b66f90>
```

- Print the Results

```
spot_2_XGB.print_results()
```

```
min y: 1.3263745845108854
x0: -2.872730773493426
x1: 10.874313833535739
```

```
[['x0', -2.872730773493426], ['x1', 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.

10.6.1.3 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.Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
spot_1.run()
```

```
spotPython tuning: 0.03475493366922229 [####------] 40.00%
spotPython tuning: 0.03475483461229862 [#####-----] 50.00%
spotPython tuning: 0.03475338954992179 [######----] 60.00%
spotPython tuning: 0.03437475313644103 [#######---] 70.00%
spotPython tuning: 0.015290217643803946 [########--] 80.00%
spotPython tuning: 0.0017932523576966073 [#########-] 90.00%
spotPython tuning: 8.771851669068651e-06 [##########] 100.00% Done...
```

fun_evals: 5. min_y: 0.034755
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 6. min_y: 0.034753
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 7. min_y: 0.034375
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 8. min_y: 0.01529
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

- Print the Results

```
spot_1.print_results()
```

```
min y: 8.771851669068651e-06
x0: 0.002961731194600322

[['x0', 0.002961731194600322]]
```

- Show the Progress

```
spot_1.plot_progress(log_y=True)
```

- The method `plot_model` plots the final surrogate:

```
spot_1.plot_model()
```

10.6.1.4 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.Spot(fun=fun,
                       fun_control=fun_control,
                       design_control=design_control,
                       surrogate = S_XGB)
spot_1_XGB.run()
```

```
spotPython tuning: 0.0018828816523185745 [####------] 40.00%
spotPython tuning: 0.0018828816523185745 [#####-----] 50.00%
spotPython tuning: 0.0018828816523185745 [######----] 60.00%
```

```
spotPython tuning: 0.0018828816523185745 [#######---] 70.00%
spotPython tuning: 0.0018828816523185745 [########--] 80.00%
spotPython tuning: 0.0018828816523185745 [#########-] 90.00%
spotPython tuning: 0.0018828816523185745 [##########] 100.00% Done...
```

fun_evals: 6. min_y: 0.001883
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 7. min_y: 0.001883
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 8. min_y: 0.001883
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 9. min_y: 0.001883
Model
True function
Samples
Last Sample
Best Sample
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

```
spot_1_XGB.print_results()
```

```
min y: 0.0018828816523185745
x0: 0.04339218423078717

[['x0', 0.04339218423078717]]
```

```
spot_1_XGB.plot_progress(log_y=True)
```

```
spot_1_XGB.plot_model()
```

10.7 Jupyter Notebook

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

11 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.build.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
```

11.1 Gaussian Processes Regression: Basic Introductory `scikit-learn` Example

- This is the example from scikit-learn: https://scikit-learn.org/stable/auto_examples/gaussian_process/pl
- 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.

11.1.1 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]
```

11.1.2 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)
```

11.1.3 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")
```

11.1.4 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")
```

11.1.5 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")
```

11.2 Exercises

11.2.1 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.

11.2.2 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)
```

11.2.3 `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)
```

11.2.4 `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)
```

11.2.5 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, n_theta=1, noise=True)`

is used?

12 Expected Improvement

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.

12.1 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_in
import matplotlib.pyplot as plt
```

12.1.1 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 7.5, 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)
```

```
Created spot_tensorboard_path: runs/spot_logs/07_Y_p040025_2024-02-24_00-23-17 for SummaryW
```

```
spot_1 = spot.Spot(
            fun=fun,
            fun_control=fun_control,
            design_control=design_control)
spot_1.run()
```

```
spotPython tuning: 1.2026789271012512e-09 [####------] 44.00%
spotPython tuning: 1.2026789271012512e-09 [#####-----] 48.00%
spotPython tuning: 1.2026789271012512e-09 [#####-----] 52.00%
spotPython tuning: 1.2026789271012512e-09 [######----] 56.00%
spotPython tuning: 3.7010904275056666e-10 [######----] 60.00%
spotPython tuning: 3.7010904275056666e-10 [######----] 64.00%
spotPython tuning: 3.7010904275056666e-10 [#######---] 68.00%
spotPython tuning: 3.7010904275056666e-10 [#######---] 72.00%
spotPython tuning: 3.7010904275056666e-10 [########--] 76.00%
spotPython tuning: 3.7010904275056666e-10 [########--] 80.00%
spotPython tuning: 3.7010904275056666e-10 [########--] 84.00%
spotPython tuning: 3.7010904275056666e-10 [#########-] 88.00%
spotPython tuning: 2.802111689321758e-11 [#########-] 92.00%
spotPython tuning: 2.802111689321758e-11 [##########] 96.00%
spotPython tuning: 2.802111689321758e-11 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2cb4d6810>
```

12.1.2 Results

```
spot_1.print_results()
```

```
min y: 2.802111689321758e-11
x0: -5.293497604912803e-06

[['x0', -5.293497604912803e-06]]
```

```
spot_1.plot_progress(log_y=True)
```

Figure 12.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

12.2 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")
```

```
Created spot_tensorboard_path: runs/spot_logs/07_EI_ISO_p040025_2024-02-24_00-23-19 for Summa
```

```
spot_1_ei = spot.Spot(fun=fun,
                      fun_control=fun_control)
spot_1_ei.run()
```

```
spotPython tuning: 9.993558891826623e-09 [####------] 44.00%
spotPython tuning: 9.993558891826623e-09 [#####-----] 48.00%
spotPython tuning: 9.993558891826623e-09 [#####-----] 52.00%
spotPython tuning: 9.993558891826623e-09 [######----] 56.00%
spotPython tuning: 3.016921825539976e-12 [######----] 60.00%
spotPython tuning: 3.016921825539976e-12 [######----] 64.00%
spotPython tuning: 3.016921825539976e-12 [#######---] 68.00%
spotPython tuning: 3.016921825539976e-12 [#######---] 72.00%
spotPython tuning: 3.016921825539976e-12 [########--] 76.00%
spotPython tuning: 3.016921825539976e-12 [########--] 80.00%
spotPython tuning: 3.016921825539976e-12 [########--] 84.00%
spotPython tuning: 3.016921825539976e-12 [#########-] 88.00%
spotPython tuning: 3.016921825539976e-12 [#########-] 92.00%
spotPython tuning: 3.016921825539976e-12 [##########] 96.00%
spotPython tuning: 3.016921825539976e-12 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2cc76ec90>
```

```
spot_1_ei.plot_progress(log_y=True)
```

```
spot_1_ei.print_results()
```

```
min y: 3.016921825539976e-12
x0: 1.7369288487269638e-06

[['x0', 1.7369288487269638e-06]]
```

Figure 12.2: TensorBoard visualization of the spotPython optimization process and the surrogate model. Expected improvement, isotropic Kriging.

12.3 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(
    n_theta=2,
    noise=False,
    )
```

```
Created spot_tensorboard_path: runs/spot_logs/07_EI_NONISO_p040025_2024-02-24_00-23-20 for Su
```

```
spot_2_ei_noniso = spot.Spot(fun=fun,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)
spot_2_ei_noniso.run()
```

```
spotPython tuning: 2.035369116580917e-05 [####------] 44.00%
spotPython tuning: 2.035369116580917e-05 [#####-----] 48.00%
spotPython tuning: 2.035369116580917e-05 [#####-----] 52.00%
spotPython tuning: 1.0764759208059285e-05 [######----] 56.00%
spotPython tuning: 1.0764759208059285e-05 [######----] 60.00%
spotPython tuning: 1.2512039520452527e-07 [######----] 64.00%
spotPython tuning: 1.2512039520452527e-07 [#######---] 68.00%
spotPython tuning: 1.2512039520452527e-07 [#######---] 72.00%
spotPython tuning: 1.2512039520452527e-07 [########--] 76.00%
spotPython tuning: 1.2512039520452527e-07 [########--] 80.00%
spotPython tuning: 1.2512039520452527e-07 [########--] 84.00%
spotPython tuning: 1.2512039520452527e-07 [#########-] 88.00%
spotPython tuning: 1.2512039520452527e-07 [#########-] 92.00%
spotPython tuning: 1.2512039520452527e-07 [##########] 96.00%
spotPython tuning: 1.2512039520452527e-07 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2cc723810>
```

```
spot_2_ei_noniso.plot_progress(log_y=True)
```

```
spot_2_ei_noniso.print_results()
```

```
min y: 1.2512039520452527e-07
x0: -0.00023903776922459534
x1: 0.0002607323150065108

[['x0', -0.00023903776922459534], ['x1', 0.0002607323150065108]]
```

```
spot_2_ei_noniso.surrogate.plot()
```

Figure 12.3: TensorBoard visualization of the spotPython optimization process and the surrogate model. Expected improvement, isotropic Kriging.

12.4 Using `sklearn` Surrogates

12.4.1 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 12.4: 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.

12.4.2 spot: The Initial Model

12.4.2.1 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.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: 0.1377171852680486 [####------] 40.00%
spotPython tuning: 0.008763557388693657 [#####-----] 46.67%
spotPython tuning: 0.002832279071142736 [#####-----] 53.33%
spotPython tuning: 0.0008138662965600185 [######----] 60.00%
spotPython tuning: 0.00036637583790222027 [#######---] 66.67%
spotPython tuning: 0.00036006945938022686 [#######---] 73.33%
spotPython tuning: 0.0003591078890308837 [########--] 80.00%
spotPython tuning: 0.00032713515580249373 [#########-] 86.67%
spotPython tuning: 0.0002785854368057176 [#########-] 93.33%
spotPython tuning: 0.0001638494253170647 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2cc79e8d0>
```

```
spot_ei.plot_progress()
```

```
np.min(spot_1.y), np.min(spot_ei.y)
```

```
(2.802111689321758e-11, 0.0001638494253170647)
```

12.4.3 Init: Build Initial Design

```
from spotPython.design.spacefilling import spacefilling
from spotPython.build.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 ]]))
```

12.4.4 Evaluate

12.4.5 Build Surrogate

12.4.6 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.

12.5 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.

```python
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.build.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")
```

12.6 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.Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate=S,
                    design_control=design_control)
spot_GP.run()
```

```
spotPython tuning: 24.51465459019188 [####------] 40.00%
spotPython tuning: 11.003077541587748 [#####-----] 46.67%
spotPython tuning: 11.003077541587748 [#####-----] 53.33%
```

```
spotPython tuning: 7.281227279299504 [######----] 60.00%
spotPython tuning: 7.281227279299504 [#######---] 66.67%
spotPython tuning: 7.281227279299504 [#######---] 73.33%
spotPython tuning: 2.9519489314482 [########--] 80.00%
spotPython tuning: 2.9519489314482 [#########-] 86.67%
spotPython tuning: 2.104972804244822 [#########-] 93.33%
spotPython tuning: 1.9431600962086772 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2cc991b90>
```

```
spot_GP.y
```

```
array([ 69.32459936, 152.38491454, 107.92560483,  24.51465459,
        76.73500031,  86.30425303,  11.00307754,  16.11742138,
         7.28122728,  21.82317903,  10.96088904,   2.95194893,
         3.02910742,   2.1049728 ,   1.9431601 ])
```

```
spot_GP.plot_progress()
```

```
spot_GP.print_results()
```

```
min y: 1.9431600962086772
x0: 10.0
x1: 2.9985482809555464

[['x0', 10.0], ['x1', 2.9985482809555464]]
```

12.7 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.build.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",
              n_theta=1,
              noise=False,
              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.Spot(fun=fun,
                     fun_control=fun_control,
                     surrogate=S_K,
                     design_control=design_control,
                     surrogate_control=surrogate_control)
spot_S_K.run()
```

```
spotPython tuning: 0.13771718778810743 [##--------] 24.00%
spotPython tuning: 0.008768000187888899 [###-------] 28.00%
spotPython tuning: 0.0028300907437246053 [###-------] 32.00%
spotPython tuning: 0.0008148020998531609 [####------] 36.00%
spotPython tuning: 0.00036681248440550095 [####------] 40.00%
spotPython tuning: 0.00035607605553701025 [####------] 44.00%
spotPython tuning: 0.00035607605553701025 [#####-----] 48.00%
spotPython tuning: 0.00033033596693814263 [#####-----] 52.00%
spotPython tuning: 0.0002774179969789593 [######----] 56.00%
spotPython tuning: 0.00016886412273302311 [######----] 60.00%
spotPython tuning: 2.0349536932144563e-05 [######----] 64.00%
spotPython tuning: 1.6621220007683266e-06 [#######---] 68.00%
spotPython tuning: 4.905822935561126e-07 [#######---] 72.00%
spotPython tuning: 4.7634545282279014e-07 [########--] 76.00%
spotPython tuning: 3.966290585455581e-07 [########--] 80.00%
spotPython tuning: 1.9602185212475464e-07 [########--] 84.00%
spotPython tuning: 1.7115221726800905e-07 [#########-] 88.00%
spotPython tuning: 1.7115221726800905e-07 [#########-] 92.00%
spotPython tuning: 1.7115221726800905e-07 [##########] 96.00%
spotPython tuning: 1.7115221726800905e-07 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2ccd2cf90>
```

```
spot_S_K.plot_progress(log_y=True)
```

```
spot_S_K.surrogate.plot()
```

```
spot_S_K.print_results()
```

```
min y: 1.7115221726800905e-07
x0: 0.0003105897139994429
x1: 0.0002732878460995902

[['x0', 0.0003105897139994429], ['x1', 0.0002732878460995902]]
```

12.7.1 Optimize on Surrogate

12.7.2 Evaluate on Real Objective

12.7.3 Impute / Infill new Points

12.8 Tests

```python
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.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  ]
[[0.00159092 0.00410652]
 [0.00190779 0.00379162]]
```

12.9 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.build.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, n_theta=1, noise=False, cod_type="norm")
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.log
```

```
{'negLnLike': array([1.20788205]),
 'theta': array([-0.9900252]),
 'p': [],
 'Lambda': []}
```

12.10 EI: The Forrester Example

```
from spotPython.build.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, n_theta=1, noise=False, cod_type="norm")
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")
```

```
Created spot_tensorboard_path: runs/spot_logs/07_EI_FORRESTER_p040025_2024-02-24_00-23-36 fo
```

```
#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")
```

12.11 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.build.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,
            n_theta=1,
            noise=False)
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")
```

```
Created spot_tensorboard_path: runs/spot_logs/07_Y_p040025_2024-02-24_00-23-36 for SummaryWr
[[ 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.log
```

```
{'negLnLike': array([26.18505386]),
 'theta': array([-1.10547474]),
 'p': [],
 'Lambda': []}
```

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=True)
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.log
```

```
{'negLnLike': array([21.82059174]),
 'theta': array([-2.96946062]),
 'p': [],
 'Lambda': array([4.28985898e-05])}
```

12.12 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.build.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, n_theta=1, noise=False)
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")
```

```
Created spot_tensorboard_path: runs/spot_logs/07_Y_p040025_2024-02-24_00-23-36 for SummaryWr
[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]
```

```
 [ 3.4468512 ]
 [ 6.36049088]
 [-7.77978539]]
[ 2.56406437e-01 -6.93071067e+01 -8.56027124e-02  8.93405931e+02
 -5.82561927e+02 -4.76028022e+01  7.16445311e+01  4.09512920e+01
  2.57319028e+02 -4.70871982e+02]
```

```
S = Kriging(name='kriging',  seed=123, log_level=0, n_theta=1, noise=True)
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.build.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, n_theta=1, noise=False)
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")
```

```
Created spot_tensorboard_path: runs/spot_logs/07_Y_p040025_2024-02-24_00-23-37 for SummaryWri
[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]
 [ 3.4468512 ]
 [ 6.36049088]
 [-7.77978539]]
[0.712453   0.05595118 0.83735691 0.0106654  0.01413372 0.07074765
 0.05479457 0.07763503 0.02412205 0.01625354]
```

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=True)
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")
```

12.13 Modifying Lambda Search Space

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=True,
            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 Lambda sear
```

12.14 Factors

```
["num"] * 3
```

```
['num', 'num', 'num']
```

```
from spotPython.design.spacefilling import spacefilling
from spotPython.build.kriging import Kriging
from spotPython.fun.objectivefunctions import analytical
import numpy as np
```

```
gen = spacefilling(2)
n = 30
rng = np.random.RandomState(1)
lower = np.array([-5,-0])
```

```
upper = np.array([10,15])
fun = analytical().fun_branin_factor
#fun = analytical(sigma=0).fun_sphere

X0 = gen.scipy_lhd(n, lower=lower, upper = upper)
X1 = np.random.randint(low=1, high=3, size=(n,))
X = np.c_[X0, X1]
y = fun(X)
S = Kriging(name='kriging',  seed=123, log_level=50, n_theta=3, noise=False, var_type=["num"
S.fit(X, y)
Sf = Kriging(name='kriging',  seed=123, log_level=50, n_theta=3, noise=False, var_type=["num"
Sf.fit(X, y)
n = 50
X0 = gen.scipy_lhd(n, lower=lower, upper = upper)
X1 = np.random.randint(low=1, high=3, size=(n,))
X = np.c_[X0, X1]
y = fun(X)
s=np.sum(np.abs(S.predict(X)[0] - y))
sf=np.sum(np.abs(Sf.predict(X)[0] - y))
sf - s
```

```
-55.49075685088792
```

```
# vars(S)
```

```
# vars(Sf)
```

13 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.

13.1 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_i

PREFIX = "08"
```

13.1.1 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()
```

13.1.2 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: [0.99264427]
2: [1.02575851]
```

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: [0.99264427]
2: [1.02575851]
```

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)}")
```

```
Created spot_tensorboard_path: runs/spot_logs/08_p040025_2024-02-24_00-23-56 for SummaryWri
0: [0.98021757]
1: [0.98021757]
2: [0.98021757]
```

13.2 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,
)
```

```
Created spot_tensorboard_path: runs/spot_logs/08_p040025_2024-02-24_00-23-56 for SummaryWrite
```

`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.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(noise=True))
```

```
spot_1_noisy.run()
```

```
spotPython tuning: 0.034754930918721325 [####------] 40.00%
spotPython tuning: 0.03339769765455772 [#####-----] 50.00%
spotPython tuning: 0.015786142156557437 [######----] 60.00%
spotPython tuning: 0.0005791214064992311 [#######---] 70.00%
spotPython tuning: 3.5506618676925576e-05 [########--] 80.00%
spotPython tuning: 5.325186406243954e-07 [#########-] 90.00%
spotPython tuning: 4.335518265610199e-07 [##########] 100.00% Done...
```

fun_evals: 10. min_y (noise): 0.033398 min_mean_y: 0.033398
Model
True function
Samples
Last Sample
Best Sample (mean)
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 12. min_y (noise): 0.015786 min_mean_y: 0.015786
Model
True function
Samples
Last Sample
Best Sample (mean)
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 14. min_y (noise): 0.000579 min_mean_y: 0.000579
Model
True function
Samples
Last Sample
Best Sample (mean)
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 16. min_y (noise): 3.6e-05 min_mean_y: 3.6e-05
Model
True function
Samples
Last Sample
Best Sample (mean)
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 18. min_y (noise): 1e-06 min_mean_y: 1e-06
Model
True function
Samples
Last Sample
Best Sample (mean)
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 20. min_y (noise): 0.0 min_mean_y: 0.0
Model
True function
Samples
Last Sample
Best Sample (mean)
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

13.3 Print the Results

```
spot_1_noisy.print_results()
```

```
min y: 4.335518265610199e-07
x0: 0.0006584465252099216
min mean y: 4.335518265610199e-07
x0: 0.0006584465252099216

[['x0', 0.0006584465252099216], ['x0', 0.0006584465252099216]]
```

```
spot_1_noisy.plot_progress(log_y=False,
    filename="./figures/" + PREFIX + "_progress.png")
```

Figure 13.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

13.4 Noise and Surrogates: The Nugget Effect

13.4.1 The Noisy Sphere

13.4.1.1 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.build.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
```

```
Created spot_tensorboard_path: runs/spot_logs/08_p040025_2024-02-24_00-24-00 for SummaryWrite
```

- A surrogate without nugget is fitted to these data:

```
S = Kriging(name='kriging',
            n_theta=1,
            noise=False)
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',
            n_theta=1,
            noise=True)
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
```

```
0.00055921881757264
```

- We see:
  - the first model `S` has no nugget,
  - whereas the second model has a nugget value (`Lambda`) larger than zero.

13.5 Exercises

13.5.1 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])
```

13.5.2 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)
```

13.5.3 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)
```

13.5.4 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)
```

14 Optimal Computational Budget Allocation in Spot

This chapter demonstrates how noisy functions can be handled with Optimal Computational Budget Allocation (OCBA) by Spot.

14.1 Example: Spot, 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_control_i

PREFIX = "09"
```

14.1.1 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)
```

```
Created spot_tensorboard_path: runs/spot_logs/09_p040025_2024-02-24_00-24-18 for SummaryWrite
```

A plot 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()
```

`Spot` 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: 2)

```
spot_1_noisy = spot.Spot(fun=fun,
                    fun_control=fun_control_init(
                    lower = np.array([-1]),
                    upper = np.array([1]),
                    fun_evals = 20,
                    fun_repeats = 2,
                    infill_criterion="ei",
                    noise = True,
                    tolerance_x=0.0,
                    ocba_delta = 1,
```

```
                    show_models=True),
                    design_control=design_control_init(init_size=3, repeats=2),
                    surrogate_control=surrogate_control_init(noise=True))
```

```
spot_1_noisy.run()
```

```
spotPython tuning: 0.03475493366922229 [####------] 40.00%
spotPython tuning: 0.0004463018568303854 [#####-----] 50.00%
spotPython tuning: 0.0004463018568303854 [######----] 60.00%
spotPython tuning: 0.0001590474610240226 [#######---] 70.00%
spotPython tuning: 4.2454542934289965e-09 [########--] 80.00%
spotPython tuning: 2.2370853591440457e-10 [#########-] 90.00%
spotPython tuning: 2.2370853591440457e-10 [##########] 100.00% Done...
```

fun_evals: 10. min_y (noise): 0.000446 min_mean_y: 0.000446
Model
True function
Samples
Last Sample
Best Sample (mean)
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 12. min_y (noise): 0.000446 min_mean_y: 0.000446
Model
True function
Samples
Last Sample
Best Sample (mean)
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 14. min_y (noise): 0.000159 min_mean_y: 0.000159
Model
True function
Samples
Last Sample
Best Sample (mean)
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 16. min_y (noise): 0.0 min_mean_y: 0.0
Model
True function
Samples
Last Sample
Best Sample (mean)
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 18. min_y (noise): 0.0 min_mean_y: 0.0
Model
True function
Samples
Last Sample
Best Sample (mean)
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

fun_evals: 20. min_y (noise): 0.0 min_mean_y: 0.0
Model
True function
Samples
Last Sample
Best Sample (mean)
y
x
1.0
0.8
0.6
0.4
0.2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

14.2 Print the Results

```
spot_1_noisy.print_results()
```

```
min y: 2.2370853591440457e-10
x0: -1.4956889245909544e-05
min mean y: 2.2370853591440457e-10
x0: -1.4956889245909544e-05
```

```
[['x0', -1.4956889245909544e-05], ['x0', -1.4956889245909544e-05]]
```

```
spot_1_noisy.plot_progress(log_y=False)
```

14.3 Noise and Surrogates: The Nugget Effect

14.3.1 The Noisy Sphere

14.3.1.1 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.build.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,
            n_theta=1,
            noise=False)
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,
            n_theta=1,
            noise=True)
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
```

```
8.374496269458742e-05
```

We see:

- the first model `S` has no nugget,
- whereas the second model has a nugget value (`Lambda`) larger than zero.

14.4 Exercises

14.4.1 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])
```

14.4.2 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)
```

14.4.3 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}
```

14.4.4 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)
```

15 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.

15.1 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"
```

15.1.1 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]))
```

```
Created spot_tensorboard_path: runs/spot_logs/015_p040025_2024-02-24_00-24-44 for SummaryWrit
```

- 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.Spot(fun=fun,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)

spot_2.run()
```

```
spotPython tuning: 1.801603872454505e-05 [#######---] 73.33%
spotPython tuning: 1.801603872454505e-05 [########--] 80.00%
spotPython tuning: 1.801603872454505e-05 [#########-] 86.67%
spotPython tuning: 1.801603872454505e-05 [#########-] 93.33%
spotPython tuning: 1.801603872454505e-05 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2c763c0d0>
```

15.1.2 Results

```
spot_2.print_results()
```

```
min y: 1.801603872454505e-05
x0: 0.0019077911677074135
x1: 0.003791618596979743

[['x0', 0.0019077911677074135], ['x1', 0.003791618596979743]]
```

```
spot_2.plot_progress(log_y=True)
```

```
spot_2.surrogate.plot()
```

15.2 Example With Modified p

- We can use set p 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.Spot(fun=fun,
                     fun_control=fun_control,
                     surrogate_control=surrogate_control)
spot_2_p1.run()
```

```
spotPython tuning: 1.801603872454505e-05 [#######---] 73.33%
spotPython tuning: 1.801603872454505e-05 [########--] 80.00%
spotPython tuning: 1.801603872454505e-05 [#########-] 86.67%
spotPython tuning: 1.801603872454505e-05 [#########-] 93.33%
spotPython tuning: 1.801603872454505e-05 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2ccb69a90>
```

- 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: 1.801603872454505e-05
x0: 0.0019077911677074135
x1: 0.003791618596979743

[['x0', 0.0019077911677074135], ['x1', 0.003791618596979743]]
```

```
spot_2_p1.surrogate.plot()
```

15.2.1 Taking a Look at the `p` Values

15.2.1.1 `p` Values from the `spot` Model

- We can check, which `p` values the `spot` model has used:
- The `p` values from the surrogate can be printed as follows:

```
spot_2_p1.surrogate.p
```

```
array([1.])
```

- 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
```

```
array([2.])
```

15.3 Optimization of the p Values

```
surrogate_control = surrogate_control_init(n_p=1,
                                           optim_p=True)
spot_2_pm= spot.Spot(fun=fun,
                     fun_control=fun_control,
                     surrogate_control=surrogate_control)
spot_2_pm.run()
```

```
spotPython tuning: 1.893023485380876e-05 [#######---] 73.33%
spotPython tuning: 1.893023485380876e-05 [########--] 80.00%
spotPython tuning: 1.893023485380876e-05 [#########-] 86.67%
spotPython tuning: 1.893023485380876e-05 [#########-] 93.33%
spotPython tuning: 1.893023485380876e-05 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2d51bf390>
```

```
spot_2_pm.plot_progress(log_y=True)
```

```
spot_2_pm.print_results()
```

```
min y: 1.893023485380876e-05
x0: 0.0017549984724977892
x1: 0.003981232876300906

[['x0', 0.0017549984724977892], ['x1', 0.003981232876300906]]
```

```
spot_2_pm.surrogate.plot()
```

```
spot_2_pm.surrogate.p
```

```
array([1.77398298])
```

15.4 Optimization of Multiple p Values

```
surrogate_control = surrogate_control_init(n_p=2,
                                           optim_p=True)
spot_2_pmo= spot.Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate_control=surrogate_control)
spot_2_pmo.run()
```

```
spotPython tuning: 2.162397189403005e-05 [#######---] 73.33%
```

```
spotPython tuning: 2.162397189403005e-05 [########--] 80.00%
spotPython tuning: 2.162397189403005e-05 [#########-] 86.67%
spotPython tuning: 2.162397189403005e-05 [#########-] 93.33%
spotPython tuning: 2.162397189403005e-05 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2d2b398d0>
```

```
spot_2_pmo.plot_progress(log_y=True)
```

```
spot_2_pmo.print_results()
```

```
min y: 2.162397189403005e-05
x0: 0.0018245082309241386
x1: 0.00427728203527896

[['x0', 0.0018245082309241386], ['x1', 0.00427728203527896]]
```

```
spot_2_pmo.surrogate.plot()
```

```
spot_2_pmo.surrogate.p
```

```
array([1.09037777, 1.76346322])
```

15.5 Exercises

15.5.1 `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`.
- 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,
```

15.5.2 `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`.

15.5.3 `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`.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

15.5.4 `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`.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

15.6 Jupyter Notebook

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

Part III

Hyperparameter Tuning with Sklearn

16 HPT: sklearn

16.1 Introduction to sklearn

17 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.

17.1 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 = 10
PREFIX = "10"
```

17.2 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 gen_design_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_2024_02_24_00_26_15
Created spot_tensorboard_path: runs/spot_logs/10_p040025_2024-02-24_00-26-15 for SummaryWrite
```

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.

17.3 Step 3: SKlearn Load Data (Classification)

Randomly generate classification data.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons, make_circles, make_classification
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

```
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

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()
```

```
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})
```

17.4 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(),
     )
```

17.5 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 SVC
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
```

17.6 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 **?@sec-modification-of-hyperparameters-14**.

17.6.1 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, "probability", bounds=[0, 0])
fun_control["core_model_hyper_dict"]["tol"]
```

```
{'type': 'float',
 'default': 0.001,
 'transform': 'None',
 'lower': 1e-05,
 'upper': 0.001}
```

17.6.2 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 `SVC` model can be modified as follows:

```
from spotPython.hyperparameters.values import modify_hyper_parameter_levels
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}
```

17.6.3 Optimizers

Optimizers are described in **?@sec-optimizers-14**.

17.7 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_score, lo
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`.

17.7.1 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,
               })
```

17.8 Step 8: Calling the SPOT Function

17.8.1 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)
```

17.8.2 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(noise=True,
                                           n_theta=2)
from spotPython.spot import spot
spot_tuner = spot.Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control,
                   surrogate_control=surrogate_control)
spot_tuner.run(X_start=X_start)
```

```
spotPython tuning: 5.734217584632275 [----------] 1.30%
spotPython tuning: 5.734217584632275 [----------] 2.89%
spotPython tuning: 5.734217584632275 [----------] 4.80%
spotPython tuning: 5.734217584632275 [#---------] 6.57%
spotPython tuning: 5.734217584632275 [#---------] 8.88%
spotPython tuning: 5.734217584632275 [#---------] 10.69%
spotPython tuning: 5.734217584632275 [#---------] 13.16%
spotPython tuning: 5.734217584632275 [##--------] 17.97%
spotPython tuning: 5.734217584632275 [##--------] 23.20%
spotPython tuning: 5.734217584632275 [###-------] 28.72%
spotPython tuning: 5.734217584632275 [###-------] 33.98%
spotPython tuning: 5.734217584632275 [####------] 38.42%
spotPython tuning: 5.734217584632275 [####------] 41.90%
spotPython tuning: 5.734217584632275 [#####-----] 45.97%
spotPython tuning: 5.734217584632275 [#####-----] 49.16%
spotPython tuning: 5.734217584632275 [#####-----] 52.60%
spotPython tuning: 5.734217584632275 [######----] 58.53%
spotPython tuning: 5.734217584632275 [######----] 63.49%
spotPython tuning: 5.734217584632275 [#######---] 66.74%
spotPython tuning: 5.734217584632275 [#######---] 69.99%
spotPython tuning: 5.734217584632275 [#######---] 73.72%
spotPython tuning: 5.734217584632275 [########--] 77.53%
spotPython tuning: 5.734217584632275 [########--] 81.16%
spotPython tuning: 5.734217584632275 [#########-] 90.34%
spotPython tuning: 5.734217584632275 [##########] 98.97%
spotPython tuning: 5.734217584632275 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2d15b5a50>
```

17.8.3 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"
```

> Tip: TENSORBOARD_PATH
>
> The TensorBoard path can be printed with the following command:
>
> ```
> from spotPython.utils.init import get_tensorboard_path
> 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 17.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

17.9 Step 9: Results

After the hyperparameter tuning run is finished, 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(PREFIX)
SAVE_AND_LOAD = False
if SAVE_AND_LOAD == True:
    save_pickle(spot_tuner, experiment_name)
    spot_tuner = load_pickle(experiment_name)
```

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, filename="./figures/" + experiment_name+"_progress.pdf")
```

Results can also be printed in tabular form.

```
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

```
| name        | type   | default   |   lower |   upper | tuned                | transform
|-------------|--------|-----------|---------|---------|----------------------|-------------
| C           | float  | 1.0       |     0.1 |    10.0 | 2.394471655384338    | 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 | 0                    | None
| probability | factor | 0         |     0.0 |     0.0 | 0                    | None
| tol         | float  | 0.001     |   1e-05 |   0.001 | 0.000982585315792582 | None
| cache_size  | float  | 200.0     |   100.0 |   400.0 | 375.6371648003268    | 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, filename="./figures/" + experiment_name+"_impor
```

17.10 Get Default Hyperparameters

The default hyperparameters, whihc 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)
model_default
```

```
SVC(break_ties=0, cache_size=200.0, probability=0, shrinking=0)
```

17.11 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)
```

17.11.1 Plot: Compare Predictions

```
from spotPython.plot.validation import plot_roc
plot_roc(model_list=[model_default, model_spot], fun_control= fun_control, model_names=["Defa
```

```
from spotPython.plot.validation import plot_confusion_matrix
plot_confusion_matrix(model=model_default, fun_control=fun_control, title = "Default")
```

```
plot_confusion_matrix(model=model_spot, fun_control=fun_control, title="SPOT")
```

```
min(spot_tuner.y), max(spot_tuner.y)
```

```
(5.734217584632275, 7.782152436286657)
```

17.11.2 Detailed Hyperparameter Plots

```
spot_tuner.plot_important_hyperparameter_contour(filename=None)
```

```
C:  0.1341541283230725
kernel:  100.0
gamma:  4.673523120804703
tol:  6.573927267560208
break_ties:  3.5982840289498093
impo: [['C', 0.1341541283230725], ['kernel', 100.0], ['gamma', 4.673523120804703], ['shrinkin
impo after select: [['C', 0.1341541283230725], ['kernel', 100.0], ['gamma', 4.67352312080470
```

Surrogate
x1: kernel
x0: C
Surrogate
x2: gamma
x0: C
Surrogate
x4: tol
x0: C
x0: C
x1: kernel
x0: C
x2: gamma
x0: C
x4: tol

Surrogate
x6: break_ties
x0: C
7.95
7.80
7.65
7.50
7.35
7.20
7.05
6.90
6.75
6.60
x0: C
x6: break_ties
Surrogate
x2: gamma
x1: kernel
7.14
7.08
7.02
6.96
6.90
6.84
x1: kernel
x2: gamma
Surrogate
x4: tol
x1: kernel
7.14
7.08
7.02
6.96
6.90
6.84
x1: kernel
x4: tol

Surrogate
x6: break_ties
x1: kernel
Surrogate
x4: tol
x2: gamma
Surrogate
x6: break_ties
x2: gamma

17.11.3 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
```

17.11.4 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)
```

Part IV

Hyperparameter Tuning with River

18 HPT: River

18.1 Introduction to River

19 Simplifying Hyperparameter Tuning in Online Machine Learning—The spotRiverGUI

19.1 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 19.2 describes how to install the software. It also explains how the `spotRiverGUI` can be started. Section 19.3 describes the binary classification task and the options available in the `spotRiverGUI`. Section 19.4 provides information about the planned regression task. Section 19.5 describes how the data can be visualized in the `spotRiverGUI`. Section 19.6 provides information about saving and loading experiments. Section 19.7 describes how to start an experiment and how the associated tensorboard process can be started and stopped. Section 19.8 provides information about the analysis of the results from the hyperparameter tuning process. Section 19.9 concludes the article and provides an outlook.

19.2 Installation and Starting

19.2.1 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
```

19.2.2 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 19.1.

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.

19.3 Binary Classification

19.3.1 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 19.1: spotriver GUI

19.3.1.1 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/

19.3.1.2 User Data Sets

Besides the `river` data sets described in Section 19.3.1.1, 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.

19.3.1.3 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).

19.3.1.4 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.

19.3.2 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.

19.3.3 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.

19.3.4 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/l 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 19.2 shows the `spotRiverGUI` when the `forest.AMFClassifier` is selected and Figure 19.3 shows the `spotRiverGUI` when the `tree.HoeffdingTreeClassifier` is selected.

Numerical and categorical hyperparameters are treated differently in the `spotRiverGUI`:

- 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 19.3. 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

Figure 19.2: `spotRiverGUI` when `forest.AMFClassifier` is selected

Figure 19.3: `spotRiverGUI` when `tree.HoeffdingAdaptiveTreeClassifier` is selected

`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.

19.4 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.

19.5 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 19.4) shows histograms of the target variables in the train and test data sets. The second figure (Figure 19.5) shows scatter plots of the features in the train data set. The third figure (Figure 19.6) shows the corresponding scatter plots of the features in the test data set.

> **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 19.7, Figure 19.8, and Figure 19.9: 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.

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:

Figure 19.4: Output from the `spotRiverGUI` when `Bananas` data is selected for the `Show Data` option

Figure 19.5: Visualization of the train data. Output from the `spotRiverGUI` when `Bananas` data is selected for the `Show Data` option

Figure 19.6: Visualization of the test data. Output from the `spotRiverGUI` when `Bananas` data is selected for the `Show Data` option

Figure 19.7: Output from the `spotRiverGUI` when `HTTP` data is selected for the `Show Data` option. The target variable is biased.

Figure 19.8: 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 19.9: 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.

```
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
```

19.6 Saving and Loading

19.6.1 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 19.1.

Table 19.1: 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
remove_poor_attrs	factor	0	0	1	None
merit_preprune	factor	0	0	1	None

19.6.2 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.

19.7 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.

19.7.1 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 19.10 and Figure 19.11 show the tensorboard page when the `spotRiverGUI` is performing the tuning process.

Figure 19.10: Tensorboard visualization of the hyperparameter tuning process

Figure 19.11: Tensorboard. Parallel coordinates plot

`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.

19.8 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
- Parallel coordinates plot

Figure 19.12 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 19.13. The default versus tuned hyperparameters are shown in Figure 19.14. The surrogate plot is shown in Figure 19.15, Figure 19.16, and Figure 19.17.

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| .   |
```

Figure 19.12: Progress plot of the hyperparameter tuning process

Figure 19.13: Confusion matrix

Figure 19.14: Default versus tuned hyperparameters

Figure 19.15: Surrogate plot based on the Kriging model. `x0` and `x1` plotted against each other.

Figure 19.16: Surrogate plot based on the Kriging model. `x1` and `x2` plotted against each other.

Figure 19.17: Surrogate plot based an the Kriging model. `x0` and `x2` plotted against each other.

```
|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| *** |
```

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.

19.9 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.

20 `river` Hyperparameter Tuning: Hoeffding Adaptive Tree Regressor with Friedman Drift Data

This chapter demonstrates hyperparameter tuning for `river`'s `Hoeffding Adaptive Tree Regressor` with the Friedman drift data set [SOURCE]. The `Hoeffding Adaptive Tree Regressor` is a decision tree that uses the Hoeffding bound to limit the number of splits evaluated at each node. The `Hoeffding Adaptive Tree Regressor` is a regression tree, i.e., it predicts a real value for each sample. The `Hoeffding Adaptive Tree Regressor` is a drift aware model, i.e., it can handle concept drifts.

20.1 Setup

Before we consider the detailed experimental setup, we select the parameters that affect run time, initial design size, size of the data set, and the experiment name.

- `MAX_TIME`: The maximum run time in seconds for the hyperparameter tuning process.
- `INIT_SIZE`: The initial design size for the hyperparameter tuning process.
- `PREFIX`: The prefix for the experiment name.
- `K`: The factor that determines the number of samples in the data set.

> 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.
> - `K` is the multiplier for the number of samples. If it is set to 1, then `100_000`samples are taken. It is set to 0.1 for demonstration purposes. For real experiments, this should be increased to at least 1.

```
MAX_TIME = 1
INIT_SIZE = 5
```

```
PREFIX="24-river"
K = 0.1
```

- This notebook exemplifies hyperparameter tuning with SPOT (spotPython and spotRiver).
- The hyperparameter software SPOT is available in Python. It was developed in R (statistical programming language), see Open Access book "Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide", available here: https://link.springer.com/book/10.1007/978-981-19-5170-1.
- This notebook demonstrates hyperparameter tuning for `river`. It is based on the notebook "Incremental decision trees in river: the Hoeffding Tree case", see: https://riverml.xyz/0.15.0/recipes/on-hoeffding-trees/#42-regression-tree-splitters.
- Here we will use the river `HTR` and `HATR` functions as in "Incremental decision trees in river: the Hoeffding Tree case", see: https://riverml.xyz/0.15.0/recipes/on-hoeffding-trees/#42-regression-tree-splitters.

20.2 Initialization of the `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
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_2024_02_24_00_31_24
Created spot_tensorboard_path: runs/spot_logs/24-river_p040025_2024-02-24_00-31-24 for Summa
```

> 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.
> - Section 21.8.3 describes how to start TensorBoard and access the TensorBoard dashboard.

- 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.

20.3 Load Data: The Friedman Drift Data

We will use the Friedman synthetic dataset with concept drifts [SOURCE]. 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. There are two points of concept drift. At the second point of drift the old concept reoccurs.

The following parameters are used to generate and handle the data set:

- horizon: The prediction horizon in hours.
- n_samples: The number of samples in the data set.
- p_1: The position of the first concept drift.
- p_2: The position of the second concept drift.
- position: The position of the concept drifts.
- n_train: The number of samples used for training.

```
horizon = 7*24
n_samples = int(K*100_000)
p_1 = int(K*25_000)
p_2 = int(K*50_000)
position=(p_1, p_2)
n_train = 1_000
```

```
from river.datasets import synth
import pandas as pd
dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=position,
   seed=123
)
```

- We will use `spotRiver`'s `convert_to_df` function [SOURCE] to convert the `river` data set to a `pandas` data frame.

```
from spotRiver.utils.data_conversion import convert_to_df
target_column = "y"
df = convert_to_df(dataset, target_column=target_column, n_total=n_samples)
```

- 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.
- Then split the data frame into a training and test data set. The train and test data sets are stored in the `fun_control` dictionary.

```
from spotPython.hyperparameters.values import set_control_key_value
df.columns = [f"x{i}" for i in range(1, 11)] + ["y"]
set_control_key_value(control_dict=fun_control,
                        key="train",
                        value=df[:n_train],
                        replace=True)
set_control_key_value(fun_control, "test", df[n_train:], True)
set_control_key_value(fun_control, "n_samples", n_samples, replace=True)
set_control_key_value(fun_control, "target_column", target_column, replace=True)
```

20.4 Specification of the Preprocessing Model

- We use the `StandardScaler` [SOURCE] from `river` as the preprocessing model. The `StandardScaler` is used to standardize the data set, i.e., it has zero mean and unit variance.

```
from river import preprocessing
prep_model = preprocessing.StandardScaler()
set_control_key_value(fun_control, "prep_model", prep_model, replace=True)
```

20.5 SelectSelect Model (`algorithm`) and `core_model_hyper_dict`

`spotPython` hyperparameter tuning approach uses two components:

1. a model (class) and
2. an associated hyperparameter dictionary.

Here, the `river` model class `HoeffdingAdaptiveTreeRegressor` [SOURCE] is selected.

The corresponding hyperparameters are loaded from the associated dictionary, which is stored as a JSON file [SOURCE]. The JSON file contains hyperparameter type information, names, and bounds.

The method `add_core_model_to_fun_control` adds the model and the hyperparameter dictionary to the `fun_control` dictionary.

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.

```
from river.tree import HoeffdingAdaptiveTreeRegressor
from spotRiver.data.river_hyper_dict import RiverHyperDict
from spotPython.hyperparameters.values import add_core_model_to_fun_control
core_model  = HoeffdingAdaptiveTreeRegressor
add_core_model_to_fun_control(core_model=core_model,
                              fun_control=fun_control,
                              hyper_dict=RiverHyperDict,
                              filename=None)
```

20.6 Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model`

After the `core_model` and the `core_model_hyper_dict` are added to the `fun_control` dictionary, the hyperparameter tuning can be started. However, in some settings, the user wants to modify the hyperparameters of the `core_model_hyper_dict`. This can be done with the `modify_hyper_parameter_bounds` and `modify_hyper_parameter_levels` functions [SOURCE].

The following code shows how hyperparameter of type numeric and integer (boolean) can be modified. The `modify_hyper_parameter_bounds` function is used to modify the bounds of the hyperparameter `delta` and `merit_preprune`. Similar option exists for the `modify_hyper_parameter_levels` function to modify the levels of categorical hyperparameters.

```
from spotPython.hyperparameters.values import set_control_hyperparameter_value
set_control_hyperparameter_value(fun_control, "delta", [1e-10, 1e-6])
set_control_hyperparameter_value(fun_control, "merit_preprune", [0, 0])
```

> **Note: Active and Inactive Hyperparameters**
>
> Hyperparameters can be excluded from the tuning procedure by selecting identical values for the lower and upper bounds. For example, the hyperparameter `merit_preprune` is excluded from the tuning procedure by setting the bounds to `[0, 0]`.

`spotPython`'s method `gen_design_table` summarizes the experimental design that is used for the hyperparameter tuning:

```
from spotPython.utils.eda import gen_design_table
print(gen_design_table(fun_control))
```

```
| name                    | type   | default           |      lower |    upper | transform
|-------------------------|--------|-------------------|------------|----------|---------------
| grace_period            | int    | 200               |     10     | 1000     | None
| max_depth               | int    | 20                |      2     |   20     | transform_powe
| delta                   | float  | 1e-07             |      1e-10 |    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
| bootstrap_sampling      | factor | 0                 |      0     |    1     | None
| drift_window_threshold  | int    | 300               |    100     |  500     | None
| switch_significance     | float  | 0.05              |      0.01  |    0.1   | None
| binary_split            | factor | 0                 |      0     |    1     | None
| max_size                | float  | 500.0             |    100     | 1000     | None
| memory_estimate_period  | int    | 1000000           | 100000     |    1e+06 | None
| stop_mem_management     | factor | 0                 |      0     |    1     | None
| remove_poor_attrs       | factor | 0                 |      0     |    1     | None
| merit_preprune          | factor | 0                 |      0     |    1     | None
```

20.7 Selection of the Objective (Loss) Function

The `metric_sklearn` is used for the sklearn based evaluation via `eval_oml_horizon` [SOURCE]. Here we use the `mean_absolute_error` [SOURCE] as the objective function.

> **Note: Additional metrics**
>
> `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.

`spotRiver` provides 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.

> **Note: Weights**
>
> 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 `oml_grace_period` defines the number of observations that are used for the initial training of the model. The `step` defines the iteration number at which to yield results. This only takes into account the predictions, and not the training steps. 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.

```
import numpy as np
from sklearn.metrics import mean_absolute_error

weights = np.array([1, 1/1000, 1/1000])*10_000.0
oml_grace_period = 2
step = 100
weight_coeff = 1.0

set_control_key_value(control_dict=fun_control,
                        key="horizon",
                        value=horizon,
                        replace=True)
set_control_key_value(fun_control, "oml_grace_period", oml_grace_period, True)
set_control_key_value(fun_control, "weights", weights, True)
set_control_key_value(fun_control, "step", step, True)
set_control_key_value(fun_control, "weight_coeff", weight_coeff, True)
set_control_key_value(fun_control, "metric_sklearn", mean_absolute_error, True)
```

20.8 Calling the SPOT Function

20.8.1 The Objective Function

The objective function `fun_oml_horizon` [SOURCE] is selected next.

```
from spotRiver.fun.hyperriver import HyperRiver
fun = HyperRiver().fun_oml_horizon
```

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)
```

20.8.2 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(noise=True,
                                           n_theta=2)
from spotPython.spot import spot
spot_tuner = spot.Spot(fun=fun,
                    fun_control=fun_control,
                    design_control=design_control,
                    surrogate_control=surrogate_control)
spot_tuner.run(X_start=X_start)
```

```
spotPython tuning: 21772.114628482777 [#---------] 9.25%
spotPython tuning: 21772.114628482777 [#---------] 13.72%
spotPython tuning: 21772.114628482777 [##--------] 20.79%
spotPython tuning: 21772.114628482777 [#####-----] 47.24%
spotPython tuning: 21772.114628482777 [##########] 99.79%
spotPython tuning: 21772.114628482777 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2c46a8350>
```

20.8.3 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"
```

> Tip: TENSORBOARD_PATH
>
> The TensorBoard path can be printed with the following command:
>
> ```
> from spotPython.utils.init import get_tensorboard_path
> 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 20.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

20.8.4 Results

After the hyperparameter tuning run is finished, 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(PREFIX)
SAVE_AND_LOAD = False
if SAVE_AND_LOAD == True:
    save_pickle(spot_tuner, experiment_name)
    spot_tuner = load_pickle(experiment_name)
```

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, filename="./figures/" + experiment_name+"_progress.pdf")
```

Results can also be printed in tabular form.

```
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

```
| name                    | type   | default           |    lower |     upper | tuned
|-------------------------|--------|-------------------|----------|-----------|---------------
| grace_period            | int    | 200               |     10.0 |    1000.0 | 271.0
| max_depth               | int    | 20                |      2.0 |      20.0 | 13.0
| delta                   | float  | 1e-07             |    1e-10 |     1e-06 | 1.560124326429
| tau                     | float  | 0.05              |     0.01 |       0.1 | 0.078681307407
| leaf_prediction         | factor | mean              |      0.0 |       2.0 | model
| leaf_model              | factor | LinearRegression  |      0.0 |       2.0 | LinearRegressi
| model_selector_decay    | float  | 0.95              |      0.9 |      0.99 | 0.973379790035
| splitter                | factor | EBSTSplitter      |      0.0 |       2.0 | QOSplitter
| min_samples_split       | int    | 5                 |      2.0 |      10.0 | 9.0
| bootstrap_sampling      | factor | 0                 |      0.0 |       1.0 | 1
| drift_window_threshold  | int    | 300               |    100.0 |     500.0 | 379.0
| switch_significance     | float  | 0.05              |     0.01 |       0.1 | 0.023590637180
| binary_split            | factor | 0                 |      0.0 |       1.0 | 0
| max_size                | float  | 500.0             |    100.0 |    1000.0 | 241.5226665924
| memory_estimate_period  | int    | 1000000           | 100000.0 | 1000000.0 | 686536.0
| stop_mem_management     | factor | 0                 |      0.0 |       1.0 | 0
| remove_poor_attrs       | factor | 0                 |      0.0 |       1.0 | 1
```

```
| merit_preprune          | factor | 0                  |       0.0 |        1.0 | 0
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=0.0025, filename="./figures/" + experiment_name+"_impor
```

20.9 The Larger Data Set

After the hyperparamter were tuned on a small data set, we can now apply the hyperparameter configuration to a larger data set. The following code snippet shows how to generate the larger data set.

> Caution: Increased Friedman-Drift Data Set
>
> - The Friedman-Drift Data Set is increased by a factor of two to show the transferability of the hyperparameter tuning results.
> - Larger values of `K` lead to a longer run time.

```
K = 0.2
n_samples = int(K*100_000)
p_1 = int(K*25_000)
p_2 = int(K*50_000)
position=(p_1, p_2)
```

```
dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=position,
   seed=123
)
```

The larger data set is converted to a Pandas data frame and passed to the `fun_control` dictionary.

```
df = convert_to_df(dataset, target_column=target_column, n_total=n_samples)
df.columns = [f"x{i}" for i in range(1, 11)] + ["y"]
set_control_key_value(fun_control, "train", df[:n_train], True)
set_control_key_value(fun_control, "test", df[n_train:], True)
set_control_key_value(fun_control, "n_samples", n_samples, True)
set_control_key_value(fun_control, "target_column", target_column, True)
```

20.10 Get Default Hyperparameters

The default hyperparameters, whihc 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)
model_default
```

```
HoeffdingAdaptiveTreeRegressor (
  grace_period=200
  max_depth=1048576
  delta=1e-07
  tau=0.05
  leaf_prediction="mean"
  leaf_model=LinearRegression (
    optimizer=SGD (
      lr=Constant (
        learning_rate=0.01
      )
    )
    loss=Squared ()
```

```
    l2=0.
    l1=0.
    intercept_init=0.
    intercept_lr=Constant (
      learning_rate=0.01
    )
    clip_gradient=1e+12
    initializer=Zeros ()
  )
  model_selector_decay=0.95
  nominal_attributes=None
  splitter=EBSTSplitter ()
  min_samples_split=5
  bootstrap_sampling=0
  drift_window_threshold=300
  drift_detector=ADWIN (
    delta=0.002
    clock=32
    max_buckets=5
    min_window_length=5
    grace_period=10
  )
  switch_significance=0.05
  binary_split=0
  max_size=500.
  memory_estimate_period=1000000
  stop_mem_management=0
  remove_poor_attrs=0
  merit_preprune=0
  seed=None
)
```

> **Note:** `spotPython` tunes numpy arrays
>
> - `spotPython` tunes numpy arrays, i.e., the hyperparameters are stored in a numpy array.

The model with the default hyperparameters can be trained and evaluated with the following commands:

```
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., scaoe (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_
df_labels=["default"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default], log_y=False, df_labels=df_labels, r
```

20.10.1 Show Predictions

- 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)
```

```
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b]], target_column=target_colu
```

20.11 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_label
```

```
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b], df_true_spot[a:b]], target_
```

```
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["Pred
plot_actual_vs_predicted(y_test=df_true_spot[target_column], y_pred=df_true_spot["Prediction"
```

20.12 Visualize Regression Trees

```
dataset_f = dataset.take(n_samples)
for x, y in dataset_f:
    model_default.learn_one(x, y)
```

> 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': 35,
 'n_branches': 17,
 'n_leaves': 18,
 'n_active_leaves': 96,
 'n_inactive_leaves': 0,
 'height': 6,
```

```
 'total_observed_weight': 39002.0,
 'n_alternate_trees': 21,
 'n_pruned_alternate_trees': 6,
 'n_switch_alternate_trees': 2}
```

20.12.1 Spot Model

```
dataset_f = dataset.take(n_samples)
for x, y in dataset_f:
    model_spot.learn_one(x, y)
```

> 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': 57,
 'n_branches': 28,
 'n_leaves': 29,
 'n_active_leaves': 117,
 'n_inactive_leaves': 0,
 'height': 10,
 'total_observed_weight': 39002.0,
 'n_alternate_trees': 22,
 'n_pruned_alternate_trees': 2,
 'n_switch_alternate_trees': 0}
```

```
from spotPython.utils.eda import compare_two_tree_models
print(compare_two_tree_models(model_default, model_spot))
```

```
| Parameter                 |   Default |   Spot |
|---------------------------|-----------|--------|
| n_nodes                   |        35 |     57 |
| n_branches                |        17 |     28 |
```

```
| n_leaves                 |        18 |     29 |
| n_active_leaves          |        96 |    117 |
| n_inactive_leaves        |         0 |      0 |
| height                   |         6 |     10 |
| total_observed_weight    |     39002 |  39002 |
| n_alternate_trees        |        21 |     22 |
| n_pruned_alternate_trees |         6 |      2 |
| n_switch_alternate_trees |         2 |      0 |
```

20.13 Detailed Hyperparameter Plots

```
filename = "./figures/" + experiment_name
spot_tuner.plot_important_hyperparameter_contour(filename=filename)
```

```
grace_period:  1.3693049100964632
max_depth:  28.2636211827359
delta:  0.6616996339249895
tau:  0.03721608337334179
leaf_prediction:  11.585743446068548
model_selector_decay:  0.43987546575077235
splitter:  14.271906480144308
min_samples_split:  100.0
bootstrap_sampling:  5.432308980187159
drift_window_threshold:  0.11074108800880438
switch_significance:  10.334833189391416
binary_split:  44.53602574812833
max_size:  4.82707281124688
stop_mem_management:  0.7001482817882747
merit_preprune:  0.07171769145566809
impo: [['grace_period', 1.3693049100964632], ['max_depth', 28.2636211827359], ['delta', 0.66
impo after select: [['grace_period', 1.3693049100964632], ['max_depth', 28.2636211827359], [
```

Surrogate
1e−9+4.265271177e4
x1: max_depth
x0: grace_period
x2: delta
x3: tau
1e−6
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000

Surrogate
1e-9+4.265271177e4
x4: leaf_prediction
x0: grace_period
x6: model_selector_decay
x7: splitter
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000

Surrogate
1e+4.265271177e4
x8: min_samples_split
x0: grace_period
x9: bootstrap_sampling
x10: drift_window_threshold
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000

Surrogate
1e+4.265271177e4
x11: switch_significance
x0: grace_period
8.396
6.796
5.195
3.602
2.001
0.400
x0: grace_period
x11: switch_significance

Surrogate
1e+4.265271177e4
x12: binary_split
x0: grace_period
8.396
6.796
5.195
3.602
2.001
0.400
x0: grace_period
x12: binary_split

Surrogate
1e+4.265271177e4
x13: max_size
x0: grace_period
8.396
6.796
5.195
3.602
2.001
0.400
x0: grace_period
x13: max_size

Surrogate
1e−9+4.265271177e4
x15: stop_mem_management
x0: grace_period
x0: grace_period
x5: stop_mem_managem

Surrogate
1e−9+4.265271177e4
x17: merit_preprune
x0: grace_period
x0: grace_period
x17: merit_preprune

Surrogate
1e−6
1e−9+4.265271177e4
x2: delta
x1: max_depth
x1: max_depth
x2: delta

Surrogate
1e−9+4.265271177e4
x3: tau
x1: max_depth
x4: leaf_prediction
x6: model_selector_decay

Surrogate
1e+4.265271177e4
x7: splitter
x1: max_depth
Surrogate
1e+4.265271177e4
x8: min_samples_split
x1: max_depth
Surrogate
1e+4.265271177e4
x9: bootstrap_sampling
x1: max_depth

Surrogate
1e−14.265271177e4
x10: drift_window_threshold
x1: max_depth
8.396
6.796
5.195
3.602
2.001
0.400
Surrogate
1e−14.265271177e4
x11: switch_significance
x1: max_depth
Surrogate
1e−14.265271177e4
x12: binary_split
x1: max_depth
44000
43000
42000
41000

Surrogate
1e+4.265271177e4
x13: max_size
x1: max_depth
8.396
6.796
5.195
3.602
2.001
0.400

x1: max_depth
x13: max_size

Surrogate
1e+4.265271177e4
x15: stop_mem_management
x1: max_depth
8.396
6.796
5.195
3.602
2.001
0.400

x1: max_depth

Surrogate
1e+4.265271177e4
x17: merit_preprune
x1: max_depth
8.396
6.796
5.195
3.602
2.001
0.400

x1: max_depth
x17: merit_preprune

Surrogate
1e−9 +4.265271177e4
x3: tau
x2: delta
1e−6
x4: leaf_prediction
x6: model_selector_decay
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000

Surrogate
1e−9+4.265271177e4
x7: splitter
x2: delta
1e−6
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
Surrogate
1e−9+4.265271177e4
x8: min_samples_split
x2: delta
1e−6
Surrogate
1e−9+4.265271177e4
x9: bootstrap_sampling
x2: delta
1e−6

Surrogate
1e−9+4.265271177e4
x10: drift_window_threshold
x2: delta
1e−6
8.396
6.796
5.195
3.602
2.001
0.400
x11: switch_significance
x12: binary_split

Surrogate
1e−9+4.265271177e4
x13: max_size
x2: delta
1e−6
8.396
6.796
5.195
3.602
2.001
0.400

x2: delta
1e−6
x13: max_size

Surrogate
1e−9+4.265271177e4
x15: stop_mem_management
x2: delta
1e−6
8.396
6.796
5.195
3.602
2.001
0.400

x2: delta
1e−6

Surrogate
1e−9+4.265271177e4
x17: merit_preprune
x2: delta
1e−6
8.396
6.796
5.195
3.602
2.001
0.400

x2: delta
1e−6
x17: merit_preprune

Surrogate
1e−9+4.265271177e4
x4: leaf_prediction
x3: tau
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x4: leaf_prediction
Surrogate
1e−9+4.265271177e4
x6: model_selector_decay
x3: tau
x6: model_selector_deca
Surrogate
1e−9+4.265271177e4
x7: splitter
x3: tau
x7: splitter

Surrogate
1e+4.265271177e4
x8: min_samples_split
x3: tau
8.396
6.796
5.195
3.602
2.001
0.400
Surrogate
1e+4.265271177e4
x9: bootstrap_sampling
x3: tau
Surrogate
1e+4.265271177e4
x10: drift_window_threshold
x3: tau

Surrogate
1e-9+4.265271177e4
x11: switch_significance
x3: tau
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x11: switch_significance

Surrogate
1e-9+4.265271177e4
x12: binary_split
x3: tau
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x12: binary_split

Surrogate
1e-9+4.265271177e4
x13: max_size
x3: tau
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x13: max_size

Surrogate
1e+4.265271177e4
x15: stop_mem_management
x3: tau
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x3: tau
x15: stop_mem_management

Surrogate
1e+4.265271177e4
x17: merit_preprune
x3: tau
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x3: tau
x17: merit_preprune

Surrogate
1e+4.265271177e4
x6: model_selector_decay
x4: leaf_prediction
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x4: leaf_prediction
x6: model_selector_decay

Surrogate
1e−9+4.265271177e4
x7: splitter
x4: leaf_prediction
Surrogate
1e−9+4.265271177e4
x8: min_samples_split
x4: leaf_prediction
Surrogate
1e−9+4.265271177e4
x9: bootstrap_sampling
x4: leaf_prediction

Surrogate
1e−9+4.265271177e4
x10: drift_window_threshold
x4: leaf_prediction
x11: switch_significance
x12: binary_split
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000

Surrogate
1e+4.265271177e4
x13: max_size
x4: leaf_prediction
8.396
6.796
5.195
3.602
2.001
0.400

x4: leaf_prediction
x13: max_size

Surrogate
1e+4.265271177e4
x15: stop_mem_management
x4: leaf_prediction
8.396
6.796
5.195
3.602
2.001
0.400

x4: leaf_prediction

Surrogate
1e+4.265271177e4
x17: merit_preprune
x4: leaf_prediction
8.396
6.796
5.195
3.602
2.001
0.400

x4: leaf_prediction
x17: merit_preprune

Surrogate
1e−9+4.265271177e4
x7: splitter
x6: model_selector_decay
8.396
6.796
5.195
3.602
2.001
0.400
Surrogate
1e−9+4.265271177e4
x8: min_samples_split
x6: model_selector_decay
Surrogate
1e−9+4.265271177e4
x9: bootstrap_sampling
x6: model_selector_decay
x7: splitter
x8: min_samples_split
x9: bootstrap_sampling

Surrogate
1e−14.265271177e4
x10: drift_window_threshold
x6: model_selector_decay
x11: switch_significance
x12: binary_split
8.396
6.796
5.195
3.602
2.001
0.400

Surrogate
1e+4.265271177e4
x13: max_size
x6: model_selector_decay
8.396
6.796
5.195
3.602
2.001
0.400

x6: model_selector_decay
x13: max_size

Surrogate
1e+4.265271177e4
x15: stop_mem_management
x6: model_selector_decay
8.396
6.796
5.195
3.602
2.001
0.400

x6: model_selector_decay

Surrogate
1e+4.265271177e4
x17: merit_preprune
x6: model_selector_decay
8.396
6.796
5.195
3.602
2.001
0.400

x6: model_selector_decay
x17: merit_preprune

Surrogate
1e−9+4.265271177e4
x8: min_samples_split
x7: splitter
Surrogate
1e−9+4.265271177e4
x9: bootstrap_sampling
x7: splitter
Surrogate
1e−9+4.265271177e4
x10: drift_window_threshold
x7: splitter
x7: splitter
x8: min_samples_split
x7: splitter
x9: bootstrap_sampling
x7: splitter
x10: drift_window_thresh

Surrogate
1e+4.265271177e4
x11: switch_significance
x7: splitter
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x11: switch_significance

Surrogate
1e+4.265271177e4
x12: binary_split
x7: splitter
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x12: binary_split

Surrogate
1e+4.265271177e4
x13: max_size
x7: splitter
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x13: max_size

Surrogate
1e+4.265271177e4
x15: stop_mem_management
x7: splitter
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x7: splitter
x15: stop_mem_management

Surrogate
1e+4.265271177e4
x17: merit_preprune
x7: splitter
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x7: splitter
x17: merit_preprune

Surrogate
1e+4.265271177e4
x9: bootstrap_sampling
x8: min_samples_split
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x8: min_samples_split
x9: bootstrap_sampling

Surrogate
1e−14.265271177e4
x10: drift_window_threshold
x8: min_samples_split
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000

Surrogate
1e−14.265271177e4
x11: switch_significance
x8: min_samples_split
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000

Surrogate
1e−14.265271177e4
x12: binary_split
x8: min_samples_split
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000

Surrogate
1e+4.265271177e4
x13: max_size
x8: min_samples_split
8.396
6.796
5.195
3.602
2.001
0.400

x8: min_samples_split
x13: max_size

Surrogate
1e+4.265271177e4
x15: stop_mem_management
x8: min_samples_split
8.396
6.796
5.195
3.602
2.001
0.400

x8: min_samples_split

Surrogate
1e+4.265271177e4
x17: merit_preprune
x8: min_samples_split
8.396
6.796
5.195
3.602
2.001
0.400

x8: min_samples_split
x17: merit_preprune

Surrogate
1e−9+4.265271177e4
x10: drift_window_threshold
x9: bootstrap_sampling
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000
x11: switch_significance
x12: binary_split

Surrogate
1e+4.265271177e4
x13: max_size
x9: bootstrap_sampling
8.396
6.796
5.195
3.602
2.001
0.400

x9: bootstrap_sampling
x13: max_size

Surrogate
1e+4.265271177e4
x15: stop_mem_management
x9: bootstrap_sampling
8.396
6.796
5.195
3.602
2.001
0.400

x9: bootstrap_sampling

Surrogate
1e+4.265271177e4
x17: merit_preprune
x9: bootstrap_sampling
8.396
6.796
5.195
3.602
2.001
0.400

x9: bootstrap_sampling
x17: merit_preprune

Surrogate
1e-9+4.265271177e4
x11: switch_significance
x10: drift_window_threshold
x12: binary_split
x13: max_size
8.396
6.796
5.195
3.602
2.001
0.400

Surrogate
1e+4.265271177e4
x15: stop_mem_management
x10: drift_window_threshold
Surrogate
1e+4.265271177e4
x17: merit_preprune
x10: drift_window_threshold
Surrogate
1e+4.265271177e4
x12: binary_split
x11: switch_significance

Surrogate
1e+44.265271177e4
x13: max_size
x11: switch_significance
8.396
6.796
5.195
3.602
2.001
0.400

x11: switch_significance
x13: max_size

Surrogate
1e+44.265271177e4
x15: stop_mem_management
x11: switch_significance
8.396
6.796
5.195
3.602
2.001
0.400

x11: switch_significance

Surrogate
1e+44.265271177e4
x17: merit_preprune
x11: switch_significance
8.396
6.796
5.195
3.602
2.001
0.400

x11: switch_significance
x17: merit_preprune

Surrogate
1e+4.265271177e4
x13: max_size
x12: binary_split
8.396
6.796
5.195
3.602
2.001
0.400

x12: binary_split
x13: max_size

Surrogate
1e+4.265271177e4
x15: stop_mem_management
x12: binary_split
8.396
6.796
5.195
3.602
2.001
0.400

x12: binary_split

Surrogate
1e+4.265271177e4
x17: merit_preprune
x12: binary_split
8.396
6.796
5.195
3.602
2.001
0.400

x12: binary_split
x17: merit_preprune

Surrogate
1e+4.265271177e4
x15: stop_mem_management
x13: max_size
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000

Surrogate
1e+4.265271177e4
x17: merit_preprune
x13: max_size
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000

Surrogate
1e+4.265271177e4
x17: merit_preprune
x15: stop_mem_management
8.396
6.796
5.195
3.602
2.001
0.400
44000
43000
42000
41000

20.14 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
```

20.15 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)
```

21 `river` Hyperparameter Tuning: Mondrian Tree Regressor with Friedman Drift Data

This chapter demonstrates hyperparameter tuning for `river`'s `Mondrian Tree Regressor` 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.

21.1 Setup

Before we consider the detailed experimental setup, we select the parameters that affect run time, initial design size, size of the data set, and the experiment name.

- `MAX_TIME`: The maximum run time in seconds for the hyperparameter tuning process.
- `INIT_SIZE`: The initial design size for the hyperparameter tuning process.
- `PREFIX`: The prefix for the experiment name.
- `K`: The factor that determines the number of samples in the data set.

> 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.
> - `K` is the multiplier for the number of samples. If it is set to 1, then `100_000`samples are taken. It is set to 0.1 for demonstration purposes. For real experiments, this should be increased to at least 1.

```
MAX_TIME = 30
INIT_SIZE = 10
PREFIX="025RIVER"
K = 0.1
```

- This notebook exemplifies hyperparameter tuning with SPOT (spotPython and spotRiver).

- The hyperparameter software SPOT is available in Python. It was developed in R (statistical programming language), see Open Access book "Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide", available here: https://link.springer.com/book/10.1007/978-981-19-5170-1.
- This notebook demonstrates hyperparameter tuning for `river`. It is based on the notebook "Incremental decision trees in river: the Hoeffding Tree case", see: https://riverml.xyz/0.15.0/recipes/on-hoeffding-trees/#42-regression-tree-splitters.
- Here we will use the river `AMFRegressor` functions, see: https://riverml.xyz/0.19.0/api/forest/AMFRegressor/.

21.2 Initialization of the `fun_control` Dictionary

`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 "experiment name" to identify the hyperparameter tuning process. The experiment name is also used to create a directory for the TensorBoard files.

```
from spotPython.utils.init import fun_control_init
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_2024_02_24_01_42_10
Created spot_tensorboard_path: runs/spot_logs/025RIVER_p040025_2024-02-24_01-42-10 for Summa
```

> 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.
> - Section 21.8.3 describes how to start TensorBoard and access the TensorBoard dashboard.
> - 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.

21.3 Load Data: The Friedman Drift Data

We will use the Friedman synthetic dataset with concept drifts [SOURCE]. 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. There are two points of concept drift. At the second point of drift the old concept reoccurs.

The following parameters are used to generate and handle the data set:

- horizon: The prediction horizon in hours.
- n_samples: The number of samples in the data set.
- p_1: The position of the first concept drift.
- p_2: The position of the second concept drift.
- position: The position of the concept drifts.
- n_train: The number of samples used for training.

```
horizon = 7*24
n_samples = int(K*100_000)
p_1 = int(K*25_000)
p_2 = int(K*50_000)
position=(p_1, p_2)
n_train = 1_000
```

```
from river.datasets import synth
import pandas as pd
dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=position,
   seed=123
)
```

- We will use `spotRiver`'s `convert_to_df` function [SOURCE] to convert the `river` data set to a `pandas` data frame.

```
from spotRiver.utils.data_conversion import convert_to_df
target_column = "y"
df = convert_to_df(dataset, target_column=target_column, n_total=n_samples)
```

- 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.

- Then split the data frame into a training and test data set. The train and test data sets are stored in the `fun_control` dictionary.

```
from spotPython.hyperparameters.values import set_control_key_value
df.columns = [f"x{i}" for i in range(1, 11)] + ["y"]
set_control_key_value(control_dict=fun_control,
                        key="train",
                        value=df[:n_train],
                        replace=True)
set_control_key_value(fun_control, "test", df[n_train:], True)
set_control_key_value(fun_control, "n_samples", n_samples, replace=True)
set_control_key_value(fun_control, "target_column", target_column, replace=True)
```

21.4 Specification of the Preprocessing Model

- We use the `StandardScaler` [SOURCE] from `river` as the preprocessing model. The `StandardScaler` is used to standardize the data set, i.e., it has zero mean and unit variance.

```
from river import preprocessing
prep_model = preprocessing.StandardScaler()
set_control_key_value(fun_control, "prep_model", prep_model, replace=True)
```

21.5 SelectSelect Model (`algorithm`) and `core_model_hyper_dict`

`spotPython` hyperparameter tuning approach uses two components:

1. a model (class) and
2. an associated hyperparameter dictionary.

The corresponding hyperparameters are loaded from the associated dictionary, which is stored as a JSON file [SOURCE]. The JSON file contains hyperparameter type information, names, and bounds.

The method `add_core_model_to_fun_control` adds the model and the hyperparameter dictionary to the `fun_control` dictionary.

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.

```
from river.forest import AMFRegressor
from spotRiver.data.river_hyper_dict import RiverHyperDict
from spotPython.hyperparameters.values import add_core_model_to_fun_control
add_core_model_to_fun_control(core_model=AMFRegressor,
                              fun_control=fun_control,
                              hyper_dict=RiverHyperDict,
                              filename=None)
```

21.6 Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model`

After the `core_model` and the `core_model_hyper_dict` are added to the `fun_control` dictionary, the hyperparameter tuning can be started. However, in some settings, the user wants to modify the hyperparameters of the `core_model_hyper_dict`. This can be done with the `modify_hyper_parameter_bounds` and `modify_hyper_parameter_levels` functions [SOURCE].

The following code shows how hyperparameter of type numeric and integer (boolean) can be modified. The `modify_hyper_parameter_bounds` function is used to modify the bounds of the hyperparameter `delta` and `merit_preprune`. Similar option exists for the `modify_hyper_parameter_levels` function to modify the levels of categorical hyperparameters.

```
# from spotPython.hyperparameters.values import modify_hyper_parameter_bounds
# modify_hyper_parameter_bounds(fun_control, "n_estimators", bounds=[2,100])

from spotPython.hyperparameters.values import set_control_hyperparameter_value
set_control_hyperparameter_value(fun_control, "n_estimators", [2, 100])
```

::: {.callout-note} #### Note: Active and Inactive Hyperparameters Hyperparameters can be excluded from the tuning procedure by selecting identical values for the lower and upper bounds.

```
from spotPython.utils.eda import gen_design_table
print(gen_design_table(fun_control))
```

```
| name            | type   |   default |   lower |   upper | transform   |
|-----------------|--------|-----------|---------|---------|-------------|
| n_estimators    | int    |        10 |     2   |     100 | None        |
| step            | float  |         1 |     0.1 |      10 | None        |
| use_aggregation | factor |         1 |     0   |       1 | None        |
```

21.7 Selection of the Objective (Loss) Function

The `metric_sklearn` is used for the sklearn based evaluation via `eval_oml_horizon` [SOURCE]. Here we use the `mean_absolute_error` [SOURCE] as the objective function.

> **Note: Additional metrics**
>
> `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.

`spotRiver` provides 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.

> **Note: Weights**
>
> 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 `oml_grace_period` defines the number of observations that are used for the initial training of the model. The `step` defines the iteration number at which to yield results. This only takes into account the predictions, and not the training steps. 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.

```
import numpy as np
from sklearn.metrics import mean_absolute_error

weights = np.array([1, 1/1000, 1/1000])*10_000.0
oml_grace_period = 2
step = 100
weight_coeff = 1.0

# fun_control.update({
#                "horizon": horizon,
#                "oml_grace_period": oml_grace_period,
#                "weights": weights,
#                "step": step,
#                "weight_coeff": weight_coeff,
```

```
#                "metric_sklearn": mean_absolute_error
#                })
set_control_key_value(control_dict=fun_control,
                        key="horizon",
                        value=horizon,
                        replace=True)
set_control_key_value(fun_control, "oml_grace_period", oml_grace_period, True)
set_control_key_value(fun_control, "weights", weights, True)
set_control_key_value(fun_control, "step", step, True)
set_control_key_value(fun_control, "weight_coeff", weight_coeff, True)
set_control_key_value(fun_control, "metric_sklearn", mean_absolute_error, True)
```

21.8 Calling the SPOT Function

21.8.1 The Objective Function

The objective function `fun_oml_horizon` [SOURCE] is selected next.

```
from spotRiver.fun.hyperriver import HyperRiver
fun = HyperRiver().fun_oml_horizon
```

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)
```

21.8.2 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(noise=True,
                                           n_theta=2)
```

```
from spotPython.spot import spot
spot_tuner = spot.Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control,
                   surrogate_control=surrogate_control)
spot_tuner.run(X_start=X_start)
```

```
spotPython tuning: 26384.951165469887 [#---------] 12.30%
spotPython tuning: 26384.951165469887 [###-------] 28.01%
spotPython tuning: 26384.951165469887 [####------] 41.20%
spotPython tuning: 26384.951165469887 [#####-----] 53.83%
spotPython tuning: 26384.951165469887 [#######---] 67.05%
spotPython tuning: 26384.951165469887 [########--] 79.66%
spotPython tuning: 26377.692711034608 [#########-] 92.97%
spotPython tuning: 26377.692711034608 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2a4cca8d0>
```

21.8.3 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"
```

> Tip: TENSORBOARD_PATH
>
> The TensorBoard path can be printed with the following command:
>
> ```
> from spotPython.utils.init import get_tensorboard_path
> 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 21.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

21.8.4 Results

After the hyperparameter tuning run is finished, 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(PREFIX)
SAVE_AND_LOAD = False
if SAVE_AND_LOAD == True:
    save_pickle(spot_tuner, experiment_name)
    spot_tuner = load_pickle(experiment_name)
```

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, filename="./figures/" + experiment_name+"_progress.pdf")
```

Results can also be printed in tabular form.

```
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

```
| name            | type   |   default |   lower |   upper |              tuned | transform
|-----------------|--------|-----------|---------|---------|--------------------|-------------
| n_estimators    | int    |      10.0 |     2.0 |     100 |               57.0 | None
| step            | float  |       1.0 |     0.1 |      10 | 6.044488025398983 | 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=0.0025, filename="./figures/" + experiment_name+"_impor
```

21.9 The Larger Data Set

After the hyperparameter were tuned on a small data set, we can now apply the hyperparameter configuration to a larger data set. The following code snippet shows how to generate the larger data set.

> Caution: Increased Friedman-Drift Data Set
>
> - The Friedman-Drift Data Set is increased by a factor of two to show the transferability of the hyperparameter tuning results.
> - Larger values of `K` lead to a longer run time.

```
K = 0.2
n_samples = int(K*100_000)
p_1 = int(K*25_000)
p_2 = int(K*50_000)
position=(p_1, p_2)
```

```
dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=position,
   seed=123
)
```

The larger data set is converted to a Pandas data frame and passed to the `fun_control` dictionary.

```
df = convert_to_df(dataset, target_column=target_column, n_total=n_samples)
df.columns = [f"x{i}" for i in range(1, 11)] + ["y"]
# fun_control.update({"train": df[:n_train],
#                     "test": df[n_train:],
#                     "n_samples": n_samples,
#                     "target_column": target_column})
set_control_key_value(fun_control, "train", df[:n_train], True)
set_control_key_value(fun_control, "test", df[n_train:], True)
set_control_key_value(fun_control, "n_samples", n_samples, True)
set_control_key_value(fun_control, "target_column", target_column, True)
```

21.10 Get Default Hyperparameters

The default hyperparameters, whihc 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)
```

> Note: `spotPython` tunes numpy arrays
>
> - `spotPython` tunes numpy arrays, i.e., the hyperparameters are stored in a numpy array.

The model with the default hyperparameters can be trained and evaluated with the following commands:

```
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., scaoe (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_
df_labels=["default"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default], log_y=False, df_labels=df_labels, r
```

21.10.1 Show Predictions

- 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)
```

```
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b]], target_column=target_colu
```

21.11 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_label
```

```
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b], df_true_spot[a:b]], target_
```

```
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["Predi
plot_actual_vs_predicted(y_test=df_true_spot[target_column], y_pred=df_true_spot["Prediction"
```

21.12 Detailed Hyperparameter Plots

```
filename = "./figures/" + experiment_name
spot_tuner.plot_important_hyperparameter_contour(filename=filename)
```

```
n_estimators:  0.21269701276875377
step:  0.21269701276875377
use_aggregation:  100.0
impo: [['n_estimators', 0.21269701276875377], ['step', 0.21269701276875377], ['use_aggregatio
impo after select: [['n_estimators', 0.21269701276875377], ['step', 0.21269701276875377], ['u
```

21.13 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
```

21.14 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)
```

Part V

Hyperparameter Tuning with PyTorch Lightning

22 HPT PyTorch Lightning: Diabetes

In this tutorial, we will show how `spotPython` can be integrated into the `PyTorch` Lightning training workflow for a regression task.

This chapter describes the hyperparameter tuning of a `PyTorch Lightning` network on the Diabetes data set. 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.

22.1 Step 1: Setup

- Before we consider the detailed experimental setup, we select the parameters that affect run time, initial design size, etc.
- The parameter `MAX_TIME` specifies the maximum run time in seconds.
- The parameter `INIT_SIZE` specifies the initial design size.
- 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.

```
from spotPython.utils.device import getDevice
from math import inf

MAX_TIME = 1
FUN_EVALS = inf
INIT_SIZE = 5
WORKERS = 0
PREFIX="031"
DEVICE = getDevice()
DEVICES = 1
TEST_SIZE = 0.1
```

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.
- `WORKERS` is set to 0 for demonstration purposes. For real experiments, this should be increased. See the warnings that are printed when the number of workers is set to 0.

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.

22.2 Step 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,
    PREFIX=PREFIX,
    TENSORBOARD_CLEAN=True,
    device=DEVICE,
    enable_progress_bar=False,
    fun_evals=FUN_EVALS,
    log_level=10,
    max_time=MAX_TIME,
    num_workers=WORKERS,
    show_progress=True,
    test_size=0.1,
    tolerance_x=np.sqrt(np.spacing(1)),
    )
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2024_02_24_03_00_31
Created spot_tensorboard_path: runs/spot_logs/031_p040025_2024-02-24_03-00-31 for SummaryWrit
```

22.3 Step 3: Loading the Diabetes Data Set

```python
from spotPython.hyperparameters.values import set_control_key_value
from spotPython.data.diabetes import Diabetes
dataset = Diabetes()
set_control_key_value(control_dict=fun_control,
                        key="data_set",
                        value=dataset,
                        replace=True)
print(len(dataset))
```

```
442
```

Note: Data Set and Data Loader

- As shown below, a DataLoader from `torch.utils.data` can be used to check the data.

```python
# 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.])
```

22.4 Step 4: Preprocessing

Preprocessing is handled by `Lightning` and `PyTorch`. It is described in the LIGHTNING-DATAMODULE documentation. Here you can find information about the `transforms` methods.

22.5 Step 5: Select the Core Model (`algorithm`) and `core_model_hyper_dict`

`spotPython` includes the `NetLightRegression` class [SOURCE] for configurable neural networks. The class is imported here. It inherits from the class `Lightning.LightningModule`, which is the base class for all models in `Lightning`. `Lightning.LightningModule` is a subclass of `torch.nn.Module` and provides additional functionality for the training and testing of neural networks. The class `Lightning.LightningModule` is described in the Lightning documentation.

- Here we simply add the NN Model to the fun_control dictionary by calling the function `add_core_model_to_fun_control`:

```
from spotPython.light.regression.netlightregression import NetLightRegression
from spotPython.hyperdict.light_hyper_dict import LightHyperDict
from spotPython.hyperparameters.values import add_core_model_to_fun_control
add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=NetLightRegression,
                              hyper_dict=LightHyperDict)
```

The hyperparameters of the model are specified in the `core_model_hyper_dict` dictionary [SOURCE].

22.6 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.

> Caution: Small number of epochs for demonstration purposes
>
> - `epochs` and `patience` are set to small values for demonstration purposes. These values are too small for a real application.
> - More resonable values are, e.g.:
> - `set_control_hyperparameter_value(fun_control, "epochs", [7, 9])` and
> - `set_control_hyperparameter_value(fun_control, "patience", [2, 7])`

```
from spotPython.hyperparameters.values import set_control_hyperparameter_value

set_control_hyperparameter_value(fun_control, "l1", [4, 6])
set_control_hyperparameter_value(fun_control, "epochs", [9, 10])
set_control_hyperparameter_value(fun_control, "batch_size", [4, 5])
set_control_hyperparameter_value(fun_control, "optimizer", [
                "Adadelta",
                "Adagrad",
                "Adam",
                "AdamW",
                "Adamax",
                "NAdam",
                "RAdam",
                "RMSprop",
                "Rprop"
            ])
set_control_hyperparameter_value(fun_control, "dropout_prob", [0.01, 0.1])
set_control_hyperparameter_value(fun_control, "lr_mult", [0.5, 5.0])
set_control_hyperparameter_value(fun_control, "patience", [5, 7])
set_control_hyperparameter_value(fun_control, "act_fn",[
```

```
                "Sigmoid",
                "ReLU",
                "LeakyReLU",
                "Swish"
            ] )
```

Now, the dictionary `fun_control` contains all information needed for the hyperparameter tuning. Before the hyperparameter tuning is started, it is recommended to take a look at the experimental design. The method `gen_design_table` [SOURCE] generates a design table as follows:

```
from spotPython.utils.eda import gen_design_table
print(gen_design_table(fun_control))
```

```
| name           | type   | default   |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------|
| l1             | int    | 3         |    4    |     6   | transform_power_2_int |
| epochs         | int    | 4         |    9    |    10   | transform_power_2_int |
| batch_size     | int    | 4         |    4    |     5   | transform_power_2_int |
| act_fn         | factor | ReLU      |    0    |     3   | None                  |
| optimizer      | factor | SGD       |    0    |     8   | None                  |
| dropout_prob   | float  | 0.01      |    0.01 |     0.1 | None                  |
| lr_mult        | float  | 1.0       |    0.5  |     5   | None                  |
| patience       | int    | 2         |    5    |     7   | transform_power_2_int |
| initialization | factor | Default   |    0    |     2   | None                  |
```

This allows to check if all information is available and if the information is correct.

> **Note: Hyperparameters of the Tuned Model and the `fun_control` Dictionary**
>
> The updated `fun_control` dictionary can be shown with the command `fun_control["core_model_hyper_dict"]`.

22.7 Step 7: Data Splitting, the Objective (Loss) Function and the Metric

22.7.1 Evaluation

The evaluation procedure requires the specification of two elements:

1. the way how the data is split into a train and a test set
2. the loss function (and a metric).

> Caution: Data Splitting in Lightning
>
> The data splitting is handled by `Lightning`.

22.7.2 Loss Function

The loss function is specified in the configurable network class [SOURCE] We will use MSE.

22.7.3 Metric

- Similar to the loss function, the metric is specified in the configurable network class [SOURCE].

> Caution: Loss Function and Metric in Lightning
>
> - The loss function and the metric are not hyperparameters that can be tuned with `spotPython`.
> - They are handled by `Lightning`.

22.8 Step 8: Calling the SPOT Function

22.8.1 Preparing the SPOT Call

```
from spotPython.utils.init import design_control_init, surrogate_control_init
design_control = design_control_init(init_size=INIT_SIZE)

surrogate_control = surrogate_control_init(noise=True,
                                           n_theta=2)
```

> Note: Modifying Values in the Control Dictionaries
>
> - The values in the control dictionaries can be modified with the function `set_control_key_value` [SOURCE], for example:

```
set_control_key_value(control_dict=surrogate_control,
                      key="noise",
                      value=True,
                      replace=True)
set_control_key_value(control_dict=surrogate_control,
                      key="n_theta",
                      value=2,
                      replace=True)
```

22.8.2 The 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`.

```
from spotPython.fun.hyperlight import HyperLight
fun = HyperLight(log_level=50).fun
```

22.8.3 Showing the fun_control Dictionary

```
import pprint
pprint.pprint(fun_control)
```

```
{'CHECKPOINT_PATH': 'runs/saved_models/',
 'DATASET_PATH': 'data/',
 'PREFIX': '031',
 'RESULTS_PATH': 'results/',
 'TENSORBOARD_PATH': 'runs/',
 '_L_in': 10,
 '_L_out': 1,
 'accelerator': 'auto',
 'converters': None,
 'core_model': <class 'spotPython.light.regression.netlightregression.NetLightRegression'>,
 'core_model_hyper_dict': {'act_fn': {'class_name': 'spotPython.torch.activation',
                                      'core_model_parameter_type': 'instance()',
                                      'default': 'ReLU',
                                      'levels': ['Sigmoid',
                                                 'ReLU',
                                                 'LeakyReLU',
```

```
                        'Swish'],
           'lower': 0,
           'transform': 'None',
           'type': 'factor',
           'upper': 3},
'batch_size': {'default': 4,
               'lower': 4,
               'transform': 'transform_power_2_int',
               'type': 'int',
               'upper': 5},
'dropout_prob': {'default': 0.01,
                 'lower': 0.01,
                 'transform': 'None',
                 'type': 'float',
                 'upper': 0.1},
'epochs': {'default': 4,
           'lower': 9,
           'transform': 'transform_power_2_int',
           'type': 'int',
           'upper': 10},
'initialization': {'core_model_parameter_type': 'str',
                   'default': 'Default',
                   'levels': ['Default',
                              'Kaiming',
                              'Xavier'],
                   'lower': 0,
                   'transform': 'None',
                   'type': 'factor',
                   'upper': 2},
'l1': {'default': 3,
       'lower': 4,
       'transform': 'transform_power_2_int',
       'type': 'int',
       'upper': 6},
'lr_mult': {'default': 1.0,
            'lower': 0.5,
            'transform': 'None',
            'type': 'float',
            'upper': 5.0},
'optimizer': {'class_name': 'torch.optim',
              'core_model_parameter_type': 'str',
              'default': 'SGD',
              'levels': ['Adadelta',
```

```
                                                        'Adagrad',
                                                        'Adam',
                                                        'AdamW',
                                                        'Adamax',
                                                        'NAdam',
                                                        'RAdam',
                                                        'RMSprop',
                                                        'Rprop'],
                                                 'lower': 0,
                                                 'transform': 'None',
                                                 'type': 'factor',
                                                 'upper': 8},
                                 'patience': {'default': 2,
                                              'lower': 5,
                                              'transform': 'transform_power_2_int',
                                              'type': 'int',
                                              'upper': 7}},
 'counter': 0,
 'data': None,
 'data_dir': './data',
 'data_module': None,
 'data_set': <spotPython.data.diabetes.Diabetes object at 0x2cf420410>,
 'design': None,
 'device': 'mps',
 'devices': 1,
 'enable_progress_bar': False,
 'eval': None,
 'fun_evals': inf,
 'fun_repeats': 1,
 'horizon': None,
 'infill_criterion': 'y',
 'k_folds': 3,
 'log_graph': False,
 'log_level': 10,
 'loss_function': None,
 'lower': array([3. , 4. , 1. , 0. , 0. , 0. , 0.1, 2. , 0. ]),
 'max_time': 1,
 'metric_params': {},
 'metric_river': None,
 'metric_sklearn': None,
 'metric_torch': None,
 'model_dict': {},
 'n_points': 1,
```

'n_samples': None,
'noise': False,
'num_workers': 0,
'ocba_delta': 0,
'oml_grace_period': None,
'optimizer': None,
'path': None,
'prep_model': None,
'save_model': False,
'seed': 123,
'show_batch_interval': 1000000,
'show_models': False,
'show_progress': True,
'shuffle': None,
'sigma': 0.0,
'spot_tensorboard_path': 'runs/spot_logs/031_p040025_2024-02-24_03-00-31',
'spot_writer': <torch.utils.tensorboard.writer.SummaryWriter object at 0x2c4942390>,
'target_column': None,
'task': None,
'test': None,
'test_seed': 1234,
'test_size': 0.1,
'tolerance_x': 1.4901161193847656e-08,
'train': None,
'upper': array([8. , 9. , 4. , 5. , 11. , 0.25, 10. , 6. , 2.]),
'var_name': ['l1',
             'epochs',
             'batch_size',
             'act_fn',
             'optimizer',
             'dropout_prob',
             'lr_mult',
             'patience',
             'initialization'],
'var_type': ['int',
             'int',
             'int',
             'factor',
             'factor',
             'float',
             'float',
             'int',
             'factor'],

```
 'verbosity': 0,
 'weight_coeff': 0.0,
 'weights': 1.0}
```

22.8.4 Starting the Hyperparameter Tuning

The `spotPython` hyperparameter tuning is started by calling the `Spot` function [SOURCE].

```
from spotPython.spot import spot
spot_tuner = spot.Spot(fun=fun,
                       fun_control=fun_control,
                       design_control=design_control,
                       surrogate_control=surrogate_control)
spot_tuner.run()
```

```
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 32
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 32
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
```

```
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 3704.84765625, 'hp_metric': 3704.84765625}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 32
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 32
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 3881.30419921875, 'hp_metric': 3881.30419921875}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
```

```
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 16
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 16
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 3187.344970703125, 'hp_metric': 3187.344970703125}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 16
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
```

```
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 16
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 3847.451416015625, 'hp_metric': 3847.451416015625}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 32
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 32
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
```

```
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 2421.27490234375, 'hp_metric': 2421.27490234375}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 32
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 32
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 2654.690185546875, 'hp_metric': 2654.690185546875}
spotPython tuning: 2421.27490234375 [########--] 83.15%
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
```

```
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 32
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 32
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 3876.7275390625, 'hp_metric': 3876.7275390625}
spotPython tuning: 2421.27490234375 [##########] 100.00% Done...


       Validate metric              DataLoader 0

          hp_metric                 3704.84765625
          val_loss                  3704.84765625



       Validate metric              DataLoader 0

          hp_metric                3881.30419921875
          val_loss                 3881.30419921875
```

```
Validate metric              DataLoader 0

   hp_metric               3187.344970703125
   val_loss                3187.344970703125


Validate metric              DataLoader 0

   hp_metric               3847.451416015625
   val_loss                3847.451416015625


Validate metric              DataLoader 0

   hp_metric               2421.27490234375
   val_loss                2421.27490234375


Validate metric              DataLoader 0

   hp_metric               2654.690185546875
   val_loss                2654.690185546875


Validate metric              DataLoader 0

   hp_metric                3876.7275390625
   val_loss                 3876.7275390625


<spotPython.spot.spot.Spot at 0x2d150d110>
```

22.9 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard.

```
tensorboard --logdir="runs/"
```

Further information can be found in the PyTorch Lightning documentation for Tensorboard.

22.10 Step 10: Results

After the hyperparameter tuning run is finished, the results can be analyzed.

```
spot_tuner.plot_progress(log_y=False,
    filename="./figures/" + PREFIX +"_progress.png")
```

Figure 22.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

```
from spotPython.utils.eda import gen_design_table
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

```
| name           | type   | default   |   lower |   upper | tuned                | transform
|----------------|--------|-----------|---------|---------|----------------------|----------
| l1             | int    | 3         |     4.0 |     6.0 | 4.0                  | transform_
| epochs         | int    | 4         |     9.0 |    10.0 | 10.0                 | transform_
| batch_size     | int    | 4         |     4.0 |     5.0 | 5.0                  | transform_
```

```
| act_fn         | factor | ReLU      |     0.0 |     3.0 | ReLU                | None
| optimizer      | factor | SGD       |     0.0 |     8.0 | RAdam               | None
| dropout_prob   | float  | 0.01      |    0.01 |     0.1 | 0.035501708617280955 | None
| lr_mult        | float  | 1.0       |     0.5 |     5.0 | 4.785884240333345   | None
| patience       | int    | 2         |     5.0 |     7.0 | 5.0                 | transform_
| initialization | factor | Default   |     0.0 |     2.0 | Xavier              | None
```

```
spot_tuner.plot_importance(threshold=0.025,
    filename="./figures/" + PREFIX + "_importance.png")
```

Figure 22.2: Variable importance plot, threshold 0.025.

22.10.1 Get the Tuned Architecture

```
from spotPython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner, fun_control)
print(config)
```

```
{'l1': 16, 'epochs': 1024, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'RAdam', 'dropou
```

- Test on the full data set

```python
from spotPython.light.testmodel import test_model
test_model(config, fun_control)
```

```
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 32
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.TESTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: test
LightDataModule: test_dataloader(). Test set size: 45
LightDataModule: test_dataloader(). batch_size: 32
LightDataModule: test_dataloader(). num_workers: 0
test_model result: {'val_loss': 3394.724365234375, 'hp_metric': 3394.724365234375}
```

Test metric	DataLoader 0
hp_metric	3394.724365234375
val_loss	3394.724365234375

```
(3394.724365234375, 3394.724365234375)
```

```python
from spotPython.light.loadmodel import load_light_from_checkpoint

model_loaded = load_light_from_checkpoint(config, fun_control)
```

```
config: {'l1': 16, 'epochs': 1024, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'RAdam',
Loading model with 16_1024_32_ReLU_RAdam_0.0355_4.7859_32_Xavier_TEST from runs/saved_models/
Model: NetLightRegression(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=16, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.035501708617280955, inplace=False)
    (3): Linear(in_features=16, out_features=8, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.035501708617280955, inplace=False)
    (6): Linear(in_features=8, out_features=8, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.035501708617280955, inplace=False)
    (9): Linear(in_features=8, out_features=4, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.035501708617280955, inplace=False)
    (12): Linear(in_features=4, out_features=1, bias=True)
  )
)
```

```python
filename = "./figures/" + PREFIX
spot_tuner.plot_important_hyperparameter_contour(filename=filename)
```

```
epochs:  59.4827397464971
batch_size:  0.0430732818423805
optimizer:  100.0
lr_mult:  100.0
patience:  0.8688520741273658
impo: [['l1', 0.0034249707603736398], ['epochs', 59.4827397464971], ['batch_size', 0.0430732
impo after select: [['l1', 0.0034249707603736398], ['epochs', 59.4827397464971], ['batch_size
```

Figure 22.3: Contour plots.

Surrogate
x6: lr_mult
x1: epochs
Surrogate
1e−5+3.3801939e3
x7: patience
x1: epochs
Surrogate
1e−5+3.3801939e3
x4: optimizer
x2: batch_size

Surrogate
x6: lr_mult
x2: batch_size
3430
3410
3390
3370
3350
3330
3425
3400
3375
3350
x2: batch_size
x6: lr_mult
Surrogate
1e−5+3.3801939e3
x7: patience
x2: batch_size
4.6
4.2
3.8
3.4
3.0
2.6
4.0
3.5
3.0
x2: batch_size
x7: patience
Surrogate
x6: lr_mult
x4: optimizer
3540
3480
3420
3360
3300
3240
3500
3400
3300
x4: optimizer
x6: lr_mult

22.10.2 Parallel Coordinates Plot

```
spot_tuner.parallel_plot()
```

```
Unable to display output for mime type(s): text/html
```

Parallel coordinates plots

```
Unable to display output for mime type(s): text/html
```

22.10.3 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].

```
from spotPython.light.cvmodel import cv_model
set_control_key_value(control_dict=fun_control,
                        key="k_folds",
                        value=2,
                        replace=True)
set_control_key_value(control_dict=fun_control,
                        key="test_size",
                        value=0.6,
                        replace=True)
cv_model(config, fun_control)
```

```
k: 0
Train Dataset Size: 221
Val Dataset Size: 221
train_model result: {'val_loss': 3149.60205078125, 'hp_metric': 3149.60205078125}
k: 1
Train Dataset Size: 221
Val Dataset Size: 221
train_model result: {'val_loss': 3178.827392578125, 'hp_metric': 3178.827392578125}


       Validate metric              DataLoader 0

          hp_metric               3149.60205078125
          val_loss                3149.60205078125



       Validate metric              DataLoader 0

          hp_metric               3178.827392578125
          val_loss                3178.827392578125
```

```
3164.2147216796875
```

22.10.4 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)
```

22.10.5 Visualizing the Activation Distribution (Under Development)

> Reference:
>
> - The following code is based on [PyTorch Lightning TUTORIAL 2: ACTIVATION FUNCTIONS], Author: Phillip Lippe, License: [CC BY-SA], Generated: 2023-03-15T09:52:39.179933.

After we have trained the models, we can look at the actual activation values that find inside the model. For instance, how many neurons are set to zero in ReLU? Where do we find most values in Tanh? To answer these questions, we can write a simple function which takes a trained model, applies it to a batch of images, and plots the histogram of the activations inside the network:

```
from spotPython.torch.activation import Sigmoid, Tanh, ReLU, LeakyReLU, ELU, Swish
act_fn_by_name = {"sigmoid": Sigmoid, "tanh": Tanh, "relu": ReLU, "leakyrelu": LeakyReLU, "el
```

```
from spotPython.hyperparameters.values import get_one_config_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
config = get_one_config_from_X(X, fun_control)
model = fun_control["core_model"](**config, _L_in=64, _L_out=11)
model
```

```
NetLightRegression(
  (layers): Sequential(
    (0): Linear(in_features=64, out_features=16, bias=True)
    (1): ReLU()
```

```
    (2): Dropout(p=0.035501708617280955, inplace=False)
    (3): Linear(in_features=16, out_features=8, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.035501708617280955, inplace=False)
    (6): Linear(in_features=8, out_features=8, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.035501708617280955, inplace=False)
    (9): Linear(in_features=8, out_features=4, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.035501708617280955, inplace=False)
    (12): Linear(in_features=4, out_features=11, bias=True)
  )
)
```

```
# from spotPython.utils.eda import visualize_activations
# visualize_activations(model, color=f"C{0}")
```

23 HPT PyTorch Lightning: Diabetes Using a Recurrent Neural Network

In this tutorial, we will show how `spotPython` can be integrated into the `PyTorch` Lightning training workflow for a regression task.

This chapter describes the hyperparameter tuning of a `PyTorch Lightning` network on the Diabetes data set. 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.

23.1 Step 1: Setup

- Before we consider the detailed experimental setup, we select the parameters that affect run time, initial design size, etc.
- The parameter `MAX_TIME` specifies the maximum run time in seconds.
- The parameter `INIT_SIZE` specifies the initial design size.
- 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.

```
from spotPython.utils.device import getDevice
from math import inf
MAX_TIME = 1
FUN_EVALS = inf
INIT_SIZE = 5
WORKERS = 0
PREFIX="032"
DEVICE = getDevice()
```

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.
- `FUN_EVALS` is set to infinity.
- `INIT_SIZE` is set to 5 for demonstration purposes. For real experiments, this should be increased to at least 10.
- `WORKERS` is set to 0 for demonstration purposes. For real experiments, this should be increased. See the warnings that are printed when the number of workers is set to 0.
- `PREFIX` is set to "032". This is used for the experiment name and the name of the log file.
- `DEVICE` is set to the device that is returned by `getDevice()`, e.g., `gpu`.

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.

23.2 Step 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,
    PREFIX=PREFIX,
    TENSORBOARD_CLEAN=True,
    device=DEVICE,
    enable_progress_bar=False,
    fun_evals=FUN_EVALS,
    log_level=10,
```

```
    max_time=MAX_TIME,
    num_workers=WORKERS,
    show_progress=True,
    test_size=0.1,
    tolerance_x=np.sqrt(np.spacing(1)),
    verbosity=1
    )
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2024_02_24_03_17_19
Created spot_tensorboard_path: runs/spot_logs/032_p040025_2024-02-24_03-17-19 for SummaryWrit
```

23.3 Step 3: Loading the Diabetes Data Set

```
from spotPython.hyperparameters.values import set_control_key_value
from spotPython.data.diabetes import Diabetes
dataset = Diabetes()
set_control_key_value(control_dict=fun_control,
                        key="data_set",
                        value=dataset,
                        replace=True)
print(len(dataset))
```

```
442
```

> **Note: Data Set and Data Loader**
>
> - As shown below, a DataLoader from `torch.utils.data` can be used to check the data.

```python
# 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.])
```

23.4 Step 4: Preprocessing

Preprocessing is handled by `Lightning` and `PyTorch`. It is described in the LIGHTNING-DATAMODULE documentation. Here you can find information about the `transforms` methods.

23.5 Step 5: Select the Core Model (`algorithm`) and `core_model_hyper_dict`

`spotPython` includes the `NetLightRegression` class [SOURCE] for configurable neural networks. The class is imported here. It inherits from the class `Lightning.LightningModule`, which is the base class for all models in `Lightning`. `Lightning.LightningModule` is a subclass of `torch.nn.Module` and provides additional functionality for the training and testing of neural networks. The class `Lightning.LightningModule` is described in the Lightning documentation.

- Here we simply add the NN Model to the fun_control dictionary by calling the function `add_core_model_to_fun_control`:

```
from spotPython.light.regression.rnnlightregression import RNNLightRegression
from spotPython.hyperdict.light_hyper_dict import LightHyperDict
from spotPython.hyperparameters.values import add_core_model_to_fun_control
add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=RNNLightRegression,
                              hyper_dict=LightHyperDict)
```

The hyperparameters of the model are specified in the `core_model_hyper_dict` dictionary [SOURCE].

> **Note: User specified models and hyperparameter dictionaries**
>
> - The user can specify a model and a hyperparameter dictionary in a subfolder, e.g., `userRNN` in the current working directory.
> - The model and the hyperparameter dictionary are imported with the following code:
>
> ```
> from spotPython.hyperparameters.values import add_core_model_to_fun_control
> import sys
> sys.path.insert(0, './userRNN')
> import userrnn
> import user_hyper_dict
> add_core_model_to_fun_control(fun_control=fun_control,
>                               core_model=userrnn.RNNLightRegression,
>                               hyper_dict=user_hyper_dict.UserHyperDict)
> ```
>
> - Example files can be found in the userRNN folder.
> - These files can be modified by the user.
> - They can be used without re-compilation of the `spotPython` source code, if they

are located in a subfolder of the current working directory.

23.6 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.

Caution: Small number of epochs for demonstration purposes

- `epochs` and `patience` are set to small values for demonstration purposes. These values are too small for a real application.
- More resonable values are, e.g.:
  - `set_control_hyperparameter_value(fun_control, "epochs", [7, 9])` and
  - `set_control_hyperparameter_value(fun_control, "patience", [2, 7])`

```
from spotPython.hyperparameters.values import set_control_hyperparameter_value

set_control_hyperparameter_value(fun_control, "l1", [3, 8])
set_control_hyperparameter_value(fun_control, "epochs", [7, 9])
set_control_hyperparameter_value(fun_control, "batch_size", [2, 6])
set_control_hyperparameter_value(fun_control, "optimizer", [
                "Adadelta",
                "Adagrad",
                "Adam",
                "Adamax"])
set_control_hyperparameter_value(fun_control, "dropout_prob", [0.01, 0.25])
set_control_hyperparameter_value(fun_control, "lr_mult", [0.5, 5.0])
set_control_hyperparameter_value(fun_control, "patience", [3, 9])
set_control_hyperparameter_value(fun_control, "act_fn",["ReLU"] )
set_control_hyperparameter_value(fun_control, "initialization",["Default"] )
```

Now, the dictionary `fun_control` contains all information needed for the hyperparameter tuning. Before the hyperparameter tuning is started, it is recommended to take a look at the experimental design. The method `gen_design_table` [SOURCE] generates a design table as follows:

```python
from spotPython.utils.eda import gen_design_table
print(gen_design_table(fun_control))
```

```
| name           | type   | default   |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------|
| l1             | int    | 3         |    3    |    8    | transform_power_2_int |
| epochs         | int    | 4         |    7    |    9    | transform_power_2_int |
| batch_size     | int    | 4         |    2    |    6    | transform_power_2_int |
| act_fn         | factor | ReLU      |    0    |    0    | None                  |
| optimizer      | factor | SGD       |    0    |    3    | None                  |
| dropout_prob   | float  | 0.01      |    0.01 |    0.25 | None                  |
| lr_mult        | float  | 1.0       |    0.5  |    5    | None                  |
| patience       | int    | 2         |    3    |    9    | transform_power_2_int |
| initialization | factor | Default   |    0    |    0    | None                  |
```

This allows to check if all information is available and if the information is correct.

> **Note: Hyperparameters of the Tuned Model and the `fun_control` Dictionary**
>
> The updated `fun_control` dictionary can be shown with the command `fun_control["core_model_hyper_dict"]`.

23.7 Step 7: Data Splitting, the Objective (Loss) Function and the Metric

23.7.1 Evaluation

The evaluation procedure requires the specification of two elements:

1. the way how the data is split into a train and a test set
2. the loss function (and a metric).

> **Caution: Data Splitting in Lightning**
>
> The data splitting is handled by `Lightning`.

23.7.2 Loss Function

The loss function is specified in the configurable network class [SOURCE] We will use MSE.

23.7.3 Metric

- Similar to the loss function, the metric is specified in the configurable network class [SOURCE].

> Caution: Loss Function and Metric in Lightning
>
> - The loss function and the metric are not hyperparameters that can be tuned with `spotPython`.
> - They are handled by `Lightning`.

23.8 Step 8: Calling the SPOT Function

23.8.1 Preparing the SPOT Call

```
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()
set_control_key_value(control_dict=surrogate_control,
                        key="noise",
                        value=True,
                        replace=True)
set_control_key_value(control_dict=surrogate_control,
                        key="n_theta",
                        value=2,
                        replace=True)
```

23.8.2 The 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`.

```python
from spotPython.fun.hyperlight import HyperLight
fun = HyperLight(log_level=10).fun
```

23.8.3 Showing the fun_control Dictionary

```python
import pprint
pprint.pprint(fun_control)
```

```
{'CHECKPOINT_PATH': 'runs/saved_models/',
 'DATASET_PATH': 'data/',
 'PREFIX': '032',
 'RESULTS_PATH': 'results/',
 'TENSORBOARD_PATH': 'runs/',
 '_L_in': 10,
 '_L_out': 1,
 'accelerator': 'auto',
 'converters': None,
 'core_model': <class 'spotPython.light.regression.rnnlightregression.RNNLightRegression'>,
 'core_model_hyper_dict': {'act_fn': {'class_name': 'spotPython.torch.activation',
                                      'core_model_parameter_type': 'instance()',
                                      'default': 'ReLU',
                                      'levels': ['ReLU'],
                                      'lower': 0,
                                      'transform': 'None',
                                      'type': 'factor',
                                      'upper': 0},
                           'batch_size': {'default': 4,
                                          'lower': 2,
                                          'transform': 'transform_power_2_int',
                                          'type': 'int',
                                          'upper': 6},
                           'dropout_prob': {'default': 0.01,
                                            'lower': 0.01,
                                            'transform': 'None',
                                            'type': 'float',
                                            'upper': 0.25},
                           'epochs': {'default': 4,
                                      'lower': 7,
                                      'transform': 'transform_power_2_int',
                                      'type': 'int',
```

```
                                        'upper': 9},
                              'initialization': {'core_model_parameter_type': 'str',
                                                 'default': 'Default',
                                                 'levels': ['Default'],
                                                 'lower': 0,
                                                 'transform': 'None',
                                                 'type': 'factor',
                                                 'upper': 0},
                              'l1': {'default': 3,
                                     'lower': 3,
                                     'transform': 'transform_power_2_int',
                                     'type': 'int',
                                     'upper': 8},
                              'lr_mult': {'default': 1.0,
                                          'lower': 0.5,
                                          'transform': 'None',
                                          'type': 'float',
                                          'upper': 5.0},
                              'optimizer': {'class_name': 'torch.optim',
                                            'core_model_parameter_type': 'str',
                                            'default': 'SGD',
                                            'levels': ['Adadelta',
                                                       'Adagrad',
                                                       'Adam',
                                                       'Adamax'],
                                            'lower': 0,
                                            'transform': 'None',
                                            'type': 'factor',
                                            'upper': 3},
                              'patience': {'default': 2,
                                           'lower': 3,
                                           'transform': 'transform_power_2_int',
                                           'type': 'int',
                                           'upper': 9}},
'counter': 0,
'data': None,
'data_dir': './data',
'data_module': None,
'data_set': <spotPython.data.diabetes.Diabetes object at 0x2d8d63090>,
'design': None,
'device': 'mps',
'devices': 1,
'enable_progress_bar': False,
```

'eval': None,
'fun_evals': inf,
'fun_repeats': 1,
'horizon': None,
'infill_criterion': 'y',
'k_folds': 3,
'log_graph': False,
'log_level': 10,
'loss_function': None,
'lower': array([3. , 4. , 1. , 0. , 0. , 0. , 0.1, 2. , 0.]),
'max_time': 1,
'metric_params': {},
'metric_river': None,
'metric_sklearn': None,
'metric_torch': None,
'model_dict': {},
'n_points': 1,
'n_samples': None,
'noise': False,
'num_workers': 0,
'ocba_delta': 0,
'oml_grace_period': None,
'optimizer': None,
'path': None,
'prep_model': None,
'save_model': False,
'seed': 123,
'show_batch_interval': 1000000,
'show_models': False,
'show_progress': True,
'shuffle': None,
'sigma': 0.0,
'spot_tensorboard_path': 'runs/spot_logs/032_p040025_2024-02-24_03-17-19',
'spot_writer': <torch.utils.tensorboard.writer.SummaryWriter object at 0x2beccd050>,
'target_column': None,
'task': None,
'test': None,
'test_seed': 1234,
'test_size': 0.1,
'tolerance_x': 1.4901161193847656e-08,
'train': None,
'upper': array([8. , 9. , 4. , 1. , 11. , 0.25, 10. , 6. , 2.]),
'var_name': ['l1',

```
              'epochs',
              'batch_size',
              'act_fn',
              'optimizer',
              'dropout_prob',
              'lr_mult',
              'patience',
              'initialization'],
 'var_type': ['int',
              'int',
              'int',
              'factor',
              'factor',
              'float',
              'float',
              'int',
              'factor'],
 'verbosity': 1,
 'weight_coeff': 0.0,
 'weights': 1.0}
```

```
pprint.pprint(design_control)
```

```
{'init_size': 5, 'repeats': 1}
```

```
pprint.pprint(surrogate_control)
```

```
{'log_level': 50,
 'max_Lambda': 1,
 'max_theta': 2.0,
 'min_Lambda': 1e-09,
 'min_theta': -3.0,
 'model_fun_evals': 10000,
 'model_optimizer': <function differential_evolution at 0x17467e3e0>,
 'n_p': 1,
 'n_theta': 2,
 'noise': True,
 'optim_p': False,
 'p_val': 2.0,
 'seed': 124,
 'theta_init_zero': True,
 'var_type': None}
```

23.8.4 Starting the Hyperparameter Tuning

The `spotPython` hyperparameter tuning is started by calling the `Spot` function [SOURCE].

```
from spotPython.spot import spot
spot_tuner = spot.Spot(fun=fun,
                       fun_control=fun_control,
                       design_control=design_control,
                       surrogate_control=surrogate_control)
spot_tuner.run()
```

```
In fun(): config:
{'act_fn': ReLU(),
 'batch_size': 64,
 'dropout_prob': 0.19355651674791854,
 'epochs': 256,
 'initialization': 'Default',
 'l1': 16,
 'lr_mult': 1.5691149440098038,
 'optimizer': 'Adam',
 'patience': 32}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 64
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 64
LightDataModule: val_dataloader(). num_workers: 0
```

```
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 64
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 64
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 5832.97998046875, 'hp_metric': 5832.97998046875}

In fun(): config:
{'act_fn': ReLU(),
 'batch_size': 16,
 'dropout_prob': 0.09424169914869776,
 'epochs': 256,
 'initialization': 'Default',
 'l1': 128,
 'lr_mult': 3.35818256351233,
 'optimizer': 'Adadelta',
 'patience': 512}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 16
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
```

```
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 16
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 3765.2763671875, 'hp_metric': 3765.2763671875}

In fun(): config:
{'act_fn': ReLU(),
 'batch_size': 4,
 'dropout_prob': 0.21164199382623602,
 'epochs': 512,
 'initialization': 'Default',
 'l1': 128,
 'lr_mult': 0.9336514668325573,
 'optimizer': 'Adamax',
 'patience': 16}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 4
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 4
LightDataModule: val_dataloader(). num_workers: 0
```

```
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 4
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 4
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 2651.450439453125, 'hp_metric': 2651.450439453125}

In fun(): config:
{'act_fn': ReLU(),
 'batch_size': 8,
 'dropout_prob': 0.05728504399550885,
 'epochs': 128,
 'initialization': 'Default',
 'l1': 64,
 'lr_mult': 4.575980093998586,
 'optimizer': 'Adam',
 'patience': 32}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 8
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 8
LightDataModule: val_dataloader(). num_workers: 0
```

```
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 8
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 8
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 4523.0830078125, 'hp_metric': 4523.0830078125}

In fun(): config:
{'act_fn': ReLU(),
 'batch_size': 16,
 'dropout_prob': 0.14352914208400058,
 'epochs': 256,
 'initialization': 'Default',
 'l1': 8,
 'lr_mult': 2.4204853123355816,
 'optimizer': 'Adagrad',
 'patience': 128}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 16
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
```

```
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 16
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 8449.935546875, 'hp_metric': 8449.935546875}

In fun(): config:
{'act_fn': ReLU(),
 'batch_size': 4,
 'dropout_prob': 0.20258177417814544,
 'epochs': 512,
 'initialization': 'Default',
 'l1': 128,
 'lr_mult': 1.120935246611504,
 'optimizer': 'Adamax',
 'patience': 16}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 45
train_model(): Train set size: 359
train_model(): Batch size: 4
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 4
LightDataModule: val_dataloader(). num_workers: 0
```

```
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 4
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 4
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 4371.36474609375, 'hp_metric': 4371.36474609375}
spotPython tuning: 2651.450439453125 [##########] 100.00% Done...


       Validate metric             DataLoader 0

          hp_metric               5832.97998046875
          val_loss                5832.97998046875



       Validate metric             DataLoader 0

          hp_metric                3765.2763671875
          val_loss                 3765.2763671875



       Validate metric             DataLoader 0

          hp_metric               2651.450439453125
          val_loss                2651.450439453125



       Validate metric             DataLoader 0

          hp_metric                4523.0830078125
          val_loss                 4523.0830078125
```

```
       Validate metric              DataLoader 0

          hp_metric                8449.935546875
          val_loss                 8449.935546875



       Validate metric              DataLoader 0

          hp_metric               4371.36474609375
          val_loss                4371.36474609375


<spotPython.spot.spot.Spot at 0x2d98fbf90>
```

23.9 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard.

```
tensorboard --logdir="runs/"
```

Further information can be found in the PyTorch Lightning documentation for Tensorboard.

23.10 Step 10: Results

After the hyperparameter tuning run is finished, the results can be analyzed.

```
spot_tuner.plot_progress(log_y=False,
    filename="./figures/" + PREFIX + "_progress.png")
```

Figure 23.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

```
from spotPython.utils.eda import gen_design_table
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

```
| name           | type   | default   |   lower |   upper | tuned                | transform
|----------------|--------|-----------|---------|---------|----------------------|-----------
| l1             | int    | 3         |     3.0 |     8.0 | 7.0                  | transform_
| epochs         | int    | 4         |     7.0 |     9.0 | 9.0                  | transform_
| batch_size     | int    | 4         |     2.0 |     6.0 | 2.0                  | transform_
| act_fn         | factor | ReLU      |     0.0 |     0.0 | ReLU                 | None
| optimizer      | factor | SGD       |     0.0 |     3.0 | Adamax               | None
| dropout_prob   | float  | 0.01      |    0.01 |    0.25 | 0.21164199382623602  | None
| lr_mult        | float  | 1.0       |     0.5 |     5.0 | 0.9336514668325573   | None
| patience       | int    | 2         |     3.0 |     9.0 | 4.0                  | transform_
| initialization | factor | Default   |     0.0 |     0.0 | Default              | None
```

```
spot_tuner.plot_importance(threshold=0.025,
    filename="./figures/" + PREFIX + "_importance.png")
```

Figure 23.2: Variable importance plot, threshold 0.025.

23.10.1 Get the Tuned Architecture

```
from spotPython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner, fun_control)
print(config)
```

```
{'l1': 128, 'epochs': 512, 'batch_size': 4, 'act_fn': ReLU(), 'optimizer': 'Adamax', 'dropou
```

- Test on the full data set

```
from spotPython.light.testmodel import test_model
test_model(config, fun_control)
```

```
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
```

```
LightDataModule: setup(). stage: predict
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 39
LightDataModule: val_dataloader(). batch_size: 4
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 359
LightDataModule: train_dataloader(). batch_size: 4
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.TESTING
LightDataModule setup(): full_train_size: 0.9
LightDataModule setup(): val_size: 0.09
LightDataModule setup(): train_size: 0.81
LightDataModule setup(): test_size: 0.1
LightDataModule: setup(). stage: test
LightDataModule: test_dataloader(). Test set size: 45
LightDataModule: test_dataloader(). batch_size: 4
LightDataModule: test_dataloader(). num_workers: 0
test_model result: {'val_loss': 3670.98876953125, 'hp_metric': 3670.98876953125}
```

Test metric	DataLoader 0
hp_metric	3670.98876953125
val_loss	3670.98876953125

```
(3670.98876953125, 3670.98876953125)
```

```
from spotPython.light.loadmodel import load_light_from_checkpoint

model_loaded = load_light_from_checkpoint(config, fun_control)
```

```
config: {'l1': 128, 'epochs': 512, 'batch_size': 4, 'act_fn': ReLU(), 'optimizer': 'Adamax',
Loading model with 128_512_4_ReLU_Adamax_0.2116_0.9337_16_Default_TEST from runs/saved_models
Model: RNNLightRegression(
  (rnn_layer): RNN(10, 128, batch_first=True)
```

```
  (fc): Linear(in_features=128, out_features=128, bias=True)
  (output_layer): Linear(in_features=128, out_features=1, bias=True)
  (dropout1): Dropout(p=0.21164199382623602, inplace=False)
  (dropout2): Dropout(p=0.0, inplace=False)
  (dropout3): Dropout(p=0.0, inplace=False)
  (activation_fct): ReLU()
)
```

```
filename = "./figures/" + PREFIX
spot_tuner.plot_important_hyperparameter_contour(filename=filename)
```

```
l1:  0.5431038158307683
optimizer:  100.0
impo: [['l1', 0.5431038158307683], ['epochs', 0.0026254086940176256], ['batch_size', 0.00262
impo after select: [['l1', 0.5431038158307683], ['epochs', 0.0026254086940176256], ['batch_s
```

Figure 23.3: Contour plots.

23.10.2 Parallel Coordinates Plot

```
spot_tuner.parallel_plot()
```

```
Unable to display output for mime type(s): text/html
```

Parallel coordinates plots

```
Unable to display output for mime type(s): text/html
```

23.10.3 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].

```
from spotPython.light.cvmodel import cv_model
set_control_key_value(control_dict=fun_control,
                        key="k_folds",
                        value=2,
                        replace=True)
set_control_key_value(control_dict=fun_control,
                        key="test_size",
                        value=0.1,
                        replace=True)
cv_model(config, fun_control)
```

```
k: 0
Train Dataset Size: 221
Val Dataset Size: 221
train_model result: {'val_loss': 3672.435546875, 'hp_metric': 3672.435546875}
k: 1
Train Dataset Size: 221
Val Dataset Size: 221
train_model result: {'val_loss': 3101.556640625, 'hp_metric': 3101.556640625}


       Validate metric              DataLoader 0

          hp_metric                3672.435546875
          val_loss                 3672.435546875



       Validate metric              DataLoader 0

          hp_metric                3101.556640625
          val_loss                 3101.556640625
```

```
3386.99609375
```

23.10.4 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)
```

23.10.5 Visualizing the Activation Distribution (Under Development)

> Reference:
>
> - The following code is based on [PyTorch Lightning TUTORIAL 2: ACTIVATION FUNCTIONS], Author: Phillip Lippe, License: [CC BY-SA], Generated: 2023-03-15T09:52:39.179933.

After we have trained the models, we can look at the actual activation values that find inside the model. For instance, how many neurons are set to zero in ReLU? Where do we find most values in Tanh? To answer these questions, we can write a simple function which takes a trained model, applies it to a batch of images, and plots the histogram of the activations inside the network:

```
from spotPython.torch.activation import Sigmoid, Tanh, ReLU, LeakyReLU, ELU, Swish
act_fn_by_name = {"sigmoid": Sigmoid, "tanh": Tanh, "relu": ReLU, "leakyrelu": LeakyReLU, "el
```

```
from spotPython.hyperparameters.values import get_one_config_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
config = get_one_config_from_X(X, fun_control)
model = fun_control["core_model"](**config, _L_in=64, _L_out=11)
model
```

```
RNNLightRegression(
  (rnn_layer): RNN(64, 128, batch_first=True)
  (fc): Linear(in_features=128, out_features=128, bias=True)
  (output_layer): Linear(in_features=128, out_features=11, bias=True)
```

```
  (dropout1): Dropout(p=0.21164199382623602, inplace=False)
  (dropout2): Dropout(p=0.0, inplace=False)
  (dropout3): Dropout(p=0.0, inplace=False)
  (activation_fct): ReLU()
)
```

```
# from spotPython.utils.eda import visualize_activations
# visualize_activations(model, color=f"C{0}")
```

24 HPT PyTorch Lightning: User Specified Data Set and Regression Model

In this tutorial, we will show how `spotPython` can be integrated into the `PyTorch` Lightning training workflow for a regression task with a user specified data set and a user specified regression model.

This chapter describes the hyperparameter tuning of a `PyTorch Lightning` network on a user data set, which can be found in the subfolder of this notebook, `userData`. The network can be found in the subfolder `userModel`.

24.1 Step 1: Setup

- Before we consider the detailed experimental setup, we select the parameters that affect run time, initial design size, etc.
- The parameter `MAX_TIME` specifies the maximum run time in seconds.
- The parameter `INIT_SIZE` specifies the initial design size.
- 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.

```
from spotPython.utils.device import getDevice
from math import inf

MAX_TIME = 1
FUN_EVALS = inf
FUN_REPEATS = 1
OCBA_DELTA = 0
REPEATS = 1
INIT_SIZE = 3
WORKERS = 0
PREFIX="033"
DEVICE = getDevice()
DEVICES = 1
TEST_SIZE = 0.3
```

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 a small value for demonstration purposes. For real experiments, this should be increased to at least 10.
- `WORKERS` is set to 0 for demonstration purposes. For real experiments, this should be increased. See the warnings that are printed when the number of workers is set to 0.

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.

24.2 Step 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=6,
    _L_out=1,
    PREFIX=PREFIX,
    TENSORBOARD_CLEAN=True,
    device=DEVICE,
    enable_progress_bar=False,
    fun_evals=FUN_EVALS,
    fun_repeats=FUN_REPEATS,
    log_level=50,
    max_time=MAX_TIME,
    num_workers=WORKERS,
    ocba_delta = OCBA_DELTA,
    show_progress=True,
    test_size=TEST_SIZE,
```

```
    tolerance_x=np.sqrt(np.spacing(1)),
    verbosity=1,
    )
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2024_02_24_03_37_33
Created spot_tensorboard_path: runs/spot_logs/033_p040025_2024-02-24_03-37-33 for SummaryWrit
```

24.3 Step 3: Loading the User Specified Data Set

```
# from spotPython.hyperparameters.values import set_control_key_value
# from spotPython.data.pkldataset import PKLDataset
# import torch
# dataset = PKLDataset(directory="./userData/",
#                      filename="data_sensitive.pkl",
#                      target_column='N',
#                      feature_type=torch.float32,
#                      target_type=torch.float32,
#                      rmNA=True)
# set_control_key_value(control_dict=fun_control,
#                         key="data_set",
#                         value=dataset,
#                         replace=True)
# print(len(dataset))
```

> **Note: Data Set and Data Loader**
>
> - As shown below, a DataLoader from `torch.utils.data` can be used to check the data.

```
# if the package pyhcf is installed then print "pyhcf is installed" else print "pyhcf is no
try:
    import pyhcf
    print("pyhcf is installed")
    from pyhcf.data.loadHcfData import load_hcf_data
    dataset = load_hcf_data(A=True, H=True,
                        param_list=['H', 'D', 'L', 'K', 'E', 'I', 'N'],
                        target='N', rmNA=True, rmMF=True, scale_data=True, return_X_y=False
except ImportError:
    print("pyhcf is not installed")
    from spotPython.data.pkldataset import PKLDataset
    import torch
    dataset = PKLDataset(directory="./userData/",
                        filename="data_sensitive.pkl",
                        target_column='N',
                        feature_type=torch.float32,
                        target_type=torch.float32,
                        rmNA=True)
```

```
pyhcf is installed
Loading data for ['H', 'D', 'L', 'K', 'E', 'I', 'N']...
```

```
from spotPython.hyperparameters.values import set_control_key_value
set_control_key_value(control_dict=fun_control,
                        key="data_set",
                        value=dataset,
                        replace=True)
print(len(dataset))
```

```
41837
```

```python
# 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, 6])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[0.0033, 0.4000, 0.0000, 0.7500, 1.0000, 0.1667],
        [0.0246, 0.4000, 0.0435, 0.7500, 1.0000, 0.1667],
        [0.0275, 0.4000, 0.0435, 0.7500, 1.0000, 0.1667],
        [0.0285, 0.4000, 0.0435, 0.7500, 1.0000, 0.1667],
        [0.0285, 0.4000, 0.0435, 0.7500, 1.0000, 0.1667]])
Targets: tensor([4.5764, 4.9073, 6.2846, 5.5094, 5.6079])
```

24.4 Step 4: Preprocessing

Preprocessing is handled by `Lightning` and `PyTorch`. It is described in the LIGHTNING-DATAMODULE documentation. Here you can find information about the `transforms` methods.

24.5 Step 5: Select the Core Model (`algorithm`) and `core_model_hyper_dict`

`spotPython` includes the `NetLightRegression` class [SOURCE] for configurable neural networks. The class is imported here. It inherits from the class `Lightning.LightningModule`, which is the base class for all models in `Lightning`. `Lightning.LightningModule` is a subclass of `torch.nn.Module` and provides additional functionality for the training and testing of neural networks. The class `Lightning.LightningModule` is described in the Lightning documentation.

- Here we simply add the NN Model to the fun_control dictionary by calling the function `add_core_model_to_fun_control`:

We can use aconfiguration from the `spotPython` package:

```
from spotPython.light.regression.netlightregression import NetLightRegression
from spotPython.hyperdict.light_hyper_dict import LightHyperDict
from spotPython.hyperparameters.values import add_core_model_to_fun_control
add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=NetLightRegression,
                              hyper_dict=LightHyperDict)
```

- Alternatively, we can use a userr configuration from the subdirectory `userModel`:

```
from spotPython.hyperparameters.values import add_core_model_to_fun_control
import sys
sys.path.insert(0, './userModel')
import netlightregression
import light_hyper_dict
add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=netlightregression.NetLightRegression,
                              hyper_dict=light_hyper_dict.LightHyperDict)
```

The hyperparameters of the model are specified in the `core_model_hyper_dict` dictionary [SOURCE].

24.6 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.

Caution: Small number of epochs for demonstration purposes

- `epochs` and `patience` are set to small values for demonstration purposes. These values are too small for a real application.
- More resonable values are, e.g.:
  - `set_control_hyperparameter_value(fun_control, "epochs", [7, 9])` and
  - `set_control_hyperparameter_value(fun_control, "patience", [2, 7])`

Note: Pre-experimental Results

- The following hyperparameters {Table 24.1} have generated acceptable results (obtained in in pre-experimental runs):

Table 24.1: Table 1: Pre-experimental results for the user specified data set. The test set size is 715, the train set size is 1167, and the batch size is 16.

Hyperparameter	Value
act_fn	LeakyReLU
batch_size	16
dropout_prob	0.01
epochs	512
initialization	Default
l1	128
lr_mult	0.5
optimizer	Adagrad
patience	16

Therefore, we will use these values as the starting poing for the hyperparameter tuning.

```
from spotPython.hyperparameters.values import set_control_hyperparameter_value

set_control_hyperparameter_value(fun_control, "l1", [3, 4])
set_control_hyperparameter_value(fun_control, "epochs", [2, 4])
set_control_hyperparameter_value(fun_control, "batch_size", [3, 6])
set_control_hyperparameter_value(fun_control, "optimizer", [
                "Adadelta",
                "Adamax",
                "Adagrad"
```

```
            ])
set_control_hyperparameter_value(fun_control, "dropout_prob", [0.005, 0.25])
set_control_hyperparameter_value(fun_control, "lr_mult", [0.25, 5.0])
set_control_hyperparameter_value(fun_control, "patience", [2, 3])
set_control_hyperparameter_value(fun_control, "act_fn",[
                "ReLU",
                "LeakyReLU",
            ] )
set_control_hyperparameter_value(fun_control, "initialization",["Default"] )
```

Now, the dictionary `fun_control` contains all information needed for the hyperparameter tuning. Before the hyperparameter tuning is started, it is recommended to take a look at the experimental design. The method `gen_design_table` [SOURCE] generates a design table as follows:

```
from spotPython.utils.eda import gen_design_table
print(gen_design_table(fun_control))
```

```
| name           | type   | default   |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------|
| l1             | int    | 3         |   3     |    4    | transform_power_2_int |
| epochs         | int    | 4         |   2     |    4    | transform_power_2_int |
| batch_size     | int    | 4         |   3     |    6    | transform_power_2_int |
| act_fn         | factor | ReLU      |   0     |    1    | None                  |
| optimizer      | factor | SGD       |   0     |    2    | None                  |
| dropout_prob   | float  | 0.01      |   0.005 |    0.25 | None                  |
| lr_mult        | float  | 1.0       |   0.25  |    5    | None                  |
| patience       | int    | 2         |   2     |    3    | transform_power_2_int |
| initialization | factor | Default   |   0     |    0    | None                  |
```

This allows to check if all information is available and if the information is correct.

> **Note: Hyperparameters of the Tuned Model and the `fun_control` Dictionary**
>
> The updated `fun_control` dictionary can be shown with the command `fun_control["core_model_hyper_dict"]`.

24.7 Step 7: Data Splitting, the Objective (Loss) Function and the Metric

24.7.1 Evaluation

The evaluation procedure requires the specification of two elements:

1. the way how the data is split into a train and a test set
2. the loss function (and a metric).

> Caution: Data Splitting in Lightning
>
> The data splitting is handled by `Lightning`.

24.7.2 Loss Function

The loss function is specified in the configurable network class [SOURCE] We will use MSE.

24.7.3 Metric

- Similar to the loss function, the metric is specified in the configurable network class [SOURCE].

> Caution: Loss Function and Metric in Lightning
>
> - The loss function and the metric are not hyperparameters that can be tuned with `spotPython`.
> - They are handled by `Lightning`.

24.8 Step 8: Calling the SPOT Function

24.8.1 Preparing the SPOT Call

```
from spotPython.utils.init import design_control_init, surrogate_control_init
design_control = design_control_init(init_size=INIT_SIZE,
                                     repeats=REPEATS,)
```

```
surrogate_control = surrogate_control_init(noise=True,
                                           n_theta=2,
                                           min_Lambda=1e-6,
                                           max_Lambda=10,
                                           log_level=50,)
```

> Note: Modifying Values in the Control Dictionaries
>
> - The values in the control dictionaries can be modified with the function `set_control_key_value` [SOURCE], for example:

```
set_control_key_value(control_dict=surrogate_control,
                      key="noise",
                      value=True,
                      replace=True)
set_control_key_value(control_dict=surrogate_control,
                      key="n_theta",
                      value=2,
                      replace=True)
```

24.8.2 The 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`.

```
from spotPython.fun.hyperlight import HyperLight
fun = HyperLight(log_level=50).fun
```

24.8.3 Showing the fun_control Dictionary

```
import pprint
pprint.pprint(fun_control)
```

```
{'CHECKPOINT_PATH': 'runs/saved_models/',
 'DATASET_PATH': 'data/',
 'PREFIX': '033',
 'RESULTS_PATH': 'results/',
 'TENSORBOARD_PATH': 'runs/',
```

```
'_L_in': 6,
'_L_out': 1,
'accelerator': 'auto',
'converters': None,
'core_model': <class 'netlightregression.NetLightRegression'>,
'core_model_hyper_dict': {'act_fn': {'class_name': 'spotPython.torch.activation',
                                     'core_model_parameter_type': 'instance()',
                                     'default': 'ReLU',
                                     'levels': ['ReLU', 'LeakyReLU'],
                                     'lower': 0,
                                     'transform': 'None',
                                     'type': 'factor',
                                     'upper': 1},
                          'batch_size': {'default': 4,
                                         'lower': 3,
                                         'transform': 'transform_power_2_int',
                                         'type': 'int',
                                         'upper': 6},
                          'dropout_prob': {'default': 0.01,
                                           'lower': 0.005,
                                           'transform': 'None',
                                           'type': 'float',
                                           'upper': 0.25},
                          'epochs': {'default': 4,
                                     'lower': 2,
                                     'transform': 'transform_power_2_int',
                                     'type': 'int',
                                     'upper': 4},
                          'initialization': {'core_model_parameter_type': 'str',
                                             'default': 'Default',
                                             'levels': ['Default'],
                                             'lower': 0,
                                             'transform': 'None',
                                             'type': 'factor',
                                             'upper': 0},
                          'l1': {'default': 3,
                                 'lower': 3,
                                 'transform': 'transform_power_2_int',
                                 'type': 'int',
                                 'upper': 4},
                          'lr_mult': {'default': 1.0,
                                      'lower': 0.25,
                                      'transform': 'None',
```

```
                                        'type': 'float',
                                        'upper': 5.0},
                             'optimizer': {'class_name': 'torch.optim',
                                           'core_model_parameter_type': 'str',
                                           'default': 'SGD',
                                           'levels': ['Adadelta',
                                                      'Adamax',
                                                      'Adagrad'],
                                           'lower': 0,
                                           'transform': 'None',
                                           'type': 'factor',
                                           'upper': 2},
                             'patience': {'default': 2,
                                          'lower': 2,
                                          'transform': 'transform_power_2_int',
                                          'type': 'int',
                                          'upper': 3}},
'counter': 0,
'data': None,
'data_dir': './data',
'data_module': None,
'data_set': <torch.utils.data.dataset.TensorDataset object at 0x2d4dc8b50>,
'design': None,
'device': 'mps',
'devices': 1,
'enable_progress_bar': False,
'eval': None,
'fun_evals': inf,
'fun_repeats': 1,
'horizon': None,
'infill_criterion': 'y',
'k_folds': 3,
'log_graph': False,
'log_level': 50,
'loss_function': None,
'lower': array([3. , 4. , 1. , 0. , 0. , 0. , 0.1, 2. , 0. ]),
'max_time': 1,
'metric_params': {},
'metric_river': None,
'metric_sklearn': None,
'metric_torch': None,
'model_dict': {},
'n_points': 1,
```

```
'n_samples': None,
'noise': False,
'num_workers': 0,
'ocba_delta': 0,
'oml_grace_period': None,
'optimizer': None,
'path': None,
'prep_model': None,
'save_model': False,
'seed': 123,
'show_batch_interval': 1000000,
'show_models': False,
'show_progress': True,
'shuffle': None,
'sigma': 0.0,
'spot_tensorboard_path': 'runs/spot_logs/033_p040025_2024-02-24_03-37-33',
'spot_writer': <torch.utils.tensorboard.writer.SummaryWriter object at 0x1721b5cd0>,
'target_column': None,
'task': None,
'test': None,
'test_seed': 1234,
'test_size': 0.3,
'tolerance_x': 1.4901161193847656e-08,
'train': None,
'upper': array([ 8.  ,  9.  ,  4.  ,  5.  , 11.  ,  0.25, 10.  ,  6.  ,  2.  ]),
'var_name': ['l1',
             'epochs',
             'batch_size',
             'act_fn',
             'optimizer',
             'dropout_prob',
             'lr_mult',
             'patience',
             'initialization'],
'var_type': ['int',
             'int',
             'int',
             'factor',
             'factor',
             'float',
             'float',
             'int',
             'factor'],
```

```
 'verbosity': 1,
 'weight_coeff': 0.0,
 'weights': 1.0}
```

24.8.4 Starting the Hyperparameter Tuning

The `spotPython` hyperparameter tuning is started by calling the `Spot` function [SOURCE].

```
from spotPython.spot import spot
spot_tuner = spot.Spot(fun=fun,
                       fun_control=fun_control,
                       design_control=design_control,
                       surrogate_control=surrogate_control)
spot_tuner.run()
```

```
In fun(): config:
{'act_fn': LeakyReLU(),
 'batch_size': 16,
 'dropout_prob': 0.020345615289778483,
 'epochs': 8,
 'initialization': 'Default',
 'l1': 16,
 'lr_mult': 3.5380370864571606,
 'optimizer': 'Adamax',
 'patience': 8}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
LightDataModule setup(): test_size: 0.3
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 12552
train_model(): Train set size: 20501
train_model(): Batch size: 16
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
```

```
LightDataModule setup(): test_size: 0.3
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 8785
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 20501
LightDataModule: train_dataloader(). batch_size: 16
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
LightDataModule setup(): test_size: 0.3
LightDataModule: val_dataloader(). Training set size: 8785
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 27.36409568786621, 'hp_metric': 27.36409568786621}

In fun(): config:
{'act_fn': ReLU(),
 'batch_size': 16,
 'dropout_prob': 0.23254269132436722,
 'epochs': 4,
 'initialization': 'Default',
 'l1': 8,
 'lr_mult': 0.6593438339617097,
 'optimizer': 'Adadelta',
 'patience': 4}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
LightDataModule setup(): test_size: 0.3
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 12552
train_model(): Train set size: 20501
train_model(): Batch size: 16
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
```

```
LightDataModule setup(): test_size: 0.3
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 8785
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 20501
LightDataModule: train_dataloader(). batch_size: 16
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
LightDataModule setup(): test_size: 0.3
LightDataModule: val_dataloader(). Training set size: 8785
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 29.57465171813965, 'hp_metric': 29.57465171813965}

In fun(): config:
{'act_fn': LeakyReLU(),
 'batch_size': 32,
 'dropout_prob': 0.15478450721867254,
 'epochs': 16,
 'initialization': 'Default',
 'l1': 8,
 'lr_mult': 2.628500799878493,
 'optimizer': 'Adagrad',
 'patience': 8}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
LightDataModule setup(): test_size: 0.3
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 12552
train_model(): Train set size: 20501
train_model(): Batch size: 32
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
```

```
LightDataModule setup(): test_size: 0.3
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 8785
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 20501
LightDataModule: train_dataloader(). batch_size: 32
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
LightDataModule setup(): test_size: 0.3
LightDataModule: val_dataloader(). Training set size: 8785
LightDataModule: val_dataloader(). batch_size: 32
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 29.49106788635254, 'hp_metric': 29.49106788635254}

In fun(): config:
{'act_fn': LeakyReLU(),
 'batch_size': 16,
 'dropout_prob': 0.019641823176285617,
 'epochs': 8,
 'initialization': 'Default',
 'l1': 16,
 'lr_mult': 3.537067177180294,
 'optimizer': 'Adamax',
 'patience': 8}
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
LightDataModule setup(): test_size: 0.3
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
LightDataModule: setup(). stage: predict
train_model(): Test set size: 12552
train_model(): Train set size: 20501
train_model(): Batch size: 16
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
```

```
LightDataModule setup(): test_size: 0.3
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 8785
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 20501
LightDataModule: train_dataloader(). batch_size: 16
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.VALIDATING
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
LightDataModule setup(): test_size: 0.3
LightDataModule: val_dataloader(). Training set size: 8785
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
train_model result: {'val_loss': 26.50702667236328, 'hp_metric': 26.50702667236328}
spotPython tuning: 26.50702667236328 [##########] 100.00% Done...


       Validate metric              DataLoader 0

          hp_metric               27.36409568786621
          val_loss                27.36409568786621



       Validate metric              DataLoader 0

          hp_metric               29.57465171813965
          val_loss                29.57465171813965



       Validate metric              DataLoader 0

          hp_metric               29.49106788635254
          val_loss                29.49106788635254
```

```
       Validate metric              DataLoader 0

          hp_metric               26.50702667236328
          val_loss                26.50702667236328


<spotPython.spot.spot.Spot at 0x107949c50>
```

24.9 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard.

```
tensorboard --logdir="runs/"
```

Further information can be found in the PyTorch Lightning documentation for Tensorboard.

24.10 Step 10: Results

After the hyperparameter tuning run is finished, the results can be analyzed.

```
if spot_tuner.noise:
    print(spot_tuner.min_mean_X)
    print(spot_tuner.min_mean_y)
else:
    print(spot_tuner.min_X)
    print(spot_tuner.min_y)
```

```
[4.         3.         4.         1.         1.         0.01964182
 3.53706718 3.        ]
26.50702667236328
```

```
spot_tuner.plot_progress(log_y=False,
    filename="./figures/" + PREFIX +"_progress.png")
```

Figure 24.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

```
from spotPython.utils.eda import gen_design_table
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

```
| name           | type   | default   |   lower |   upper | tuned                | transform
|----------------|--------|-----------|---------|---------|----------------------|----------
| l1             | int    | 3         |     3.0 |     4.0 | 4.0                  | transform_
| epochs         | int    | 4         |     2.0 |     4.0 | 3.0                  | transform_
| batch_size     | int    | 4         |     3.0 |     6.0 | 4.0                  | transform_
| act_fn         | factor | ReLU      |     0.0 |     1.0 | LeakyReLU            | None
| optimizer      | factor | SGD       |     0.0 |     2.0 | Adamax               | None
| dropout_prob   | float  | 0.01      |   0.005 |    0.25 | 0.019641823176285617 | None
| lr_mult        | float  | 1.0       |    0.25 |     5.0 | 3.537067177180294    | None
| patience       | int    | 2         |     2.0 |     3.0 | 3.0                  | transform_
| initialization | factor | Default   |     0.0 |     0.0 | Default              | None
```

```
spot_tuner.plot_importance(threshold=0.025,
    filename="./figures/" + PREFIX + "_importance.png")
```

Figure 24.2: Variable importance plot, threshold 0.025.

24.10.1 Get the Tuned Architecture

```
from spotPython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner, fun_control)
print(config)
```

```
{'l1': 16, 'epochs': 8, 'batch_size': 16, 'act_fn': LeakyReLU(), 'optimizer': 'Adamax', 'dro
```

- Test on the full data set

```
from spotPython.light.testmodel import test_model
test_model(config, fun_control)
```

```
LightDataModule: setup(). stage: None
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
LightDataModule setup(): test_size: 0.3
LightDataModule: setup(). stage: fit
LightDataModule: setup(). stage: test
```

```
LightDataModule: setup(). stage: predict
LightDataModule: setup(). stage: TrainerFn.FITTING
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
LightDataModule setup(): test_size: 0.3
LightDataModule: setup(). stage: fit
LightDataModule: val_dataloader(). Training set size: 8785
LightDataModule: val_dataloader(). batch_size: 16
LightDataModule: val_dataloader(). num_workers: 0
LightDataModule: train_dataloader(). Training set size: 20501
LightDataModule: train_dataloader(). batch_size: 16
LightDataModule: train_dataloader(). num_workers: 0
LightDataModule: setup(). stage: TrainerFn.TESTING
LightDataModule setup(): full_train_size: 0.7
LightDataModule setup(): val_size: 0.21
LightDataModule setup(): train_size: 0.49
LightDataModule setup(): test_size: 0.3
LightDataModule: setup(). stage: test
LightDataModule: test_dataloader(). Test set size: 12552
LightDataModule: test_dataloader(). batch_size: 16
LightDataModule: test_dataloader(). num_workers: 0
test_model result: {'val_loss': 25.852802276611328, 'hp_metric': 25.852802276611328}
```

Test metric	DataLoader 0
hp_metric	25.852802276611328
val_loss	25.852802276611328

```
(25.852802276611328, 25.852802276611328)
```

```python
from spotPython.light.loadmodel import load_light_from_checkpoint

model_loaded = load_light_from_checkpoint(config, fun_control)
```

```
config: {'l1': 16, 'epochs': 8, 'batch_size': 16, 'act_fn': LeakyReLU(), 'optimizer': 'Adamax
Loading model with 16_8_16_LeakyReLU_Adamax_0.0196_3.5371_8_Default_TEST from runs/saved_mode
Model: NetLightRegression(
  (layers): Sequential(
```

```
    (0): Linear(in_features=6, out_features=16, bias=True)
    (1): LeakyReLU()
    (2): Dropout(p=0.019641823176285617, inplace=False)
    (3): Linear(in_features=16, out_features=8, bias=True)
    (4): LeakyReLU()
    (5): Dropout(p=0.019641823176285617, inplace=False)
    (6): Linear(in_features=8, out_features=8, bias=True)
    (7): LeakyReLU()
    (8): Dropout(p=0.019641823176285617, inplace=False)
    (9): Linear(in_features=8, out_features=4, bias=True)
    (10): LeakyReLU()
    (11): Dropout(p=0.019641823176285617, inplace=False)
    (12): Linear(in_features=4, out_features=1, bias=True)
  )
)
```

```
filename = "./figures/" + PREFIX
spot_tuner.plot_important_hyperparameter_contour(filename=filename)
```

```
l1:  100.0
act_fn:  0.05127949950884306
optimizer:  0.036389675225087334
impo: [['l1', 100.0], ['epochs', 0.0014423222599760528], ['batch_size', 0.0014423222599760528
impo after select: [['l1', 100.0], ['epochs', 0.0014423222599760528], ['batch_size', 0.00144
```

Figure 24.3: Contour plots.

24.10.2 Parallel Coordinates Plot

```
spot_tuner.parallel_plot()
```

```
Unable to display output for mime type(s): text/html
```

Parallel coordinates plots

```
Unable to display output for mime type(s): text/html
```

24.10.3 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].

```
from spotPython.light.cvmodel import cv_model
set_control_key_value(control_dict=fun_control,
                        key="k_folds",
                        value=2,
                        replace=True)
set_control_key_value(control_dict=fun_control,
                        key="test_size",
                        value=0.6,
                        replace=True)
cv_model(config, fun_control)
```

```
k: 0
Train Dataset Size: 20918
Val Dataset Size: 20919
train_model result: {'val_loss': 27.558460235595703, 'hp_metric': 27.558460235595703}
k: 1
Train Dataset Size: 20919
Val Dataset Size: 20918
train_model result: {'val_loss': 27.35320281982422, 'hp_metric': 27.35320281982422}


       Validate metric              DataLoader 0

          hp_metric              27.558460235595703
          val_loss               27.558460235595703



       Validate metric              DataLoader 0

          hp_metric               27.35320281982422
          val_loss                27.35320281982422
```

```
27.45583152770996
```

24.10.4 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)
```

24.10.5 Visualizing the Activation Distribution (Under Development)

> Reference:
>
> - The following code is based on [PyTorch Lightning TUTORIAL 2: ACTIVATION FUNCTIONS], Author: Phillip Lippe, License: [CC BY-SA], Generated: 2023-03-15T09:52:39.179933.

After we have trained the models, we can look at the actual activation values that find inside the model. For instance, how many neurons are set to zero in ReLU? Where do we find most values in Tanh? To answer these questions, we can write a simple function which takes a trained model, applies it to a batch of images, and plots the histogram of the activations inside the network:

```
from spotPython.torch.activation import Sigmoid, Tanh, ReLU, LeakyReLU, ELU, Swish
act_fn_by_name = {"sigmoid": Sigmoid, "tanh": Tanh, "relu": ReLU, "leakyrelu": LeakyReLU, "el
```

```
from spotPython.hyperparameters.values import get_one_config_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
config = get_one_config_from_X(X, fun_control)
model = fun_control["core_model"](**config, _L_in=64, _L_out=11)
model
```

```
NetLightRegression(
  (layers): Sequential(
    (0): Linear(in_features=64, out_features=16, bias=True)
    (1): LeakyReLU()
```

```
    (2): Dropout(p=0.019641823176285617, inplace=False)
    (3): Linear(in_features=16, out_features=8, bias=True)
    (4): LeakyReLU()
    (5): Dropout(p=0.019641823176285617, inplace=False)
    (6): Linear(in_features=8, out_features=8, bias=True)
    (7): LeakyReLU()
    (8): Dropout(p=0.019641823176285617, inplace=False)
    (9): Linear(in_features=8, out_features=4, bias=True)
    (10): LeakyReLU()
    (11): Dropout(p=0.019641823176285617, inplace=False)
    (12): Linear(in_features=4, out_features=11, bias=True)
  )
)
```

```
# from spotPython.utils.eda import visualize_activations
# visualize_activations(model, color=f"C{0}")
```

25 Explainable AI with SpotPython and Pytorch

```
from torch.utils.data import DataLoader
from spotPython.utils.init import fun_control_init
from spotPython.hyperparameters.values import set_control_key_value
from spotPython.data.diabetes import Diabetes
from spotPython.light.regression.netlightregression import NetLightRegression
from spotPython.hyperdict.light_hyper_dict import LightHyperDict
from spotPython.hyperparameters.values import add_core_model_to_fun_control
from spotPython.hyperparameters.values import (
        get_default_hyperparameters_as_array, get_one_config_from_X)
from spotPython.hyperparameters.values import set_control_key_value
from spotPython.plot.xai import (get_activations, get_gradients, get_weights, plot_nn_values_
fun_control = fun_control_init(
    _L_in=10, # 10: diabetes
    _L_out=1,
    )
dataset = Diabetes()
set_control_key_value(control_dict=fun_control,
                        key="data_set",
                        value=dataset,
                        replace=True)
add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=NetLightRegression,
                              hyper_dict=LightHyperDict)
X = get_default_hyperparameters_as_array(fun_control)
config = get_one_config_from_X(X, fun_control)
_L_in = fun_control["_L_in"]
_L_out = fun_control["_L_out"]
model = fun_control["core_model"](**config, _L_in=_L_in, _L_out=_L_out)
batch_size= config["batch_size"]
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=False)
```

```python
get_activations(model, fun_control=fun_control, batch_size=batch_size, device = "cpu")
```

```
net: NetLightRegression(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=8, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.01, inplace=False)
    (3): Linear(in_features=8, out_features=4, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.01, inplace=False)
    (6): Linear(in_features=4, out_features=4, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.01, inplace=False)
    (9): Linear(in_features=4, out_features=2, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.01, inplace=False)
    (12): Linear(in_features=2, out_features=1, bias=True)
  )
)
```

```
{0: array([ 1.43207282e-01,  6.29712082e-03,  1.04200497e-01, -3.79188173e-03,
        -1.74976081e-01, -7.97475874e-02, -2.00860098e-01,  2.48444736e-01,
         1.42530382e-01, -2.86847632e-03,  3.61538231e-02, -5.21567538e-02,
        -2.15294853e-01, -1.26742452e-01, -1.79230243e-01,  2.73077697e-01,
         1.36738747e-01,  8.57900176e-03,  1.01677164e-01,  3.27536091e-03,
        -1.92429125e-01, -7.95854479e-02, -1.84092522e-01,  2.72164375e-01,
         1.51459932e-01,  3.70034538e-02,  4.94864434e-02, -6.36564642e-02,
        -1.63678646e-01, -1.26617596e-01, -2.05547154e-01,  2.25242063e-01,
         1.54910132e-01,  4.92912624e-03,  6.90693632e-02, -3.28048877e-02,
        -1.77523270e-01, -1.17699921e-01, -1.95609123e-01,  2.50784487e-01,
         1.66618377e-01,  1.22015951e-02,  2.58807316e-02, -8.16192776e-02,
        -2.00623482e-01, -1.17052853e-01, -1.86843857e-01,  2.40996510e-01,
         1.80479109e-01,  3.72159854e-02,  3.55244167e-02, -3.60636115e-02,
        -2.09616780e-01, -1.19843856e-01, -1.44335642e-01,  2.73970902e-01,
         1.46006003e-01, -1.83095373e-02,  8.83664042e-02,  2.28608586e-02,
        -1.77115664e-01, -1.37761638e-01, -1.90622538e-01,  2.85049856e-01,
         1.44436464e-01,  1.36893094e-02,  6.65568933e-02, -2.01083720e-04,
        -1.99043870e-01, -1.11171007e-01, -1.76820531e-01,  2.78549373e-01,
         1.31597325e-01,  1.31126186e-02,  5.92438355e-02, -6.50760308e-02,
        -1.55642599e-01, -1.12090096e-01, -2.32182071e-01,  2.25448400e-01,
         2.09733546e-01,  4.48576249e-02,  1.76887661e-02, -7.26176351e-02,
```

```
       -1.81560591e-01, -1.18118793e-01, -1.55840069e-01,  2.45131850e-01,
        1.57539800e-01,  4.57477495e-02,  8.64019692e-02,  1.06538832e-02,
       -2.25713193e-01, -8.36062431e-02, -1.51326194e-01,  2.42097050e-01,
        1.46130219e-01, -6.08363096e-03,  4.69235368e-02, -4.06553932e-02,
       -1.90215483e-01, -1.30105391e-01, -1.91207454e-01,  2.75829703e-01,
        1.37035578e-01,  1.32784406e-02,  8.11730623e-02, -2.83420049e-02,
       -1.72134370e-01, -1.05717532e-01, -1.93411276e-01,  2.68321246e-01,
        1.24822736e-01, -2.49985531e-02,  5.46513572e-02, -3.76938097e-02,
       -2.02080101e-01, -1.29510283e-01, -1.99880868e-01,  2.84415126e-01,
        1.36025175e-01,  2.10405551e-02,  1.25923336e-01, -1.76883545e-02,
       -1.46617338e-01, -1.00234658e-01, -2.21794963e-01,  2.05139250e-01],
      dtype=float32),
3: array([-0.09106569,  0.15831017,  0.29874575, -0.05709065, -0.07168067,
        0.13238071,  0.29310873, -0.04537551, -0.08868651,  0.15093939,
        0.29576218, -0.0508837 , -0.07256822,  0.15756649,  0.29804155,
       -0.06024086, -0.07925774,  0.15159754,  0.29655144, -0.05204485,
       -0.06510481,  0.14707124,  0.2955585 , -0.05045141, -0.05945833,
        0.15397519,  0.28643152, -0.03937227, -0.0780265 ,  0.1443048 ,
        0.2993904 , -0.04338943, -0.07745007,  0.1438258 ,  0.29152495,
       -0.04569358, -0.08201659,  0.14775375,  0.3020632 , -0.06361471,
       -0.05014775,  0.16657498,  0.28808075, -0.04191205, -0.07614301,
        0.16806594,  0.29809946, -0.05615523, -0.07369395,  0.13612927,
        0.2925982 , -0.04455032, -0.08367015,  0.14735378,  0.29441217,
       -0.05101945, -0.07929114,  0.12925598,  0.29300398, -0.04631315,
       -0.09977546,  0.1741175 ,  0.30642375, -0.07330882], dtype=float32),
6: array([ 0.02894721, -0.15329668,  0.0478624 ,  0.5073338 ,  0.03414171,
       -0.1624101 ,  0.0582981 ,  0.5058923 ,  0.0301194 , -0.15560818,
        0.05099656,  0.5068564 ,  0.02897662, -0.15344843,  0.04822758,
        0.5072659 ,  0.03012998, -0.15550745,  0.05065323,  0.5069246 ,
        0.03103478, -0.1570965 ,  0.05247599,  0.50667256,  0.02730933,
       -0.15252227,  0.0509877 ,  0.5065358 ,  0.03256607, -0.15896471,
        0.05305116,  0.5067359 ,  0.03095146, -0.15756464,  0.05418621,
        0.5063291 ,  0.03229896, -0.1581411 ,  0.05142961,  0.50702184,
        0.02454497, -0.14787357,  0.04604906,  0.50718296,  0.02638294,
       -0.14930864,  0.04427201,  0.5077407 ,  0.03309863, -0.16082475,
        0.05695035,  0.50603575,  0.03071198, -0.15675312,  0.05250891,
        0.5066291 ,  0.03489432, -0.16362445,  0.05948593,  0.50574666,
        0.02671532, -0.14859803,  0.04098557,  0.5084204 ], dtype=float32),
9: array([0.04397329, 0.23183572, 0.04112439, 0.22675759, 0.0430866 ,
       0.23046201, 0.04386139, 0.2317175 , 0.04319487, 0.23055825,
       0.04269706, 0.22967225, 0.04286424, 0.23156166, 0.04263979,
       0.2289063 , 0.0421553 , 0.2292049 , 0.04312573, 0.22948454,
       0.04418794, 0.23408437, 0.04489119, 0.23388621, 0.0414625 ,
```

```
        0.22755873, 0.0426609 , 0.22978865, 0.04081305, 0.22611658,
        0.04594607, 0.23471704], dtype=float32),
 12: array([-0.30635476, -0.30988604, -0.3073418 , -0.30644947, -0.30726004,
        -0.30787635, -0.306807  , -0.308307  , -0.3082779 , -0.3078606 ,
        -0.30507785, -0.3049926 , -0.30935943, -0.30782318, -0.3103186 ,
        -0.30425358], dtype=float32)}
```

```
get_gradients(model, fun_control=fun_control, batch_size=batch_size, device = "cpu")
```

```
{'layers.0.weight': array([ 0.10417589, -0.04161514,  0.10597268,  0.02180895,  0.12001497,
         0.0289035 ,  0.01146171,  0.08183315,  0.2495192 ,  0.5108763 ,
         0.14668097, -0.07902835,  0.00912531,  0.02640062,  0.14108549,
         0.06816658,  0.14256881, -0.00347908,  0.07373644,  0.23171763,
         0.08313344, -0.0332093 ,  0.08456729,  0.01740377,  0.09577318,
         0.0230653 ,  0.00914656,  0.0653037 ,  0.1991189 ,  0.4076846 ,
         0.04405227,  0.03805925,  0.015035  ,  0.0069457 ,  0.0094994 ,
         0.03021198, -0.01876849,  0.02160799, -0.03238906, -0.02050959,
         0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        -0.05415884,  0.02163483, -0.05509295, -0.01133801, -0.06239325,
        -0.01502632, -0.0059587 , -0.04254333, -0.12971975, -0.2655938 ],
       dtype=float32),
 'layers.3.weight': array([ 0.0000000e+00,  0.0000000e+00,  0.0000000e+00,  0.0000000e+00,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00,  0.0000000e+00,
        -5.8896484e+00, -6.3058013e-01, -2.5641673e+00, -8.9936234e-02,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00, -1.0009734e+01,
         5.1539743e-01,  5.5181440e-02,  2.2438775e-01,  7.8702327e-03,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00,  8.7594193e-01,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00,  0.0000000e+00,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00,  0.0000000e+00],
       dtype=float32),
 'layers.6.weight': array([ 0.        ,  7.6445217, 15.007772 ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        , 11.027901 ,
        21.650045 ,  0.        ,  0.        ,  3.458755 ,  6.7902493,
         0.        ], dtype=float32),
 'layers.9.weight': array([ -2.3285942,   0.        ,  -3.9471323, -39.11015  ,  -4.6057286,
          0.        ,  -7.8070364, -77.35598  ], dtype=float32),
 'layers.12.weight': array([-12.126856, -64.91129 ], dtype=float32)}
```

```
get_weights(model)
```

```
{'Layer 0': array([-0.12895013,  0.01047491, -0.15705723,  0.11925378, -0.26944348,
         0.23180884, -0.22984707, -0.25141433, -0.19982024,  0.1432175 ,
        -0.11684369,  0.11833665, -0.2683918 , -0.19186287, -0.11611126,
        -0.06214499, -0.24123858,  0.20706302, -0.07457636,  0.10150522,
         0.22361842,  0.05891513,  0.08647271,  0.3052416 , -0.1426217 ,
         0.10016554, -0.14069483,  0.22599207,  0.25255734, -0.29155323,
         0.26994652,  0.1510033 ,  0.13780165,  0.13018303,  0.26287985,
        -0.04175457, -0.26743335, -0.09074122, -0.2227112 ,  0.02090477,
        -0.05904209, -0.16961981, -0.02875187,  0.2995954 , -0.0249426 ,
         0.01004026, -0.04931906,  0.04971322,  0.28176296,  0.19337103,
         0.11224869,  0.06871963,  0.07456426,  0.12216929, -0.04086405,
        -0.29390487, -0.19555901,  0.2699275 ,  0.01890202, -0.25616774,
         0.04987781,  0.26129004, -0.29883513, -0.21289697, -0.12594265,
         0.0126926 , -0.07375361, -0.03475064, -0.30828732,  0.14808287,
         0.2775668 ,  0.19329055, -0.22393112, -0.25491226,  0.13131432,
         0.00710202,  0.12963155, -0.3090024 , -0.01885445,  0.22301763],
       dtype=float32),
 'Layer 3': array([ 0.19455571,  0.12364562, -0.2711233 ,  0.2728095 ,  0.11085409,
         0.24458633, -0.13908438,  0.07495222,  0.34520328,  0.23782092,
         0.28354865, -0.07424083,  0.26936427, -0.2769144 ,  0.03057847,
        -0.19906998, -0.08245403, -0.09054411,  0.02645254,  0.32178298,
         0.17503859, -0.00149773,  0.2509683 , -0.1811804 ,  0.18221132,
        -0.03278595, -0.06152213,  0.0413917 , -0.27085608,  0.04085568,
         0.11887809,  0.302264  ], dtype=float32),
 'Layer 6': array([ 0.4752962 , -0.24824601,  0.22039747,  0.19587505,  0.13966405,
         0.39540154, -0.20208222,  0.13140953,  0.00280607, -0.3760708 ,
        -0.12140697, -0.33391154,  0.22107768,  0.04494798,  0.04898232,
        -0.15168536], dtype=float32),
 'Layer 9': array([ 0.07573527, -0.22145915, -0.30541402,  0.03821951, -0.3709231 ,
        -0.3758251 , -0.3254385 , -0.1698224 ], dtype=float32),
 'Layer 12': array([0.2738903, 0.5417278], dtype=float32)}
```

```
visualize_activations(model, fun_control=fun_control, batch_size=batch_size, device = "cpu",
```

```
net: NetLightRegression(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=8, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.01, inplace=False)
```

```
    (3): Linear(in_features=8, out_features=4, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.01, inplace=False)
    (6): Linear(in_features=4, out_features=4, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.01, inplace=False)
    (9): Linear(in_features=4, out_features=2, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.01, inplace=False)
    (12): Linear(in_features=2, out_features=1, bias=True)
  )
)
```

Activations Plot for 3
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−0.05
Value

Activations Plot for 6
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
0.5
0.4
0.3
0.2
0.1
0.0
−0.1
Value

Activations Plot for 9
0
1
2
3
4
5
0
1
2
3
4
5
0.20
0.15
0.10
0.05
0.00
Value

```
visualize_weights_distributions(model, color=f"C{0}")
```

```
n:5
```

```
visualize_gradient_distributions(model, fun_control, batch_size=batch_size, color=f"C{0}")
```

```
n:5
```

```
visualize_weights(model, absolute=True, cmap="gray", figsize=(6, 6))
```

Weights Plot for Layer 0
0
1
2
3
4
5
6
7
8
0
2
4
6
8
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Value

Weights Plot for Layer 3
0
1
2
3
4
5
0
1
2
3
4
5
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Value

Weights Plot for Layer 6
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0.4
0.3
0.2
0.1
Value

Weights Plot for Layer 9
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Value

```
visualize_gradients(model, fun_control, batch_size, absolute=True, cmap="BlueWhiteRed", figsi
```

Gradients Plot for layers.0.weight
0
1
2
3
4
5
6
7
8
0
2
4
6
8
0.5
0.4
0.3
0.2
0.1
0.0
Value

Gradients Plot for layers.3.weight
0
1
2
3
4
5
0
1
2
3
4
5
10
8
6
4
2
0
Value

Gradients Plot for layers.6.weight
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Value

Gradients Plot for layers.9.weight
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
70
60
50
40
30
20
10
0
Value

```
visualize_activations(model, fun_control=fun_control, batch_size=batch_size, device = "cpu")
```

```
net: NetLightRegression(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=8, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.01, inplace=False)
    (3): Linear(in_features=8, out_features=4, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.01, inplace=False)
    (6): Linear(in_features=4, out_features=4, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.01, inplace=False)
    (9): Linear(in_features=4, out_features=2, bias=True)
    (10): ReLU()
```

```
    (11): Dropout(p=0.01, inplace=False)
    (12): Linear(in_features=2, out_features=1, bias=True)
  )
)
```

Activations Plot for 3
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
0.30
0.25
0.20
0.15
0.10
0.05
Value

Activations Plot for 6
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
0.5
0.4
0.3
0.2
0.1
Value

Activations Plot for 9
0
1
2
3
4
5
0
1
2
3
4
5
0.20
0.15
0.10
0.05
0.00
Value

```
visualize_activations(model, fun_control=fun_control, batch_size=batch_size, device = "cpu")
```

```
net: NetLightRegression(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=8, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.01, inplace=False)
    (3): Linear(in_features=8, out_features=4, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.01, inplace=False)
    (6): Linear(in_features=4, out_features=4, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.01, inplace=False)
    (9): Linear(in_features=4, out_features=2, bias=True)
    (10): ReLU()
```

```
    (11): Dropout(p=0.01, inplace=False)
    (12): Linear(in_features=2, out_features=1, bias=True)
  )
)
```

Activations Plot for 3
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
0.30
0.25
0.20
0.15
0.10
0.05
Value

Activations Plot for 6
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
0.5
0.4
0.3
0.2
0.1
Value

Activations Plot for 9
0
1
2
3
4
5
0
1
2
3
4
5
0.20
0.15
0.10
0.05
0.00
Value

Activations Plot for 12
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0.310
0.309
0.308
0.307
0.306
0.305
Value

26 HPT PyTorch Lightning Transformer: Introduction

In this tutorial, we will introduce transformer.

26.1 Decoder-Only Transformer

26.1.1 Embedding

26.1.1.1 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. For simplicity, we will use a 2-dimensional output vector. 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.

26.1.1.2 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.

26.1.2 Masked Self-Attention

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:

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 mask-self attention value for the token.

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).

26.1.3 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.

26.1.4 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. With a decoder-only transformer, the masked self-attention method is used to take the relationships into account.

27 Example: Word Predictor

- The following code is based on https://github.com/phlippe/uvadlc_notebooks/tree/master (Author: Phillip Lippe)

```
## Standard libraries
import os
import numpy as np
import random
import math
import json
from functools import partial

## Imports for plotting
import matplotlib.pyplot as plt
plt.set_cmap('cividis')
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()

## 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

## Torchvision
import torchvision
```

```
from torchvision.datasets import CIFAR100
from torchvision import transforms

# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by defau
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial6"

# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print("Device:", device)
```

```
Device: cpu
```

```
<Figure size 1650x1050 with 0 Axes>
```

```
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)

# 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("Something went wrong. Please try to download the file from the GDrive fol
```

```
def scaled_dot_product(q, k, v, mask=None):
    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
```

```
seq_len, d_k = 3, 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],
        [ 0.2345,  0.2303],
        [-1.1229, -0.1863]])
K
 tensor([[ 2.2082, -0.6380],
        [ 0.4617,  0.2674],
```

```
        [ 0.5349,  0.8094]])
V
 tensor([[ 1.1103, -1.6898],
        [-0.9890,  0.9580],
        [ 1.3221,  0.8172]])
Values
 tensor([[ 0.5698, -0.1520],
        [ 0.5379, -0.0265],
        [ 0.2246,  0.5556]])
Attention
 tensor([[0.4028, 0.2886, 0.3086],
        [0.3538, 0.3069, 0.3393],
        [0.1303, 0.4630, 0.4067]])
```

28 HPT PyTorch Lightning Transformer: Diabetes

In this tutorial, we will show how `spotPython` can be integrated into the `PyTorch` Lightning training workflow for a regression task.

This chapter describes the hyperparameter tuning of a `PyTorch Lightning` network on the Diabetes data set. 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.

28.1 Step 1: Setup

- Before we consider the detailed experimental setup, we select the parameters that affect run time, initial design size, etc.
- The parameter `MAX_TIME` specifies the maximum run time in seconds.
- The parameter `INIT_SIZE` specifies the initial design size.
- 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.

```
from spotPython.utils.device import getDevice
from math import inf

MAX_TIME = 1
FUN_EVALS = inf
INIT_SIZE = 10
WORKERS = 0
PREFIX="036"
DEVICE = getDevice()
DEVICES = 1
TEST_SIZE = 0.3
```

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.
- `WORKERS` is set to 0 for demonstration purposes. For real experiments, this should be increased. See the warnings that are printed when the number of workers is set to 0.

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.

28.2 Step 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,
    PREFIX=PREFIX,
    TENSORBOARD_CLEAN=True,
    device=DEVICE,
    enable_progress_bar=False,
    fun_evals=FUN_EVALS,
    log_level=10,
    max_time=MAX_TIME,
    num_workers=WORKERS,
    show_progress=True,
    test_size=TEST_SIZE,
    tolerance_x=np.sqrt(np.spacing(1)),
    )
```

28.3 Step 3: Loading the Diabetes Data Set

```
from spotPython.hyperparameters.values import set_control_key_value
from spotPython.data.diabetes import Diabetes
dataset = Diabetes()
set_control_key_value(control_dict=fun_control,
                        key="data_set",
                        value=dataset,
                        replace=True)
print(len(dataset))
```

Note: 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
```

28.4 Step 4: Preprocessing

Preprocessing is handled by `Lightning` and `PyTorch`. It is described in the LIGHTNING-DATAMODULE documentation. Here you can find information about the `transforms` methods.

28.5 Step 5: Select the Core Model (`algorithm`) and `core_model_hyper_dict`

`spotPython` includes the `NetLightRegression` class [SOURCE] for configurable neural networks. The class is imported here. It inherits from the class `Lightning.LightningModule`, which is the base class for all models in `Lightning`. `Lightning.LightningModule` is a subclass of `torch.nn.Module` and provides additional functionality for the training and testing of neural networks. The class `Lightning.LightningModule` is described in the Lightning documentation.

- Here we simply add the NN Model to the fun_control dictionary by calling the function `add_core_model_to_fun_control`:

```
from spotPython.light.regression.transformerlightregression import TransformerLightRegressio
from spotPython.hyperdict.light_hyper_dict import LightHyperDict
from spotPython.hyperparameters.values import add_core_model_to_fun_control
add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=TransformerLightRegression,
                              hyper_dict=LightHyperDict)
```

The hyperparameters of the model are specified in the `core_model_hyper_dict` dictionary [SOURCE].

28.6 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.

> Caution: Small number of epochs for demonstration purposes
>
> - `epochs` and `patience` are set to small values for demonstration purposes. These values are too small for a real application.
> - More resonable values are, e.g.:
> - `set_control_hyperparameter_value(fun_control, "epochs", [7, 9])` and
> - `set_control_hyperparameter_value(fun_control, "patience", [2, 7])`

```python
from spotPython.hyperparameters.values import set_control_hyperparameter_value

# set_control_hyperparameter_value(fun_control, "l1", [2, 3])
# set_control_hyperparameter_value(fun_control, "epochs", [5, 7])
# set_control_hyperparameter_value(fun_control, "batch_size", [3, 4])
# set_control_hyperparameter_value(fun_control, "optimizer", [
#                 "Adadelta",
#                 "Adagrad",
#                 "Adam",
#                 "Adamax",
#             ])
# set_control_hyperparameter_value(fun_control, "dropout_prob", [0.01, 0.1])
# set_control_hyperparameter_value(fun_control, "lr_mult", [0.5, 5.0])
# set_control_hyperparameter_value(fun_control, "patience", [3, 5])
# set_control_hyperparameter_value(fun_control, "act_fn",[
#                 "ReLU",
#                 "LeakyReLU",
#             ] )
set_control_hyperparameter_value(fun_control, "initialization",["Default"] )
```

Now, the dictionary `fun_control` contains all information needed for the hyperparameter tuning. Before the hyperparameter tuning is started, it is recommended to take a look at the experimental design. The method `gen_design_table` [SOURCE] generates a design table as follows:

```python
from spotPython.utils.eda import gen_design_table
print(gen_design_table(fun_control))
```

This allows to check if all information is available and if the information is correct.

> **Note: Hyperparameters of the Tuned Model and the `fun_control` Dictionary**
>
> The updated `fun_control` dictionary can be shown with the command `fun_control["core_model_hyper_dict"]`.

28.7 Step 7: Data Splitting, the Objective (Loss) Function and the Metric

28.7.1 Evaluation

The evaluation procedure requires the specification of two elements:

1. the way how the data is split into a train and a test set
2. the loss function (and a metric).

> Caution: Data Splitting in Lightning
>
> The data splitting is handled by `Lightning`.

28.7.2 Loss Function

The loss function is specified in the configurable network class [SOURCE] We will use MSE.

28.7.3 Metric

- Similar to the loss function, the metric is specified in the configurable network class [SOURCE].

> Caution: Loss Function and Metric in Lightning
>
> - The loss function and the metric are not hyperparameters that can be tuned with `spotPython`.
> - They are handled by `Lightning`.

28.8 Step 8: Calling the SPOT Function

28.8.1 Preparing the SPOT Call

```
from spotPython.utils.init import design_control_init, surrogate_control_init
design_control = design_control_init(init_size=INIT_SIZE)

surrogate_control = surrogate_control_init(noise=True,
                                           n_theta=2)
```

> **Note: Modifying Values in the Control Dictionaries**
>
> - The values in the control dictionaries can be modified with the function `set_control_key_value` [SOURCE], for example:

```
set_control_key_value(control_dict=surrogate_control,
                      key="noise",
                      value=True,
                      replace=True)
set_control_key_value(control_dict=surrogate_control,
                      key="n_theta",
                      value=2,
                      replace=True)
```

28.8.2 The 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`.

```
from spotPython.fun.hyperlight import HyperLight
fun = HyperLight(log_level=10).fun
```

28.8.3 Showing the fun_control Dictionary

```
import pprint
pprint.pprint(fun_control)
```

28.8.4 Starting the Hyperparameter Tuning

The `spotPython` hyperparameter tuning is started by calling the `Spot` function [SOURCE].

```
from spotPython.spot import spot
spot_tuner = spot.Spot(fun=fun,
                       fun_control=fun_control,
                       design_control=design_control,
                       surrogate_control=surrogate_control)
spot_tuner.run()
```

28.9 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard.

```
tensorboard --logdir="runs/"
```

Further information can be found in the PyTorch Lightning documentation for Tensorboard.

28.10 Step 10: Results

After the hyperparameter tuning run is finished, the results can be analyzed.

```
spot_tuner.plot_progress(log_y=False,
    filename="./figures/" + PREFIX +"_progress.png")
```

```
from spotPython.utils.eda import gen_design_table
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

```
spot_tuner.plot_importance(threshold=50,
    filename="./figures/" + PREFIX + "_importance.png")
```

28.10.1 Get the Tuned Architecture

```
from spotPython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner, fun_control)
print(config)
```

- Test on the full data set

```
from spotPython.light.testmodel import test_model
test_model(config, fun_control)
```

```
from spotPython.light.loadmodel import load_light_from_checkpoint

model_loaded = load_light_from_checkpoint(config, fun_control)
```

```
# filename = "./figures/" + PREFIX
filename = None
spot_tuner.plot_important_hyperparameter_contour(filename=filename, threshold=50)
```

28.10.2 Parallel Coordinates Plot

```
spot_tuner.parallel_plot()
```

28.10.3 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].

```
from spotPython.light.cvmodel import cv_model
set_control_key_value(control_dict=fun_control,
                        key="k_folds",
                        value=2,
                        replace=True)
set_control_key_value(control_dict=fun_control,
                        key="test_size",
                        value=0.6,
                        replace=True)
cv_model(config, fun_control)
```

28.10.4 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)
```

28.10.5 Visualizing the Activation Distribution (Under Development)

> Reference:
>
> - The following code is based on [PyTorch Lightning TUTORIAL 2: ACTIVATION FUNCTIONS], Author: Phillip Lippe, License: [CC BY-SA], Generated: 2023-03-15T09:52:39.179933.

After we have trained the models, we can look at the actual activation values that find inside the model. For instance, how many neurons are set to zero in ReLU? Where do we find most values in Tanh? To answer these questions, we can write a simple function which takes a trained model, applies it to a batch of images, and plots the histogram of the activations inside the network:

```
from spotPython.torch.activation import Sigmoid, Tanh, ReLU, LeakyReLU, ELU, Swish
act_fn_by_name = {"sigmoid": Sigmoid, "tanh": Tanh, "relu": ReLU, "leakyrelu": LeakyReLU, "el
```

```
from spotPython.hyperparameters.values import get_one_config_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
config = get_one_config_from_X(X, fun_control)
model = fun_control["core_model"](**config, _L_in=64, _L_out=11)
model
```

```
# from spotPython.utils.eda import visualize_activations
# visualize_activations(model, color=f"C{0}")
```

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
1
0
−1
−2
−2
−1
0
1
2

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 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 .gitignc

# 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.

> **i** 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.

> **i** 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`

> ℹ 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`.

> ℹ 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.

> ℹ 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.

:::{.callout-note} #### Difference Between `checkout` and `switch` `git checkout` and `git switch` are both commands for switching between branches in a Git repository. The main difference between the two commands is that `git switch` is a newer command specifically designed for branch switching, while `git checkout` is an older command that can be used for various tasks, including branch switching.

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.

Here's an example:

```
# 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.

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/numerisc` to your local machine, add a file, and upload the changes, you can follow these steps:

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-sommersemester2023

# 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:

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.

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_`

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 Introduction

C.1 Recommendations

Beginner's Guide to Python

D Documentation of the Sequential Parameter Optimization

This document describes the `Spot` features. The official `spotPython` documentation can be found here: https://sequential-parameter-optimization.github.io/spotPython/.

D.1 An Initial Example

```
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
import matplotlib.pyplot as plt
```

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
```

```
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x,y, "k")
plt.show()
```

```
from spotPython.utils.init import fun_control_init, design_control_init, surrogate_control_in
spot_1 = spot.Spot(fun=fun,
                   fun_control=fun_control_init(
                        lower = np.array([-10]),
                        upper = np.array([100]),
                        fun_evals = 7,
                        fun_repeats = 1,
                        max_time = inf,
                        noise = False,
                        tolerance_x = np.sqrt(np.spacing(1)),
                        var_type=["num"],
                        infill_criterion = "y",
                        n_points = 1,
                        seed=123,
                        log_level = 50),
                   design_control=design_control_init(
                        init_size=5,
                        repeats=1),
                   surrogate_control=surrogate_control_init(
                        noise=False,
                        min_theta=-4,
                        max_theta=3,
                        n_theta=1,
                        model_optimizer=differential_evolution,
                        model_fun_evals=10000))
```

```
spot_1.run()
```

```
spotPython tuning: 2.0106521524877827 [#########-] 85.71%
spotPython tuning: 0.01033163973935242 [##########] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x105d5ed90>
```

D.2 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.build.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 initialzaion 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.

```
from spotPython.utils.init import fun_control_init, design_control_init, optimizer_control_in
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()
```

D.3 The Spot Object

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:

```
from spotPython.spot import spot
spot_tuner = spot.Spot(fun=fun,
                       fun_control=fun_control,
                       design_control=design_control,
                       optimizer_control=optimizer_control,
                       surrogate_control=surrogate_control)
```

D.4 Run

```
spot_tuner.run()
```

```
spotPython tuning: 1.801603872454505e-05 [#######---] 73.33%
spotPython tuning: 1.801603872454505e-05 [########--] 80.00%
spotPython tuning: 1.801603872454505e-05 [#########-] 86.67%
spotPython tuning: 1.801603872454505e-05 [#########-] 93.33%
spotPython tuning: 1.801603872454505e-05 [##########] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2cbdcd490>
```

D.5 Print the Results

```
spot_tuner.print_results()
```

```
min y: 1.801603872454505e-05
x0: 0.0019077911677074135
x1: 0.003791618596979743

[['x0', 0.0019077911677074135], ['x1', 0.003791618596979743]]
```

D.6 Show the Progress

```
spot_tuner.plot_progress()
```

D.7 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()
```

D.8 Run With a Specific Start Design

To pass a specific start design, use the `X_start` argument of the `run` method.

```
spot_x0 = spot.Spot(fun=fun,
                    fun_control=fun_control_init(
                        lower = np.array([-10]),
                        upper = np.array([100]),
                        fun_evals = 7,
                        fun_repeats = 1,
                        max_time = inf,
                        noise = False,
                        tolerance_x = np.sqrt(np.spacing(1)),
                        var_type=["num"],
                        infill_criterion = "y",
                        n_points = 1,
                        seed=123,
                        log_level = 50),
```

```
                    design_control=design_control_init(
                        init_size=5,
                        repeats=1),
                    surrogate_control=surrogate_control_init(
                        noise=False,
                        min_theta=-4,
                        max_theta=3,
                        n_theta=1,
                        model_optimizer=differential_evolution,
                        model_fun_evals=10000))
spot_x0.run(X_start=np.array([0.5, -0.5]))
spot_x0.plot_progress()
```

```
spotPython tuning: 2.0106521524877827 [#########-] 85.71%
spotPython tuning: 0.01033163973935242 [##########] 100.00% Done...
```

D.9 Init: Build Initial Design

```
from spotPython.design.spacefilling import spacefilling
from spotPython.build.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
```

```
fun_control = {"sigma": 0,
               "seed": 123}

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]
```

D.10 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 ]]))
```

D.11 Surrogates

D.11.1 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.

D.12 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.

```python
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.Spot(fun=fun,
                     fun_control=fun_control,
                     design_control=design_control)
spot_1.run()
# To check whether the run was successfully completed,
# we compare the number of evaluated points to the specified
# number of points.
assert spot_1.y.shape[0] == n
```

```
[        nan         nan -0.02203599 -0.21843718  0.78240941         nan
 -0.3923345   0.67234256  0.31802454 -0.68898927 -0.75129705  0.97550354
  0.41757584         nan  0.82585329         nan -0.49274073         nan
 -0.17991251  0.1481835 ]
[-1.]
[nan]
[-0.14624037]
[0.166475]
[nan]
[-0.3352401]
[-0.47259301]
[0.95541987]
[0.17335968]
[-0.58552368]
[-0.20126111]
```

```
[-0.60100809]
[-0.97897336]
[-0.2748985]
[0.8359486]
[0.99035591]
[0.01641232]
[0.5629346]
```

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.

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. 2023. "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.

———. 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.

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, 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.

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.

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.

Forrester, Alexander, András Sóbester, and Andy Keane. 2008. *Engineering Design via Surrogate Modelling*. Wiley.

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.

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.

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.

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.

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."

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.

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.

Putatunda, Sayan. 2021. *Practical Machine Learning for Streaming Data with Python*. Springer.

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.