

# **Hyperparameter Tuning Cookbook**

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

Thomas Bartz-Beielstein

Oct 30, 2023

# Table of contents

<b>Preface: Optimization and Hyperparameter Tuning</b>	<b>3</b>
Book Structure . . . . .	4
Software Used in this Book . . . . .	6
Citation . . . . .	6
<b>I Spot as an Optimizer</b>	<b>8</b>
<b>1 Introduction to spotPython</b>	<b>9</b>
1.1 Example: Spot and the Sphere Function . . . . .	9
1.1.1 The Objective Function: Sphere . . . . .	10
1.2 Spot Parameters: <code>fun_evals</code> , <code>init_size</code> and <code>show_models</code> . . . . .	12
1.3 Print the Results . . . . .	14
1.4 Show the Progress . . . . .	14
1.5 Visualizing the Optimization and Hyperparameter Tuning Process with TensorBoard . . . . .	14
<b>2 Multi-dimensional Functions</b>	<b>18</b>
2.1 Example: Spot and the 3-dim Sphere Function . . . . .	18
2.1.1 The Objective Function: 3-dim Sphere . . . . .	18
2.1.2 Results . . . . .	20
2.1.3 A Contour Plot . . . . .	20
2.1.4 TensorBoard . . . . .	22
2.2 Conclusion . . . . .	23
2.3 Exercises . . . . .	24
2.3.1 The Three Dimensional <code>fun_cubed</code> . . . . .	24
2.3.2 The Ten Dimensional <code>fun_wing_wt</code> . . . . .	24
2.3.3 The Three Dimensional <code>fun_runge</code> . . . . .	24
2.3.4 The Three Dimensional <code>fun_linear</code> . . . . .	25
<b>3 Isotropic and Anisotropic Kriging</b>	<b>26</b>
3.1 Example: Isotropic Spot Surrogate and the 2-dim Sphere Function . . . . .	26
3.1.1 The Objective Function: 2-dim Sphere . . . . .	26
3.1.2 Results . . . . .	27
3.2 Example With Anisotropic Kriging . . . . .	28
3.2.1 Taking a Look at the <code>theta</code> Values . . . . .	30

3.3	Exercises . . . . .	31
3.3.1	<code>fun_branin</code> . . . . .	31
3.3.2	<code>fun_sin_cos</code> . . . . .	32
3.3.3	<code>fun_runge</code> . . . . .	32
3.3.4	<code>fun_wingwt</code> . . . . .	32
<b>4</b>	<b>Using sklearn Surrogates in spotPython</b>	<b>33</b>
4.1	Example: Branin Function with <code>spotPython</code> 's Internal Kriging Surrogate . . . . .	33
4.1.1	The Objective Function Branin . . . . .	33
4.1.2	Running the surrogate model based optimizer <code>Spot</code> : . . . . .	34
4.1.3	<code>TensorBoard</code> . . . . .	35
4.1.4	Print the Results . . . . .	35
4.1.5	Show the Progress and the Surrogate . . . . .	37
4.2	Example: Using Surrogates From <code>scikit-learn</code> . . . . .	38
4.2.1	<code>GaussianProcessRegressor</code> as a Surrogate . . . . .	38
4.3	Example: One-dimensional Sphere Function With <code>spotPython</code> 's Kriging . . . . .	40
4.3.1	Results . . . . .	46
4.4	Example: Sklearn Model <code>GaussianProcess</code> . . . . .	47
4.5	Exercises . . . . .	54
4.5.1	<code>DecisionTreeRegressor</code> . . . . .	54
4.5.2	<code>RandomForestRegressor</code> . . . . .	55
4.5.3	<code>linear_model.LinearRegression</code> . . . . .	55
4.5.4	<code>linear_model.Ridge</code> . . . . .	55
4.6	Exercise 2 . . . . .	55
<b>5</b>	<b>Sequential Parameter Optimization: Using <code>scipy</code> Optimizers</b>	<b>56</b>
5.1	The Objective Function Branin . . . . .	56
5.2	The Optimizer . . . . .	57
5.2.1	<code>TensorBoard</code> . . . . .	59
5.3	Print the Results . . . . .	59
5.4	Show the Progress . . . . .	59
5.5	Exercises . . . . .	62
5.5.1	<code>dual_annealing</code> . . . . .	62
5.5.2	<code>direct</code> . . . . .	62
5.5.3	<code>shgo</code> . . . . .	62
5.5.4	<code>basinhopping</code> . . . . .	62
5.5.5	Performance Comparison . . . . .	62
<b>6</b>	<b>Sequential Parameter Optimization: Gaussian Process Models</b>	<b>64</b>
6.1	Gaussian Processes Regression: Basic Introductory <code>scikit-learn</code> Example . . . . .	64
6.1.1	Train and Test Data . . . . .	65
6.1.2	Building the Surrogate With <code>Sklearn</code> . . . . .	65
6.1.3	Plotting the <code>SklearnModel</code> . . . . .	65

6.1.4	The <code>spotPython</code> Version . . . . .	66
6.1.5	Visualizing the Differences Between the <code>spotPython</code> and the <code>sklearn</code> Model Fits . . . . .	67
6.2	Exercises . . . . .	68
6.2.1	<code>Schonlau</code> Example Function . . . . .	68
6.2.2	<code>Forrester</code> Example Function . . . . .	68
6.2.3	<code>fun_runge</code> Function (1-dim) . . . . .	69
6.2.4	<code>fun_cubed</code> (1-dim) . . . . .	70
6.2.5	The Effect of Noise . . . . .	70
<b>7</b>	<b>Expected Improvement</b>	<b>72</b>
7.1	Example: Spot and the 1-dim Sphere Function . . . . .	72
7.1.1	The Objective Function: 1-dim Sphere . . . . .	72
7.1.2	Results . . . . .	74
7.2	Same, but with EI as infill_criterion . . . . .	75
7.3	Non-isotropic Kriging . . . . .	77
7.4	Using <code>sklearn</code> Surrogates . . . . .	80
7.4.1	The spot Loop . . . . .	80
7.4.2	<code>spot</code> : The Initial Model . . . . .	81
7.4.3	<code>Init</code> : Build Initial Design . . . . .	83
7.4.4	Evaluate . . . . .	85
7.4.5	Build Surrogate . . . . .	85
7.4.6	A Simple Predictor . . . . .	85
7.5	Gaussian Processes regression: basic introductory example . . . . .	85
7.6	The Surrogate: Using scikit-learn models . . . . .	88
7.7	Additional Examples . . . . .	91
7.7.1	Optimize on Surrogate . . . . .	95
7.7.2	Evaluate on Real Objective . . . . .	95
7.7.3	Impute / Infill new Points . . . . .	95
7.8	Tests . . . . .	95
7.9	EI: The Famous Schonlau Example . . . . .	96
7.10	EI: The Forrester Example . . . . .	98
7.11	Noise . . . . .	101
7.12	Cubic Function . . . . .	104
7.13	Factors . . . . .	110
<b>8</b>	<b>Hyperparameter Tuning and Noise</b>	<b>112</b>
8.1	Example: Spot and the Noisy Sphere Function . . . . .	112
8.1.1	The Objective Function: Noisy Sphere . . . . .	112
8.2	Print the Results . . . . .	119
8.3	Noise and Surrogates: The Nugget Effect . . . . .	120
8.3.1	The Noisy Sphere . . . . .	120

8.4 Exercises . . . . .	123
8.4.1 Noisy <code>fun_cubed</code> . . . . .	123
8.4.2 <code>fun_runge</code> . . . . .	124
8.4.3 <code>fun_forrester</code> . . . . .	124
8.4.4 <code>fun_xsin</code> . . . . .	124
<b>9 Handling Noise: Optimal Computational Budget Allocation in Spot</b>	<b>125</b>
9.1 Example: Spot, OCBA, and the Noisy Sphere Function . . . . .	125
9.1.1 The Objective Function: Noisy Sphere . . . . .	125
9.2 Print the Results . . . . .	131
9.3 Noise and Surrogates: The Nugget Effect . . . . .	131
9.3.1 The Noisy Sphere . . . . .	131
9.4 Exercises . . . . .	134
9.4.1 Noisy <code>fun_cubed</code> . . . . .	134
9.4.2 <code>fun_runge</code> . . . . .	135
9.4.3 <code>fun_forrester</code> . . . . .	135
9.4.4 <code>fun_xsin</code> . . . . .	135
<b>II Hyperparameter Tuning</b>	<b>136</b>
<b>10 HPT: sklearn SVC on Moons Data</b>	<b>137</b>
10.1 Step 1: Setup . . . . .	137
10.2 Step 2: Initialization of the Empty <code>fun_control</code> Dictionary . . . . .	137
10.3 Step 3: SKlearn Load Data (Classification) . . . . .	138
10.4 Step 4: Specification of the Preprocessing Model . . . . .	140
10.5 Step 5: Select Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	140
10.6 Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	143
10.6.1 Modify hyperparameter of type numeric and integer (boolean) . . . . .	143
10.6.2 Modify hyperparameter of type factor . . . . .	144
10.6.3 Optimizers . . . . .	144
10.7 Step 7: Selection of the Objective (Loss) Function . . . . .	144
10.7.1 Predict Classes or Class Probabilities . . . . .	145
10.8 Step 8: Calling the SPOT Function . . . . .	145
10.8.1 Preparing the SPOT Call . . . . .	145
10.8.2 The Objective Function . . . . .	146
10.8.3 Run the Spot Optimizer . . . . .	146
10.8.4 Starting the Hyperparameter Tuning . . . . .	147
10.9 Step 9: Results . . . . .	148
10.9.1 Show variable importance . . . . .	150
10.9.2 Get Default Hyperparameters . . . . .	150
10.9.3 Get SPOT Results . . . . .	151

10.9.4	Plot: Compare Predictions . . . . .	152
10.9.5	Detailed Hyperparameter Plots . . . . .	154
10.9.6	Parallel Coordinates Plot . . . . .	155
10.9.7	Plot all Combinations of Hyperparameters . . . . .	155
<b>11</b>	<b>river Hyperparameter Tuning: Hoeffding Adaptive Tree Regressor with Friedman Drift Data</b>	<b>156</b>
11.1	Setup . . . . .	156
11.2	Initialization of the <code>fun_control</code> Dictionary . . . . .	157
11.3	Load Data: The Friedman Drift Data . . . . .	158
11.4	Specification of the Preprocessing Model . . . . .	159
11.5	SelectSelect Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	159
11.6	Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	160
11.7	Selection of the Objective (Loss) Function . . . . .	161
11.8	Calling the SPOT Function . . . . .	162
11.8.1	Prepare the SPOT Parameters . . . . .	162
11.8.2	The Objective Function . . . . .	163
11.8.3	Run the Spot Optimizer . . . . .	163
11.8.4	TensorBoard . . . . .	165
11.8.5	Results . . . . .	165
11.9	The Larger Data Set . . . . .	168
11.10	Get Default Hyperparameters . . . . .	169
11.10.1	Show Predictions . . . . .	170
11.11	Get SPOT Results . . . . .	171
11.12	Visualize Regression Trees . . . . .	174
11.12.1	Spot Model . . . . .	175
11.13	Detailed Hyperparameter Plots . . . . .	176
11.14	Parallel Coordinates Plots . . . . .	176
11.15	Plot all Combinations of Hyperparameters . . . . .	177
<b>12</b>	<b>river Hyperparameter Tuning: Mondrian Tree Regressor with Friedman Drift Data</b>	<b>178</b>
12.1	Setup . . . . .	178
12.2	Initialization of the <code>fun_control</code> Dictionary . . . . .	179
12.3	Load Data: The Friedman Drift Data . . . . .	180
12.4	Specification of the Preprocessing Model . . . . .	181
12.5	SelectSelect Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	181
12.6	Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	182
12.7	Selection of the Objective (Loss) Function . . . . .	183
12.8	Calling the SPOT Function . . . . .	184
12.8.1	Prepare the SPOT Parameters . . . . .	184
12.8.2	The Objective Function . . . . .	184
12.8.3	Run the Spot Optimizer . . . . .	185
12.8.4	TensorBoard . . . . .	186

12.8.5 Results . . . . .	186
12.9 The Larger Data Set . . . . .	189
12.10Get Default Hyperparameters . . . . .	190
12.10.1 Show Predictions . . . . .	191
12.11Get SPOT Results . . . . .	192
12.12Detailed Hyperparameter Plots . . . . .	195
12.13Parallel Coordinates Plots . . . . .	196
12.14Plot all Combinations of Hyperparameters . . . . .	196
<b>13 river Hyperparameter Tuning: Mondrian Tree Classifier with Bananas Data</b>	<b>197</b>
13.1 Setup . . . . .	197
13.2 Initialization of the <code>fun_control</code> Dictionary . . . . .	198
13.3 Load Data: The Bananas Dataset . . . . .	198
13.4 Specification of the Preprocessing Model . . . . .	200
13.5 SelectSelect Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	200
13.6 Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code>	201
13.7 Selection of the Objective (Loss) Function . . . . .	202
13.8 Calling the SPOT Function . . . . .	203
13.8.1 Prepare the SPOT Parameters . . . . .	203
13.8.2 The Objective Function . . . . .	203
13.8.3 Run the Spot Optimizer . . . . .	204
13.8.4 TensorBoard . . . . .	205
13.8.5 Results . . . . .	206
13.9 Get Default Hyperparameters . . . . .	208
13.9.1 Show Predictions . . . . .	209
13.10Get SPOT Results . . . . .	210
13.11Compare Predictions . . . . .	211
13.12Detailed Hyperparameter Plots . . . . .	215
13.13Parallel Coordinates Plots . . . . .	217
13.14Plot all Combinations of Hyperparameters . . . . .	217
<b>14 HPT: PyTorch With spotPython and Ray Tune on CIFAR10</b>	<b>218</b>
14.1 Step 1: Setup . . . . .	219
14.2 Step 2: Initialization of the <code>fun_control</code> Dictionary . . . . .	220
14.3 Step 3: PyTorch Data Loading . . . . .	221
14.4 Step 4: Specification of the Preprocessing Model . . . . .	221
14.5 Step 5: Select Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	222
14.5.1 The <code>Net_Core</code> class . . . . .	224
14.5.2 Comparison of the Approach Described in the PyTorch Tutorial With <code>spotPython</code> . . . . .	224
14.5.3 The Search Space: Hyperparameters . . . . .	225
14.5.4 Configuring the Search Space With Ray Tune . . . . .	225
14.5.5 Configuring the Search Space With <code>spotPython</code> . . . . .	226

14.6 Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	228
14.6.1 Optimizers . . . . .	229
14.7 Step 7: Selection of the Objective (Loss) Function . . . . .	231
14.7.1 Evaluation: Data Splitting . . . . .	231
14.7.2 Hold-out Data Split . . . . .	231
14.7.3 Cross-Validation . . . . .	232
14.7.4 Overview of the Evaluation Settings . . . . .	233
14.7.5 Evaluation: Loss Functions and Metrics . . . . .	234
14.8 Step 8: Calling the SPOT Function . . . . .	235
14.8.1 Preparing the SPOT Call . . . . .	235
14.8.2 The Objective Function <code>fun_torch</code> . . . . .	236
14.8.3 Using Default Hyperparameters or Results from Previous Runs . . . . .	236
14.8.4 Starting the Hyperparameter Tuning . . . . .	236
14.9 Step 9: Tensorboard . . . . .	243
14.9.1 Tensorboard: Start Tensorboard . . . . .	243
14.9.2 Saving the State of the Notebook . . . . .	243
14.10 Step 10: Results . . . . .	245
14.10.1 Get the Tuned Architecture (SPOT Results) . . . . .	247
14.10.2 Get Default Hyperparameters . . . . .	248
14.10.3 Evaluation of the Default Architecture . . . . .	248
14.10.4 Evaluation of the Tuned Architecture . . . . .	250
14.10.5 Detailed Hyperparameter Plots . . . . .	252
14.11 Summary and Outlook . . . . .	254
14.12 Appendix . . . . .	254
14.12.1 Sample Output From Ray Tune's Run . . . . .	254
<b>15 HPT: <code>sklearn RandomForestClassifier</code> VBDP Data</b>	<b>256</b>
15.1 Step 1: Setup . . . . .	256
15.2 Step 2: Initialization of the Empty <code>fun_control</code> Dictionary . . . . .	257
15.3 Step 3: PyTorch Data Loading . . . . .	257
15.3.1 Load Data: Classification VBDP . . . . .	257
15.3.2 Holdout Train and Test Data . . . . .	258
15.4 Step 4: Specification of the Preprocessing Model . . . . .	259
15.5 Step 5: Select Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	260
15.6 Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	261
15.6.1 Modify hyperparameter of type numeric and integer (boolean) . . . . .	261
15.6.2 Modify hyperparameter of type factor . . . . .	262
15.6.3 Optimizers . . . . .	262
15.6.4 Selection of the Objective: Metric and Loss Functions . . . . .	262
15.7 Step 7: Selection of the Objective (Loss) Function . . . . .	263
15.7.1 Metric Function . . . . .	263

15.7.2	Evaluation on Hold-out Data . . . . .	264
15.7.3	OOB Score . . . . .	264
15.8	Step 8: Calling the SPOT Function . . . . .	265
15.8.1	Preparing the SPOT Call . . . . .	265
15.8.2	The Objective Function . . . . .	266
15.8.3	Run the Spot Optimizer . . . . .	266
15.9	Step 9: Tensorboard . . . . .	269
15.10	Step 10: Results . . . . .	269
15.10.1	Show variable importance . . . . .	270
15.10.2	Get Default Hyperparameters . . . . .	270
15.10.3	Get SPOT Results . . . . .	271
15.10.4	Evaluate SPOT Results . . . . .	272
15.10.5	Handling Non-deterministic Results . . . . .	273
15.10.6	Evalution of the Default Hyperparameters . . . . .	273
15.10.7	Plot: Compare Predictions . . . . .	274
15.10.8	Cross-validated Evaluations . . . . .	276
15.10.9	Detailed Hyperparameter Plots . . . . .	277
15.10.10	Parallel Coordinates Plot . . . . .	283
15.10.11	Plot all Combinations of Hyperparameters . . . . .	283
<b>16</b>	<b>HPT: sklearn XGB Classifier VBDP Data</b>	<b>284</b>
16.1	Step 1: Setup . . . . .	284
16.2	Step 2: Initialization of the Empty <code>fun_control</code> Dictionary . . . . .	285
16.3	Step 3: PyTorch Data Loading . . . . .	285
16.3.1	1. Load Data: Classification VBDP . . . . .	285
16.3.2	Holdout Train and Test Data . . . . .	286
16.4	Step 4: Specification of the Preprocessing Model . . . . .	287
16.5	Step 5: Select Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	287
16.6	Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	289
16.6.1	Modify hyperparameter of type numeric and integer (boolean) . . . . .	289
16.6.2	Modify hyperparameter of type factor . . . . .	290
16.6.3	Optimizers . . . . .	290
16.7	Step 7: Selection of the Objective (Loss) Function . . . . .	290
16.7.1	Evaluation . . . . .	290
16.7.2	Selection of the Objective: Metric and Loss Functions . . . . .	290
16.7.3	Loss Function . . . . .	291
16.7.4	Metric Function . . . . .	291
16.7.5	Evaluation on Hold-out Data . . . . .	292
16.8	Step 8: Calling the SPOT Function . . . . .	293
16.8.1	Preparing the SPOT Call . . . . .	293
16.8.2	The Objective Function . . . . .	293
16.8.3	Run the Spot Optimizer . . . . .	294

16.9 Step 9: Tensorboard . . . . .	295
16.10 Step 10: Results . . . . .	296
16.10.1 Show variable importance . . . . .	297
16.10.2 Get Default Hyperparameters . . . . .	297
16.10.3 Get SPOT Results . . . . .	298
16.10.4 Evaluate SPOT Results . . . . .	299
16.10.5 Handling Non-deterministic Results . . . . .	300
16.10.6 Evaluation of the Default Hyperparameters . . . . .	300
16.10.7 Plot: Compare Predictions . . . . .	301
16.10.8 Cross-validated Evaluations . . . . .	303
16.10.9 Detailed Hyperparameter Plots . . . . .	304
16.10.10 Parallel Coordinates Plot . . . . .	308
16.10.11 Plot all Combinations of Hyperparameters . . . . .	308
<b>17 HPT: sklearn SVC VBDP Data</b>	<b>309</b>
17.1 Step 1: Setup . . . . .	309
17.2 Step 2: Initialization of the Empty <code>fun_control</code> Dictionary . . . . .	310
17.3 Step 3: PyTorch Data Loading . . . . .	310
17.3.1 1. Load Data: Classification VBDP . . . . .	310
17.3.2 Holdout Train and Test Data . . . . .	311
17.4 Step 4: Specification of the Preprocessing Model . . . . .	312
17.5 Step 5: Select Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	312
17.6 Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	314
17.6.1 Modify hyperparameter of type numeric and integer (boolean) . . . . .	314
17.6.2 Modify hyperparameter of type factor . . . . .	314
17.6.3 Optimizers . . . . .	315
17.6.4 Selection of the Objective: Metric and Loss Functions . . . . .	315
17.7 Step 7: Selection of the Objective (Loss) Function . . . . .	315
17.7.1 Metric Function . . . . .	315
17.7.2 Evaluation on Hold-out Data . . . . .	316
17.8 Step 8: Calling the SPOT Function . . . . .	317
17.8.1 Preparing the SPOT Call . . . . .	317
17.8.2 The Objective Function . . . . .	318
17.8.3 Run the Spot Optimizer . . . . .	318
17.9 Step 9: Tensorboard . . . . .	321
17.10 Step 10: Results . . . . .	321
17.10.1 Show variable importance . . . . .	322
17.10.2 Get Default Hyperparameters . . . . .	323
17.10.3 Get SPOT Results . . . . .	324
17.10.4 Evaluate SPOT Results . . . . .	325
17.10.5 Handling Non-deterministic Results . . . . .	326
17.10.6 Evaluation of the Default Hyperparameters . . . . .	326

17.10.7 Plot: Compare Predictions . . . . .	327
17.10.8 Cross-validated Evaluations . . . . .	328
17.10.9 Detailed Hyperparameter Plots . . . . .	329
17.10.10 Parallel Coordinates Plot . . . . .	331
17.10.11 Plot all Combinations of Hyperparameters . . . . .	331
<b>18 HPT: sklearn KNN Classifier VBDP Data</b>	<b>332</b>
18.1 Step 1: Setup . . . . .	332
18.2 Step 2: Initialization of the Empty <code>fun_control</code> Dictionary . . . . .	333
18.2.1 Load Data: Classification VBDP . . . . .	333
18.2.2 Holdout Train and Test Data . . . . .	334
18.3 Step 4: Specification of the Preprocessing Model . . . . .	335
18.4 Step 5: Select Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	335
18.5 Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	337
18.5.1 Modify hyperparameter of type numeric and integer (boolean) . . . . .	337
18.5.2 Modify hyperparameter of type factor . . . . .	337
18.5.3 Optimizers . . . . .	338
18.5.4 Selection of the Objective: Metric and Loss Functions . . . . .	338
18.6 Step 7: Selection of the Objective (Loss) Function . . . . .	338
18.6.1 Metric Function . . . . .	338
18.6.2 Evaluation on Hold-out Data . . . . .	339
18.7 Step 8: Calling the SPOT Function . . . . .	340
18.7.1 Preparing the SPOT Call . . . . .	340
18.7.2 The Objective Function . . . . .	341
18.7.3 Run the Spot Optimizer . . . . .	341
18.8 Step 9: Tensorboard . . . . .	344
18.9 Step 10: Results . . . . .	344
18.9.1 Show variable importance . . . . .	345
18.9.2 Get Default Hyperparameters . . . . .	346
18.9.3 Get SPOT Results . . . . .	347
18.9.4 Evaluate SPOT Results . . . . .	347
18.9.5 Handling Non-deterministic Results . . . . .	348
18.9.6 Evalution of the Default Hyperparameters . . . . .	349
18.9.7 Plot: Compare Predictions . . . . .	349
18.9.8 Cross-validated Evaluations . . . . .	351
18.9.9 Detailed Hyperparameter Plots . . . . .	352
18.9.10 Parallel Coordinates Plot . . . . .	353
18.9.11 Plot all Combinations of Hyperparameters . . . . .	353
<b>19 HPT PyTorch Lightning: VBDP</b>	<b>354</b>
19.1 Step 1: Setup . . . . .	355
19.2 Step 2: Initialization of the <code>fun_control</code> Dictionary . . . . .	355

19.3 Step 3: PyTorch Data Loading . . . . .	356
19.3.1 Lightning Dataset and DataModule . . . . .	356
19.4 Step 4: Preprocessing . . . . .	356
19.5 Step 5: Select the NN Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	357
19.6 Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	357
19.7 Step 7: Data Splitting, the Objective (Loss) Function and the Metric . . . . .	359
19.7.1 Evaluation . . . . .	359
19.7.2 Loss Functions and Metrics . . . . .	359
19.7.3 Metric . . . . .	359
19.8 Step 8: Calling the SPOT Function . . . . .	359
19.8.1 Preparing the SPOT Call . . . . .	359
19.8.2 The Objective Function <code>fun</code> . . . . .	360
19.8.3 Starting the Hyperparameter Tuning . . . . .	360
19.9 Step 9: Tensorboard . . . . .	363
19.10 Step 10: Results . . . . .	364
19.10.1 Get the Tuned Architecture . . . . .	365
19.10.2 Cross Validation With Lightning . . . . .	366
19.10.3 Detailed Hyperparameter Plots . . . . .	370
19.10.4 Parallel Coordinates Plot . . . . .	371
19.10.5 Plot all Combinations of Hyperparameters . . . . .	371
19.10.6 Visualizing the Activation Distribution . . . . .	372
19.11 Submission . . . . .	373
19.12 Appendix . . . . .	375
19.12.1 Differences to the spotPython Approaches for <code>torch</code> , <code>sklearn</code> and <code>river</code> . . . . .	375
19.12.2 Taking a Look at the Data . . . . .	376
19.12.3 The MAPK Metric . . . . .	377
<b>III Optimization</b>	<b>378</b>
<b>20 Introduction to <code>scipy.optimize</code></b>	<b>379</b>
<b>21 Lecture 01: Background</b>	<b>380</b>
21.1 Contents of this Course . . . . .	380
21.2 Surrogates . . . . .	380
21.3 Surrogate Example: Data . . . . .	381
21.4 Surrogate Example: Cubic Regression Model . . . . .	381
21.5 Extrapolation . . . . .	381
21.6 New Modeling Approach . . . . .	381
21.7 Comparison . . . . .	381
21.8 Benefits . . . . .	381
21.9 Costs of Simulation . . . . .	381

21.10 Mathematical Models and Meta-Models . . . . .	381
21.11 Surrogates = Trained Meta-models . . . . .	382
21.12 Computer Experiments . . . . .	382
21.13 Experimentation is Changing . . . . .	382
21.14 Examples . . . . .	382
21.15 Limits of Mathematical Modeling . . . . .	383
21.16 Example: Why Computer Simulations are Necessary . . . . .	383
21.17 Simulation Requirements . . . . .	383
21.18 Approaches: RSM and DACE (GP) . . . . .	383
21.19 Response Surface Methods: Pros and Cons . . . . .	384
21.20 Response Surface Methods: Related Fields . . . . .	384
21.21 RSM Applications . . . . .	384
21.22 What is RSM? . . . . .	385
21.23 RSM Example . . . . .	385
<b>22 Lecture 02</b>	<b>386</b>
22.1 RSM: Example . . . . .	386
22.2 3d Plot . . . . .	386
22.3 Contour plot . . . . .	387
22.4 Problems in Practice . . . . .	388
22.5 RSM: Strategies . . . . .	388
22.6 RSM: Fitting an Empirical Model . . . . .	389
22.7 RSM: Equations of the Empirical Model . . . . .	389
22.8 RSM: Noise in the Empirical Model . . . . .	389
22.9 RSM: Natural and Coded Variables . . . . .	389
<b>23 Lecture 02: RSM Low-order Polynomials</b>	<b>391</b>
23.1 Simplifying Assumptions . . . . .	391
23.2 First-Order Models (Main Effects Model) . . . . .	391
23.3 First-Order Model in python Evaluated on a Grid . . . . .	391
23.4 First-Order Model Properties . . . . .	392
23.5 First-order Model with Interactions in python . . . . .	393
23.6 First-order Model with Interactions . . . . .	394
23.7 Observations: First-Order Model with Interactions . . . . .	394
23.8 Second-Order Models . . . . .	394
23.9 Second-Order Models: Properties . . . . .	396
23.10 Example: Stationary Ridge . . . . .	396
23.11 Observations: Second-Order Model (Ridge) . . . . .	397
23.12 Example: Rising Ridge . . . . .	397
23.13 Summary: Rising Ridge . . . . .	398
23.14 Falling Ridge . . . . .	399
23.15 Saddle Point . . . . .	399
23.16 Interpretation: Saddle Points . . . . .	400

23.17	Summary: Ridge Analysis . . . . .	400
23.18	General RSM Models . . . . .	401
23.19	Ordinary Least Squares . . . . .	401
23.20	Designs . . . . .	401
23.21	Different Designs . . . . .	401
23.22	RSM Experimentation: First Step . . . . .	402
23.23	RSM Experimentation: Second Step . . . . .	402
23.24	RSM Experimentation: Third Step . . . . .	402
<b>24</b>	<b>Example 1 (DOE)</b>	<b>403</b>
<b>25</b>	<b>Example 2 (DOE)</b>	<b>404</b>
<b>26</b>	<b>RSM: Review and General Considerations</b>	<b>405</b>
26.1	First Glimpse . . . . .	405
26.2	RSM Downside: Inefficiency . . . . .	405
26.3	RSM Downside: Locality . . . . .	405
26.4	RSM Downside: Expert Knowledge . . . . .	405
26.5	RSM Downside: Replicability . . . . .	406
26.6	Historical Considerations about RSM . . . . .	406
26.7	Status Quo . . . . .	406
26.8	The Role of Statistics . . . . .	406
26.9	New RSM is needed: DACE . . . . .	407
<b>27</b>	<b>Exercises</b>	<b>408</b>
<b>28</b>	<b>Lecture 04: DACE and Aircraft Wing Example</b>	<b>409</b>
28.1	Design and Analysis of Computer Experiments . . . . .	409
28.2	Research Questions for DACE . . . . .	409
28.3	Remaining Differences Between RSM and DACE: Noise . . . . .	410
28.4	DACE Literature . . . . .	410
<b>29</b>	<b>Aircraft Wing Weight Example</b>	<b>411</b>
29.1	AWWE Equation . . . . .	411
29.2	AWWE Parameters (Part 1) . . . . .	411
29.3	Discussing the AWWE Parameters and Equations . . . . .	412
29.4	Mathematical Properties of the AWWE Equation . . . . .	412
29.5	Goals: Understanding and Optimization . . . . .	412
29.6	AWWE: Python Code . . . . .	413
29.7	AWWE: Python Code . . . . .	413
29.8	Properties of the Python “Solver” . . . . .	414
29.9	AWWE Visualization . . . . .	414
29.10	Plot 1: Load Factor ( $N_z$ ) and Aspect Ratio ( $A$ ) . . . . .	415
29.11	Variations of the Contour Plots . . . . .	416

29.12	Plot 1: Interpretation of the AWWE Plot . . . . .	416
29.13	Plot 2: Taper Ratio and Fuel Weight . . . . .	417
29.14	Plot 2: Interpretation of Taper Ratio ( $l$ ) and Fuel Weight ( $W_{fw}$ ) . . . . .	418
<b>30</b>	<b>The Big Picture</b>	<b>419</b>
30.1	Combining all Variables . . . . .	419
30.2	Plotting the Big Picture . . . . .	421
30.3	AWWE Landscape . . . . .	423
30.3.1	Our Observations . . . . .	423
30.3.2	Expert Knowledge . . . . .	424
30.4	Summary of the First Experiments . . . . .	424
<b>31</b>	<b>Exercise 1</b>	<b>425</b>
31.0.1	Adding Paint Weight . . . . .	425
<b>32</b>	<b>Lecture 05: Kriging</b>	<b>426</b>
32.1	Introduction . . . . .	426
<b>33</b>	<b>Gaussian Process Basics</b>	<b>427</b>
33.1	Gaussian Process Prior . . . . .	427
33.2	Covariance . . . . .	427
33.3	Positive Definiteness . . . . .	427
33.4	Generating GP Random Functions . . . . .	428
<b>34</b>	<b>Example in 1 Dimension</b>	<b>429</b>
34.1	Input Grid . . . . .	429
34.2	Covariance Matrix . . . . .	429
34.3	Multivariate Normal Distribution . . . . .	429
34.4	Plot the Results . . . . .	430
<b>35</b>	<b>Properties of the 1d Example</b>	<b>431</b>
35.1	Several Bumps . . . . .	431
35.2	Smooth Look . . . . .	431
35.3	Scale of Two . . . . .	431
35.4	Background: Expectation, Mean . . . . .	431
35.5	Example . . . . .	432
35.6	Sample Mean . . . . .	432
35.7	Variance and Standard Deviation . . . . .	432
35.8	Varianc . . . . .	432
35.9	Standard Deviation . . . . .	433
35.10	Calculation of the Standard Deviation with Python . . . . .	433
35.10.1	Single Versus Double Precision . . . . .	434
35.11	Visualization of the Standard Deviation . . . . .	435
35.12	Standardization . . . . .	437

35.13 Random Numbers in Python . . . . .	437
35.14 The Multivariate Normal Distribution . . . . .	438
35.15 Kriging . . . . .	439
35.15.1 The Kriging Covariance Matrix . . . . .	439
35.15.2 Building the Kriging Model . . . . .	440
35.15.3 Example: Correlation Matrix . . . . .	440
35.15.4 Generate sample from $N(\mu, \Psi)$ . . . . .	443
<b>36 Kriging in a Nutshell</b>	<b>444</b>
36.0.1 The Kriging Model . . . . .	444
36.0.2 Correlations . . . . .	444
36.0.3 MLE to estimate $\theta$ and $p$ . . . . .	445
36.1 Tuning $\theta$ and $p$ . . . . .	446
36.2 Kriging Prediction . . . . .	446
36.2.1 Properties of the Predictor . . . . .	447
36.3 Example: Sinusoid Function . . . . .	447
36.4 Calculating the Correlation Matrix $\Psi$ . . . . .	447
36.5 Computing the $\psi$ Vector . . . . .	448
36.5.1 Based on Distances Between Testing and Training Data Locations . . . . .	448
36.6 Predictive Equations . . . . .	449
<b>37 Exercises</b>	<b>451</b>
37.1 1 Number of Sample Points . . . . .	451
37.2 2 Modified $\theta$ values . . . . .	451
37.3 3 Prediction Interval . . . . .	451
<b>38 DOE 1</b>	<b>452</b>
38.1 Two factors experiment . . . . .	453
38.2 Negative Interaction . . . . .	455
<b>39 DOE 2</b>	<b>459</b>
39.1 first full factorial . . . . .	463
39.2 Second Step . . . . .	464
39.3 Third Step . . . . .	464
<b>40 Final Implementation</b>	<b>465</b>
<b>41 Exercise RBF</b>	<b>467</b>
41.1 Package Loading . . . . .	467
41.2 1.2 Define a small number . . . . .	467
41.3 1.3 The Sampling Plan ( $X$ ) . . . . .	467
41.4 1.4 The Objective Function . . . . .	468
<b>42 2 The Gram Matrix</b>	<b>469</b>

<b>43 3 The Radial Basis Functions</b>	<b>470</b>
43.1 4 The $\Psi$ Matrix . . . . .	470
<b>44 5 Inverting <math>\Psi</math> via Cholesky Factorization</b>	<b>471</b>
<b>45 6 Predictions</b>	<b>472</b>
45.1 6.1 The Predictor . . . . .	472
45.2 6.2 Testing some Example Points . . . . .	472
45.3 6.1 The RBF Prediction $\hat{f}$ . . . . .	472
45.4 6.2 The Original (True) Value $f$ . . . . .	472
45.5 6.3 Visualizations . . . . .	473
<b>46 7 Note</b>	<b>474</b>
<b>47 8 Cholesky Factorization</b>	<b>475</b>
47.1 8.1 $A = U^T U$ . . . . .	475
47.2 8.2 $A = LL^T$ . . . . .	475
47.3 8.3 Example . . . . .	476
47.4 8.4 Check: Is $A$ positive definite? . . . . .	476
47.5 8.5.1 $A = U^T U$ . . . . .	476
47.6 8.5.2 $A = LL^T$ . . . . .	477
<b>48 9 Exercises</b>	<b>478</b>
48.1 9.1 Gaussian Basis Function . . . . .	478
48.1.1 9.1.1 . . . . .	478
48.1.2 9.1.2 . . . . .	478
48.2 9.2 Linear Basis Function . . . . .	478
<b>Appendices</b>	<b>479</b>
<b>A Introduction to Jupyter Notebook</b>	<b>479</b>
<b>B Different Notebook cells</b>	<b>480</b>
B.1 Code cells . . . . .	480
B.2 Markdown cells . . . . .	480
B.3 Raw cells . . . . .	480
B.3.1 Usage . . . . .	481
<b>C Install packages</b>	<b>482</b>
<b>D Load packages</b>	<b>483</b>
<b>E Functions in Python</b>	<b>484</b>

<b>F List of Useful Jupyter Notebook Shortcuts</b>	<b>485</b>
<b>G Documentation of the Sequential Parameter Optimization</b>	<b>487</b>
G.1 Example: spot . . . . .	487
G.1.1 The Objective Function . . . . .	487
G.1.2 External Parameters . . . . .	489
G.2 The <code>fun_control</code> Dictionary . . . . .	492
G.3 The <code>design_control</code> Dictionary . . . . .	492
G.4 The <code>surrogate_control</code> Dictionary . . . . .	493
G.5 The <code>optimizer_control</code> Dictionary . . . . .	493
G.6 Run . . . . .	494
G.7 Print the Results . . . . .	496
G.8 Show the Progress . . . . .	496
G.9 Visualize the Surrogate . . . . .	496
G.10 Run With a Specific Start Design . . . . .	497
G.11 Init: Build Initial Design . . . . .	498
G.12 Replicability . . . . .	499
G.13 Surrogates . . . . .	500
G.13.1 A Simple Predictor . . . . .	500
G.14 Demo/Test: Objective Function Fails . . . . .	500
G.15 PyTorch: Detailed Description of the Data Splitting . . . . .	503
G.15.1 Description of the " <code>train_hold_out</code> " Setting . . . . .	503
<b>References</b>	<b>514</b>

# Preface: Optimization and Hyperparameter Tuning

This document provides a comprehensive guide to hyperparameter tuning using spotPython for scikit-learn, PyTorch, and river. The first part introduces spotPython’s surrogate model-based optimization process, while the second 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, and river. This publication is under development, with updates available on the corresponding webpage.

The goal of hyperparameter tuning is to optimize the hyperparameters in a way that improves the performance of the machine learning or deep learning model. Hyperparameters are parameters that are not learned during the training process, but are set before the training process begins. Hyperparameter tuning is an important, but often difficult and computationally intensive task. Changing the architecture of a neural network or the learning rate of an optimizer can have a significant impact on the performance.

Hyperparameter tuning is referred to as “hyperparameter optimization” (HPO) in the literature. However, since we do not consider the optimization, but also the understanding of the hyperparameters, we use the term “hyperparameter tuning” in this book. See also the discussion in Chapter 2 of Bartz et al. (2022), which lays the groundwork and presents an introduction to the process of tuning Machine Learning and Deep Learning hyperparameters and the respective methodology. Since the key elements such as the hyperparameter tuning process and measures of tunability and performance are presented in Bartz et al. (2022), we refer to this chapter for details.

The simplest, but also most computationally expensive, hyperparameter tuning approach uses manual search (or trial-and-error (Meignan et al. 2015)). Commonly encountered is simple random search, i.e., random and repeated selection of hyperparameters for evaluation, and lattice search (“grid search”). In addition, methods that perform directed search and other

model-free algorithms, i.e., algorithms that do not explicitly rely on a model, e.g., evolution strategies (Bartz-Beielstein et al. 2014) or pattern search (Lewis, Torczon, and Trosset 2000) play an important role. Also, “hyperband”, i.e., a multi-armed bandit strategy that dynamically allocates resources to a set of random configurations and uses successive bisections to stop configurations with poor performance (Li et al. 2016), is very common in hyperparameter tuning. The most sophisticated and efficient approaches are the Bayesian optimization and surrogate model based optimization methods, which are based on the optimization of cost functions determined by simulations or experiments.

We consider a surrogate optimization based hyperparameter tuning approach that uses the Python version of the SPOT (“Sequential Parameter Optimization Toolbox”) (Bartz-Beielstein, Lasarczyk, and Preuss 2005), which is suitable for situations where only limited resources are available. This may be due to limited availability and cost of hardware, or due to the fact that confidential data may only be processed locally, e.g., due to legal requirements. Furthermore, in our approach, the understanding of algorithms is seen as a key tool for enabling transparency and explainability. This can be enabled, for example, by quantifying the contribution of machine learning and deep learning components (nodes, layers, split decisions, activation functions, etc.). Understanding the importance of hyperparameters and the interactions between multiple hyperparameters plays a major role in the interpretability and explainability of machine learning models. SPOT provides statistical tools for understanding hyperparameters and their interactions. Last but not least, it should be noted that the SPOT software code is available in the open source `spotPython` package on github<sup>1</sup>, allowing replicability of the results. This tutorial describes the Python variant of SPOT, which is called `spotPython`. The R implementation is described in Bartz et al. (2022). SPOT is an established open source software that has been maintained for more than 15 years (Bartz-Beielstein, Lasarczyk, and Preuss 2005) (Bartz et al. 2022).

! 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 two parts. The first part describes the surrogate model based optimization process and the second part describes the hyperparameter tuning.

The first part is structured as follows: The concept of the hyperparameter tuning software `spotPython` is described in Chapter 1. This introduction is based on one-dimensional examples. Higher-dimensional examples are presented in Chapter 2. Chapter 3 describes

<sup>1</sup><https://github.com/sequential-parameter-optimization>

isotropic and anisotropic kriging. How different surrogate models from `scikit-learn` can be used as surrogates in `spotPython` optimization runs is explained in Chapter 4. Chapter 5 describes how different optimizers from the `scipy optimize` package can be used on the surrogate. The differences between the Kriging implementation in `spotPython` and the `GaussianProcessRegressor` in `scikit-learn` are explained in Chapter 6. Chapter 7 describes the expected improvement approach. How noisy functions can be handled is described in Chapter 8. Chapter 9 demonstrates how noisy functions can be handled with Optimal Computational Budget Allocation (OCBA) by Spot.

The second part is structured as follows: Chapter 10 describes the hyperparameter tuning of a `support vector classifier` from `scikit-learn` with `spotPython`. Chapter 11 illustrates the hyperparameter tuning of a `Hoeffding Adaptive Tree Regressor` from `river` with `spotPython`.

Chapter 14 describes the execution of the example from the tutorial “Hyperparameter Tuning with Ray Tune” (PyTorch 2023a). The integration of `spotPython` into the PyTorch training workflow is described in detail in the following sections. Section 14.1 describes the setup of the tuners. Section 14.3 describes the data loading. Section 14.5 describes the model to be tuned. The search space is introduced in Section 14.5.3. Optimizers are presented in Section 14.6.1. How to split the data in train, validation, and test sets is described in Section 14.7.1. The selection of the loss function and metrics is described in Section 14.7.5. Section 14.8.1 describes the preparation of the `spotPython` call. The objective function is described in Section 14.8.2. How to use results from previous runs and default hyperparameter configurations is described in Section 14.8.3. Starting the tuner is shown in Section 14.8.4. TensorBoard can be used to visualize the results as shown in Section 14.9. Results are discussed and explained in Section 14.10. Section 14.11 presents a summary and an outlook for the execution of the example from the tutorial “Hyperparameter Tuning with Ray Tune”.

Four more examples are presented in the following sections: Chapter 15 describes the hyperparameter tuning of a `random forest classifier` from `scikit-learn` with `spotPython`. Chapter 16 describes the hyperparameter tuning of an `XGBoost classifier` from `scikit-learn` with `spotPython`. Chapter 17 describes the hyperparameter tuning of a `support vector classifier` from `scikit-learn` with `spotPython`. Chapter 18 describes the hyperparameter tuning of a `k-nearest neighbors classifier` from `scikit-learn` with `spotPython`.

This part of the book is concluded with a description of the most recent PyTorch hyperparameter tuning approach, which is the integration of `spotPython` into the PyTorch Lightning training workflow. This is described in Chapter 19. This is considered as the most effective, efficient, and flexible way to integrate `spotPython` into the PyTorch training workflow.

### 💡 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](https://github.com/sequential-parameter-optimization/spotPython/blob/main/notebooks/14_spot_ray_hpt_torch_cifar10.ipynb).

## Software Used in this Book

`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](#).

`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.

`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**

## **Spot as an Optimizer**

# 1 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. Got 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.

## 1.1 Example: Spot and the Sphere Function

```
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

```

### 1.1.1 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

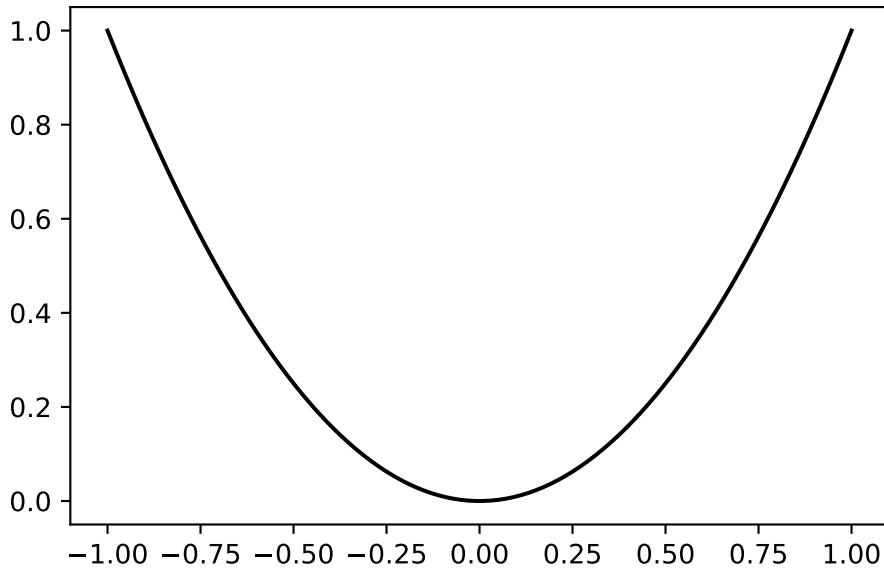
```

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()

```



```

spot_0 = spot.Spot(fun=fun,
                    lower = np.array([-1]),
                    upper = np.array([1]))

spot_0.run()

spotPython tuning: 1.2459257396367542e-08 [#####---] 73.33%
spotPython tuning: 1.2459257396367542e-08 [#####---] 80.00%
spotPython tuning: 1.2459257396367542e-08 [#####---] 86.67%
spotPython tuning: 1.2459257396367542e-08 [#####---] 93.33%
spotPython tuning: 4.897545259852824e-10 [#####---] 100.00% Done...

```

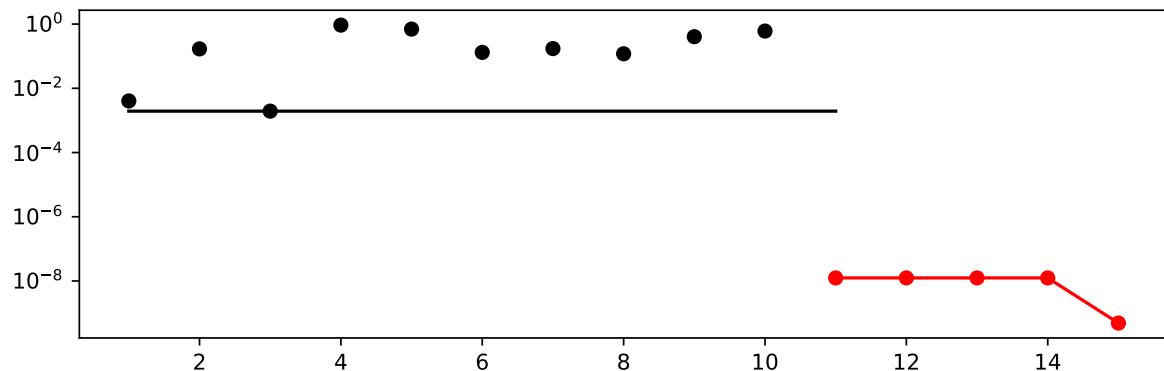
<spotPython.spot.spot.Spot at 0x105a07130>

```
spot_0.print_results()
```

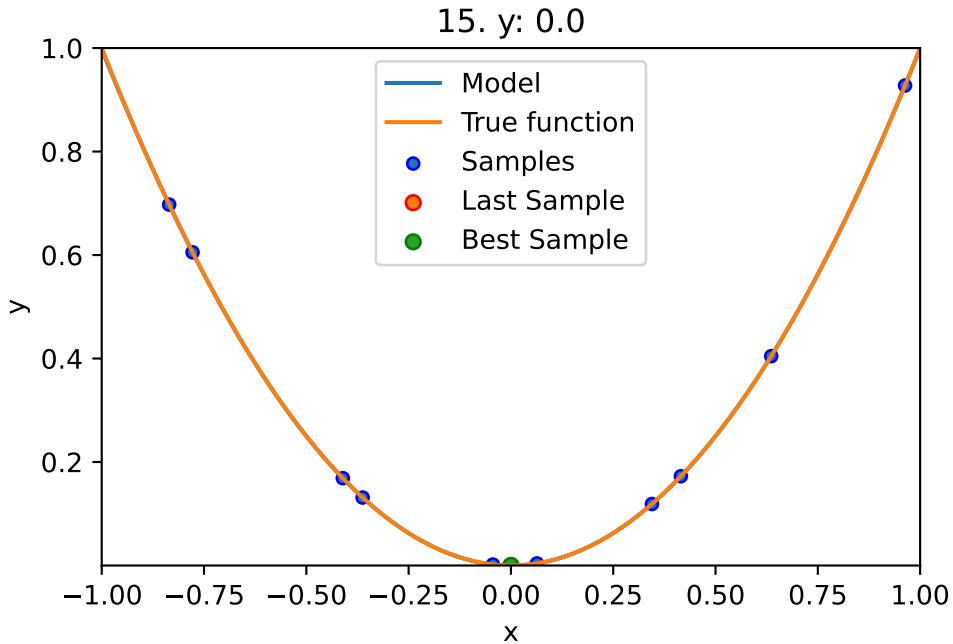
```
min y: 4.897545259852824e-10
x0: 2.2130398233770724e-05
```

```
[['x0', 2.2130398233770724e-05]]
```

```
spot_0.plot_progress(log_y=True)
```



```
spot_0.plot_model()
```



## 1.2 Spot Parameters: `fun_evals`, `init_size` and `show_models`

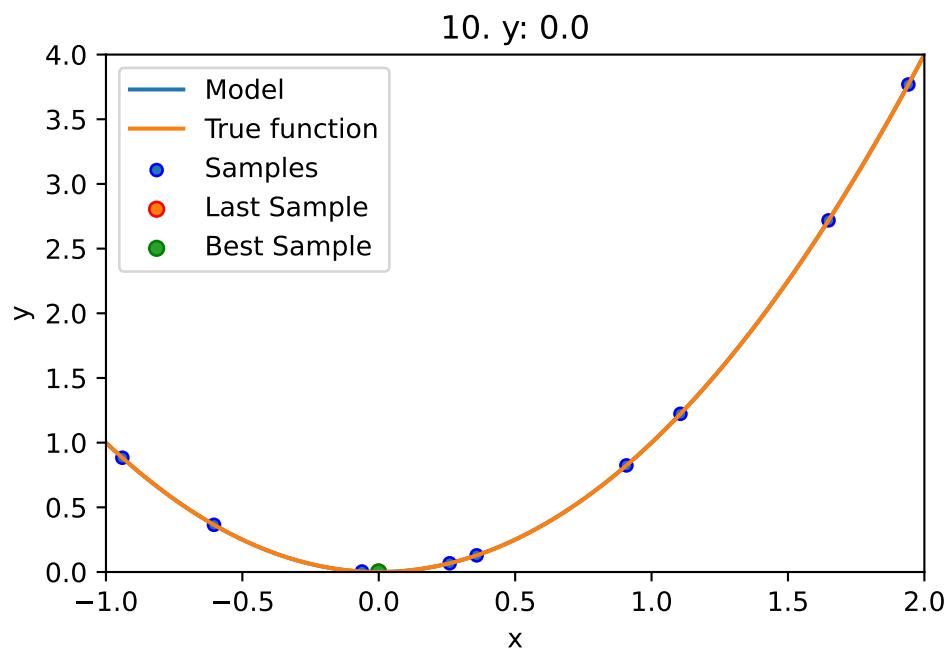
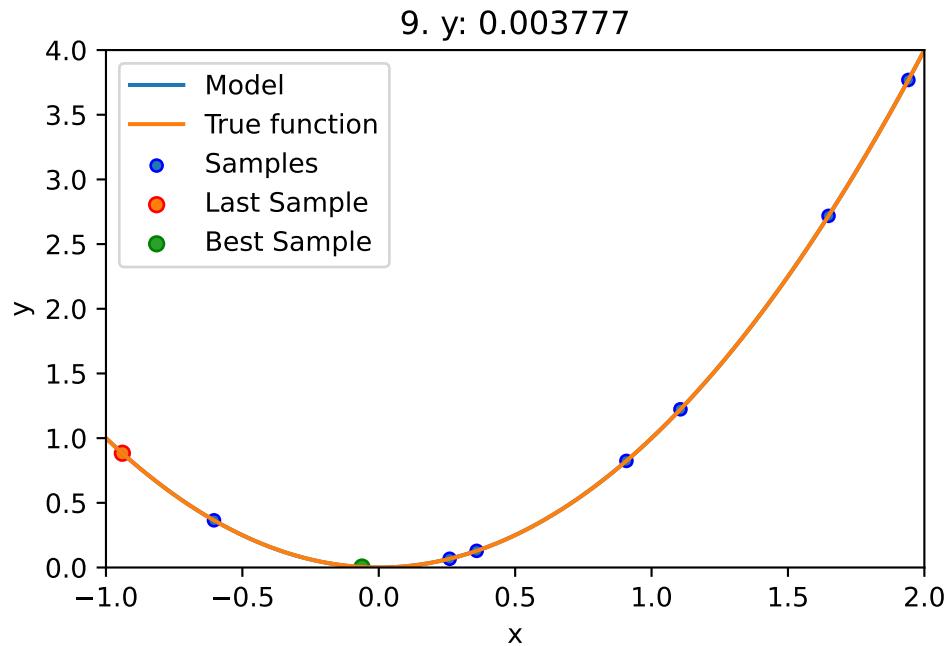
We will modify three parameters:

1. The number of function evaluations (`fun_evals`)
2. The size of the initial design (`init_size`)
3. The parameter `show_models`, which visualizes the search process for 1-dim functions.

The full list of the Spot parameters is shown in the Help System and in the notebook `spot_doc.ipynb`.

```
spot_1 = spot.Spot(fun=fun,
                    lower = np.array([-1]),
                    upper = np.array([2]),
                    fun_evals= 10,
                    seed=123,
                    show_models=True,
                    design_control={"init_size": 9})
```

```
spot_1.run()
```



```
spotPython tuning: 3.648984784366253e-07 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2ff541270>
```

### 1.3 Print the Results

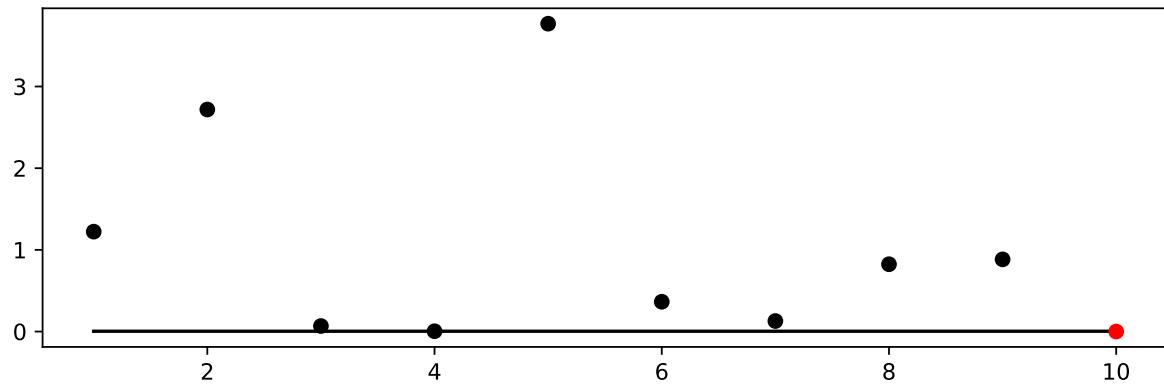
```
spot_1.print_results()
```

```
min y: 3.648984784366253e-07  
x0: -0.0006040682729929005
```

```
[['x0', -0.0006040682729929005]]
```

### 1.4 Show the Progress

```
spot_1.plot_progress()
```



### 1.5 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 “experiment name” to identify the hyperparameter tuning process. The experiment name is used to create a directory for the TensorBoard files.

```
from spotPython.utils.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "01"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)

fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
```

01\_maans14\_2023-10-30\_23-06-24

Since the `spot_tensorboard_path` is defined, `spotPython` will log the optimization process in the TensorBoard files. The TensorBoard files are stored in the directory `spot_tensorboard_path`. We can pass the TensorBoard information to the `Spot` method via the `fun_control` dictionary.

```
spot_tuner = spot.Spot(fun=fun,
                       lower = np.array([-1]),
                       upper = np.array([2]),
                       fun_evals= 10,
                       seed=123,
                       show_models=False,
                       design_control={"init_size": 5},
                       fun_control=fun_control,)

spot_tuner.run()
```

spotPython tuning: 2.760068954719313e-05 [#####----] 60.00%

spotPython tuning: 7.588618329369276e-07 [#####---] 70.00%

spotPython tuning: 7.546340185833067e-07 [#####----] 80.00%

spotPython tuning: 3.3653559447366466e-07 [#####----] 90.00%

spotPython tuning: 7.948275967360275e-11 [#####----] 100.00% Done...

```
<spotPython.spot.spot.Spot at 0x31ab6f730>
```

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`.

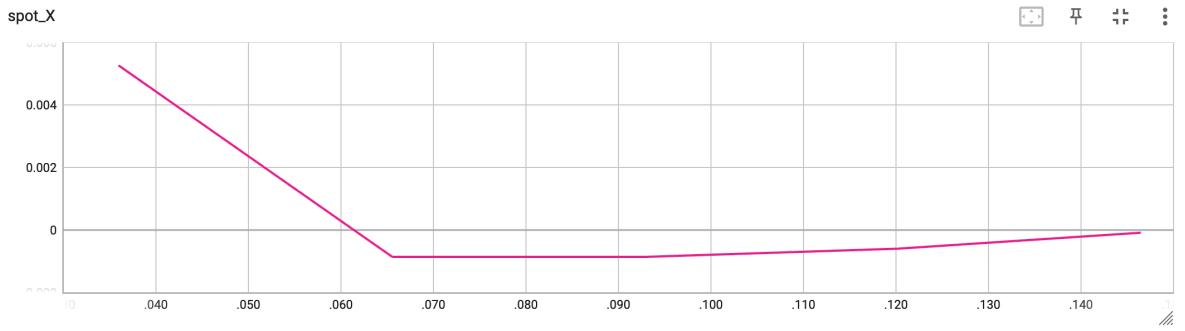
TensorBoard will start a web server on port 6006. We can access the TensorBoard web server with the following URL:

```
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 vi-

sualizes 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  $\theta$  of the Kriging surrogate is plotted against the number of optimization steps.

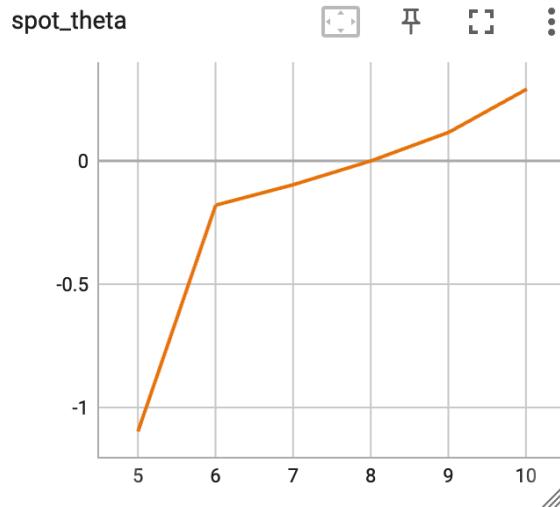


Figure 1.1: TensorBoard visualization of the spotPython process.

## 2 Multi-dimensional Functions

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

### 2.1 Example: Spot and the 3-dim Sphere Function

```
import numpy as np
from spotPython.fun.objectivefunctions import analytical
from spotPython.spot import spot
```

#### 2.1.1 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:

$$f(x) = \sum_i^n x_i^2$$

- Here we will use  $n = 3$ .

```
fun = analytical().fun_sphere
```

- The size of the `lower` bound vector determines the problem dimension.
- Here we will use `-1.0 * np.ones(3)`, i.e., a three-dim function.
- We will use three different `theta` values (one for each dimension), i.e., we set  
`surrogate_control={"n_theta": 3}.`

#### i TensorBoard

Similar to the one-dimensional case, which was introduced in Section [Section 1.5](#), 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.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "02"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)

fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
```

02\_maans14\_2023-10-30\_23-06-34

```
spot_3 = spot.Spot(fun=fun,
                    lower = -1.0*np.ones(3),
                    upper = np.ones(3),
                    var_name=["Pressure", "Temp", "Lambda"],
                    show_progress=True,
                    surrogate_control={"n_theta": 3},
                    fun_control=fun_control,)

spot_3.run()

spotPython tuning: 0.03443367156190887 [#####---] 73.33%

spotPython tuning: 0.031348911082058686 [#####---] 80.00%

spotPython tuning: 0.0009629115535977041 [#####---] 86.67%

spotPython tuning: 8.600065786394651e-05 [#####---] 93.33%

spotPython tuning: 5.9908343748136085e-05 [#####] 100.00% Done...

<spotPython.spot.Spot at 0x177b0d8a0>
```

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/
```

### 2.1.2 Results

```
spot_3.print_results()
```

```
min y: 5.9908343748136085e-05
```

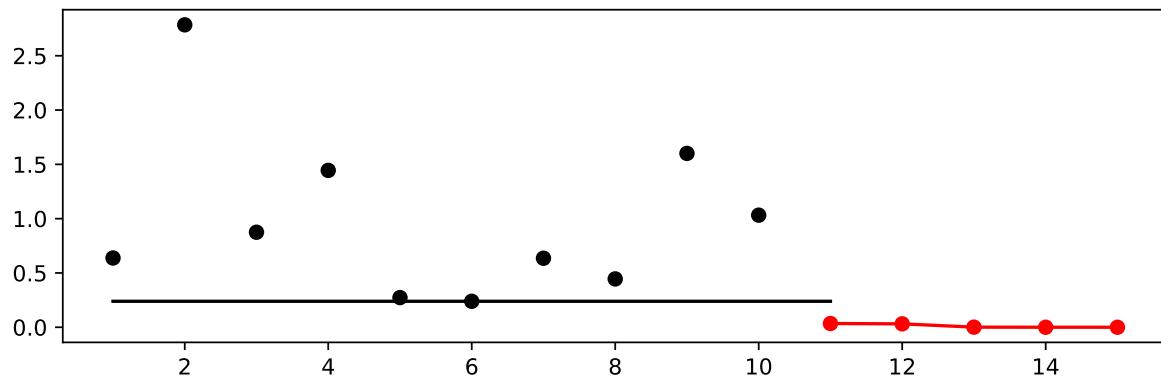
```
Pressure: 0.005157864627379999
```

```
Temp: 0.00195710957248863
```

```
Lambda: 0.005429042121316765
```

```
[['Pressure', 0.005157864627379999],  
 ['Temp', 0.00195710957248863],  
 ['Lambda', 0.005429042121316765]]
```

```
spot_3.plot_progress()
```

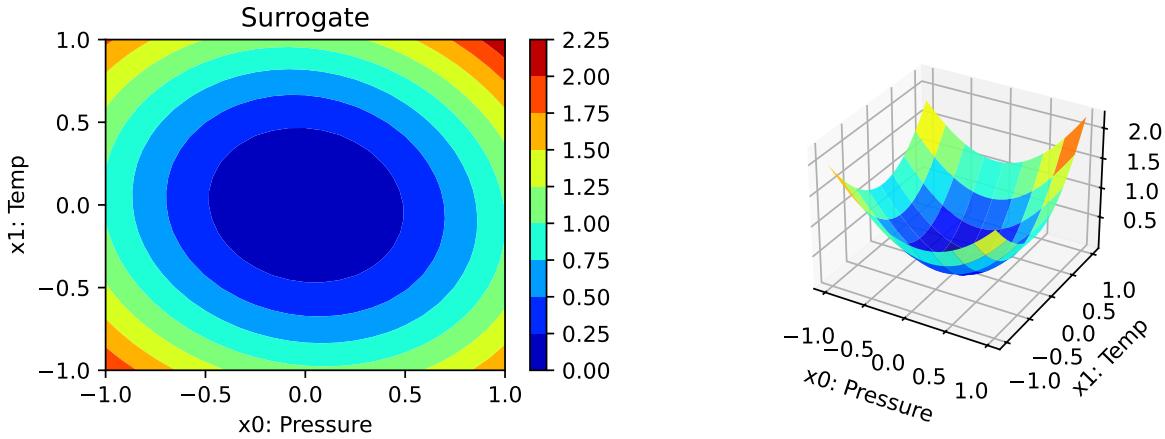


### 2.1.3 A Contour Plot

- We can select two dimensions, say  $i = 0$  and  $j = 1$ , and generate a contour plot as follows.

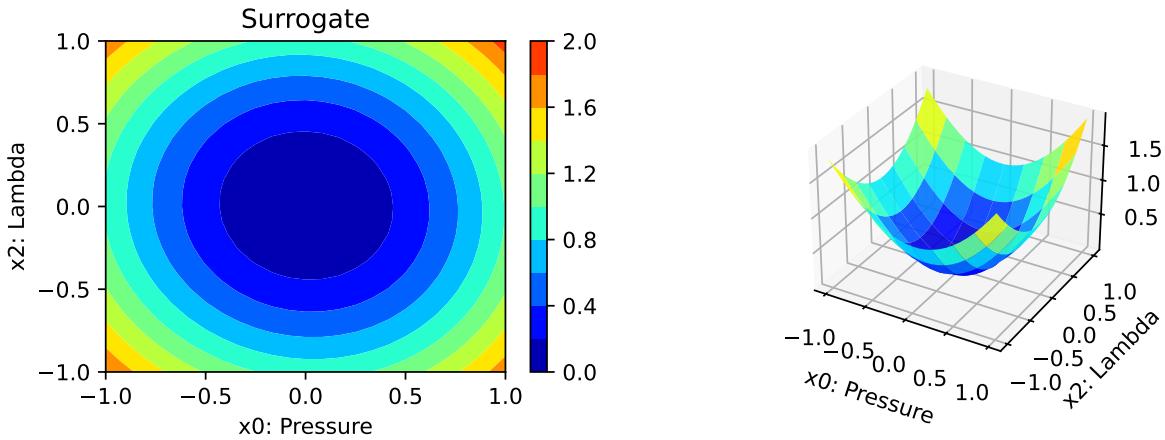
- Note: We have specified identical `min_z` and `max_z` values to generate comparable plots!

```
spot_3.plot_contour(i=0, j=1, min_z=0, max_z=2.25)
```



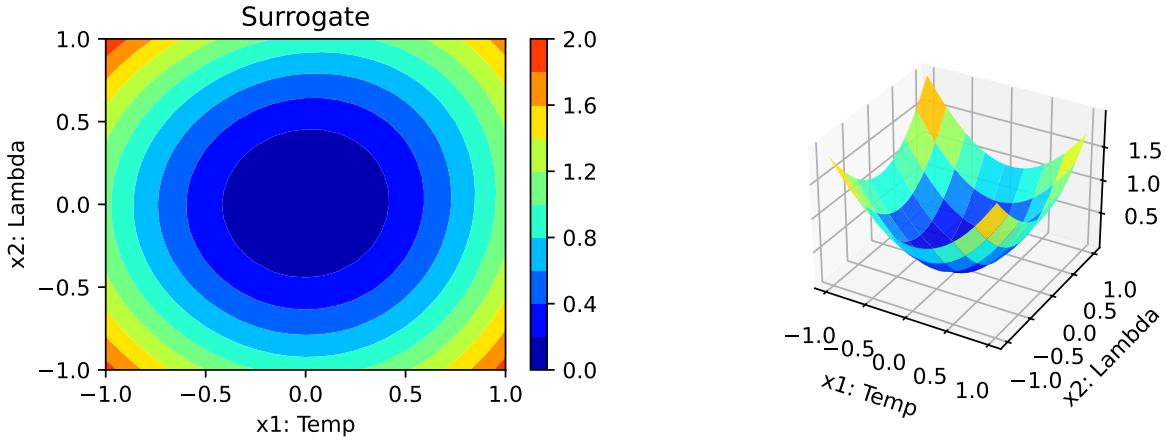
- In a similar manner, we can plot dimension  $i = 0$  and  $j = 2$ :

```
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:

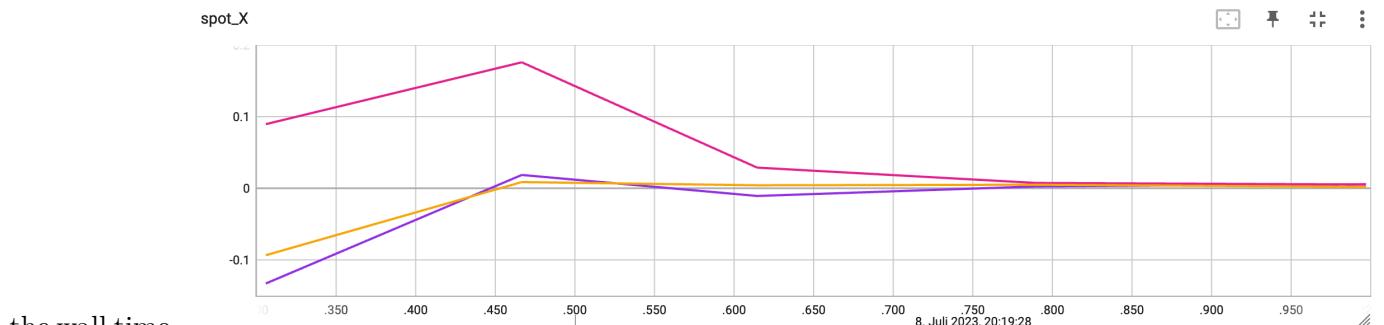
```
spot_3.print_importance()
```

```
Pressure: 100.0
Temp: 99.78247670817808
Lambda: 94.72233826625329
```

```
[['Pressure', 100.0],
 ['Temp', 99.78247670817808],
 ['Lambda', 94.72233826625329]]
```

## 2.1.4 TensorBoard

The second TensorBoard visualization shows the input values, i.e.,  $x_0, \dots, x_2$ , plotted against



the wall time.

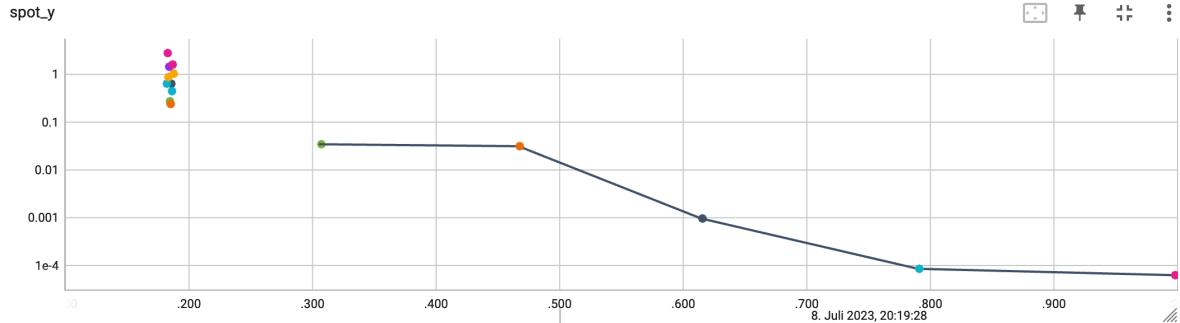


Figure 2.1: TensorBoard visualization of the spotPython process. Objective function values plotted against wall time.

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  $\theta$  of the Kriging surrogate is plotted against the number of optimization steps.

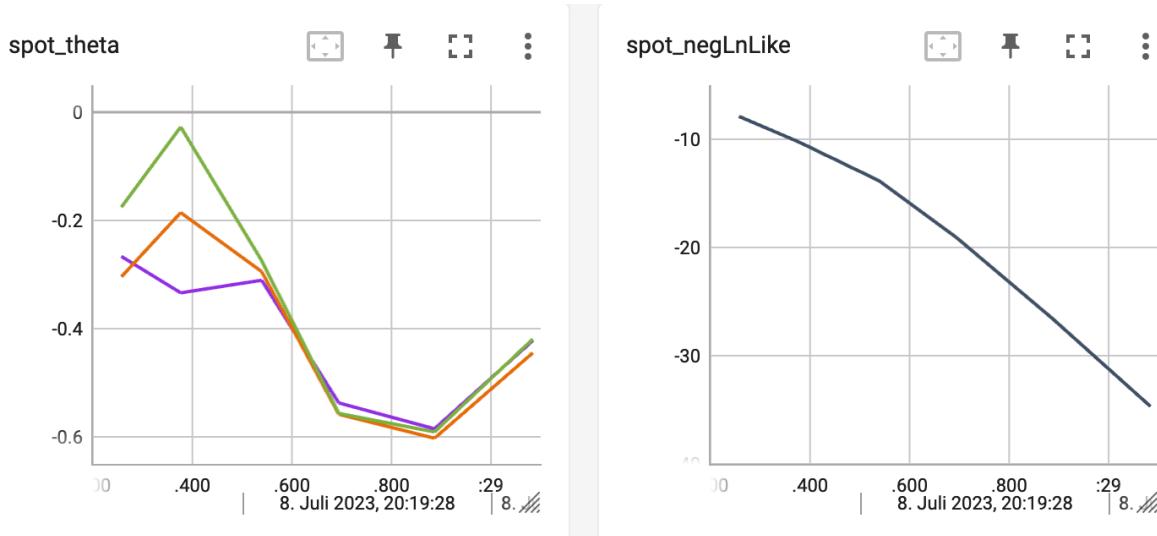


Figure 2.2: TensorBoard visualization of the spotPython surrogate model.

## 2.2 Conclusion

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

## 2.3 Exercises

- Important:
  - Results from these exercises should be added to this document, i.e., you should submit an updated version of this notebook.
  - Please combine your results using this notebook.
  - Only one notebook from each group!
  - Presentation is based on this notebook. No additional slides are required!
  - spotPython version 0.16.11 (or greater) is required

### 2.3.1 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?

### 2.3.2 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?

### 2.3.3 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?

#### **2.3.4 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?

# 3 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.

## 3.1 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
```

### 3.1.1 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:

$$f(x, y) = x^2 + y^2$$

```
fun = analytical().fun_sphere
fun_control = {"sigma": 0,
               "seed": 123}
```

- 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.

```
spot_2 = spot.Spot(fun=fun,
                    lower = np.array([-1, -1]),
                    upper = np.array([1, 1]))
```

```

spot_2.run()

spotPython tuning: 1.8750731199649933e-05 [#####---] 73.33%
spotPython tuning: 1.8750731199649933e-05 [#####--] 80.00%
spotPython tuning: 1.8750731199649933e-05 [#####---] 86.67%
spotPython tuning: 1.8750731199649933e-05 [#####---] 93.33%
spotPython tuning: 1.8750731199649933e-05 [#####---] 100.00% Done...
<spotPython.spot.spot.Spot at 0x16e451ab0>

```

### 3.1.2 Results

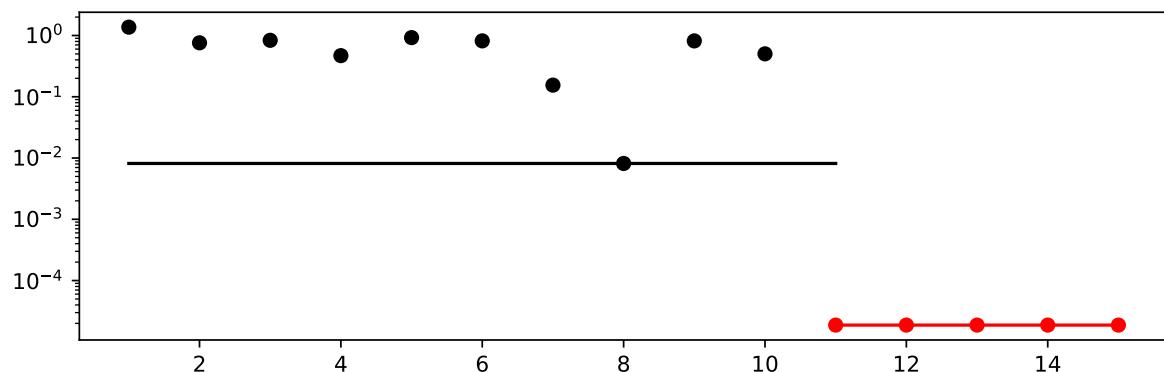
```

spot_2.print_results()

min y: 1.8750731199649933e-05
x0: 0.0015130475553084242
x1: 0.0040572673433020325

[['x0', 0.0015130475553084242], ['x1', 0.0040572673433020325]]
```

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



## 3.2 Example With Anisotropic Kriging

- 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={"n_theta": 2}` to enable this behavior (2 is the problem dimension).

### i TensorBoard

Similar to the one-dimensional case, which was introduced in Section [Section 1.5](#), 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.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "03"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)

fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
```

03\_maans14\_2023-10-30\_23-06-48

```
spot_2_anisotropic = spot.Spot(fun=fun,
                                lower = np.array([-1, -1]),
                                upper = np.array([1, 1]),
                                surrogate_control={"n_theta": 2},
                                fun_control=fun_control)
spot_2_anisotropic.run()
```

spotPython tuning: 1.7904944376943484e-05 [#####---] 73.33%

spotPython tuning: 1.7904944376943484e-05 [#####---] 80.00%

```
spotPython tuning: 1.7904944376943484e-05 [#####--] 86.67%
```

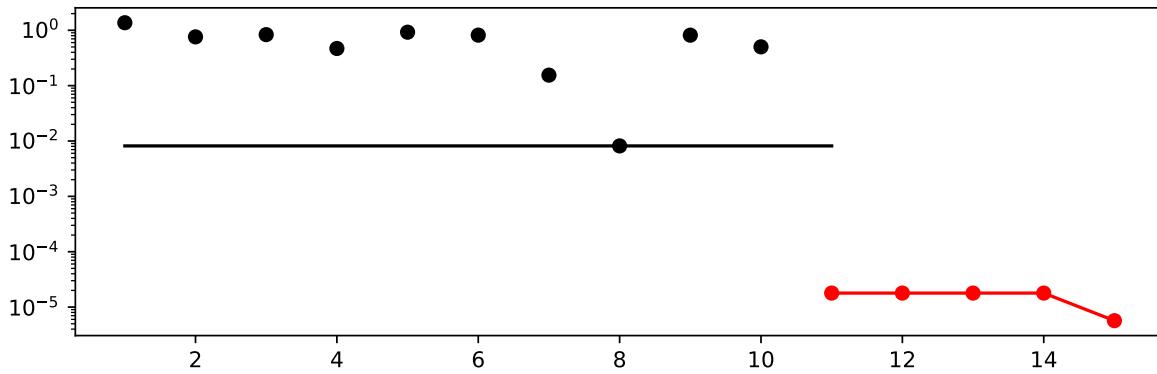
```
spotPython tuning: 1.7904944376943484e-05 [#####--] 93.33%
```

```
spotPython tuning: 5.68261952864018e-06 [#####--] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x309237ee0>
```

- The search progress of the optimization with the anisotropic model can be visualized:

```
spot_2_anisotropic.plot_progress(log_y=True)
```

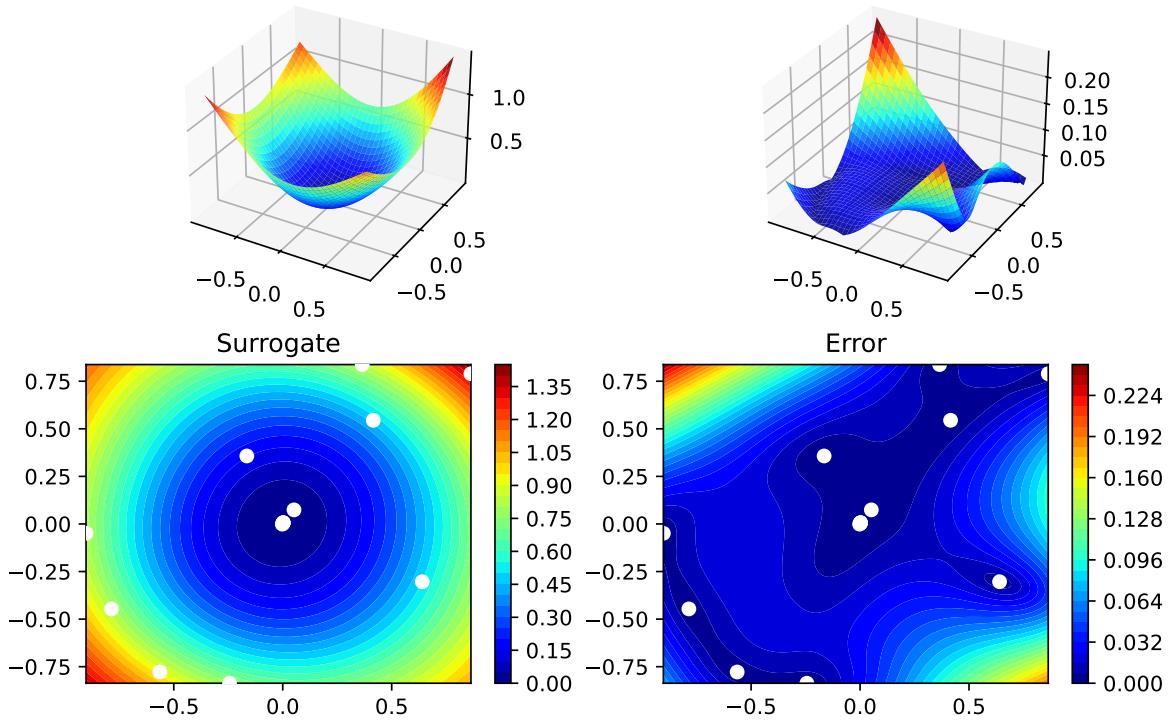


```
spot_2_anisotropic.print_results()
```

```
min y: 5.68261952864018e-06
x0: -0.002137037687426695
x1: -0.0010562620182313395
```

```
[['x0', -0.002137037687426695], ['x1', -0.0010562620182313395]]
```

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



### 3.2.1 Taking a Look at the theta Values

#### 3.2.1.1 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([0.16545225, 0.28999215])
```

- 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([0.26287446])
```

### 3.2.1.2 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  $\theta$  of the Kriging surrogate is plotted against the number of optimization steps.

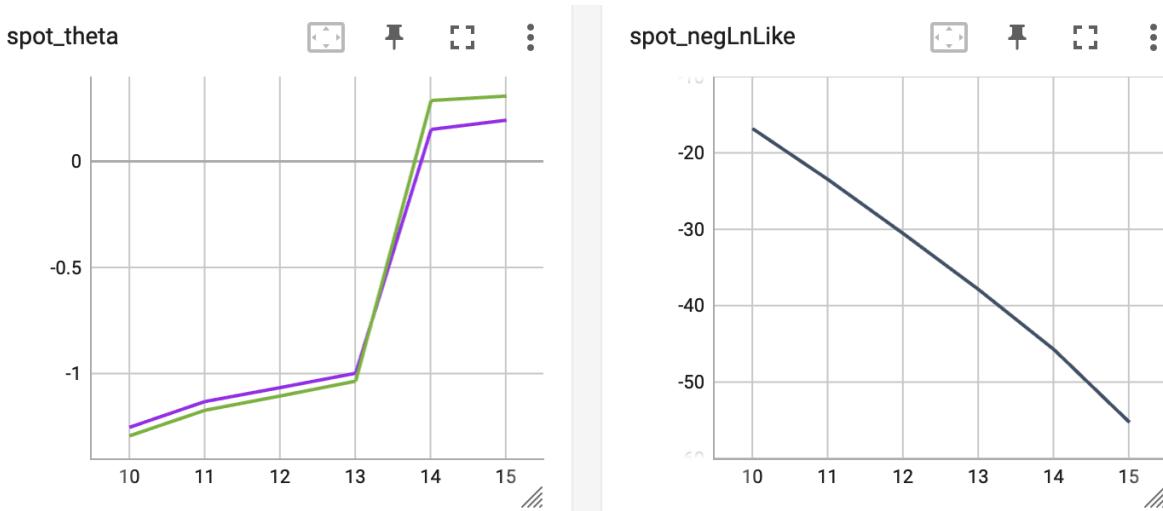


Figure 3.1: TensorBoard visualization of the `spotPython` surrogate model.

## 3.3 Exercises

### 3.3.1 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` 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):

```
fun_evals=inf,
max_time=1,
```

### 3.3.2 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`.

### 3.3.3 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` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

### 3.3.4 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` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

# 4 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
```

## 4.1 Example: Branin Function with spotPython's Internal Kriging Surrogate

### 4.1.1 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
lower = np.array([-5,-0])
upper = np.array([10,15])
```

```
fun = analytical().fun_branin
```

### i TensorBoard

Similar to the one-dimensional case, which was introduced in Section Section 1.5, 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.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "04"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)

fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
```

```
04_maans14_2023-10-30_23-07-10
```

#### 4.1.2 Running the surrogate model based optimizer Spot:

```
spot_2 = spot.Spot(fun=fun,
                    lower = lower,
                    upper = upper,
                    fun_evals = 20,
                    max_time = inf,
                    seed=123,
                    design_control={"init_size": 10},
                    fun_control=fun_control)

spot_2.run()
```

```
spotPython tuning: 3.447460568213552 [#####----] 55.00%
```

```
spotPython tuning: 3.447460568213552 [#####----] 60.00%
```

```
spotPython tuning: 3.0394923470341615 [#####----] 65.00%
```

```
spotPython tuning: 3.0394923470341615 [#####---] 70.00%
spotPython tuning: 1.1632551812894665 [#####---] 75.00%
spotPython tuning: 0.612453922191154 [#####---] 80.00%
spotPython tuning: 0.4576355201245761 [#####---] 85.00%
spotPython tuning: 0.3983178342401956 [#####---] 90.00%
spotPython tuning: 0.3983178342401956 [#####---] 95.00%
spotPython tuning: 0.3983178342401956 [#####---] 100.00% Done...
<spotPython.spot.spot.Spot at 0x2ebbebf40>
```

#### 4.1.3 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  $\theta$  of the Kriging surrogate is plotted against the number of optimization steps.

#### 4.1.4 Print the Results

```
spot_2.print_results()
```

```
min y: 0.3983178342401956
x0: 3.135416996435963
x1: 2.2955490975636685

[['x0', 3.135416996435963], ['x1', 2.2955490975636685]]
```

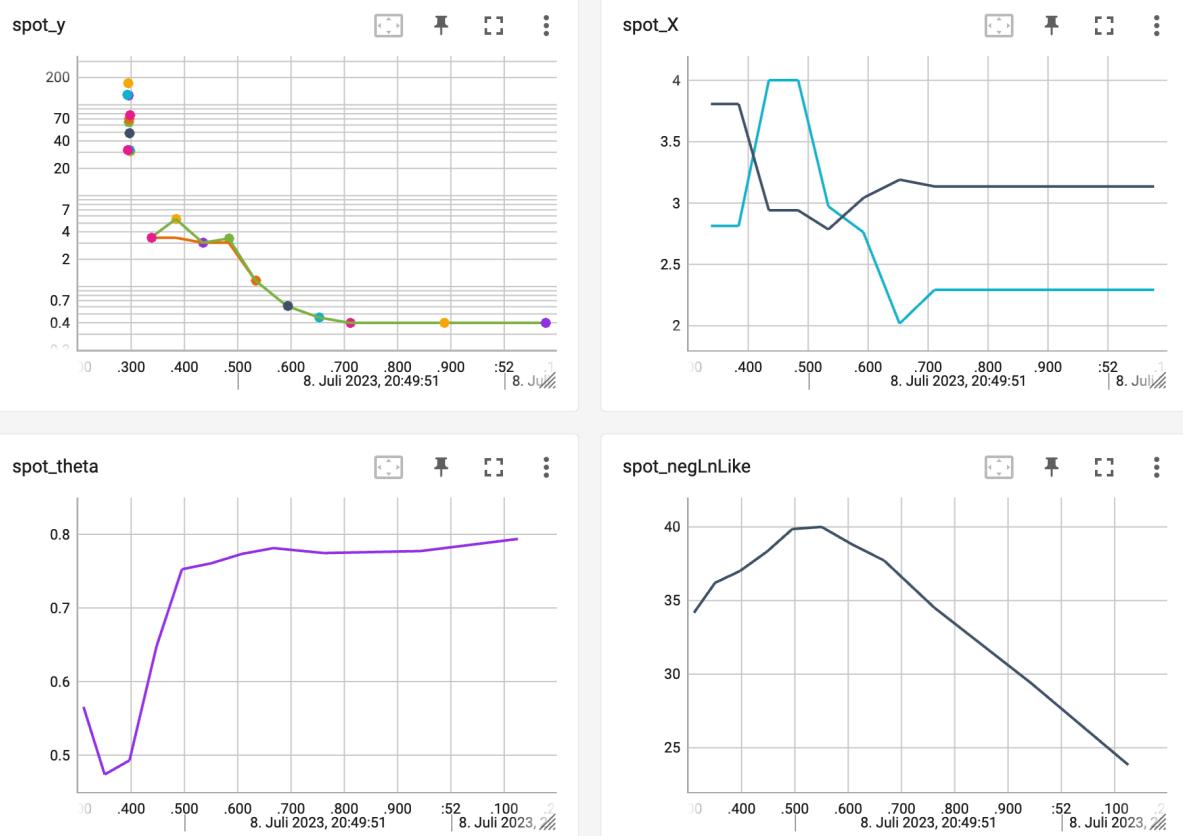
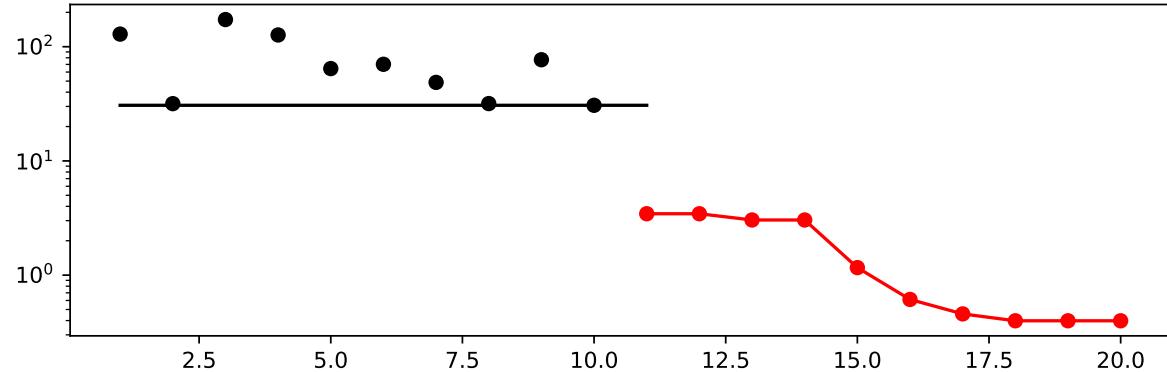


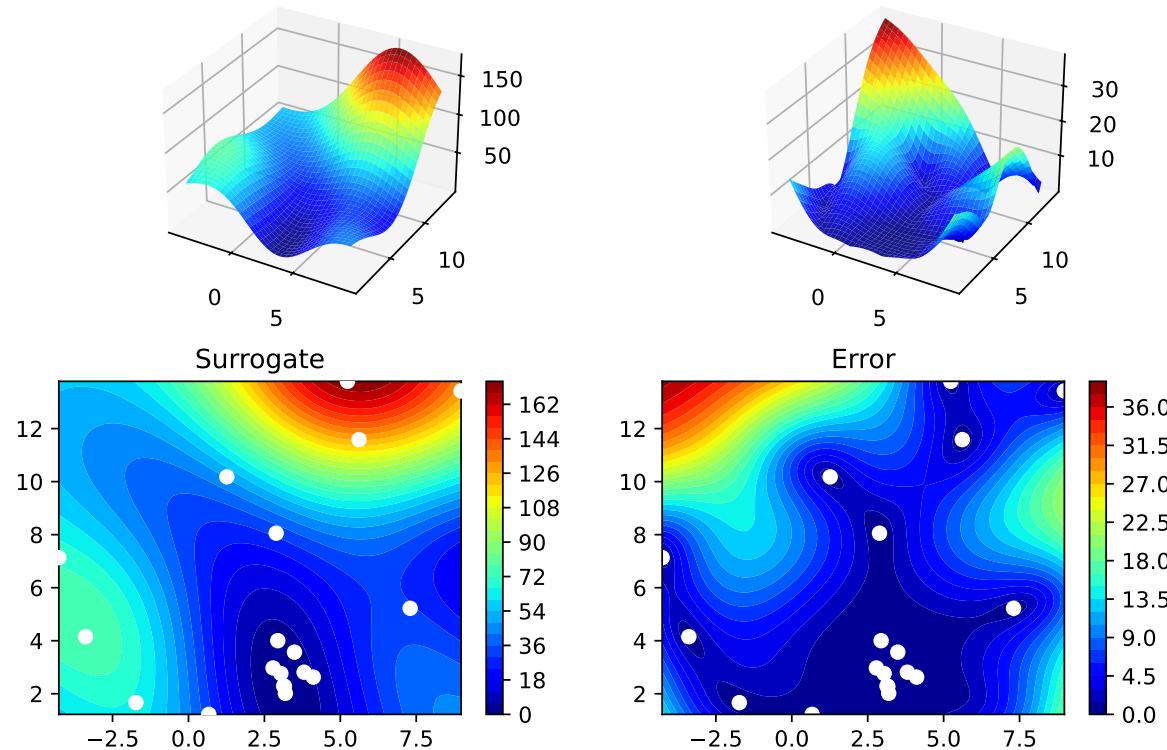
Figure 4.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

#### 4.1.5 Show the Progress and the Surrogate

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



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



## 4.2 Example: Using Surrogates From scikit-learn

- Default is the `spotPython` (i.e., the internal) `Kriging` surrogate.
- It can be called explicitly 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)
```

### 4.2.1 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 surrogate 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,
                      lower = lower,
                      upper = upper,
                      fun_evals = 20,
                      seed=123,
                      design_control={"init_size": 10},
                      surrogate = S_GP)

spot_2_GP.run()
```

```
spotPython tuning: 18.865102092040814 [#####----] 55.00%
```

```
spotPython tuning: 4.067063943633956 [#####----] 60.00%
```

```
spotPython tuning: 3.461921636551292 [#####----] 65.00%
```

```
spotPython tuning: 3.461921636551292 [#####---] 70.00%
```

```
spotPython tuning: 1.3283243814960493 [#####---] 75.00%
```

```
spotPython tuning: 0.9549503108510411 [#####---] 80.00%
```

```
spotPython tuning: 0.9352250003678275 [#####---] 85.00%
```

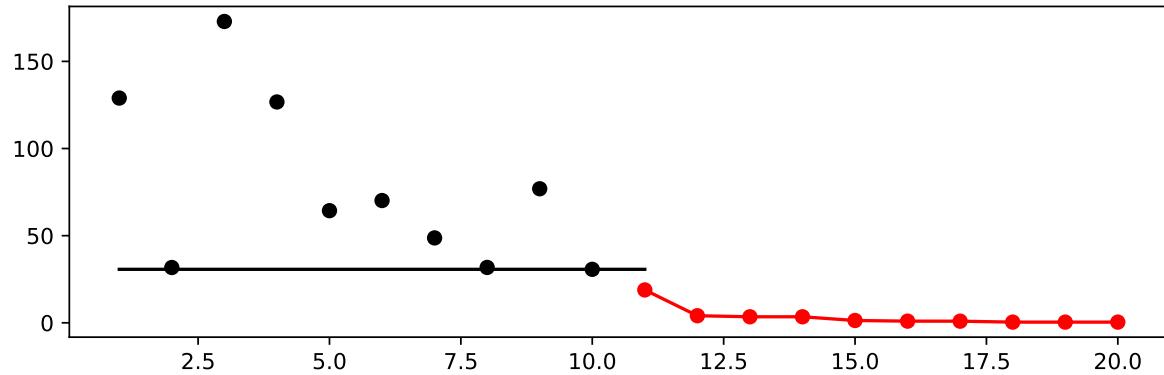
```
spotPython tuning: 0.39960822331589974 [#####---] 90.00%
```

```
spotPython tuning: 0.3981631812933486 [#####] 95.00%
```

```
spotPython tuning: 0.3981631812933486 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2ed3c5840>
```

```
spot_2_GP.plot_progress()
```



```
spot_2_GP.print_results()
```

```
min y: 0.3981631812933486
x0: 3.1491652274330053
x1: 2.2698186003445153
```

```
[['x0', 3.1491652274330053], ['x1', 2.2698186003445153]]
```

### 4.3 Example: One-dimensional Sphere Function With spotPython's Kriging

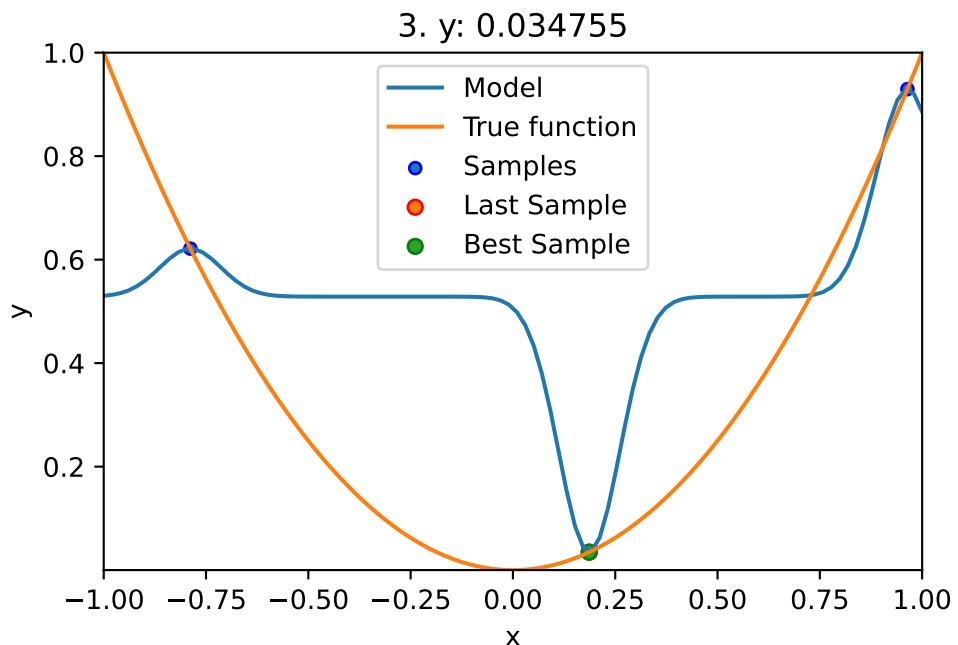
- In this example, we will use an one-dimensional function, which allows us to visualize the optimization process.
  - `show_models= True` is added to the argument list.

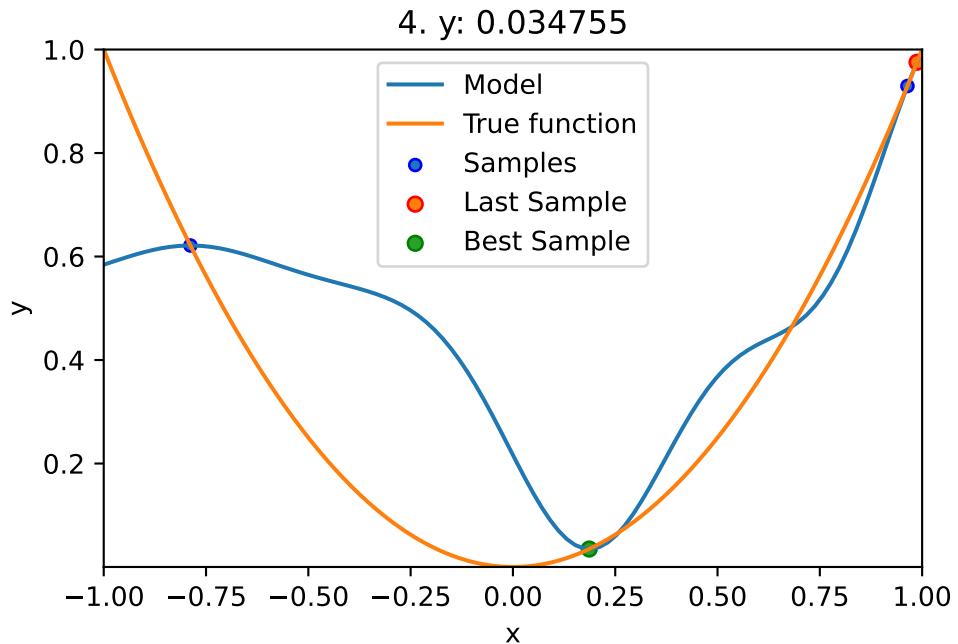
```

from spotPython.fun.objectivefunctions import analytical
lower = np.array([-1])
upper = np.array([1])
fun = analytical(seed=123).fun_sphere

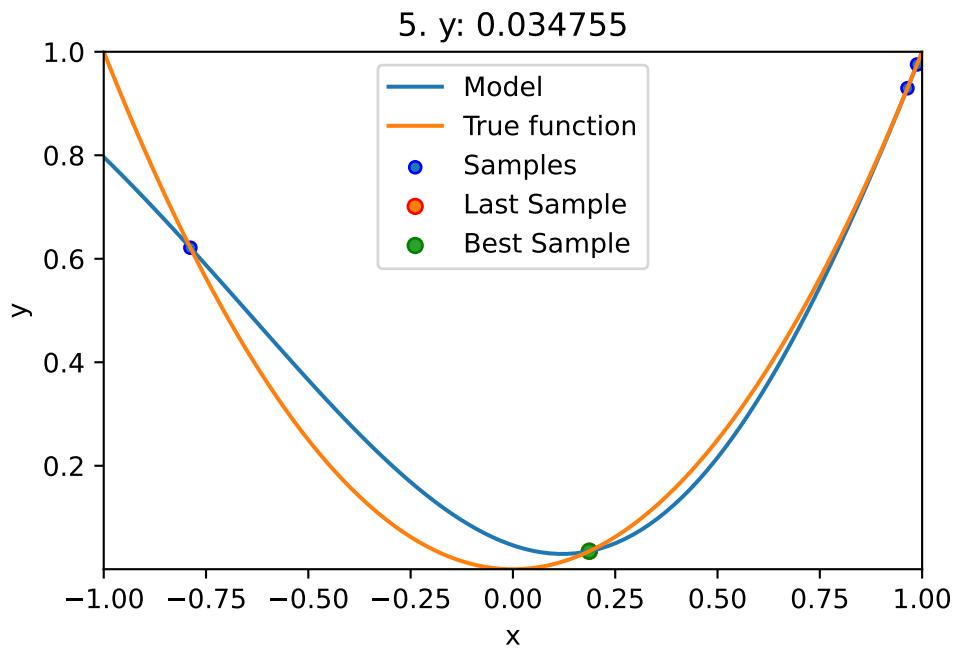
spot_1 = spot.Spot(fun=fun,
                    lower = lower,
                    upper = upper,
                    fun_evals = 10,
                    max_time = inf,
                    seed=123,
                    show_models= True,
                    tolerance_x = np.sqrt(np.spacing(1)),
                    design_control={"init_size": 3},)
spot_1.run()

```

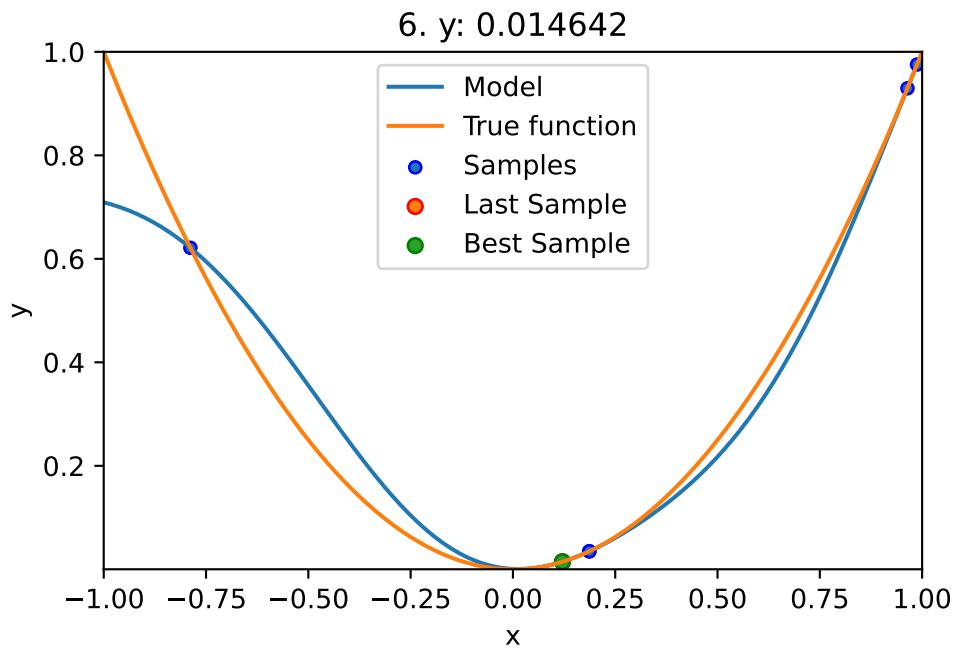




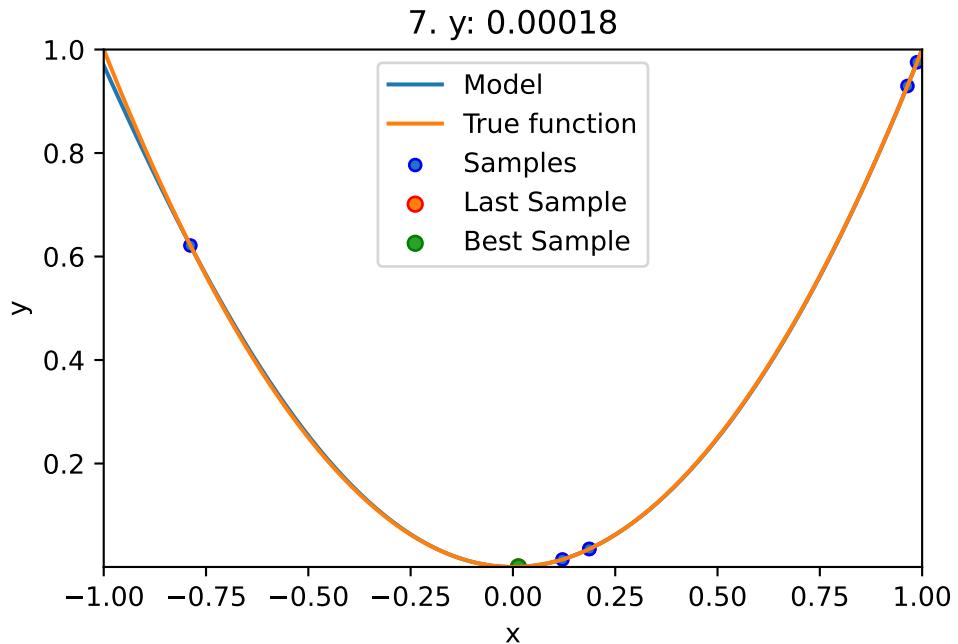
```
spotPython tuning: 0.03475493366922229 [#####-----] 40.00%
```



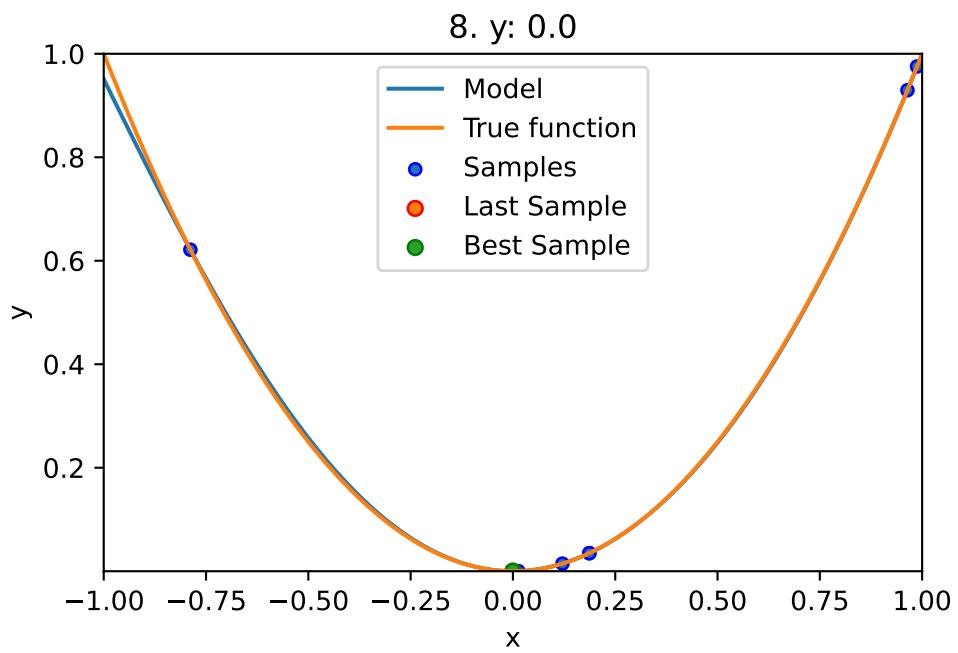
```
spotPython tuning: 0.03475493366922229 [#####----] 50.00%
```



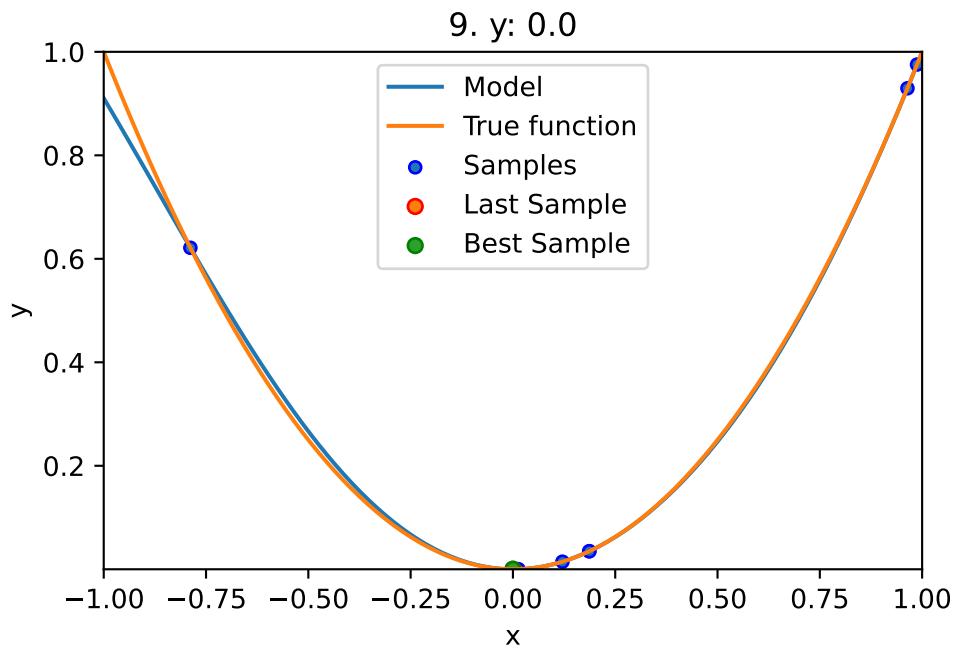
```
spotPython tuning: 0.014642358641673271 [#####----] 60.00%
```



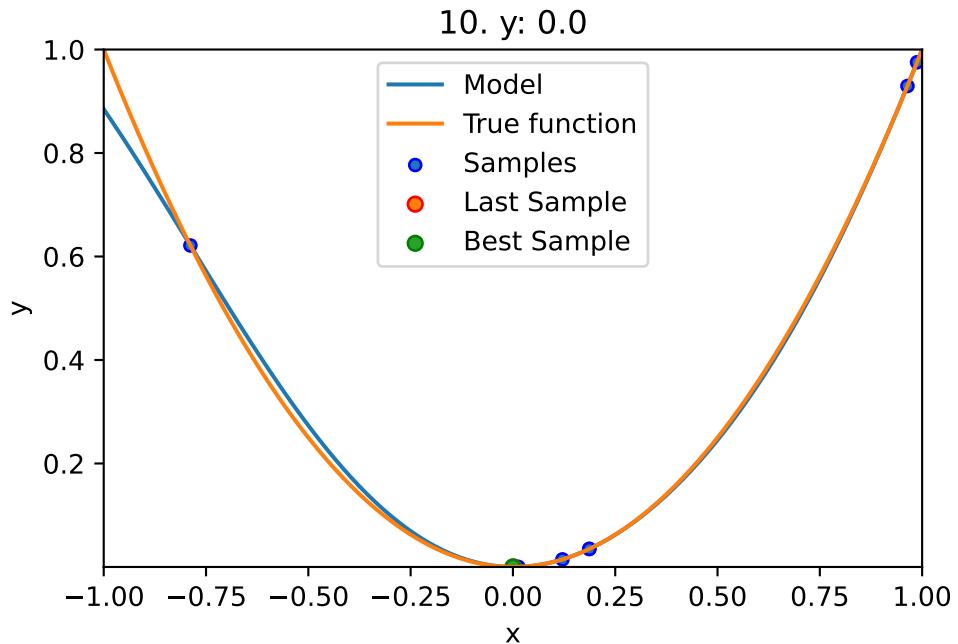
```
spotPython tuning: 0.00018032497380230452 [#####---] 70.00%
```



```
spotPython tuning: 2.1786524623022742e-08 [#####--] 80.00%
```



```
spotPython tuning: 2.1786524623022742e-08 [#####--] 90.00%
```



```
spotPython tuning: 2.1786524623022742e-08 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2ff079b40>
```

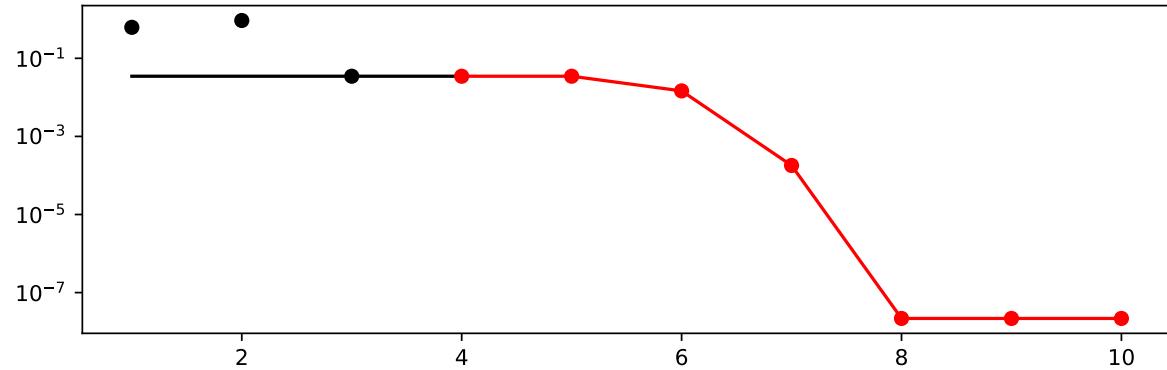
#### 4.3.1 Results

```
spot_1.print_results()
```

```
min y: 2.1786524623022742e-08
x0: -0.00014760259016366462
```

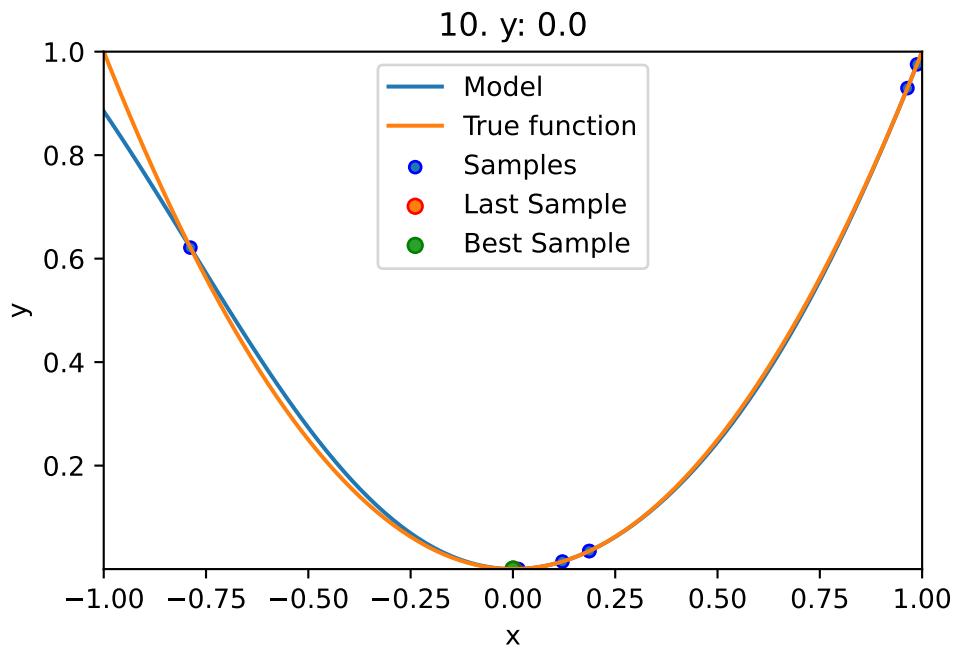
```
[['x0', -0.00014760259016366462]]
```

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



- The method `plot_model` plots the final surrogate:

```
spot_1.plot_model()
```



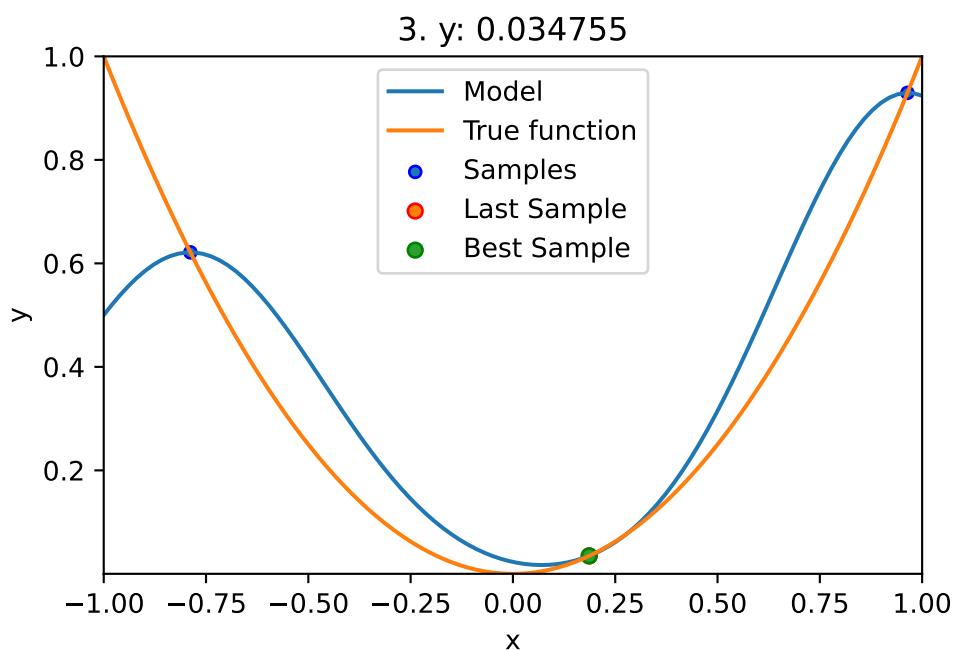
#### 4.4 Example: Sklearn Model GaussianProcess

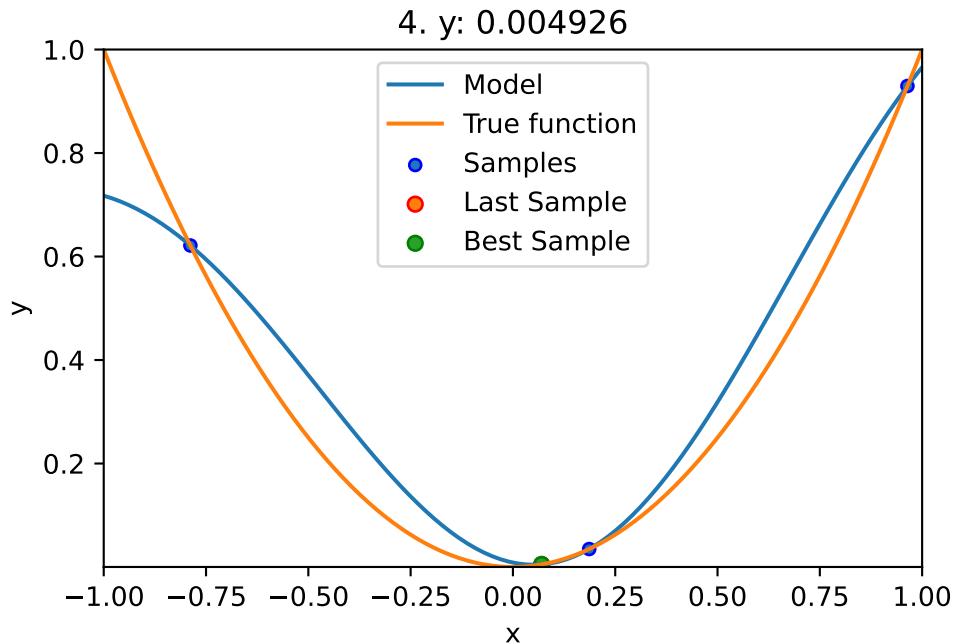
- This example visualizes the search process on the `GaussianProcessRegression` surrogate from `sklearn`.
- Therefore `surrogate = S_GP` is added to the argument list.

```

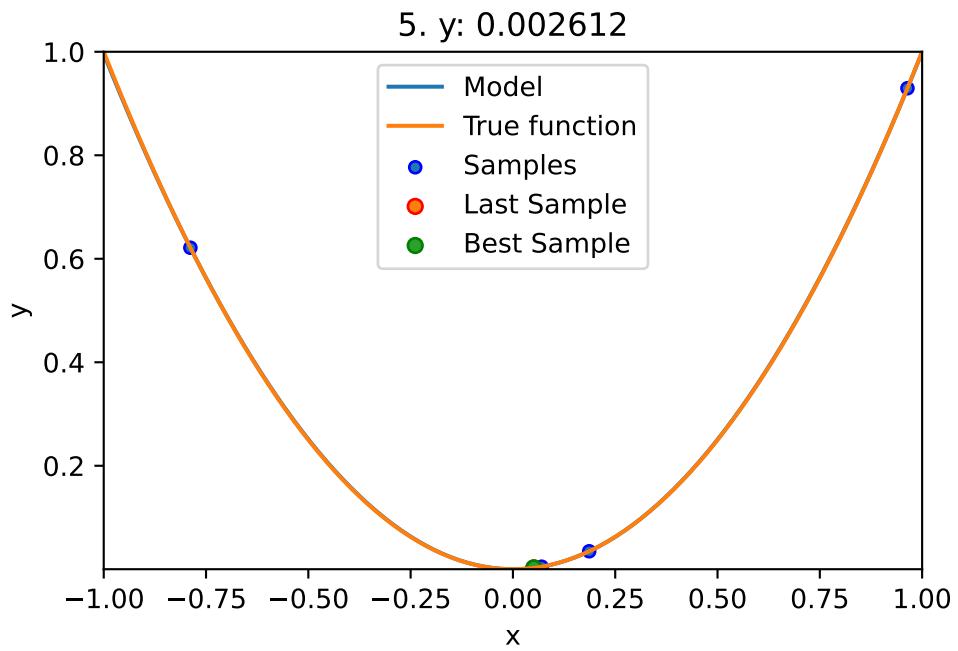
fun = analytical(seed=123).fun_sphere
spot_1_GP = spot.Spot(fun=fun,
                      lower = lower,
                      upper = upper,
                      fun_evals = 10,
                      max_time = inf,
                      seed=123,
                      show_models= True,
                      design_control={"init_size": 3},
                      surrogate = S_GP)
spot_1_GP.run()

```

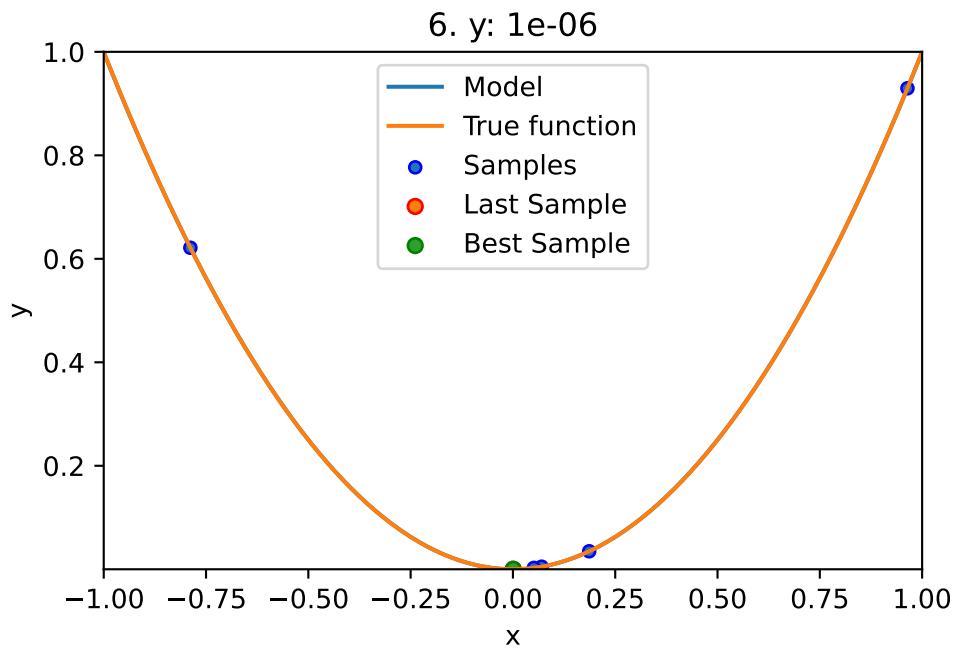




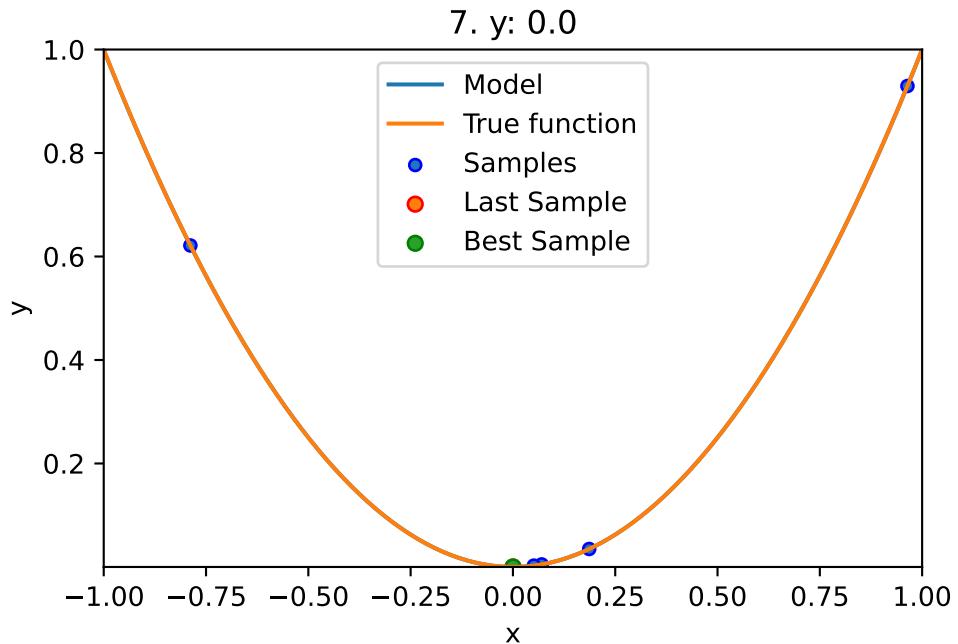
spotPython tuning: 0.0049257617153734565 [####-----] 40.00%



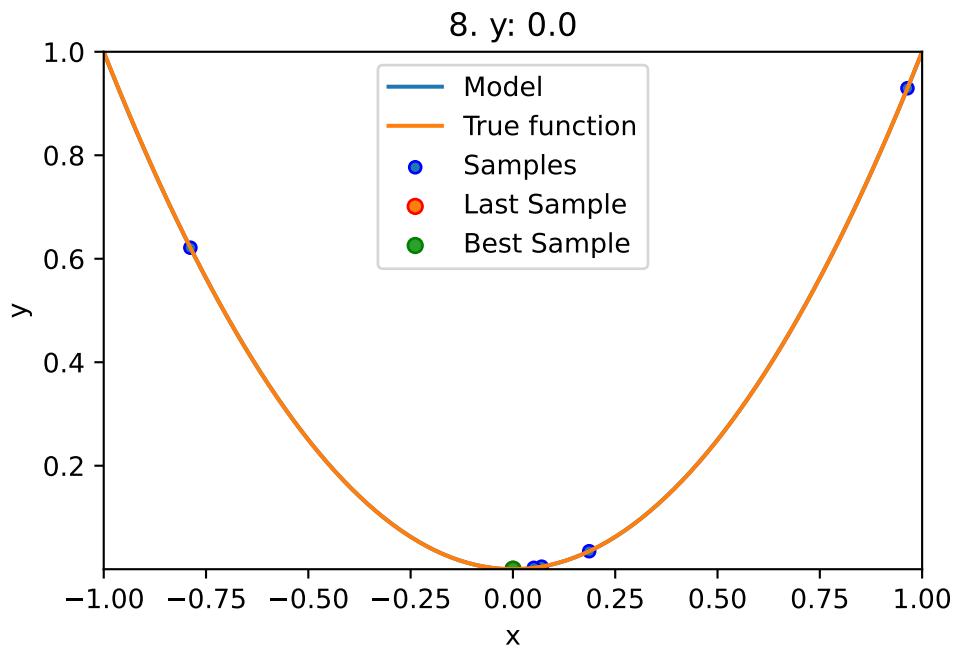
```
spotPython tuning: 0.002612076484523571 [#####----] 50.00%
```



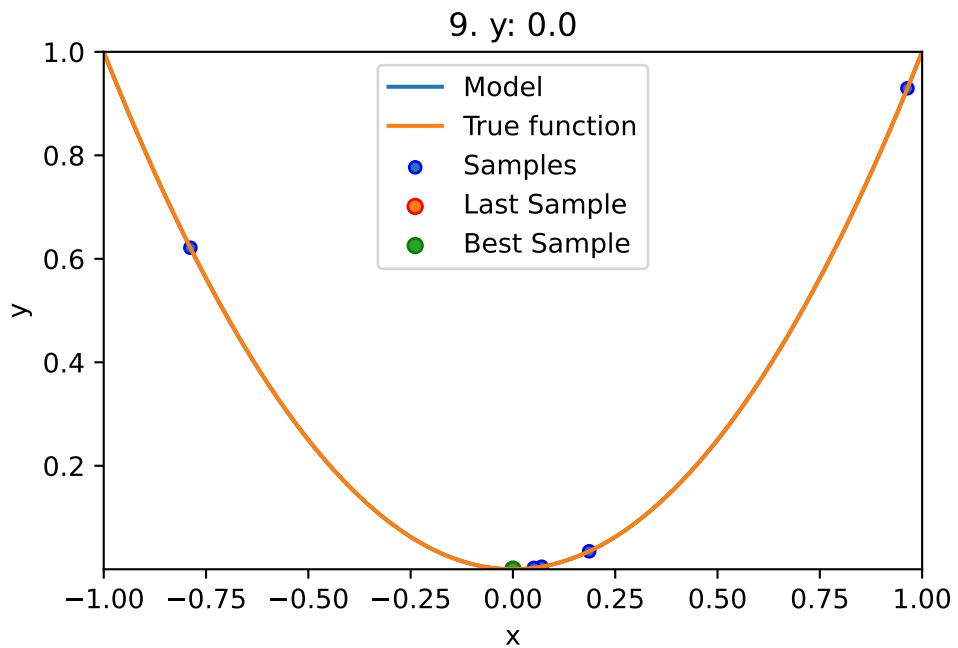
```
spotPython tuning: 5.912935436232656e-07 [#####----] 60.00%
```



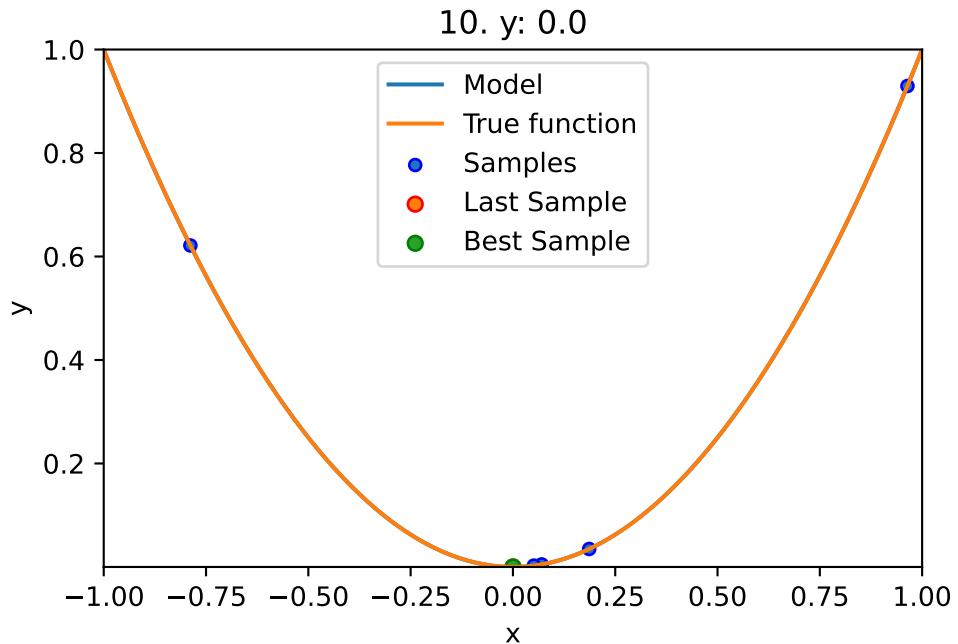
```
spotPython tuning: 4.211659449449057e-08 [#####---] 70.00%
```



```
spotPython tuning: 4.211659449449057e-08 [#####--] 80.00%
```



```
spotPython tuning: 1.6119389142446866e-09 [#####--] 90.00%
```



```
spotPython tuning: 1.6119389142446866e-09 [#####] 100.00% Done...
```

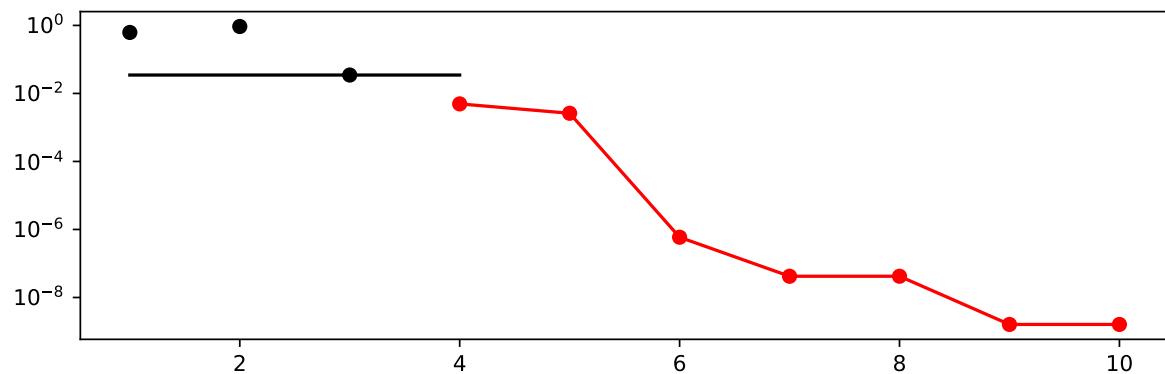
```
<spotPython.spot.spot.Spot at 0x309676f80>
```

```
spot_1_GP.print_results()
```

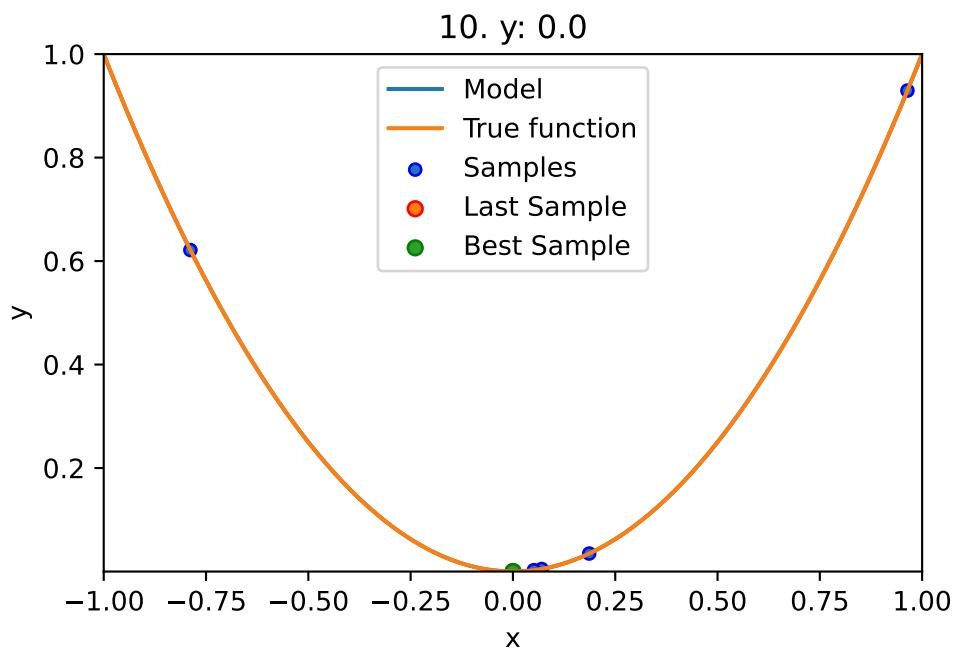
```
min y: 1.6119389142446866e-09
x0: 4.0148959068009304e-05
```

```
[['x0', 4.0148959068009304e-05]]
```

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



```
spot_1_GP.plot_model()
```



## 4.5 Exercises

### 4.5.1 DecisionTreeRegressor

- Describe the surrogate model.
- Use the surrogate as the model for optimization.

#### **4.5.2 RandomForestRegressor**

- Describe the surrogate model.
- Use the surrogate as the model for optimization.

#### **4.5.3 linear\_model.LinearRegression**

- Describe the surrogate model.
- Use the surrogate as the model for optimization.

#### **4.5.4 linear\_model.Ridge**

- Describe the surrogate model.
- Use the surrogate as the model for optimization.

### **4.6 Exercise 2**

- Compare the performance of the five different surrogates on both objective functions:
  - spotPython's internal Kriging
  - DecisionTreeRegressor
  - RandomForestRegressor
  - linear\_model.LinearRegression
  - linear\_model.Ridge

# 5 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
```

## 5.1 The Objective Function Branin

- The `spotPython` package provides several classes of objective functions.
- We will use an analytical objective function, i.e., a function that can be described by a (closed) formula.
- Here we will use the Branin function. The 2-dim Branin function is

$$y = a * (x_2 - b * x_1 ** 2 + c * x_1 - r) ** 2 + s * (1 - t) * \cos(x_1) + s,$$

where values of a, b, c, r, s and t are:  $a = 1$ ,  $b = 5.1/(4 * pi * 2)$ ,  $c = 5/pi$ ,  $r = 6$ ,  $s = 10$  and  $t = 1/(8 * pi)$ .

- It has three global minima:

$f(x) = 0.397887$  at  $(-\pi, 12.275)$ ,  $(\pi, 2.275)$ , and  $(9.42478, 2.475)$ .

- Input Domain: This function is usually evaluated on the square  $x_1$  in  $[-5, 10]$  x  $x_2$  in  $[0, 15]$ .

```
from spotPython.fun.objectivefunctions import analytical
lower = np.array([-5,-0])
upper = np.array([10,15])

fun = analytical(seed=123).fun_branin
```

## 5.2 The Optimizer

- Differential Evolution from the `scikit.optimize` package, see [https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential\\_evolution.html#scipy.optimize.differential\\_evolution](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 optimizers that are available in `spotPython`:
  - `dual_annealing`
  - `direct`
  - `shgo`
  - `basinhopping`, see <https://docs.scipy.org/doc/scipy/reference/optimize.html#global-optimization>.
- These can be selected as follows:
 

```
surrogate_control = "model_optimizer": differential_evolution
```
- 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](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 was introduced in Section [Section 1.5](#), 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.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "05_DE_"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)

fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))

```

05\_DE\_maans14\_2023-10-30\_23-07-33

```

spot_de = spot.Spot(fun=fun,
                     lower = lower,
                     upper = upper,
                     fun_evals = 20,
                     max_time = inf,
                     seed=125,
                     noise=False,
                     show_models= False,
                     design_control={"init_size": 10},
                     surrogate_control={"n_theta": len(lower),
                                        "model_optimizer": differential_evolution,
                                        "model_fun_evals": 1000,
                                        },
                     fun_control=fun_control)
spot_de.run()

```

spotPython tuning: 5.213735995388665 [#####----] 55.00%

spotPython tuning: 5.213735995388665 [#####----] 60.00%

spotPython tuning: 2.5179173635657266 [#####----] 65.00%

spotPython tuning: 1.016872525620073 [#####---] 70.00%

spotPython tuning: 0.4160579721446034 [#####--] 75.00%

```
spotPython tuning: 0.4096599475220657 [#####--] 80.00%
spotPython tuning: 0.4096599475220657 [#####--] 85.00%
spotPython tuning: 0.3999223417849329 [#####--] 90.00%
spotPython tuning: 0.39969164980122684 [#####] 95.00%
spotPython tuning: 0.39969164980122684 [#####] 100.00% Done...
<spotPython.spot.spot.Spot at 0x105796680>
```

### 5.2.1 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  $\theta$  of the Kriging surrogate is plotted against the number of optimization steps.

## 5.3 Print the Results

```
spot_de.print_results()

min y: 0.39969164980122684
x0: -3.158224446089584
x1: 12.293182279400076

[['x0', -3.158224446089584], ['x1', 12.293182279400076]]
```

## 5.4 Show the Progress

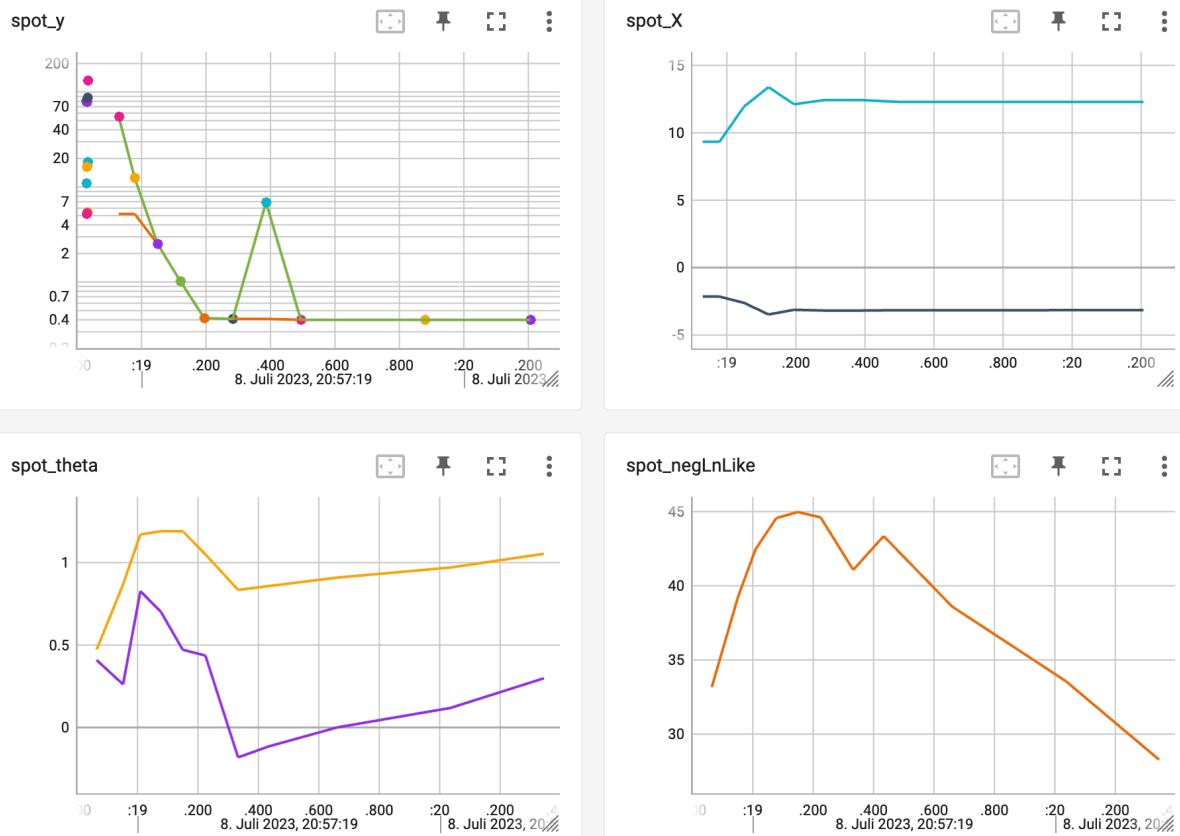
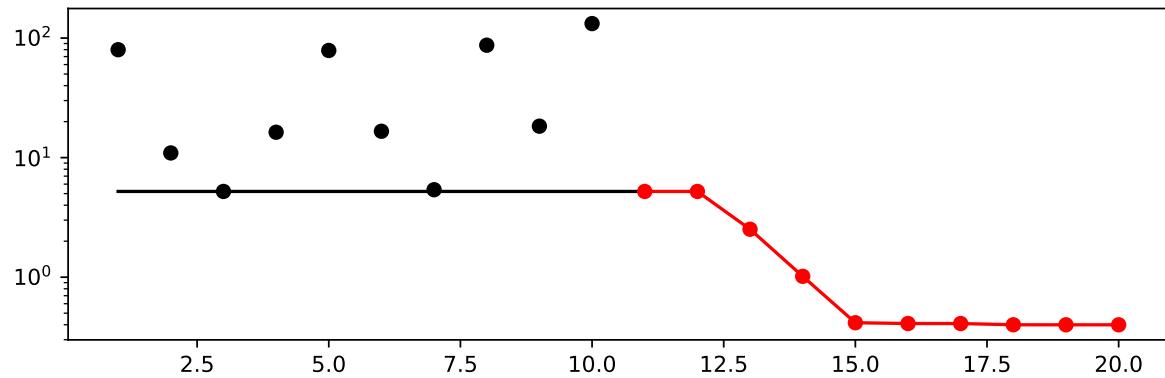
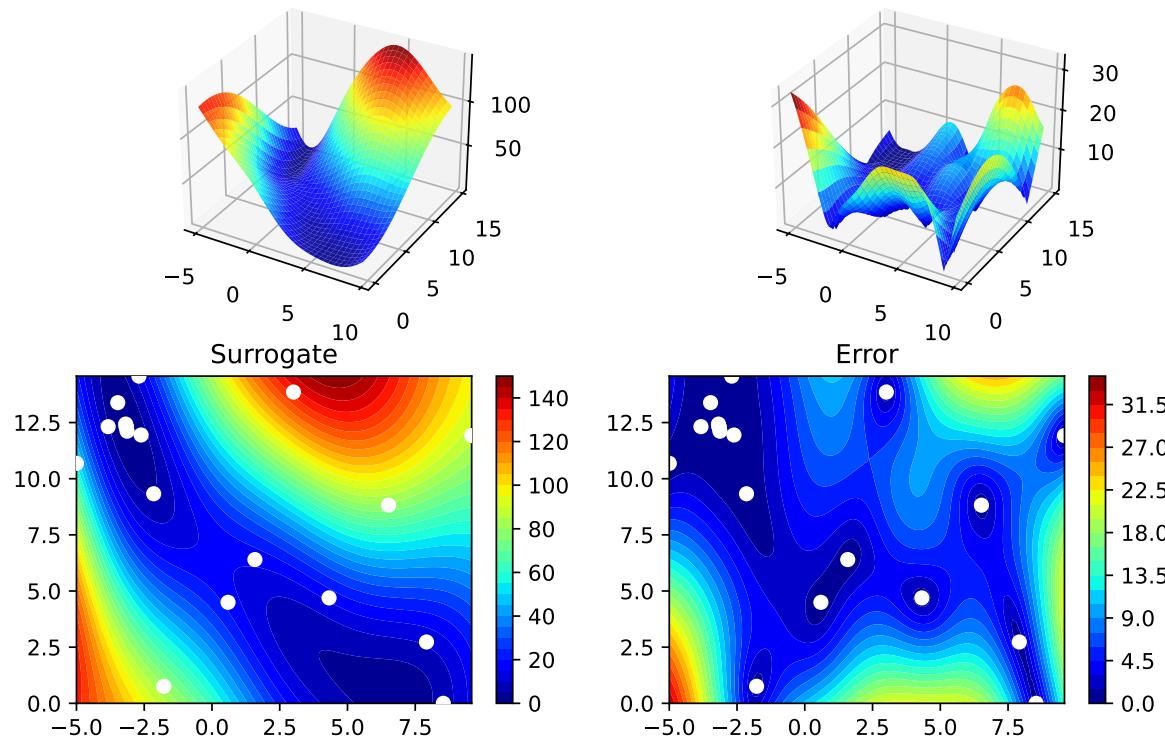


Figure 5.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

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



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



## 5.5 Exercises

### 5.5.1 `dual_annealing`

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

### 5.5.2 `direct`

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

### 5.5.3 `shgo`

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

### 5.5.4 `basinhopping`

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

### 5.5.5 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` change this behavior?

# 6 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
```

## 6.1 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](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.

### 6.1.1 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]
```

### 6.1.2 Building the Surrogate With Sklearn

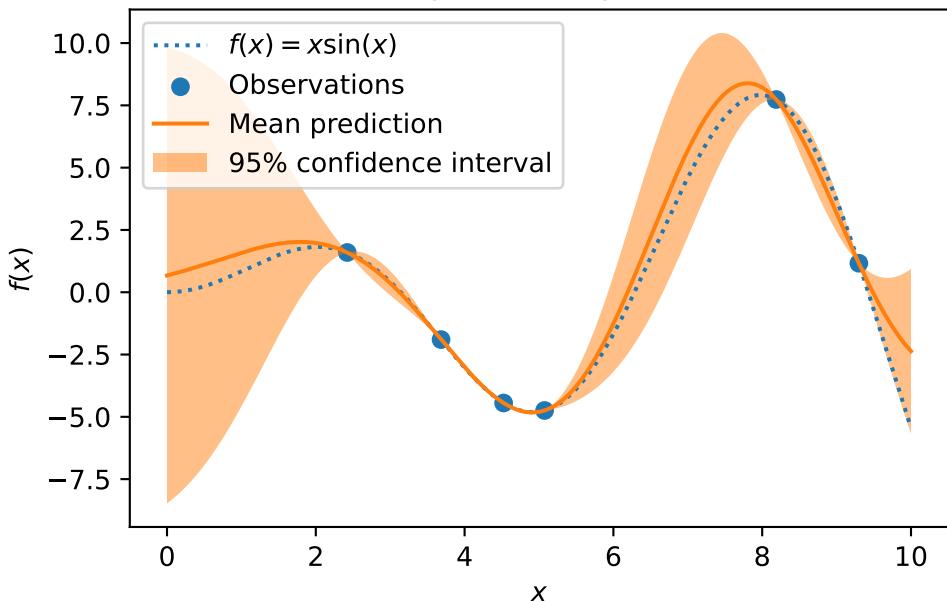
- The model building with `sklearn` consists of three steps:
  1. Instantiating the model, then
  2. fitting the model (using `fit`), and
  3. making predictions (using `predict`)

```
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)
```

### 6.1.3 Plotting the SklearnModel

```
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")
```

sk-learn Version: Gaussian process regression on noise-free dataset



#### 6.1.4 The spotPython Version

- The spotPython version is very similar:
  - Instantiating the model, then
  - fitting the model and
  - 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")

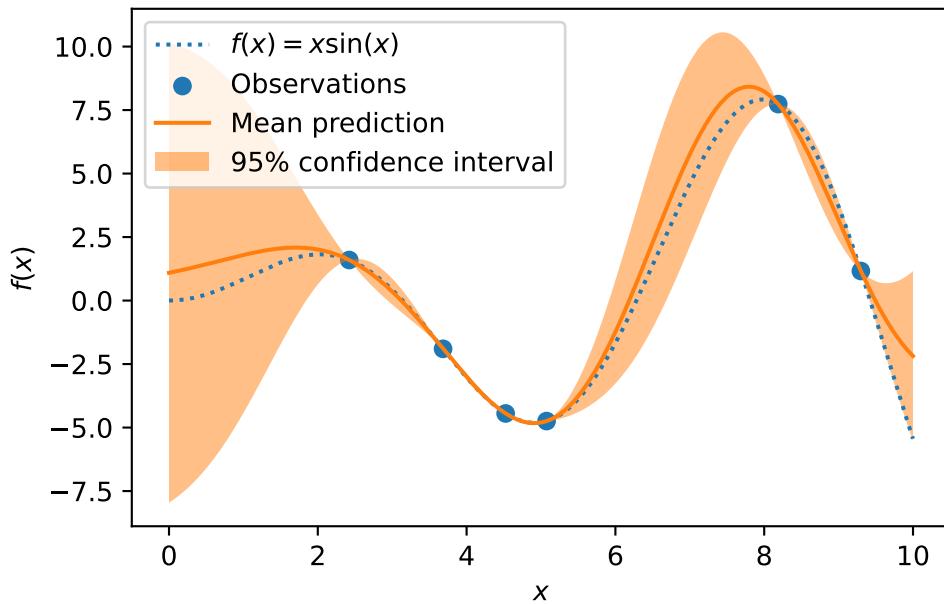
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, S_mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
    S_mean_prediction - 1.96 * S_std_prediction,
    S_mean_prediction + 1.96 * S_std_prediction,
    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")

```

spotPython Version: Gaussian process regression on noise-free dataset

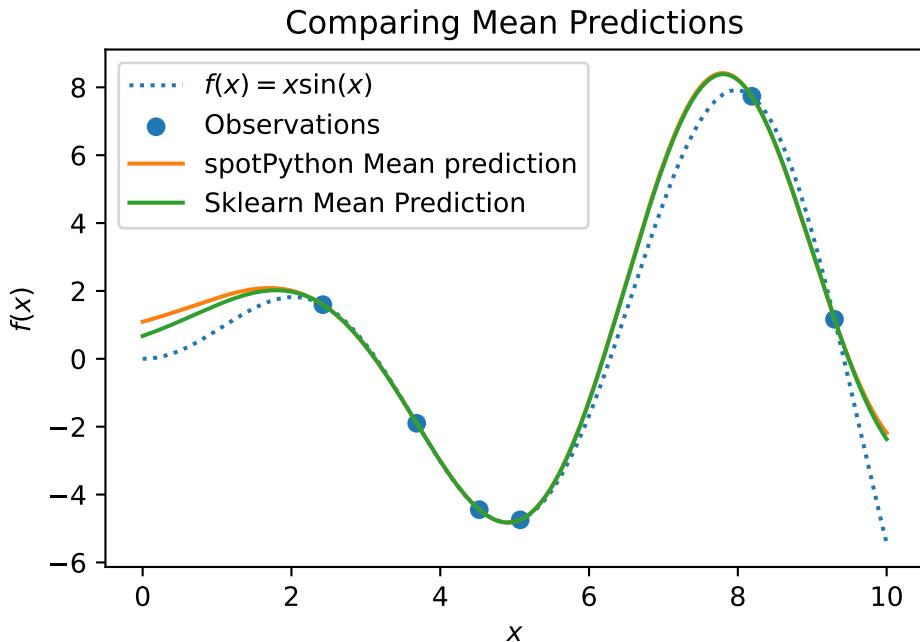


### 6.1.5 Visualizing the Differences Between the spotPython and the sklearn Model Fits

```

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")

```



## 6.2 Exercises

### 6.2.1 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.

### 6.2.2 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:

```
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 = {"sigma": 0.1,
                "seed": 123}
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 = {"sigma": 0.2}
```

### 6.2.3 `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 = {"sigma": 0.025,
                "seed": 123}
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 = {"sigma": 0.5}
```

#### 6.2.4 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)
lower = np.array([-10])
upper = np.array([10])
fun = analytical().fun_cubed
fun_control = {"sigma": 0.025,
               "seed": 123}
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 = {"sigma": 0.05}
```

#### 6.2.5 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?

# 7 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.

## 7.1 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
import matplotlib.pyplot as plt
```

### 7.1.1 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
```

```
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.

#### i TensorBoard

Similar to the one-dimensional case, which was introduced in Section Section 1.5, 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.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "07_Y"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)

fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=0,
    seed=123,)
```

07\_Y\_maans14\_2023-10-30\_23-08-14

```

spot_1 = spot.Spot(fun=fun,
                    fun_evals = 25,
                    lower = np.array([-1]),
                    upper = np.array([1]),
                    design_control={"init_size": 10},
                    tolerance_x = np.sqrt(np.spacing(1)),
                    fun_control = fun_control,)

spot_1.run()
```

spotPython tuning: 1.2459257396367542e-08 [#####----] 44.00%

spotPython tuning: 1.2459257396367542e-08 [#####----] 48.00%

spotPython tuning: 1.2459257396367542e-08 [#####----] 52.00%

spotPython tuning: 1.2459257396367542e-08 [#####----] 56.00%

spotPython tuning: 4.897545259852824e-10 [#####----] 60.00%

spotPython tuning: 4.897545259852824e-10 [#####----] 64.00%

spotPython tuning: 4.897545259852824e-10 [#####---] 68.00%

```

spotPython tuning: 4.897545259852824e-10 [#####---] 72.00%
spotPython tuning: 4.897545259852824e-10 [#####---] 76.00%
spotPython tuning: 4.897545259852824e-10 [#####---] 80.00%
spotPython tuning: 1.9335518024989866e-10 [#####---] 84.00%
spotPython tuning: 1.9335518024989866e-10 [#####---] 88.00%
spotPython tuning: 1.9335518024989866e-10 [#####---] 92.00%
spotPython tuning: 1.9335518024989866e-10 [#####---] 96.00%
spotPython tuning: 2.135607331180881e-12 [#####---] 100.00% Done...
<spotPython.spot.spot.Spot at 0x1079ba230>

```

### 7.1.2 Results

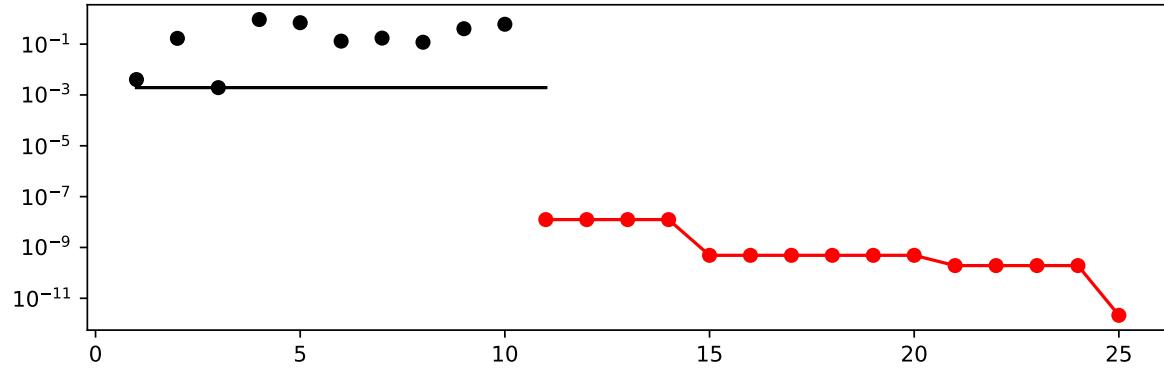
```

spot_1.print_results()

min y: 2.135607331180881e-12
x0: -1.4613717292943917e-06

[['x0', -1.4613717292943917e-06]]
```

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



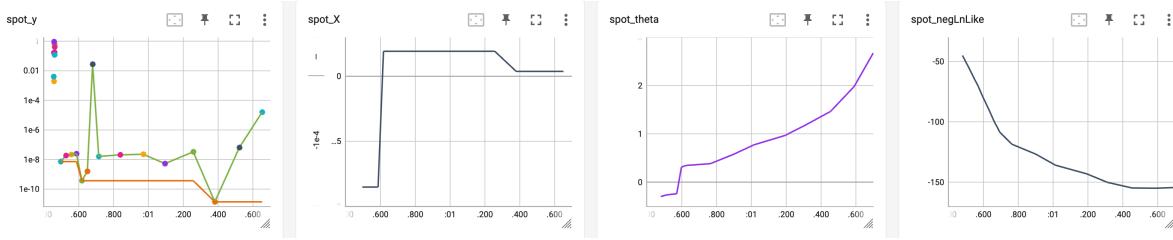


Figure 7.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

## 7.2 Same, but with EI as infill\_criterion

```
PREFIX = "07_EI_ISO"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=0,
    seed=123,)
```

07\_EI\_ISO\_maans14\_2023-10-30\_23-08-16

```
spot_1_ei = spot.Spot(fun=fun,
                      lower = np.array([-1]),
                      upper = np.array([1]),
                      fun_evals = 25,
                      tolerance_x = np.sqrt(np.spacing(1)),
                      infill_criterion = "ei",
                      design_control={"init_size": 10},
                      fun_control = fun_control,)

spot_1_ei.run()
```

spotPython tuning: 8.79000773789907e-08 [#####-----] 44.00%

spotPython tuning: 2.6197300077861015e-08 [#####-----] 48.00%

spotPython tuning: 2.6197300077861015e-08 [#####-----] 52.00%

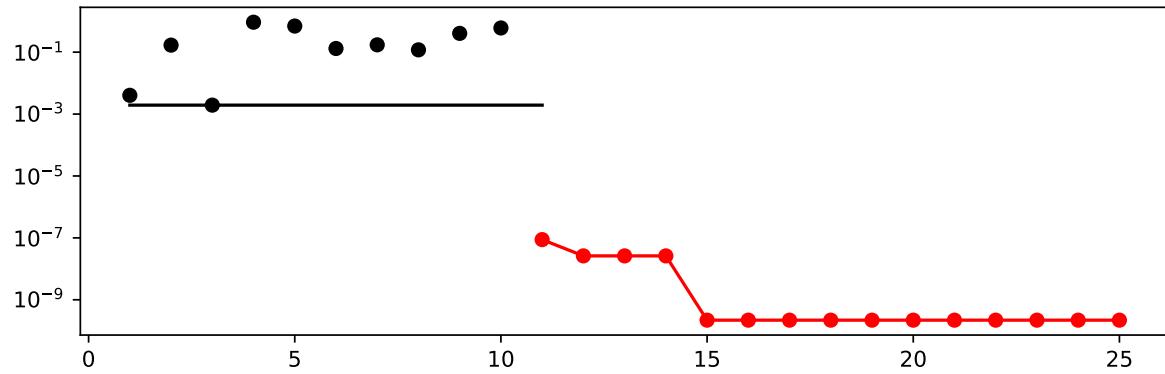
```

spotPython tuning: 2.6197300077861015e-08 [#####----] 56.00%
spotPython tuning: 2.1963022660037201e-10 [#####----] 60.00%
spotPython tuning: 2.1963022660037201e-10 [#####----] 64.00%
spotPython tuning: 2.1963022660037201e-10 [#####----] 68.00%
spotPython tuning: 2.1963022660037201e-10 [#####----] 72.00%
spotPython tuning: 2.1963022660037201e-10 [#####---] 76.00%
spotPython tuning: 2.1963022660037201e-10 [#####---] 80.00%
spotPython tuning: 2.1963022660037201e-10 [#####---] 84.00%
spotPython tuning: 2.1963022660037201e-10 [#####---] 88.00%
spotPython tuning: 2.1963022660037201e-10 [#####---] 92.00%
spotPython tuning: 2.1963022660037201e-10 [#####---] 96.00%
spotPython tuning: 2.1963022660037201e-10 [#####---] 100.00% Done...

```

<spotPython.spot.spot.Spot at 0x326722c20>

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



```

spot_1_ei.print_results()

min y: 2.1963022660037201e-10
x0: 1.4819926673245452e-05

[['x0', 1.4819926673245452e-05]]

```

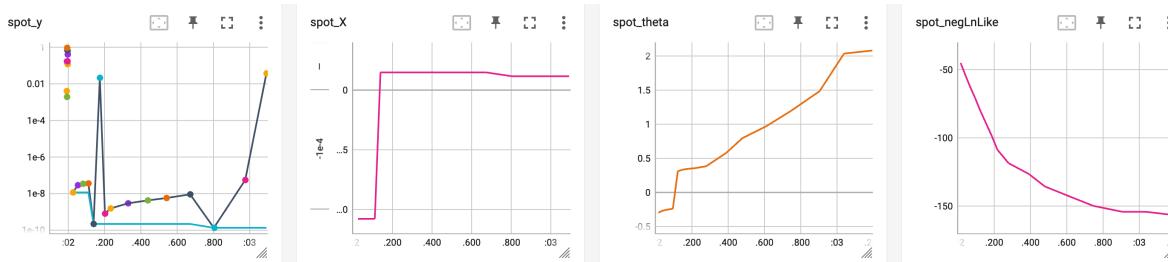


Figure 7.2: TensorBoard visualization of the spotPython optimization process and the surrogate model. Expected improvement, isotropic Kriging.

### 7.3 Non-isotropic Kriging

```

PREFIX = "07_EI_NONISO"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=0,
    seed=123,)

```

07\_EI\_NONISO\_maans14\_2023-10-30\_23-08-18

```

spot_2_ei_noniso = spot.Spot(fun=fun,
    lower = np.array([-1, -1]),
    upper = np.array([1, 1]),
    fun_evals = 25,
    tolerance_x = np.sqrt(np.spacing(1)),
    infill_criterion = "ei",
    show_models=True,

```

```
    design_control={"init_size": 10},
    surrogate_control={"noise": False,
                      "cod_type": "norm",
                      "min_theta": -4,
                      "max_theta": 3,
                      "n_theta": 2,
                      "model_fun_evals": 1000,
                      },
    fun_control=fun_control,)

spot_2_ei_noniso.run()
```

```
spotPython tuning: 1.8247169797759505e-05 [#####-----] 44.00%
```

```
spotPython tuning: 1.8247169797759505e-05 [#####-----] 48.00%
```

```
spotPython tuning: 1.8247169797759505e-05 [#####-----] 52.00%
```

```
spotPython tuning: 1.0281222147432436e-05 [#####-----] 56.00%
```

```
spotPython tuning: 1.0281222147432436e-05 [#####-----] 60.00%
```

```
spotPython tuning: 1.088759927339735e-07 [#####-----] 64.00%
```

```
spotPython tuning: 1.088759927339735e-07 [#####-----] 68.00%
```

```
spotPython tuning: 1.088759927339735e-07 [#####-----] 72.00%
```

```
spotPython tuning: 1.088759927339735e-07 [#####-----] 76.00%
```

```
spotPython tuning: 1.088759927339735e-07 [#####-----] 80.00%
```

```
spotPython tuning: 1.088759927339735e-07 [#####-----] 84.00%
```

```
spotPython tuning: 1.088759927339735e-07 [#####-----] 88.00%
```

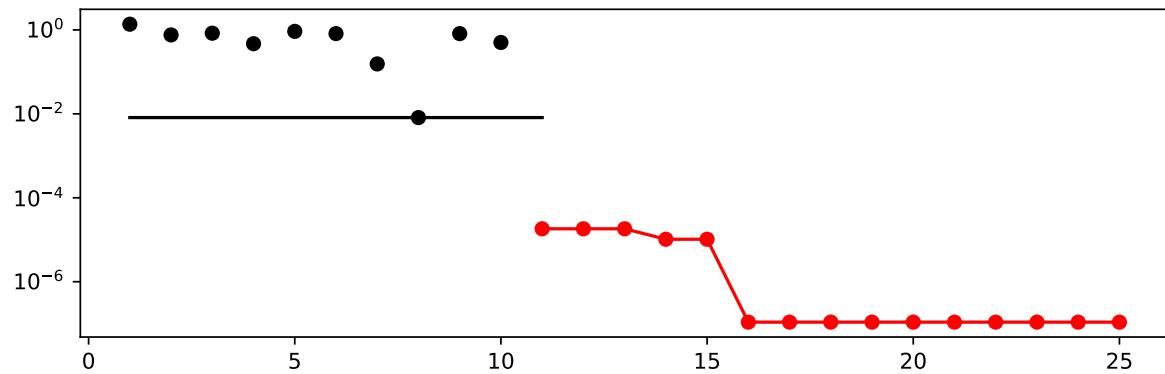
```
spotPython tuning: 1.088759927339735e-07 [#####-----] 92.00%
```

```
spotPython tuning: 1.088759927339735e-07 [#####] 96.00%
```

```
spotPython tuning: 1.088759927339735e-07 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x326e79330>
```

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



```
spot_2_ei_noniso.print_results()
```

```
min y: 1.088759927339735e-07
x0: -0.0002833471276146305
x1: 0.00016908695398081962
```

```
[['x0', -0.0002833471276146305], ['x1', 0.00016908695398081962]]
```

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

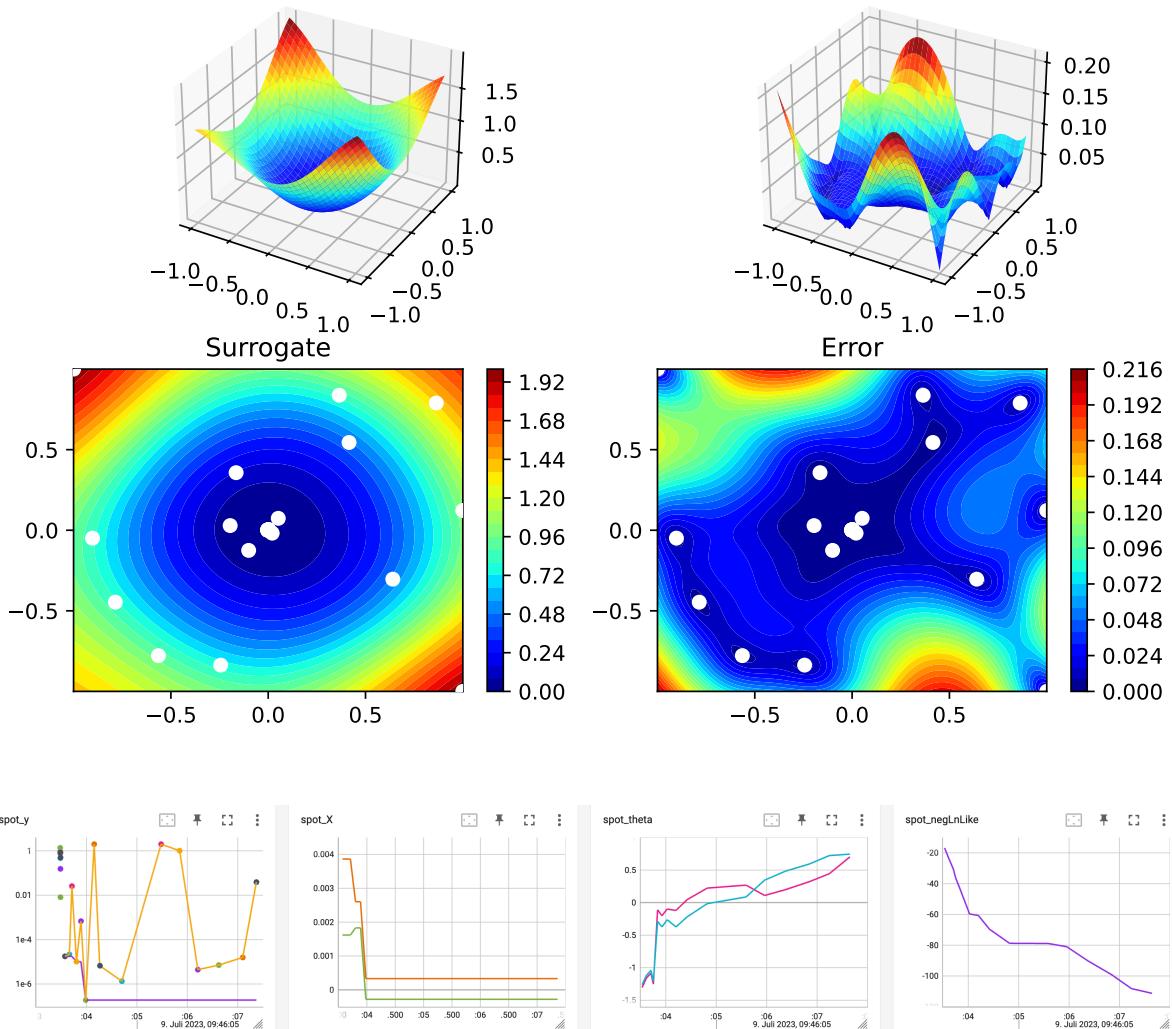


Figure 7.3: TensorBoard visualization of the spotPython optimization process and the surrogate model. Expected improvement, isotropic Kriging.

## 7.4 Using sklearn Surrogates

### 7.4.1 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:

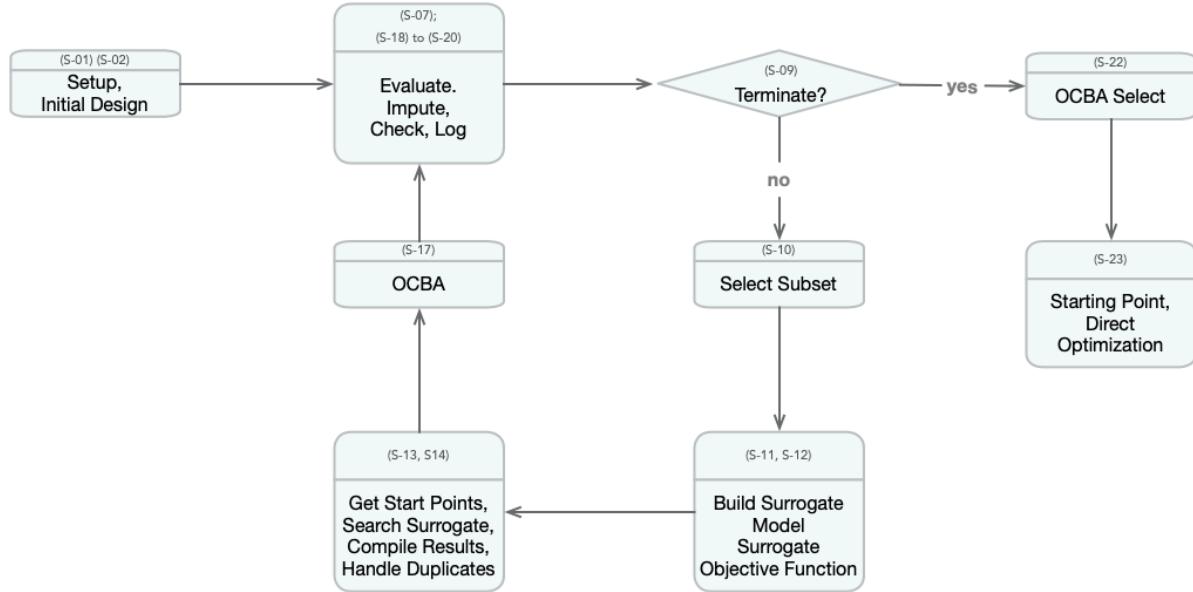


Figure 7.4: Visual representation of the model based search with SPOT. Taken from: Bartz-Beielstein, T., and Zaefferer, M. Hyperparameter tuning approaches. In Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide, E. Bartz, T. Bartz-Beielstein, M. Zaefferer, and O. Mersmann, Eds. Springer, 2022, ch. 4, pp. 67–114.

## 7.4.2 spot: The Initial Model

### 7.4.2.1 Example: Modifying the initial design size

This is the “Example: Modifying the initial design size” from Chapter 4.5.1 in [bart21i].

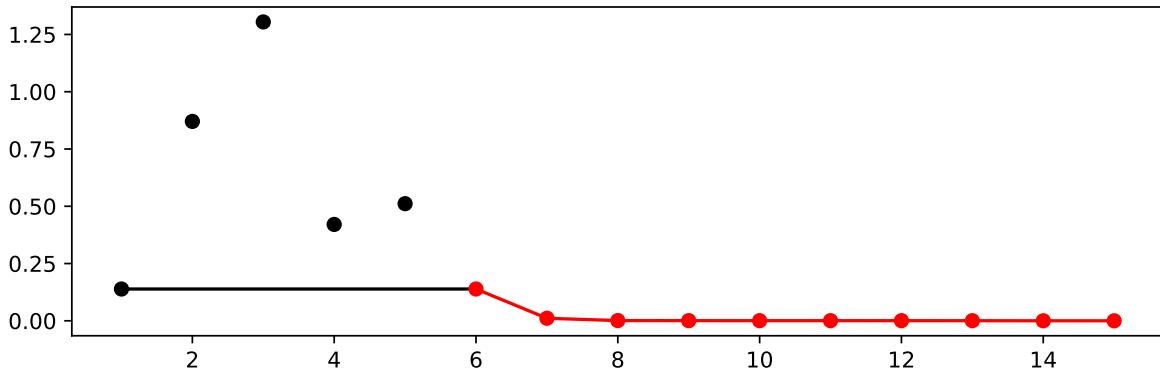
```

spot_ei = spot.Spot(fun=fun,
                    lower = np.array([-1,-1]),
                    upper= np.array([1,1]),
                    design_control={"init_size": 5})
  
```

```
spot_ei.run()

spotPython tuning: 0.13881986540743513 [#####-----] 40.00%
spotPython tuning: 0.011157100173301121 [#####-----] 46.67%
spotPython tuning: 0.0010077722891862157 [#####-----] 53.33%
spotPython tuning: 0.0006326308401677749 [#####-----] 60.00%
spotPython tuning: 0.0005880000745278913 [#####-----] 66.67%
spotPython tuning: 0.0005853974252148365 [#####-----] 73.33%
spotPython tuning: 0.0005615353015376504 [#####-----] 80.00%
spotPython tuning: 0.0004470375728318479 [#####-----] 86.67%
spotPython tuning: 6.506371306758665e-05 [#####-----] 93.33%
spotPython tuning: 1.881581967484049e-05 [#####-----] 100.00% Done...
<spotPython.spot.spot.Spot at 0x3274efd60>
```

```
spot_ei.plot_progress()
```



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

```
(2.135607331180881e-12, 1.881581967484049e-05)
```

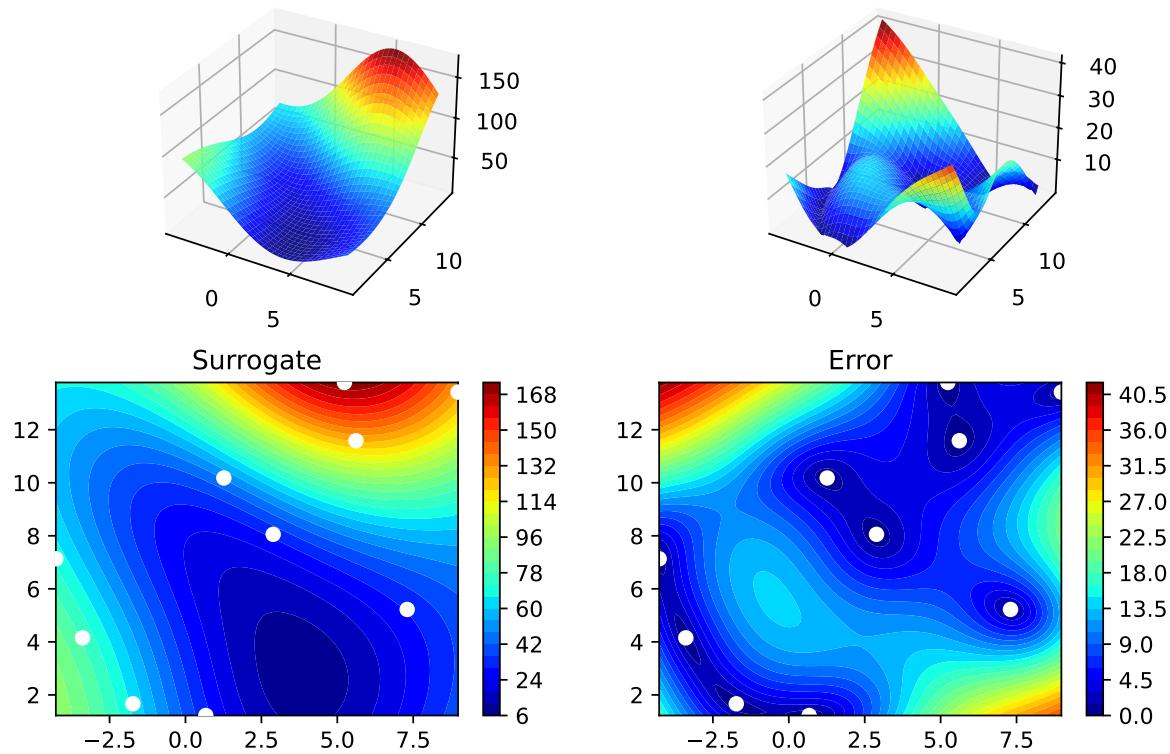
#### 7.4.3 Init: Build Initial Design

```
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 ]]))
```

#### 7.4.4 Evaluate

#### 7.4.5 Build Surrogate

#### 7.4.6 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  $f$ .

### 7.5 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.

```

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]

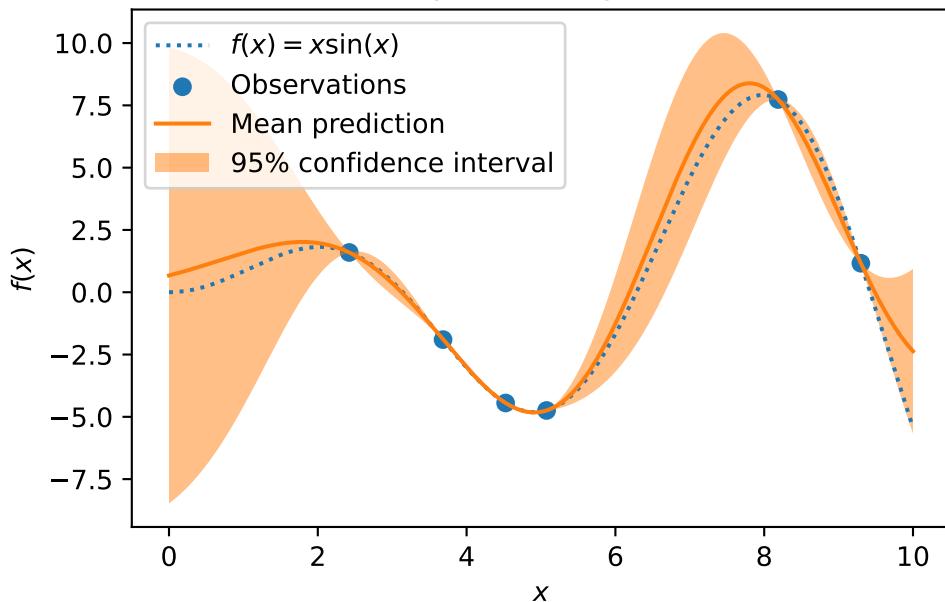
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
gaussian_process = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
gaussian_process.fit(X_train, y_train)
gaussian_process.kernel_

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")

```

## 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

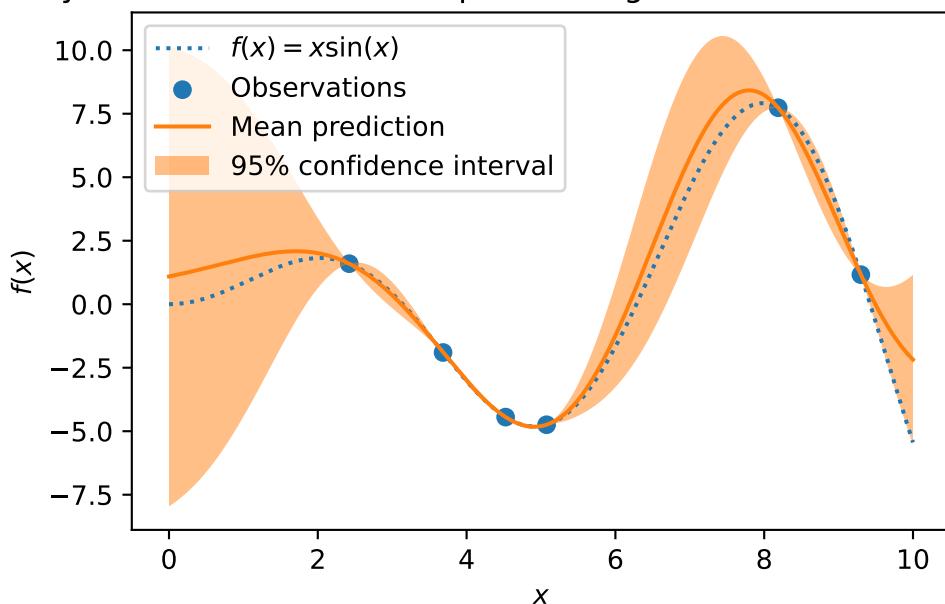
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
plt.fill_between(
```

```

        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")

```

spotPython Version: Gaussian process regression on noise-free dataset



## 7.6 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
lower = np.array([-5,-0])
upper = np.array([10,15])
design_control={"init_size": 5}
surrogate_control={
    "infill_criterion": None,
    "n_points": 1,
}
spot_GP = spot.Spot(fun=fun, lower = lower, upper= upper, surrogate=S,
                     fun_evals = 15, noise = False, log_level = 50,
                     design_control=design_control,
                     surrogate_control=surrogate_control)

```

```
spot_GP.run()
```

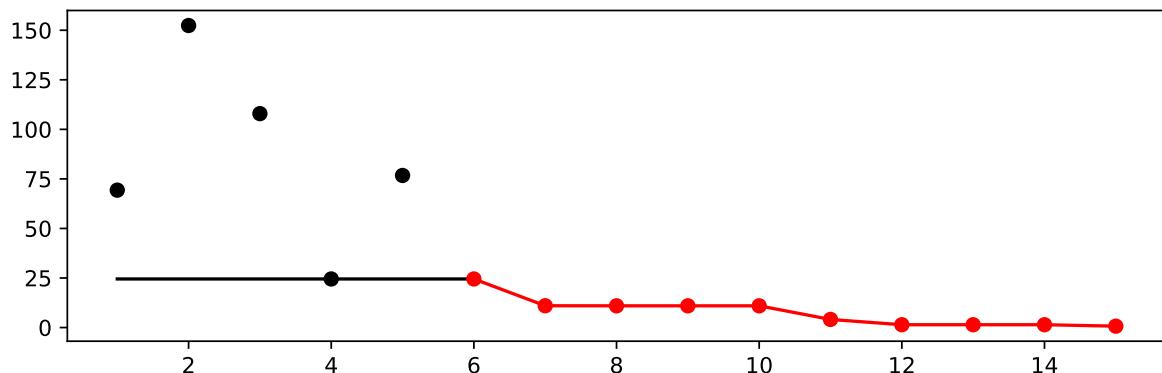
```
spotPython tuning: 24.51465459019188 [#####-----] 40.00%
spotPython tuning: 11.003078163486554 [#####-----] 46.67%
spotPython tuning: 10.960665185123245 [#####-----] 53.33%
spotPython tuning: 10.960665185123245 [#####----] 60.00%
spotPython tuning: 10.960665185123245 [#####---] 66.67%
spotPython tuning: 4.0894841491438765 [#####---] 73.33%
spotPython tuning: 1.4230377508791392 [#####--] 80.00%
spotPython tuning: 1.4230377508791392 [#####-] 86.67%
spotPython tuning: 1.4230377508791392 [#####-] 93.33%
spotPython tuning: 0.6989341031319167 [#####-] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x32a7d7370>
```

```
spot_GP.y
```

```
array([ 69.32459936, 152.38491454, 107.92560483, 24.51465459,
       76.73500031, 86.304256 , 11.00307816, 10.96066519,
      16.06668258, 24.08432082, 4.08948415, 1.42303775,
      1.47359526, 16.04703294, 0.6989341 ])
```

```
spot_GP.plot_progress()
```



```
spot_GP.print_results()
```

```
min y: 0.6989341031319167
x0: 3.358292789592623
x1: 2.3886120108545597
```

```
[['x0', 3.358292789592623], ['x1', 2.3886120108545597]]
```

## 7.7 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
lower = np.array([-1,-1])
upper = np.array([1,1])

design_control={"init_size": 10}
surrogate_control={
    "n_points": 1,
}
spot_S_K = spot.Spot(fun=fun,
                      lower = lower,
                      upper= upper,
                      surrogate=S_K,
                      fun_evals = 25,
                      noise = False,
                      log_level = 50,
                      design_control=design_control,
                      surrogate_control=surrogate_control)

spot_S_K.run()

```

spotPython tuning: 2.1370719642847402e-05 [#####----] 44.00%

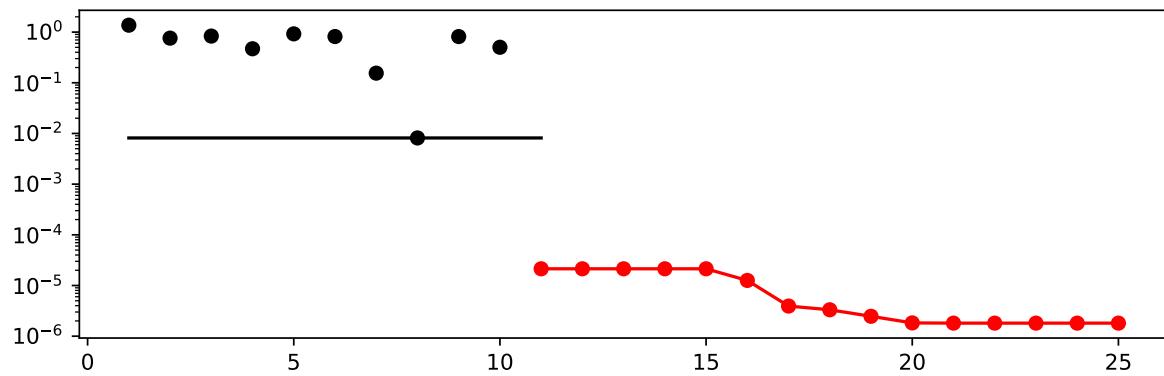
spotPython tuning: 2.1370719642847402e-05 [#####----] 48.00%

spotPython tuning: 2.1370719642847402e-05 [#####----] 52.00%

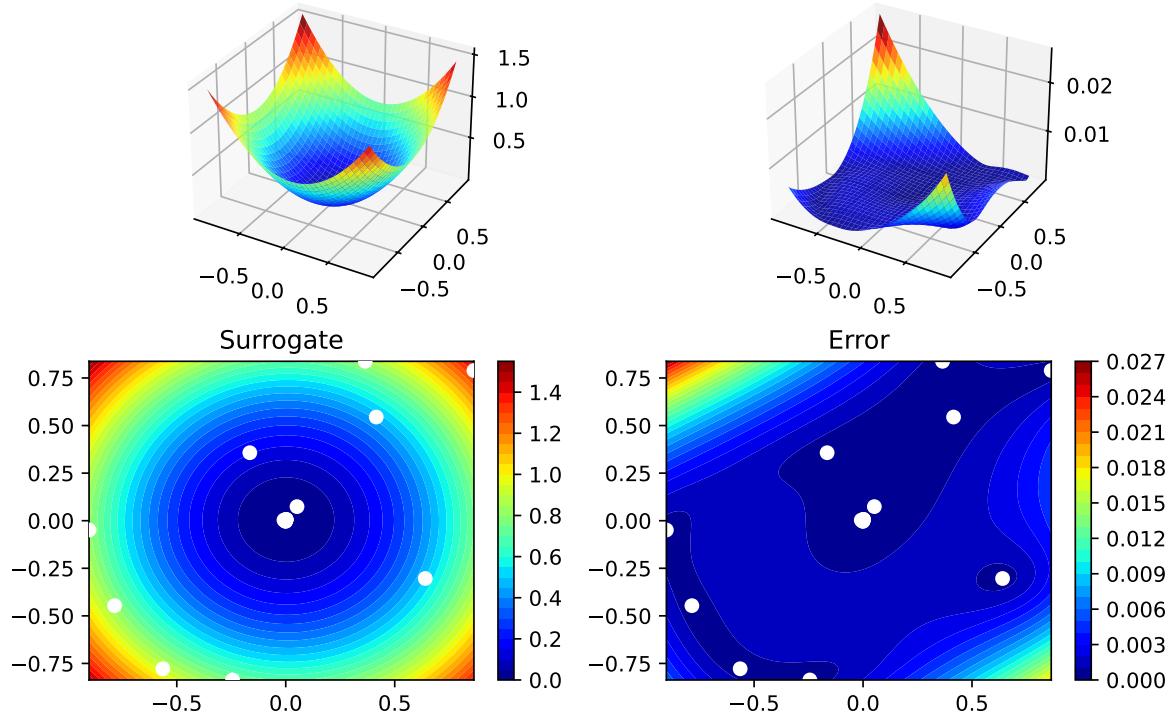
spotPython tuning: 2.1370719642847402e-05 [#####----] 56.00%

```
spotPython tuning: 2.1370719642847402e-05 [#####----] 60.00%
spotPython tuning: 1.2590483826517302e-05 [#####----] 64.00%
spotPython tuning: 3.930538349742746e-06 [#####---] 68.00%
spotPython tuning: 3.3191760809461184e-06 [#####---] 72.00%
spotPython tuning: 2.4684282727935e-06 [#####---] 76.00%
spotPython tuning: 1.8279736801432919e-06 [#####---] 80.00%
spotPython tuning: 1.809224307539433e-06 [#####---] 84.00%
spotPython tuning: 1.809224307539433e-06 [#####---] 88.00%
spotPython tuning: 1.809224307539433e-06 [#####---] 92.00%
spotPython tuning: 1.809224307539433e-06 [#####---] 96.00%
spotPython tuning: 1.809224307539433e-06 [#####---] 100.00% Done...
<spotPython.spot.spot.Spot at 0x32a780df0>
```

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



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



```
spot_S_K.print_results()
```

```
min y: 1.809224307539433e-06
x0: -0.001330101474082372
x1: 0.0002001358942901893
```

```
[['x0', -0.001330101474082372], ['x1', 0.0002001358942901893]]
```

### 7.7.1 Optimize on Surrogate

### 7.7.2 Evaluate on Real Objective

### 7.7.3 Impute / Infill new Points

## 7.8 Tests

```
import numpy as np
from spotPython.spot import spot
from spotPython.fun.objectivefunctions import analytical

fun_sphere = analytical().fun_sphere
spot_1 = spot.Spot(
    fun=fun_sphere,
    lower=np.array([-1, -1]),
    upper=np.array([1, 1]),
    n_points = 2
)

# (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.surrogate.fit(spot_1.X, spot_1.y)
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   ]
[[0.00151305  0.00405727]
 [0.00151305  0.00405727]]

```

## 7.9 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="nor")
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")

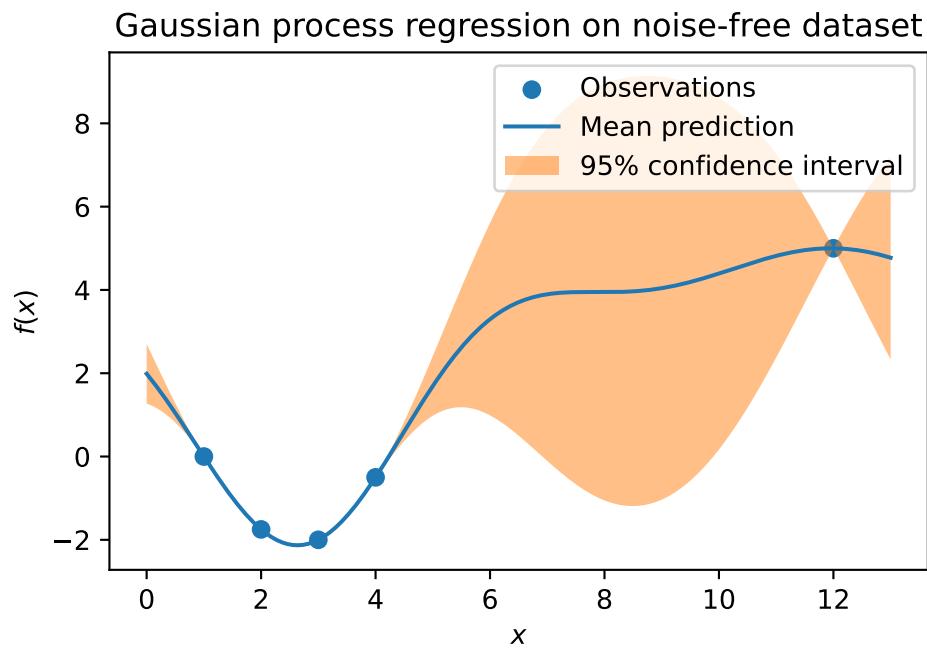
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
if True:
    plt.fill_between(
        X.ravel(),
        mean_prediction - 2 * std_prediction,
        mean_prediction + 2 * std_prediction,
        alpha=0.5,
        label=r"95% confidence interval",
    )
plt.legend()
plt.xlabel("$x$")

```

```

plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")

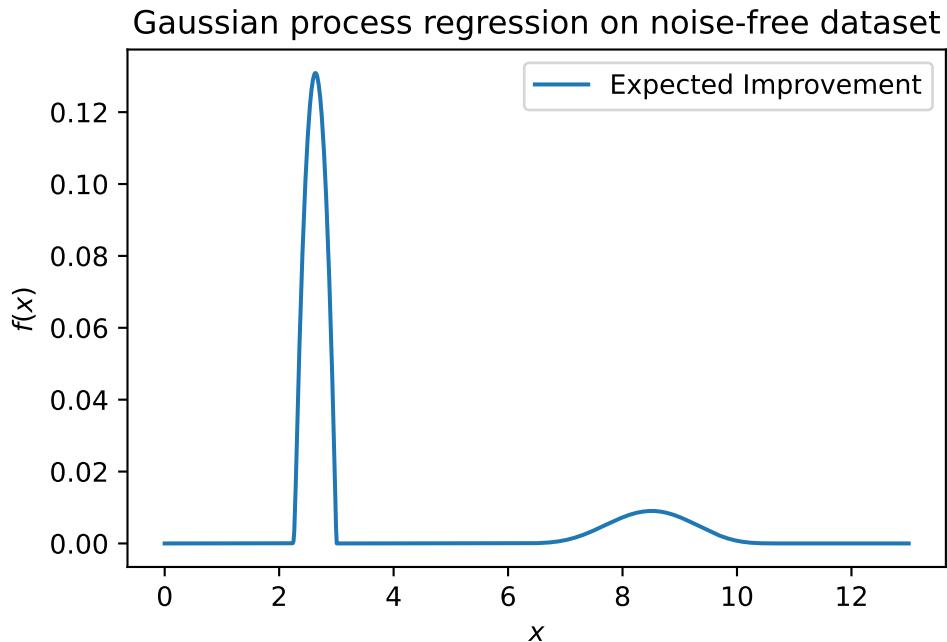
```



```

# 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([1.09275997]),
 'p': [],
 'Lambda': []}
```

## 7.10 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(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=1.0,
    seed=123,)
y_train = fun(X_train, fun_control=fun_control)

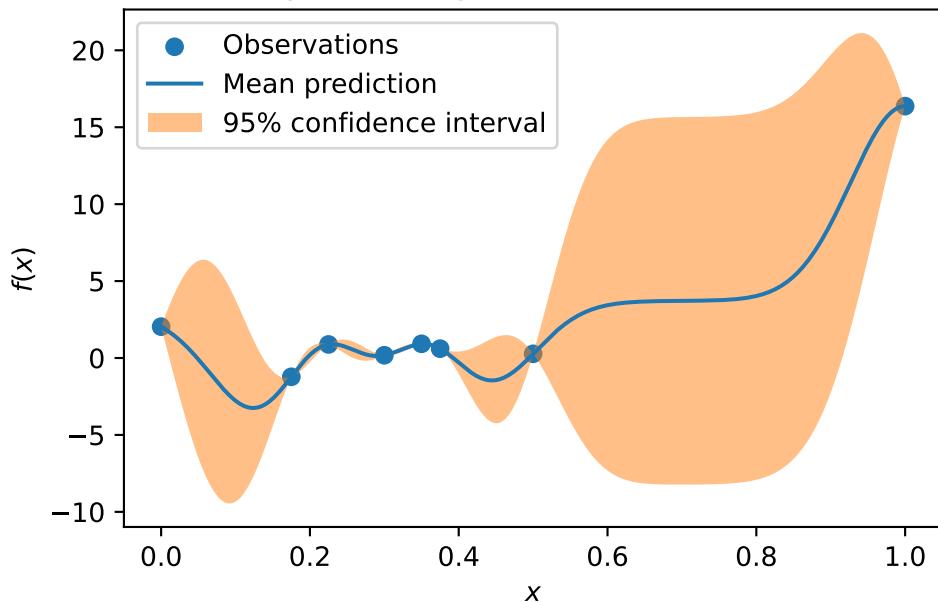
S = Kriging(name='kriging', seed=123, log_level=50, n_theta=1, noise=False, cod_type="nor"
S.fit(X_train, y_train)

X = np.linspace(start=0, stop=1, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X, return_val="all")

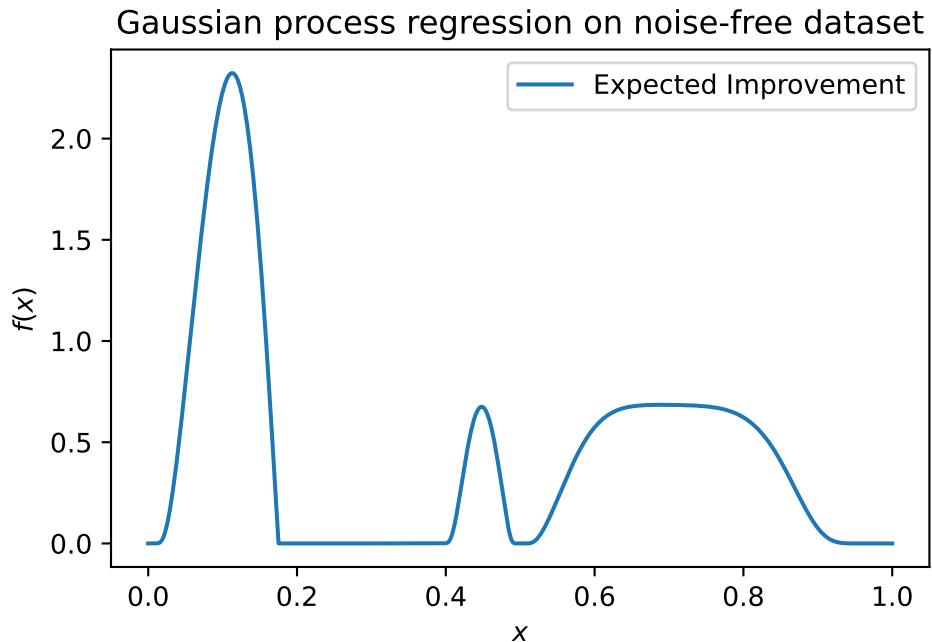
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
if True:
    plt.fill_between(
        X.ravel(),
        mean_prediction - 2 * std_prediction,
        mean_prediction + 2 * std_prediction,
        alpha=0.5,
        label=r"95% confidence interval",
    )
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")

```

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")
```



## 7.11 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(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    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

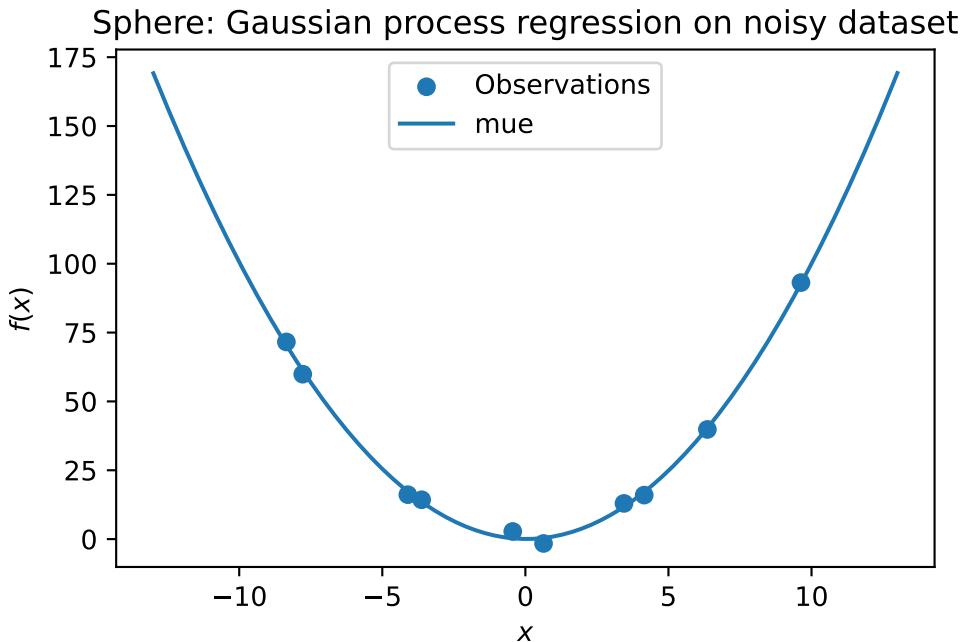
S = Kriging(name='kriging',
             seed=123,
             log_level=50,
             n_theta=1,
             noise=False)
S.fit(X_train, y_train)

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")

[[ 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([25.26601608]),
  'theta': array([-1.98024606]),
  '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")

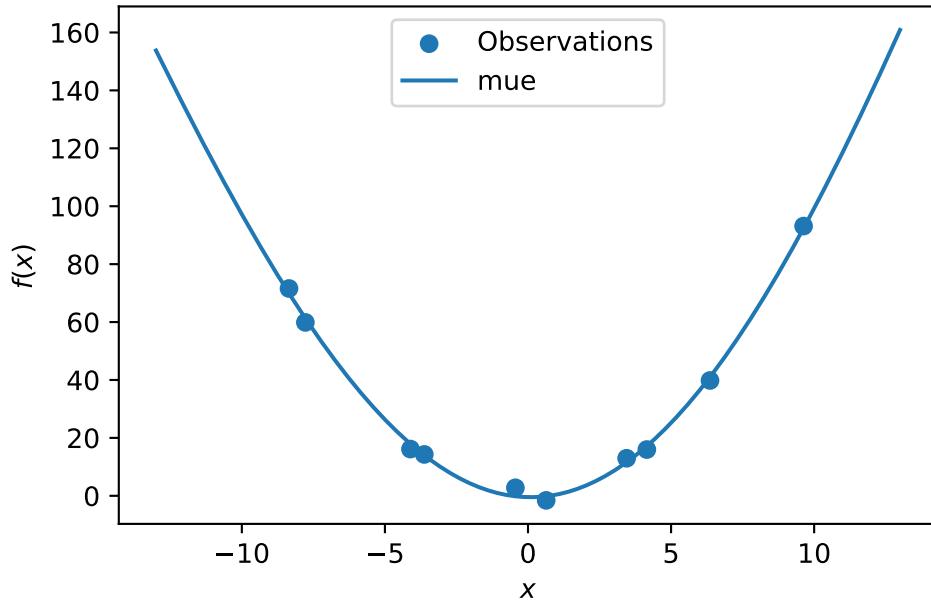
# plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
# plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mle")
```

```

plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression with nugget on noisy dataset")

```

Sphere: Gaussian process regression with nugget on noisy dataset



S.log

```

{'negLnLike': array([21.82530943]),
 'theta': array([-0.41935831]),
 'p': [],
 'Lambda': array([5.20850895e-05])}

```

## 7.12 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(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    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")

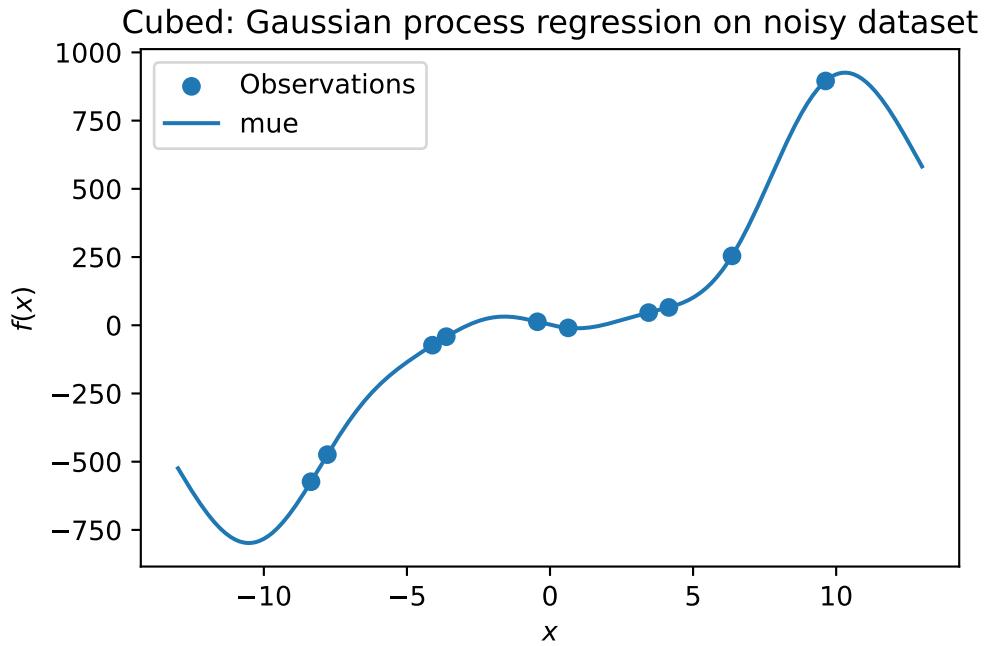
```

```

[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]]

```

```
[ 3.4468512 ]
[ 6.36049088]
[-7.77978539]
[ -9.63480707 -72.98497325   12.7936499   895.34567477 -573.35961837
-41.83176425   65.27989461   46.37081417   254.1530734  -474.09587355]
```

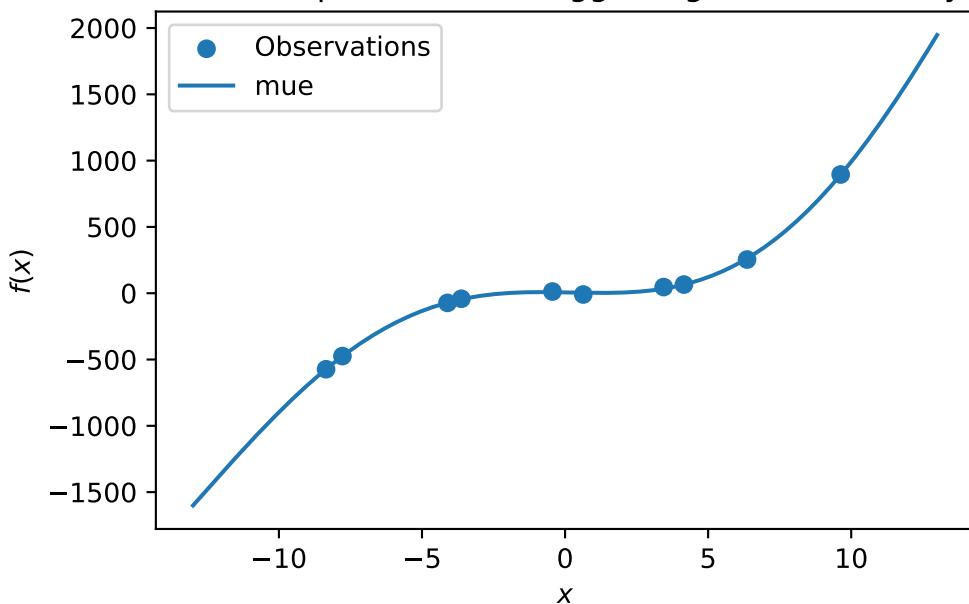


```
S = Kriging(name='kriging', seed=123, log_level=0, n_theta=1, noise=True)
S.fit(X_train, y_train)

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")
```

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(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    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)

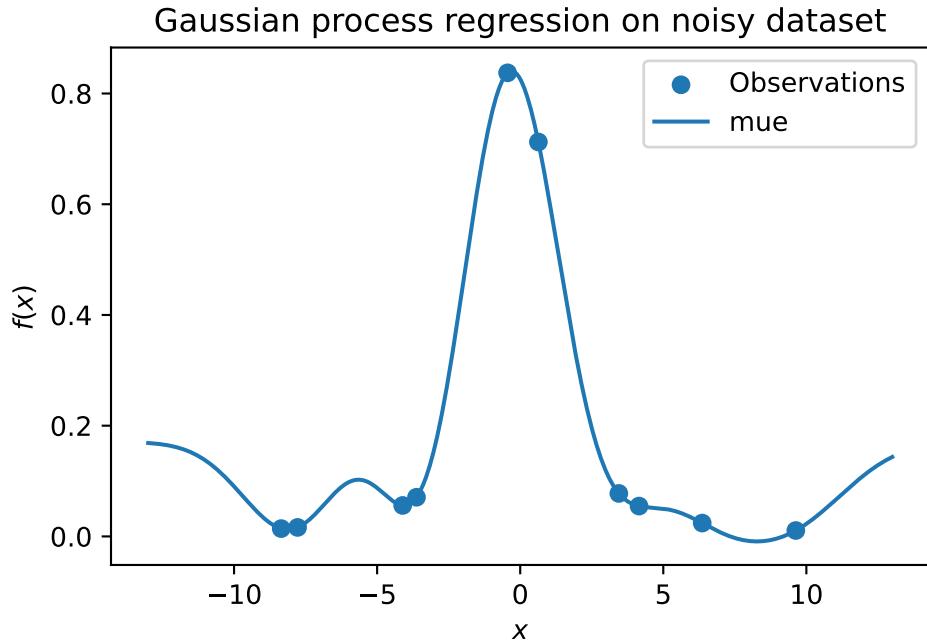
X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mu")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noisy dataset")

```

```

[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]
 [ 3.4468512 ]
 [ 6.36049088]
 [-7.77978539]]
[0.712453  0.05595118 0.83735691 0.0106654  0.01413372 0.07074765
 0.05479457 0.07763503 0.02412205 0.01625354]
```

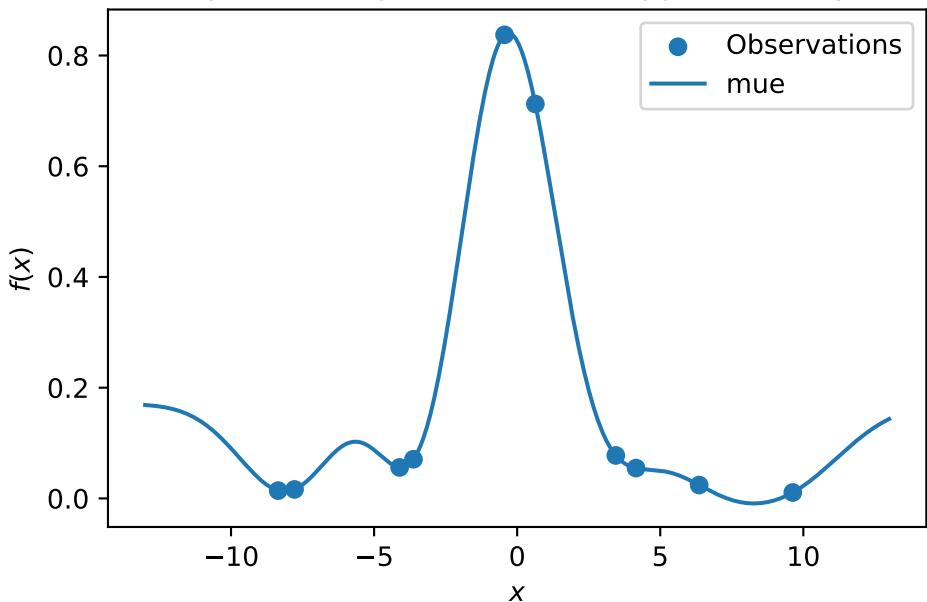


```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=True)
S.fit(X_train, y_train)

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="mle")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression with nugget on noisy dataset")
```

Gaussian process regression with nugget on noisy dataset



## 7.13 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=["nu"])
S.fit(X, y)
Sf = Kriging(name='kriging', seed=123, log_level=50, n_theta=3, noise=False, var_type=["nu"])
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

```

-40.513457642582125

```
# vars(S)
```

```
# vars(Sf)
```

# 8 Hyperparameter Tuning and Noise

This chapter demonstrates how noisy functions can be handled by Spot.

## 8.1 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.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "08"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)
```

08\_maans14\_2023-10-30\_23-08-46

### 8.1.1 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:

$$f(x) = x^2 + \epsilon$$

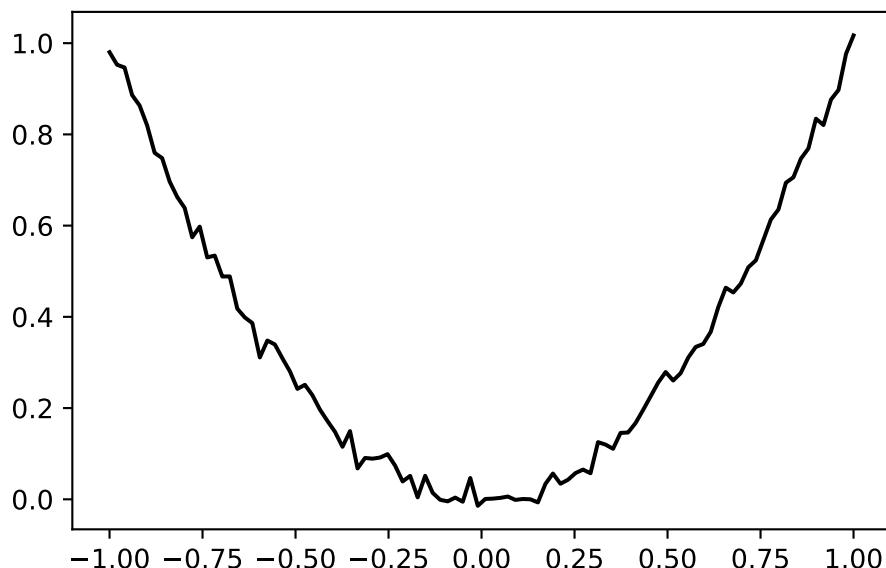
- Since `sigma` is set to 0.1, noise is added to the function:

```
fun = analytical().fun_sphere
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
```

```
sigma=0.02,  
seed=123,)
```

- 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()
```



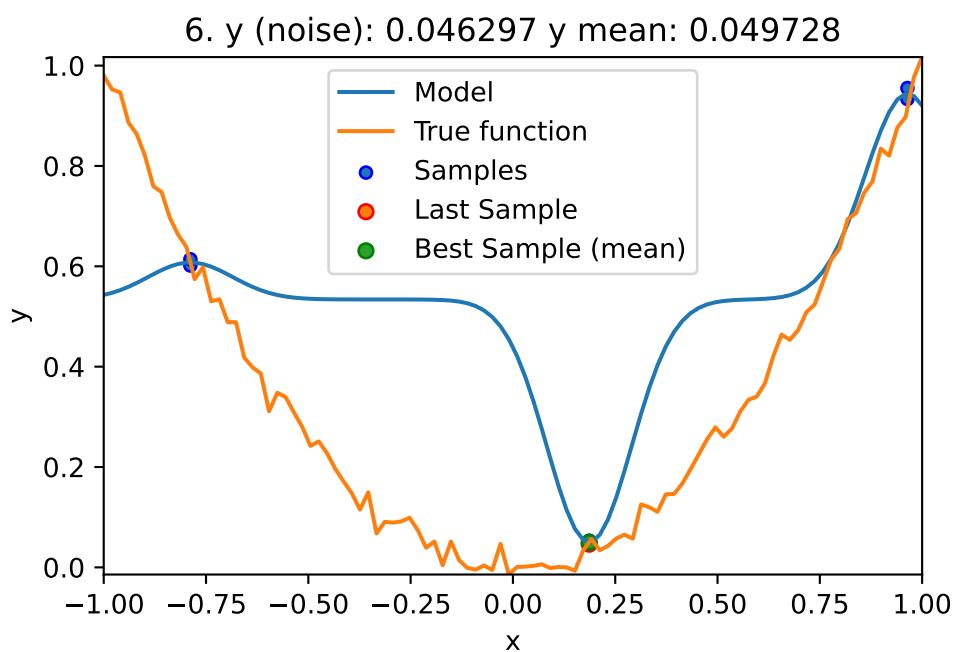
Spot is adopted as follows to cope with noisy functions:

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)

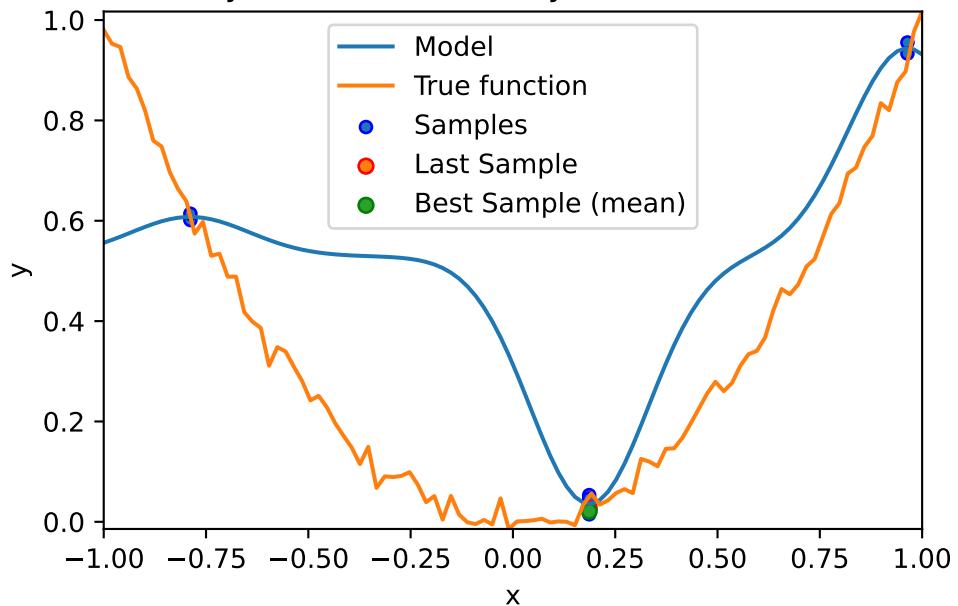
```
spot_1_noisy = spot.Spot(fun=fun,  
                         lower = np.array([-1]),  
                         upper = np.array([1]),  
                         fun_evals = 20,  
                         fun_repeats = 2,
```

```
noise = True,  
seed=123,  
show_models=True,  
design_control={"init_size": 3,  
                "repeats": 2},  
surrogate_control={"noise": True},  
fun_control=fun_control,)
```

```
spot_1_noisy.run()
```

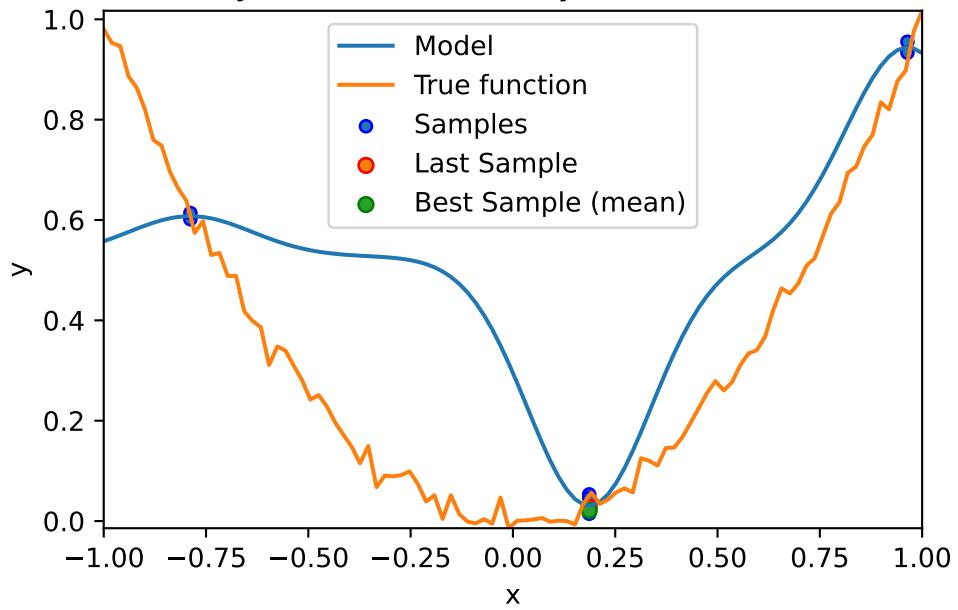


8. y (noise): 0.014973 y mean: 0.021186

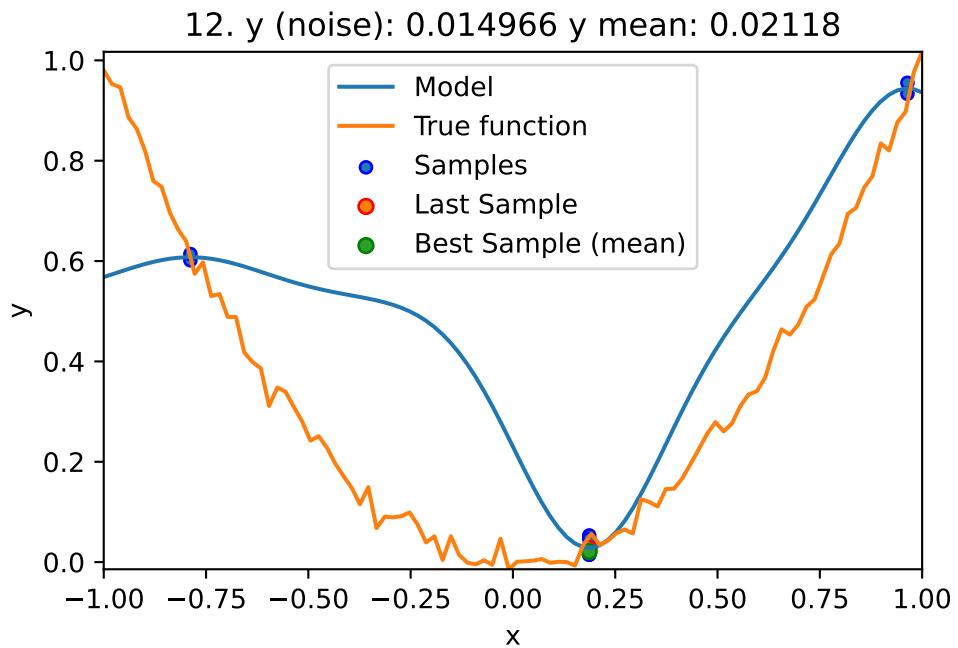


spotPython tuning: 0.01497250376483504 [#####-----] 40.00%

10. y (noise): 0.014972 y mean: 0.021186

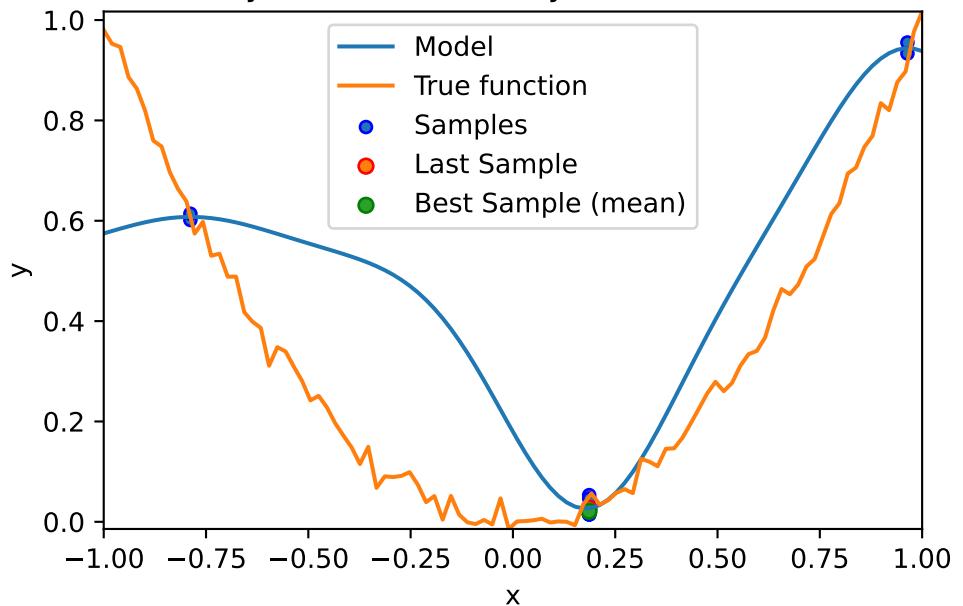


```
spotPython tuning: 0.014972272755587455 [#####----] 50.00%
```



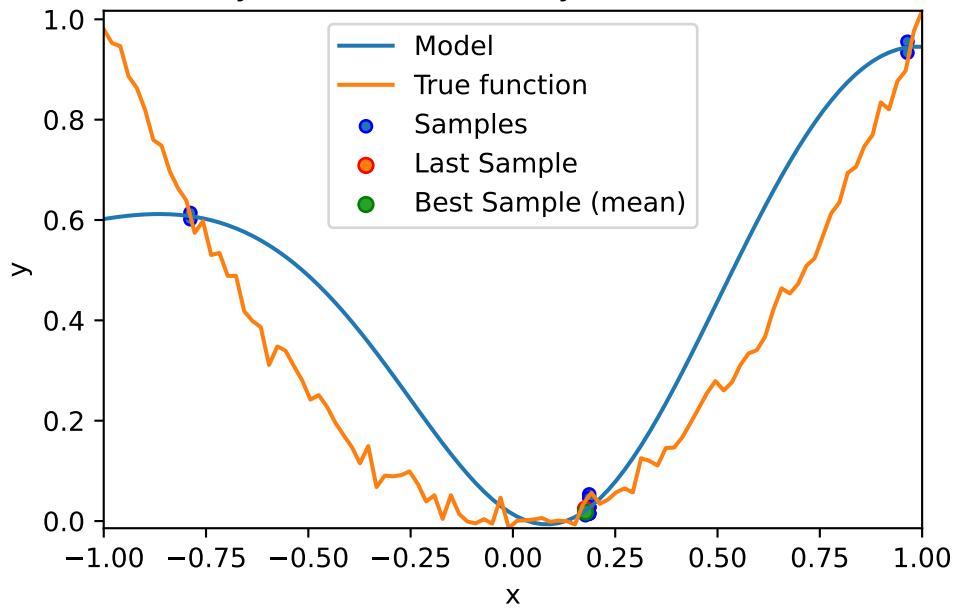
```
spotPython tuning: 0.014966273462465166 [#####----] 60.00%
```

14. y (noise): 0.01481 y mean: 0.021023

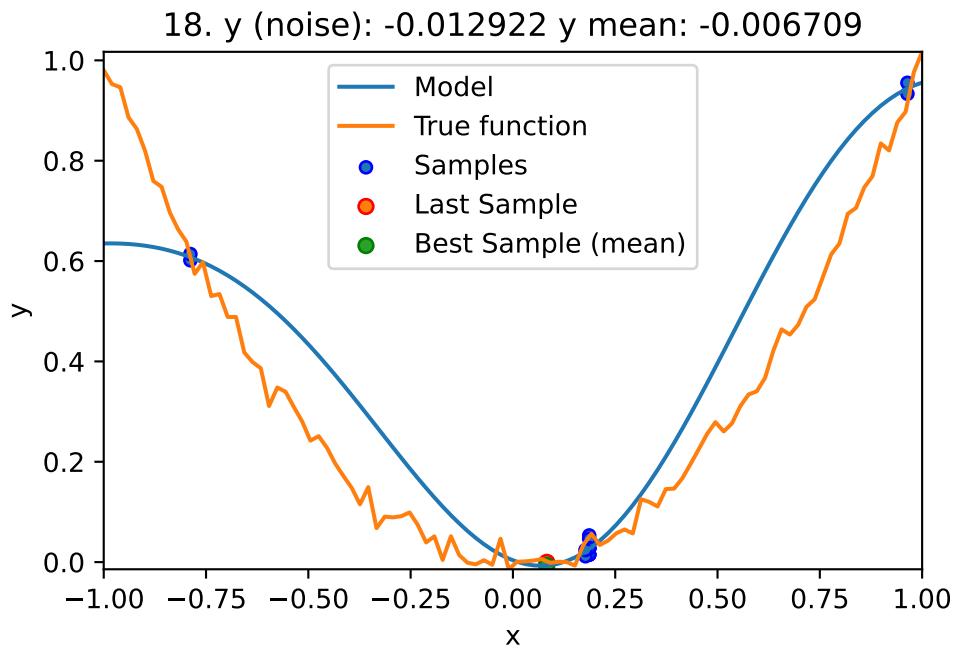


spotPython tuning: 0.01480994923420837 [#####---] 70.00%

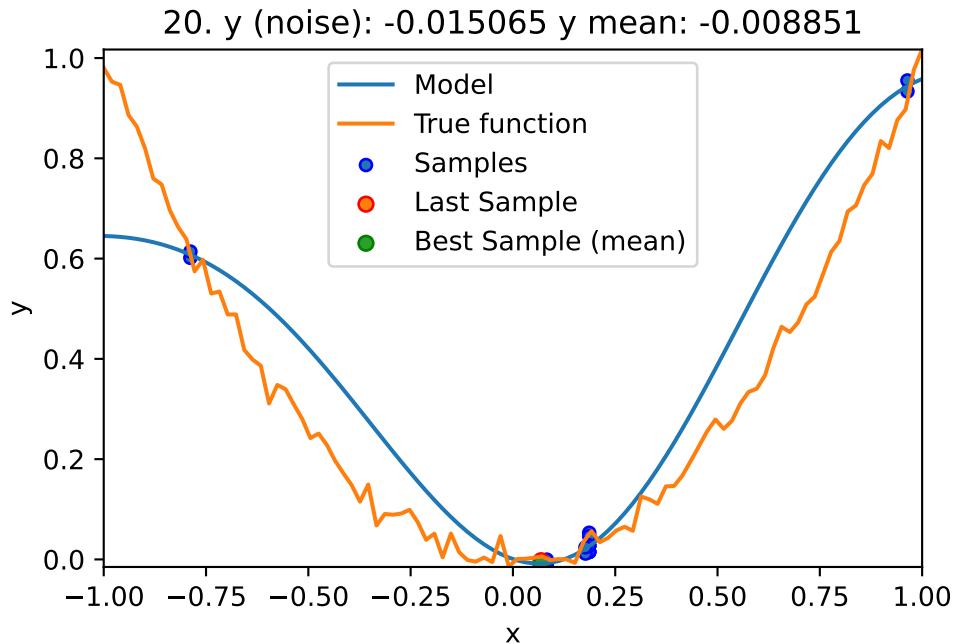
16. y (noise): 0.011666 y mean: 0.017879



```
spotPython tuning: 0.011665893638594611 [#####--] 80.00%
```



```
spotPython tuning: -0.012922167154961792 [#####--] 90.00%
```



```
spotPython tuning: -0.015064679263867698 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2e3c6bc10>
```

## 8.2 Print the Results

```
spot_1_noisy.print_results()
```

```
min y: -0.015064679263867698
x0: 0.0686858627600274
min mean y: -0.008851332275068022
x0: 0.0686858627600274
```

```
[['x0', 0.0686858627600274], ['x0', 0.0686858627600274]]
```

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

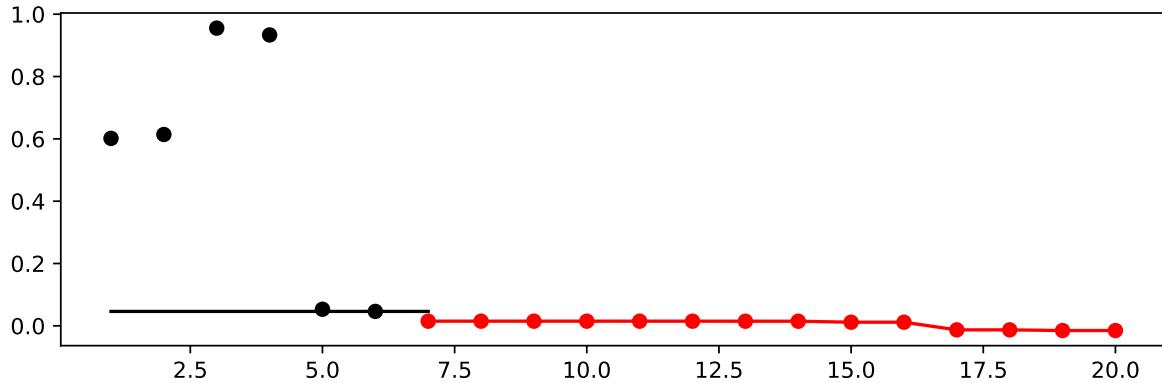


Figure 8.1: Progress plot. *Black dots* denote results from the initial design. *Red dots* illustrate the improvement found by the surrogate model based optimization.

## 8.3 Noise and Surrogates: The Nugget Effect

### 8.3.1 The Noisy Sphere

#### 8.3.1.1 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(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=2,
    seed=123,)
```

```

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:

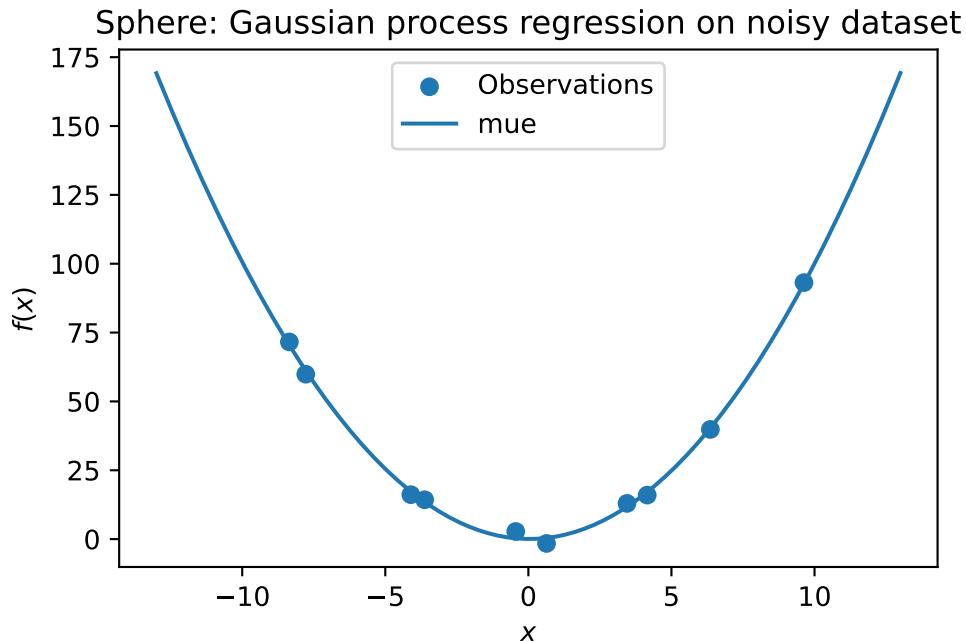
```

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="mu_e")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression on noisy dataset")

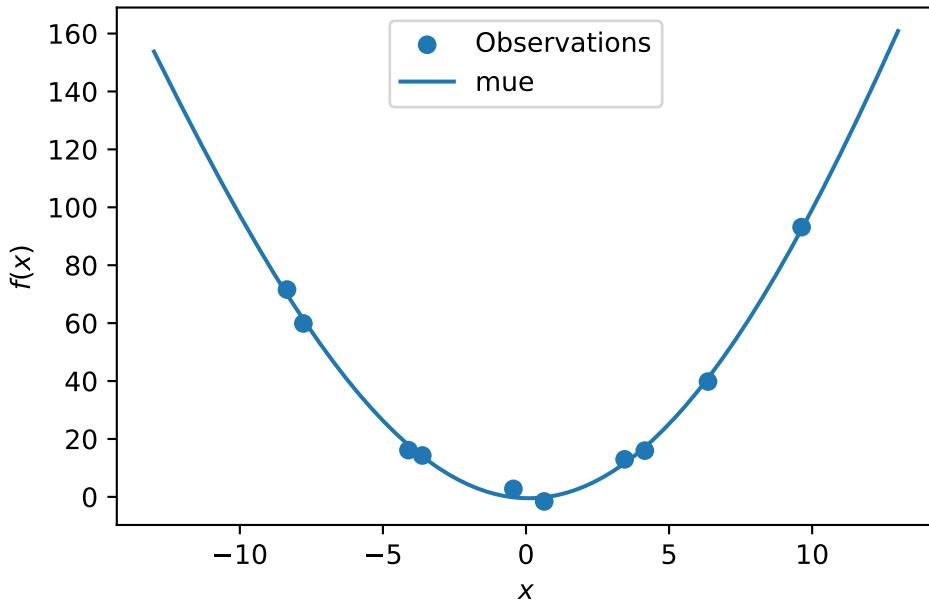
```



- 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="mle")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression with nugget on noisy dataset")
```

## 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
```

```
5.208508947162493e-05
```

- We see:

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

## 8.4 Exercises

### 8.4.1 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])
```

#### 8.4.2 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,)
```

#### 8.4.3 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 = fun_control_init(
    sigma=5,
    seed=123,)
```

#### 8.4.4 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,)
```

# 9 Handling Noise: Optimal Computational Budget Allocation in Spot

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

## 9.1 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.file import get_experiment_name
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_spot_tensorboard_path

PREFIX = "09"
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)
```

09\_maans14\_2023-10-30\_23-08-58

### 9.1.1 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(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    sigma=0.1,
    seed=123)

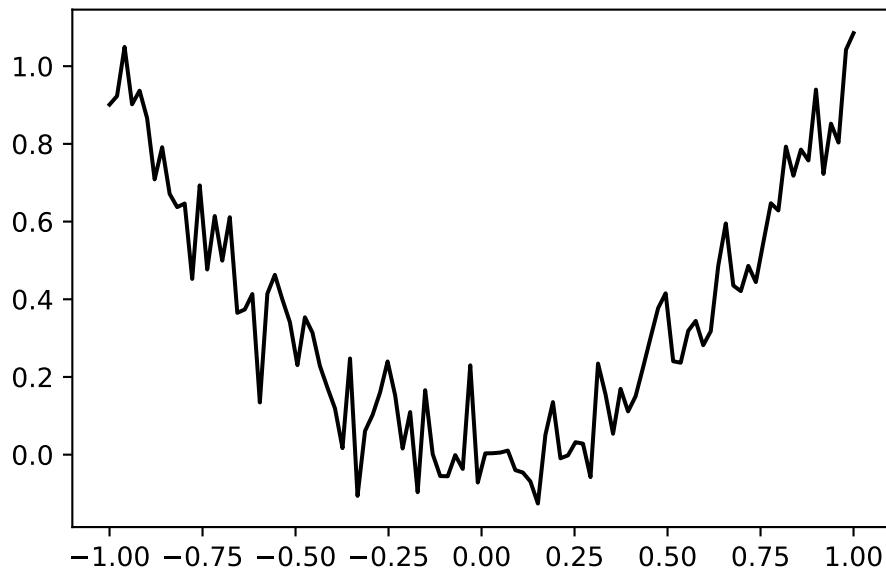
```

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,
                         lower = np.array([-1]),

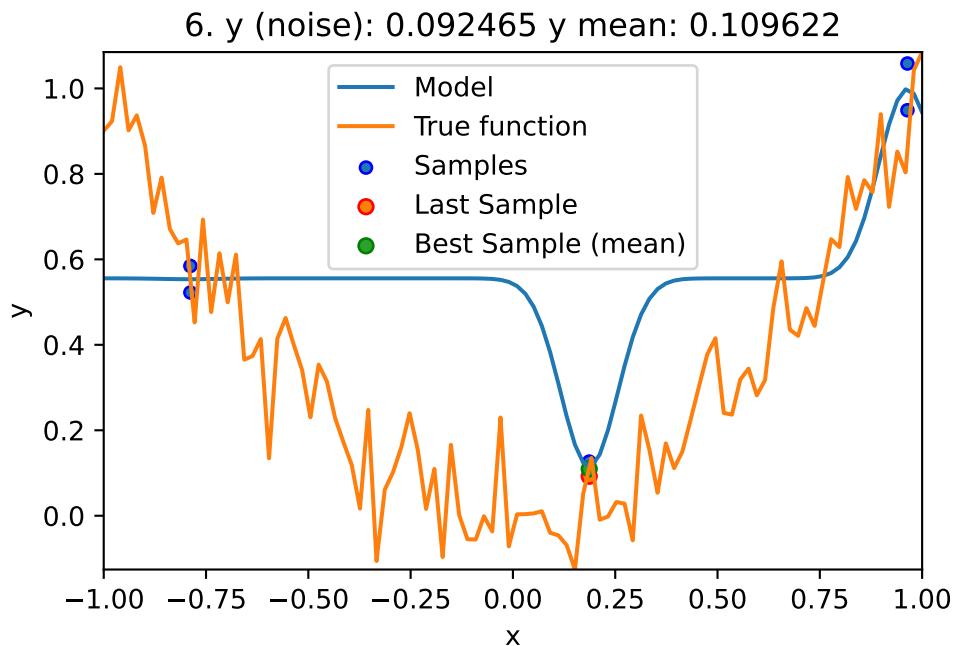
```

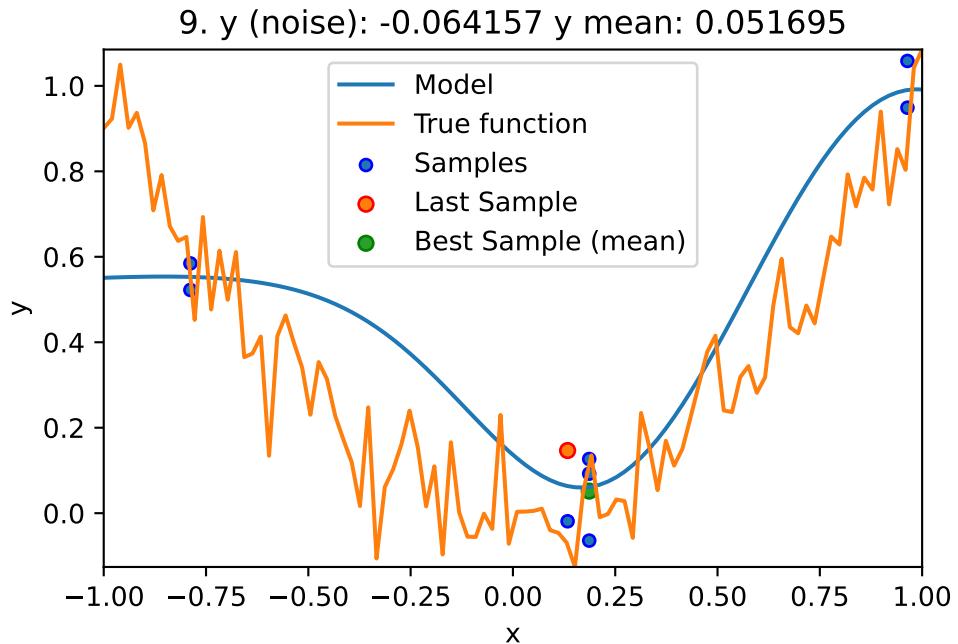
```

upper = np.array([1]),
fun_evals = 20,
fun_repeats = 2,
infill_criterion="ei",
noise = True,
tolerance_x=0.0,
ocba_delta = 1,
seed=123,
show_models=True,
fun_control = fun_control,
design_control={"init_size": 3,
                 "repeats": 2},
surrogate_control={"noise": True})

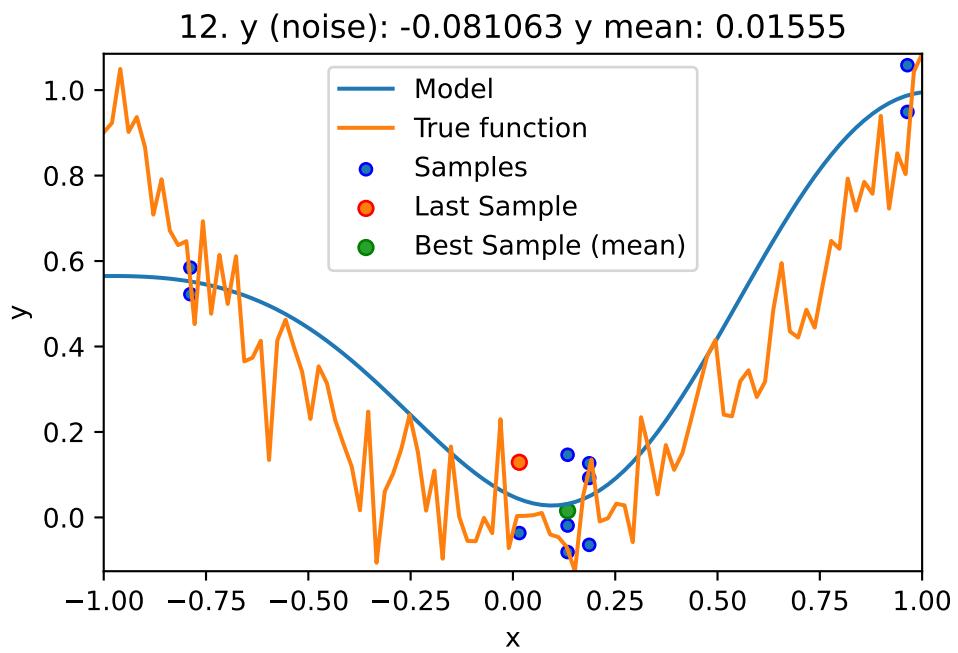
```

```
spot_1_noisy.run()
```

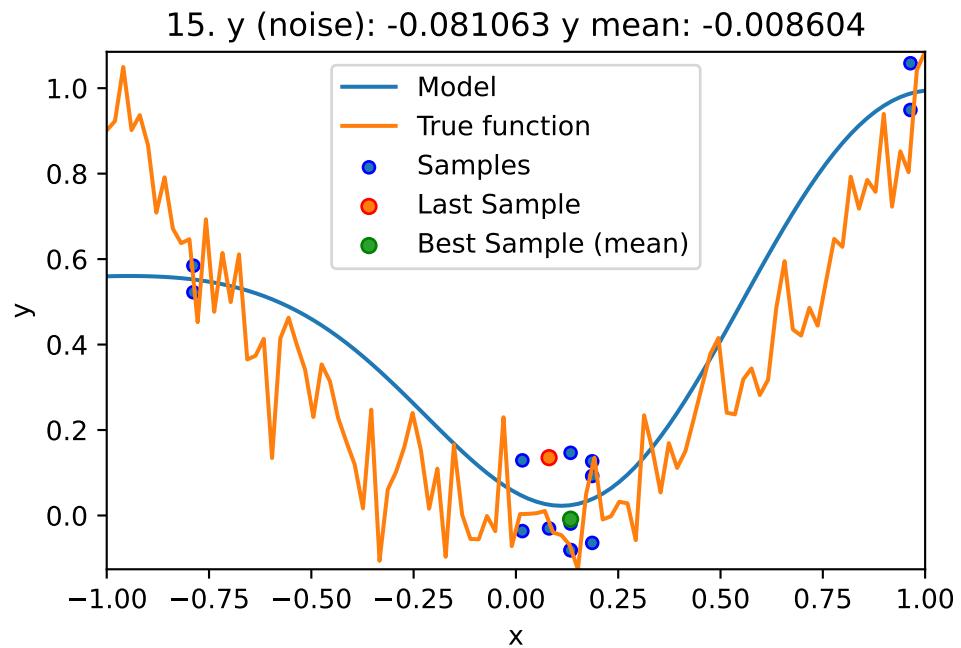




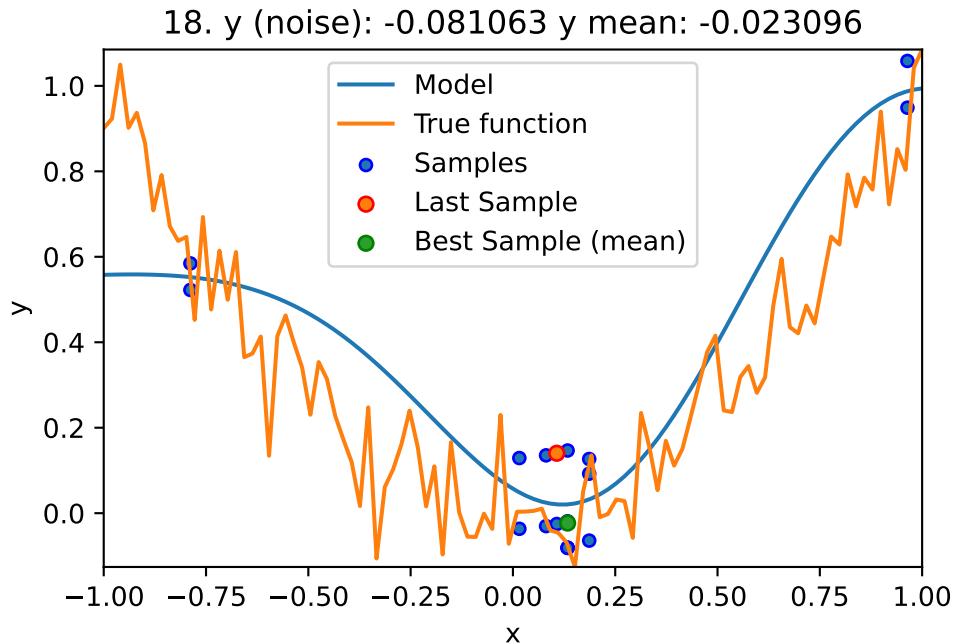
spotPython tuning: -0.0641572013655628 [#####-----] 45.00%



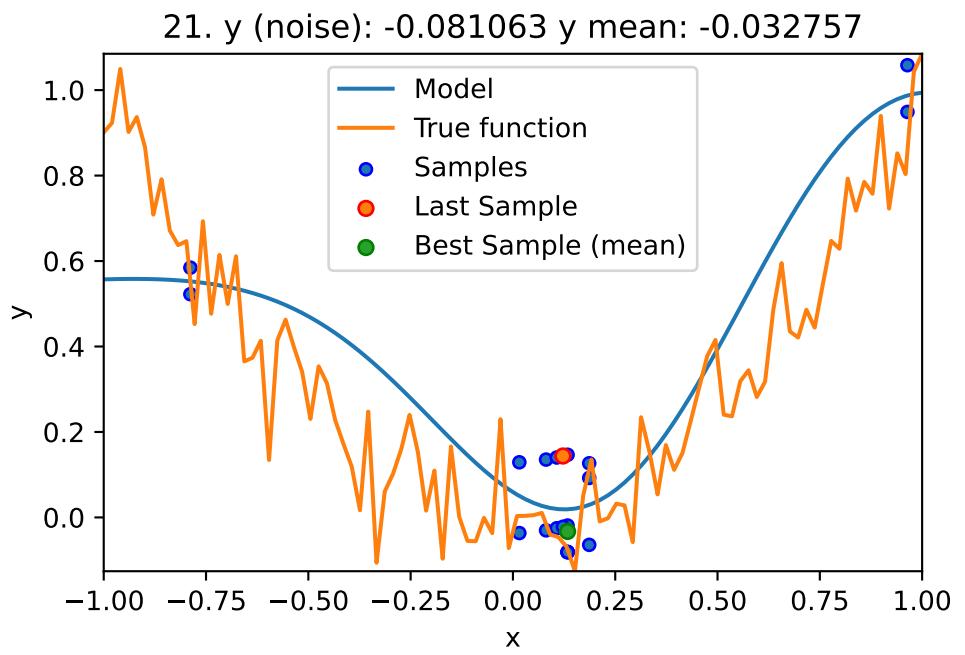
```
spotPython tuning: -0.08106318979737473 [#####----] 60.00%
```



```
spotPython tuning: -0.08106318979737473 [#####----] 75.00%
```



spotPython tuning: -0.08106318979737473 [#####-] 90.00%



```
spotPython tuning: -0.08106318979737473 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2feba80d0>
```

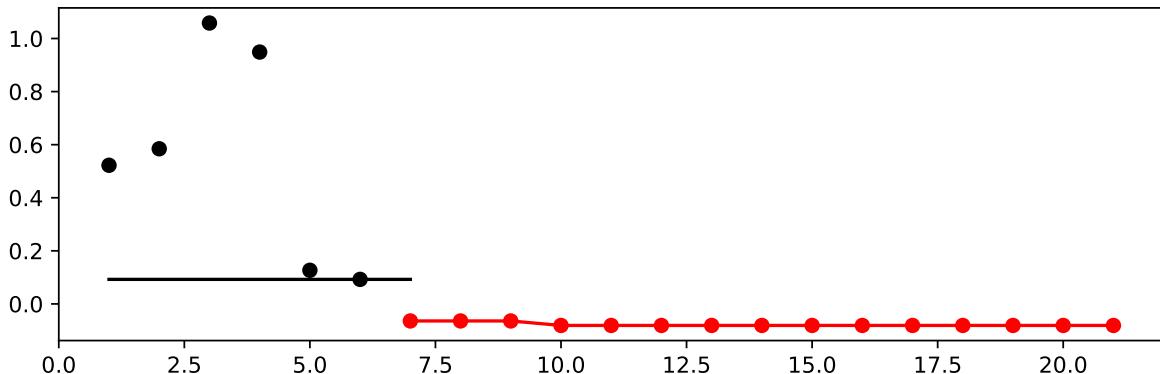
## 9.2 Print the Results

```
spot_1_noisy.print_results()
```

```
min y: -0.08106318979737473
x0: 0.13359994475077583
min mean y: -0.03275683462209028
x0: 0.13359994475077583
```

```
[['x0', 0.13359994475077583], ['x0', 0.13359994475077583]]
```

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



## 9.3 Noise and Surrogates: The Nugget Effect

### 9.3.1 The Noisy Sphere

#### 9.3.1.1 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:

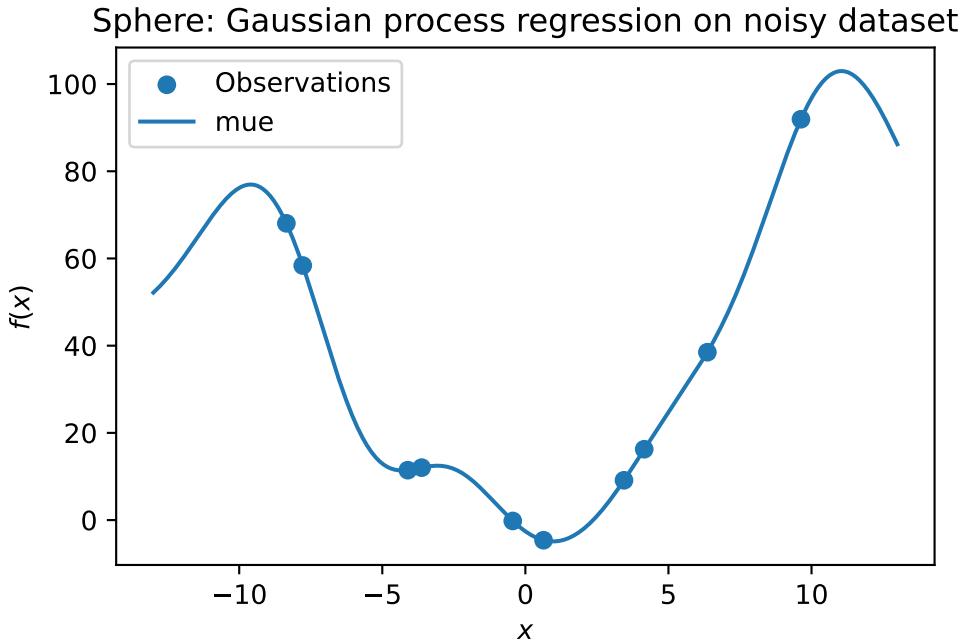
```

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="mu")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression on noisy dataset")

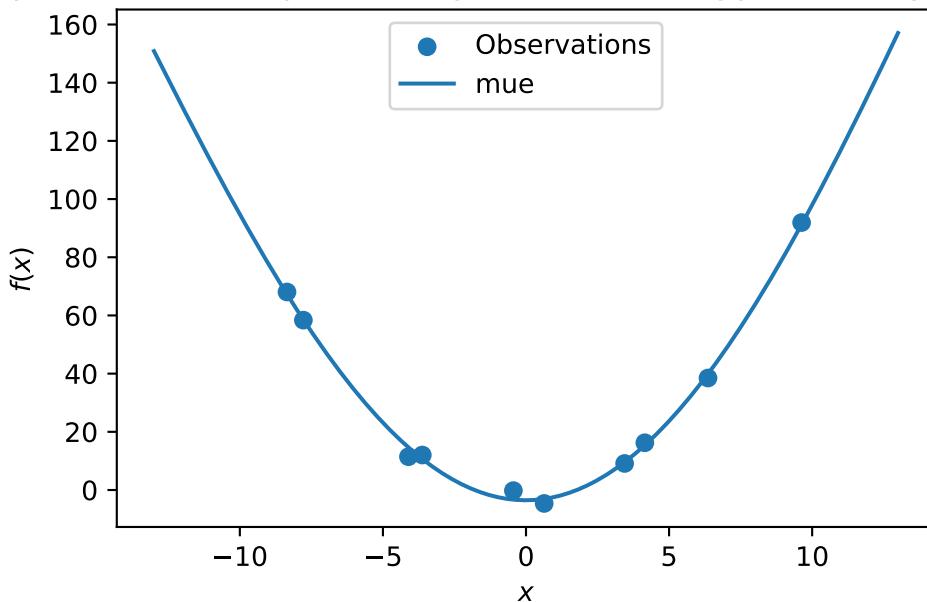
```



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="mle")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression with nugget on noisy dataset")
```

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
```

```
9.088149959982792e-05
```

We see:

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

## 9.4 Exercises

### 9.4.1 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])

```

#### **9.4.2 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)

```

#### **9.4.3 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}

```

#### **9.4.4 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)

```

## **Part II**

# **Hyperparameter Tuning**

# 10 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.

## 10.1 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"
```

## 10.2 Step 2: Initialization of the Empty `fun_control` Dictionary

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.utils.file import get_experiment_name, get_spot_tensorboard_path  
from spotPython.utils.device import getDevice  
  
experiment_name = get_experiment_name(prefix=PREFIX)  
  
fun_control = fun_control_init()
```

```

task="classification",
spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
TENSORBOARD_CLEAN=True)

```

## 10.3 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

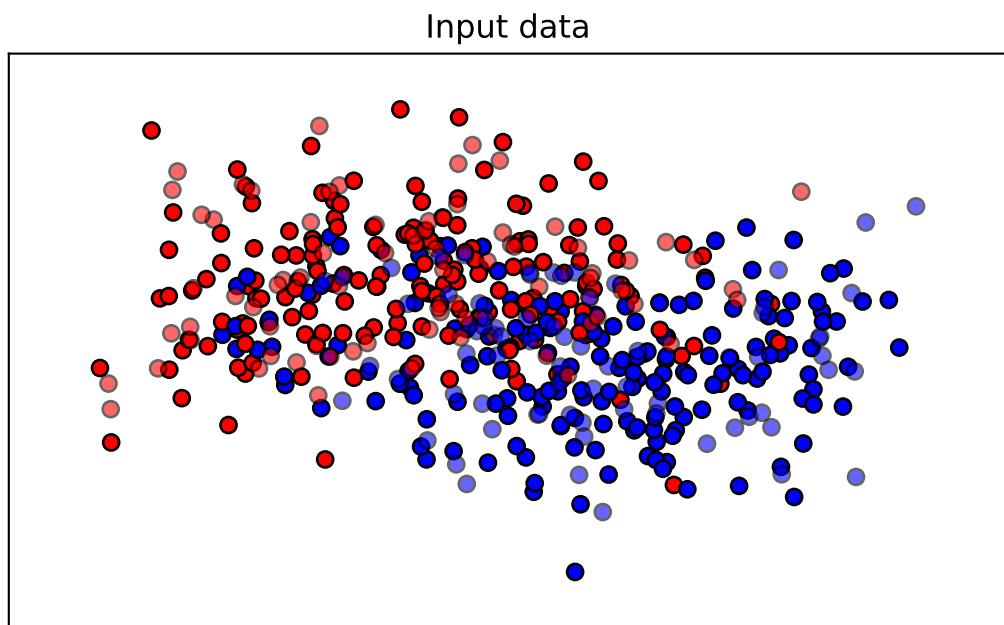
x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

```

```

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})
```

## 10.4 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(),  
)
```

## 10.5 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.data.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
```

## 10.6 Step 6: Modify hyper\_dict Hyperparameters for the Selected Algorithm aka core\_model

`spotPython` provides functions for modifying the hyperparameters, their bounds and factors as well as for activating and de-activating hyperparameters without re-compilation of the Python source code. These functions were described in Section 14.6.

### 10.6.1 Modify hyperparameter of type numeric and integer (boolean)

Numeric and boolean values can be modified using the `modify_hyper_parameter_bounds` method.

#### i `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}
```

### 10.6.2 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}
```

### 10.6.3 Optimizers

Optimizers are described in Section 14.6.1.

## 10.7 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,
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`.

### 10.7.1 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,
})
```

## 10.8 Step 8: Calling the SPOT Function

### 10.8.1 Preparing the SPOT Call

The following code passes the information about the parameter ranges and bounds to `spot`.

```
# extract the variable types, names, and bounds
from spotPython.hyperparameters.values import (
    get_var_name,
    get_var_type,
    get_bound_values
)
var_type = get_var_type(fun_control)
var_name = get_var_name(fun_control)
lower = get_bound_values(fun_control, "lower")
upper = get_bound_values(fun_control, "upper")
```

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

name	type	default	lower	upper	transform
C	float	1.0	0.1	10	None
kernel	factor	rbf	0	1	None
degree	int	3	3	3	None
gamma	factor	scale	0	1	None
coef0	float	0.0	0	0	None
shrinking	factor	0	0	1	None
probability	factor	0	0	0	None
tol	float	0.001	1e-05	0.001	None
cache_size	float	200.0	100	400	None
break_ties	factor	0	0	1	None

### 10.8.2 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

from spotPython.hyperparameters.values import get_default_hyperparameters_as_array
# X_start = get_default_hyperparameters_as_array(fun_control)
```

### 10.8.3 Run the Spot Optimizer

- Run SPOT for approx. x mins (`max_time`).
- Note: the run takes longer, because the evaluation time of initial design (here: `initi_size`, 20 points) is not considered.

#### 10.8.4 Starting the Hyperparameter Tuning

```
import numpy as np
from spotPython.spot import spot
from math import inf
spot_tuner = spot.Spot(fun=fun,
                       lower = lower,
                       upper = upper,
                       fun_evals = inf,
                       fun_repeats = 1,
                       max_time = MAX_TIME,
                       noise = False,
                       tolerance_x = np.sqrt(np.spacing(1)),
                       var_type = var_type,
                       var_name = var_name,
                       infill_criterion = "y",
                       n_points = 1,
                       seed=123,
                       log_level = 50,
                       show_models= False,
                       show_progress= True,
                       fun_control = fun_control,
                       design_control={"init_size": INIT_SIZE,
                                       "repeats": 1},
                       surrogate_control={"noise": True,
                                         "cod_type": "norm",
                                         "min_theta": -4,
                                         "max_theta": 3,
                                         "n_theta": len(var_name),
                                         "model_fun_evals": 10_000,
                                         "log_level": 50
                                         })
spot_tuner.run()
```

spotPython tuning: 5.734217584632275 [-----] 1.79%

spotPython tuning: 5.734217584632275 [-----] 4.05%

spotPython tuning: 5.734217584632275 [#-----] 7.16%

spotPython tuning: 5.734217584632275 [#-----] 10.37%

```
spotPython tuning: 5.734217584632275 [#-----] 13.47%
spotPython tuning: 5.734217584632275 [##-----] 16.04%
spotPython tuning: 5.734217584632275 [##-----] 18.28%
spotPython tuning: 5.734217584632275 [##-----] 23.27%
spotPython tuning: 5.734217584632275 [###-----] 28.83%
spotPython tuning: 5.734217584632275 [####-----] 35.07%
spotPython tuning: 5.734217584632275 [#####-----] 41.84%
spotPython tuning: 5.734217584632275 [#####-----] 48.99%
spotPython tuning: 5.734217584632275 [#####-----] 55.19%
spotPython tuning: 5.734217584632275 [#####-----] 58.76%
spotPython tuning: 5.734217584632275 [#####-----] 63.08%
spotPython tuning: 5.734217584632275 [#####-----] 76.08%
spotPython tuning: 5.734217584632275 [#####-----] 91.01%
spotPython tuning: 5.734217584632275 [#####-----] 100.00% Done...
<spotPython.spot.spot.Spot at 0x2e54c5120>
```

## 10.9 Step 9: Results

```
from spotPython.utils.file import save_pickle
save_pickle(spot_tuner, experiment_name)
```

```
from spotPython.utils.file import load_pickle
spot_tuner = load_pickle(experiment_name)
```

- Show the Progress of the hyperparameter tuning:

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized.

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

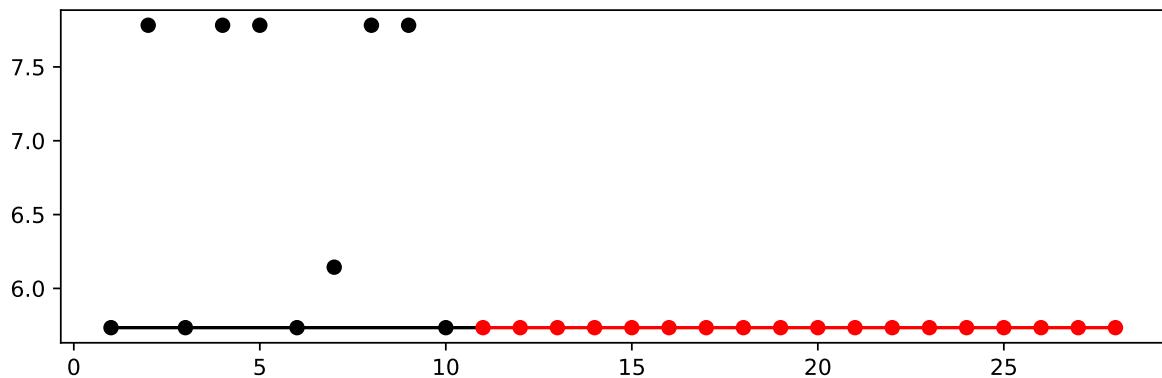


Figure 10.1: Progress plot. *Black dots* denote results from the initial design. *Red dots* illustrate the improvement found by the surrogate model based optimization.

- Print the results

```
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	1.0	None
degree	int	3	3.0	3.0	3.0	None
gamma	factor	scale	0.0	1.0	0.0	None
coef0	float	0.0	0.0	0.0	0.0	None
shrinking	factor	0	0.0	1.0	0.0	None
probability	factor	0	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	0.0	None

### 10.9.1 Show variable importance

```
spot_tuner.plot_importance(threshold=0.025, filename="./figures/" + experiment_name+_imp
```

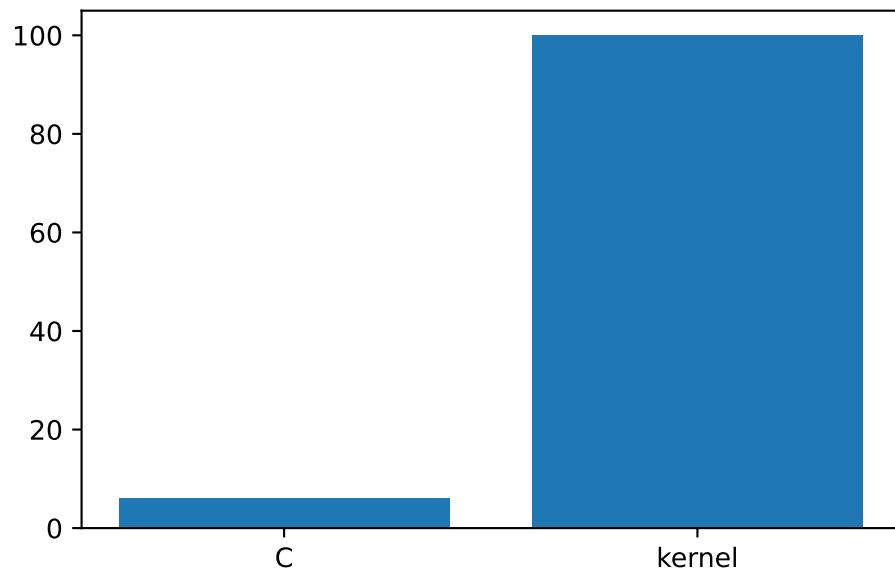


Figure 10.2: Variable importance plot, threshold 0.025.

### 10.9.2 Get Default Hyperparameters

```
from spotPython.hyperparameters.values import get_default_values, transform_hyper_parameter
values_default = get_default_values(fun_control)
values_default = transform_hyper_parameter_values(fun_control=fun_control, hyper_parameter
values_default
```

```
{'C': 1.0,
'kernel': 'rbf',
'degree': 3,
'gamma': 'scale',
'coef0': 0.0,
```

```

'shrinking': 0,
'probability': 0,
'tol': 0.001,
'cache_size': 200.0,
'break_ties': 0}

from sklearn.pipeline import make_pipeline
model_default = make_pipeline(fun_control["prep_model"], fun_control["core_model"](**value
model_default

Pipeline(steps=[('standardscaler', StandardScaler()),
               ('svc',
                SVC(break_ties=0, cache_size=200.0, probability=0,
                     shrinking=0))])

```

### 10.9.3 Get SPOT Results

```

X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
print(X)

[[2.39447166e+00 1.00000000e+00 3.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 9.82585316e-04
 3.75637165e+02 0.00000000e+00]]

from spotPython.hyperparameters.values import assign_values, return_conf_list_from_var_dic
v_dict = assign_values(X, fun_control["var_name"])
return_conf_list_from_var_dict(var_dict=v_dict, fun_control=fun_control)

[{'C': 2.394471655384338,
 'kernel': 'rbf',
 'degree': 3,
 'gamma': 'scale',
 'coef0': 0.0,
 'shrinking': 0,
 'probability': 0,
 'tol': 0.000982585315792582,
 'cache_size': 375.6371648003268,
 'break_ties': 0}]

```

```

from spotPython.hyperparameters.values import get_one_sklearn_model_from_X
model_spot = get_one_sklearn_model_from_X(X, fun_control)
model_spot

Pipeline(steps=[('standardscaler', StandardScaler()),
               ('svc',
                SVC(C=2.394471655384338, break_ties=0,
                     cache_size=375.6371648003268, probability=0, shrinking=0,
                     tol=0.000982585315792582))])

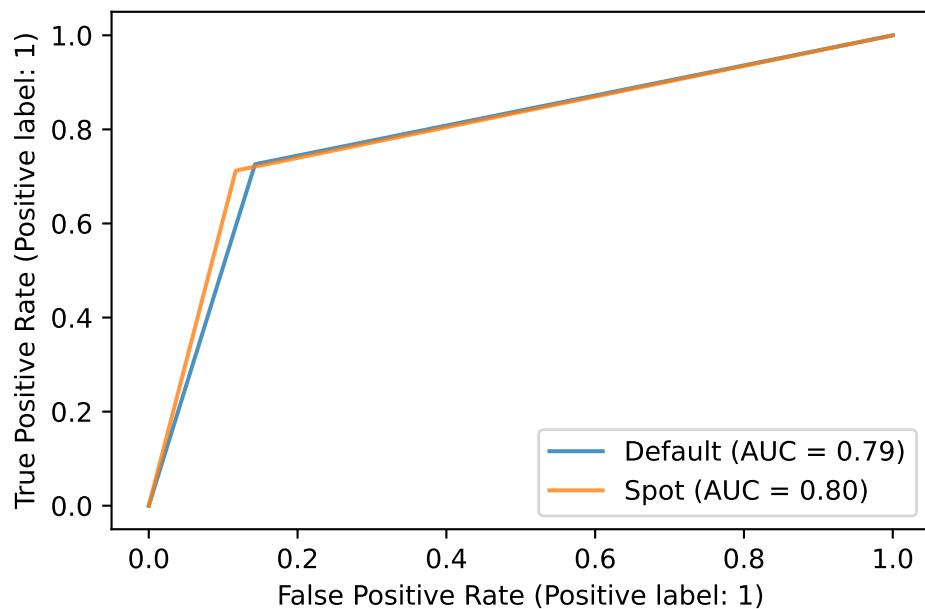
```

#### 10.9.4 Plot: Compare Predictions

```

from spotPython.plot.validation import plot_roc
plot_roc([model_default, model_spot], fun_control, model_names=["Default", "Spot"])

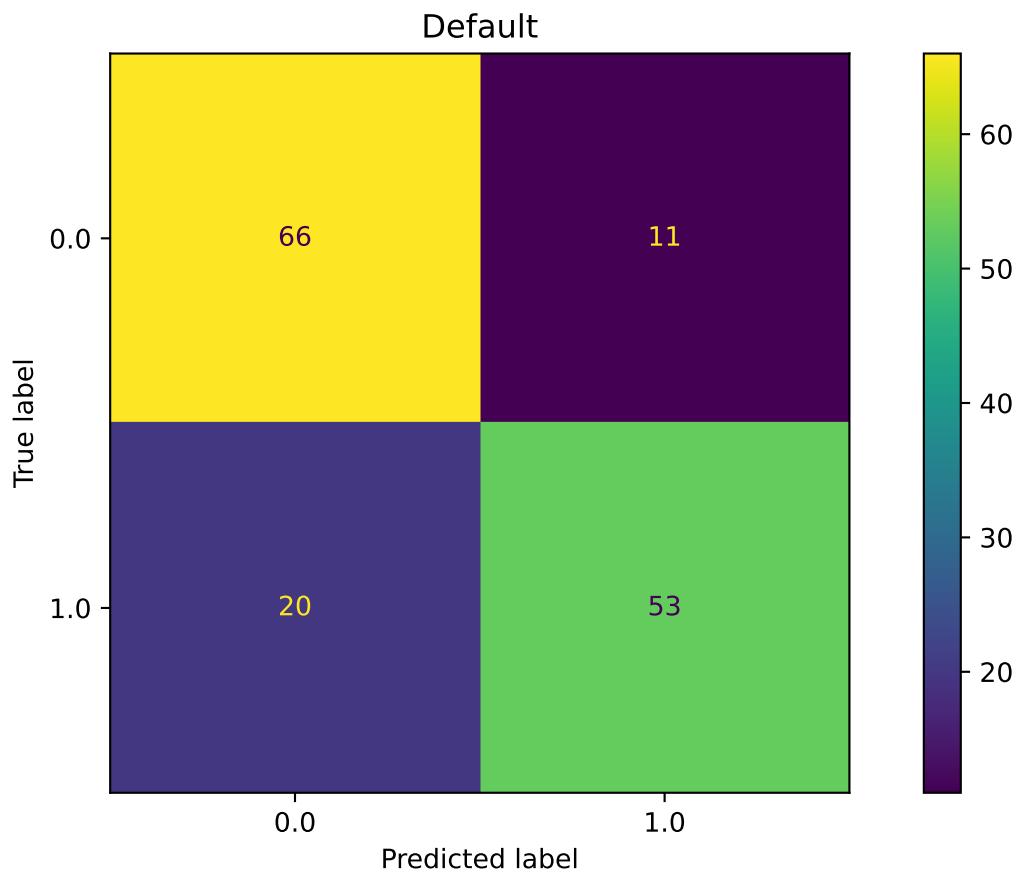
```



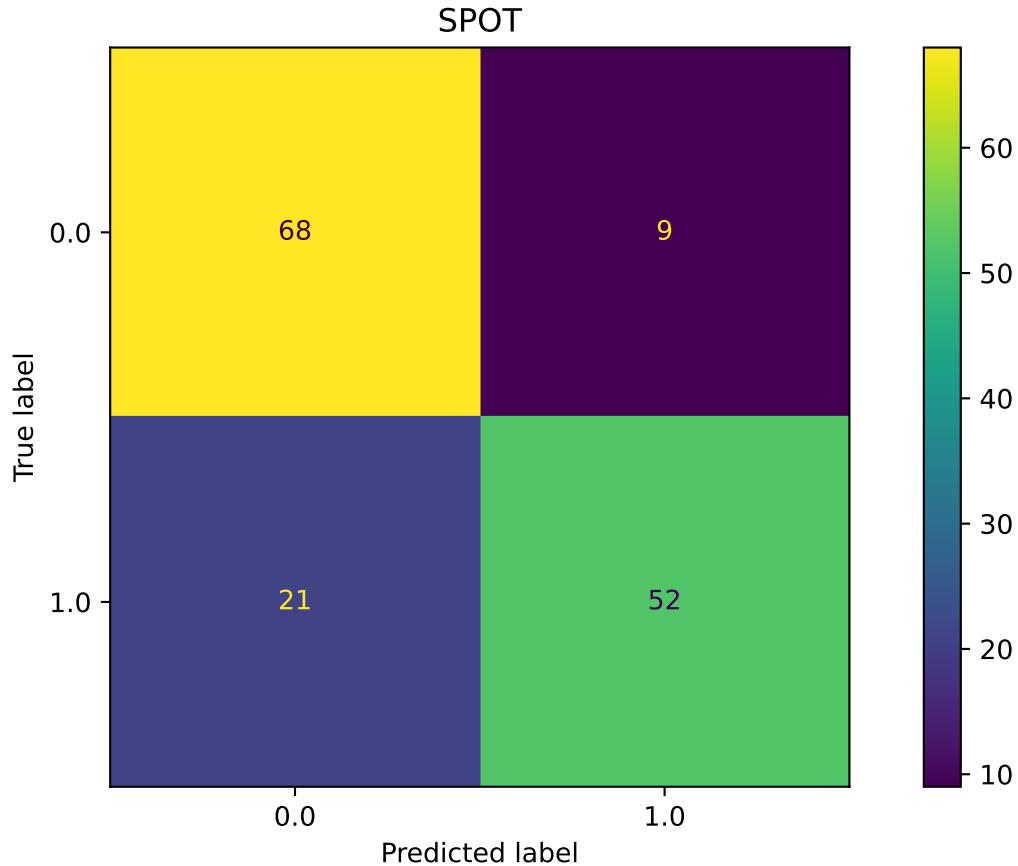
```

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

```



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



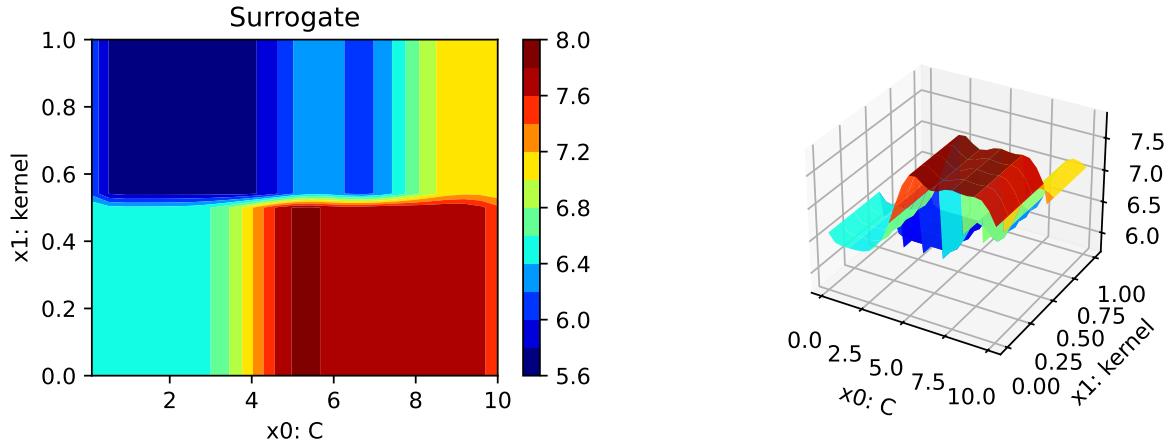
```
min(spot_tuner.y), max(spot_tuner.y)
```

```
(5.734217584632275, 7.782152436286657)
```

#### 10.9.5 Detailed Hyperparameter Plots

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

```
C: 5.974249844745921
kernel: 100.0
```



### 10.9.6 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

### 10.9.7 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)
```

# 11 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.

## 11.1 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\_000samples 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="13-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>.

## 11.2 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
from spotPython.utils.file import get_spot_tensorboard_path
import os
from spotPython.utils.file import get_experiment_name
experiment_name = get_experiment_name(prefix=PREFIX)
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    TENSORBOARD_CLEAN=True)
print(experiment_name)
```

13-river\_maans14\_2023-10-30\_23-13-42

### 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 13.8.4 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.

## 11.3 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.

```
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})
```

## 11.4 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()
fun_control.update({"prep_model": prep_model})
```

## 11.5 SelectModel (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
add_core_model_to_fun_control(core_model=HoeffdingAdaptiveTreeRegressor,
                               fun_control=fun_control,
                               hyper_dict=RiverHyperDict,
                               filename=None)
```

## 11.6 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, "delta", bounds=[1e-10, 1e-6])
modify_hyper_parameter_bounds(fun_control, "merit_preprune", [0, 0])
```

**i** 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_power
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_prune	factor	0	0	0	None

## 11.7 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.

**i** 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  $(\text{step}/n_{\text{steps}})^{\text{weight\_coeff}}$ , 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
})
```

## 11.8 Calling the SPOT Function

### 11.8.1 Prepare the SPOT Parameters

The hyperparameter tuning configuration is stored in the `fun_control` dictionary. Since Spot can be used as an optimization algorithm with a similar interface as optimization algorithms from `scipy.optimize` [LINK], the bounds and variable types have to be specified explicitly. The `get_var_type`, `get_var_name`, and `get_bound_values` functions [SOURCE] implement the required functionality.

- Get types and variable names as well as lower and upper bounds for the hyperparameters, so that they can be passed to the `Spot` function.

```
from spotPython.hyperparameters.values import (
    get_var_type,
    get_var_name,
    get_bound_values
)
var_type = get_var_type(fun_control)
var_name = get_var_name(fun_control)
lower = get_bound_values(fun_control, "lower")
upper = get_bound_values(fun_control, "upper")
```

### 11.8.2 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)
```

### 11.8.3 Run the Spot Optimizer

The class `Spot` [\[SOURCE\]](#) is the hyperparameter tuning workhorse. It is initialized with the following parameters:

- `fun`: the objective function
- `lower`: lower bounds of the hyperparameters
- `upper`: upper bounds of the hyperparameters
- `fun_evals`: number of function evaluations
- `max_time`: maximum time in seconds
- `tolerance_x`: tolerance for the hyperparameters
- `var_type`: variable types of the hyperparameters
- `var_name`: variable names of the hyperparameters
- `show_progress`: show progress bar
- `fun_control`: dictionary with control parameters for the objective function

- `design_control`: dictionary with control parameters for the initial design
- `surrogate_control`: dictionary with control parameters for the surrogate model

 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.spot import spot
from math import inf
spot_tuner = spot.Spot(fun=fun,
                       lower = lower,
                       upper = upper,
                       fun_evals = inf,
                       max_time = MAX_TIME,
                       tolerance_x = np.sqrt(np.spacing(1)),
                       var_type = var_type,
                       var_name = var_name,
                       show_progress= True,
                       fun_control = fun_control,
                       design_control={"init_size": INIT_SIZE},
                       surrogate_control={"noise": False,
                                          "cod_type": "norm",
                                          "min_theta": -4,
                                          "max_theta": 3,
                                          "n_theta": len(var_name),
                                          "model_fun_evals": 10_000})
spot_tuner.run(X_start=X_start)
```

spotPython tuning: 2.205384065994446 [##-----] 17.65%

spotPython tuning: 2.190294017596055 [###-----] 31.76%

spotPython tuning: 2.190294017596055 [####-----] 41.74%

spotPython tuning: 2.190294017596055 [#####----] 50.49%

spotPython tuning: 2.190294017596055 [#####---] 65.44%

```
spotPython tuning: 2.1762173327463197 [#####--] 78.93%
spotPython tuning: 2.1762173327463197 [#####--] 93.43%
spotPython tuning: 2.102850778058835 [#####] 100.00% Done...
<spotPython.spot.spot.Spot at 0x2aa757310>
```

#### 11.8.4 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.file 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  $\theta$  of the Kriging surrogate [\[SOURCE\]](#) is plotted against the number of optimization steps.

#### 11.8.5 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
save_pickle(spot_tuner, experiment_name)
```

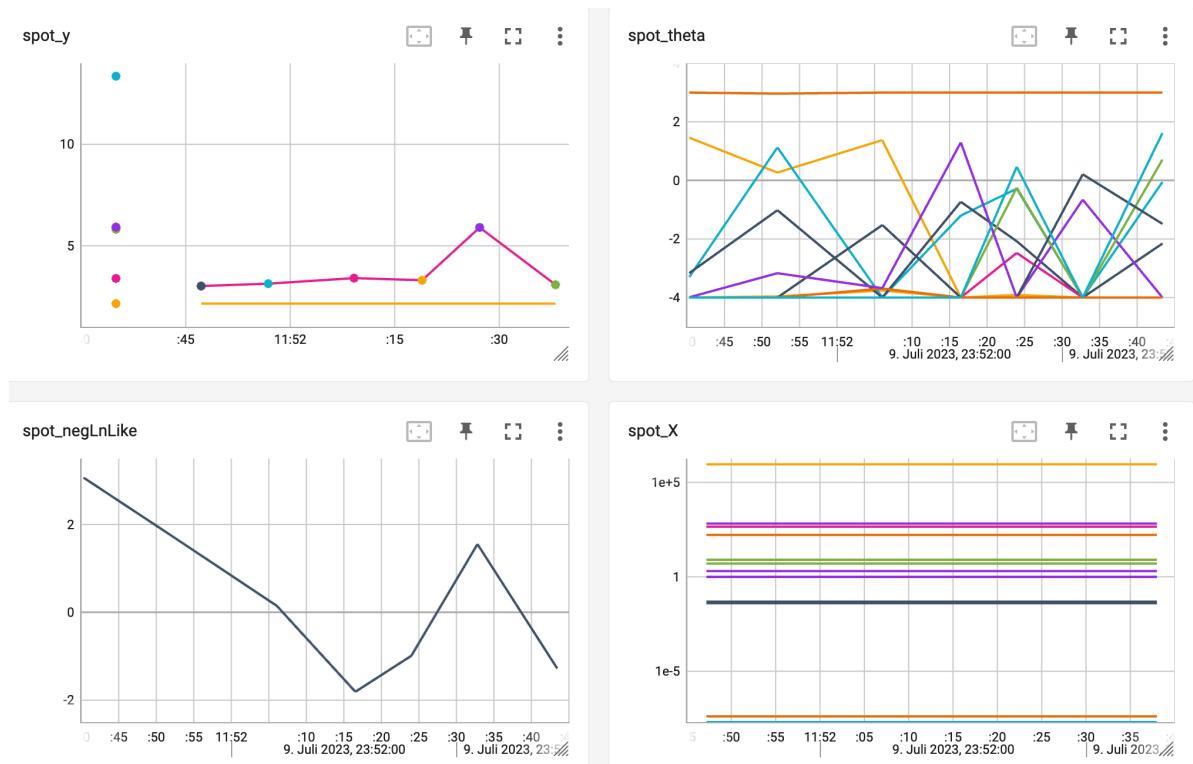
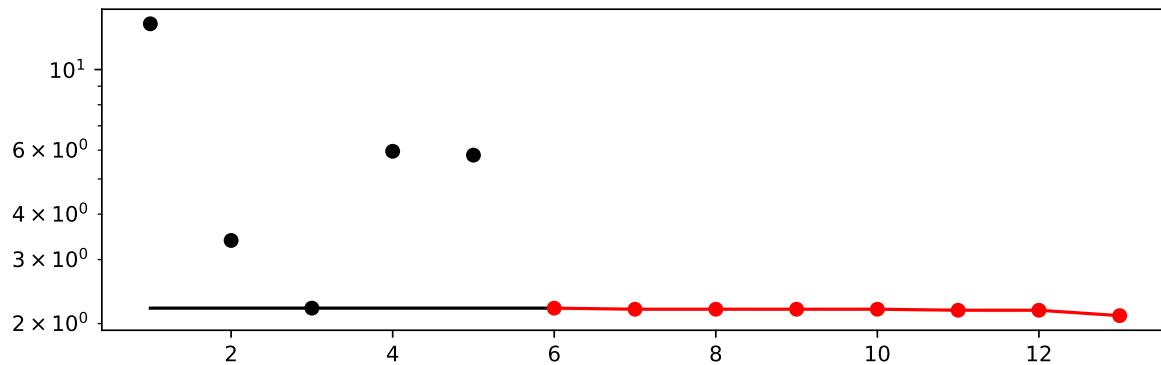


Figure 11.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

```
from spotPython.utils.file import load_pickle
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 performance values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress(log_y=True, filename="./figures/" + experiment_name + "_progress.pdf")
```



Results can also be printed in tabular form.

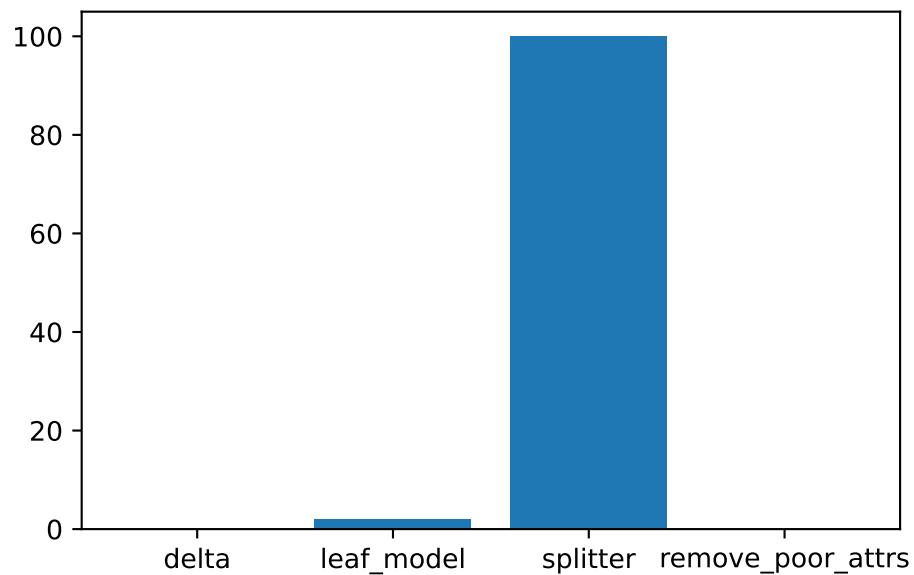
```
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

name	type	default	lower	upper	...	...
grace_period	int	200	10.0	1000.0	400	...
max_depth	int	20	2.0	20.0	2	...
delta	float	1e-07	1e-10	1e-06	1e-09	1e-06
tau	float	0.05	0.01	0.1	0.001	0.1
leaf_prediction	factor	mean	0.0	2.0	0.0	2.0
leaf_model	factor	LinearRegression	0.0	2.0	0.0	2.0
model_selector_decay	float	0.95	0.9	0.99	0.9721805031355	0.99
splitter	factor	EBSTSplitter	0.0	2.0	0.0	2.0
min_samples_split	int	5	2.0	10.0	2	10
bootstrap_sampling	factor	0	0.0	1.0	0.0	1.0
drift_window_threshold	int	300	100.0	500.0	200	500
switch_significance	float	0.05	0.01	0.1	0.001	0.1
binary_split	factor	0	0.0	1.0	0.0	1.0

max_size	float	500.0	100.0	1000.0	331.60662874071
memory_estimate_period	int	1000000	1000000.0	1000000.0	88851
stop_mem_management	factor	0	0.0	1.0	
remove_poor_attrs	factor	0	0.0	1.0	
merit_prune	factor	0	0.0	0.0	

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

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



## 11.9 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})

```

## 11.10 Get Default Hyperparameters

The default hyperparameters, which will be used for a comparison 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)

```

**i** 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"],
)

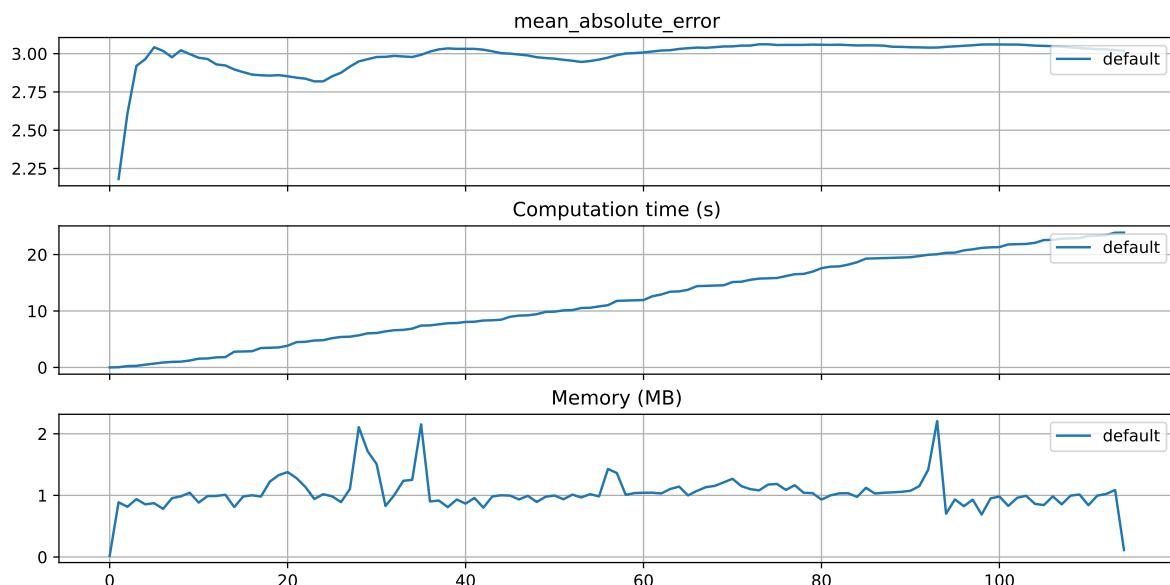
```

The three performance criteria, i.e., score (metric), runtime, and memory consumption, can be visualized with the following commands:

```

from spotRiver.evaluation.eval_bml import plot_bml_oml_horizon_metrics, plot_bml_oml_horizon
df_labels=["default"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default], log_y=False, df_labels=df_labels)

```



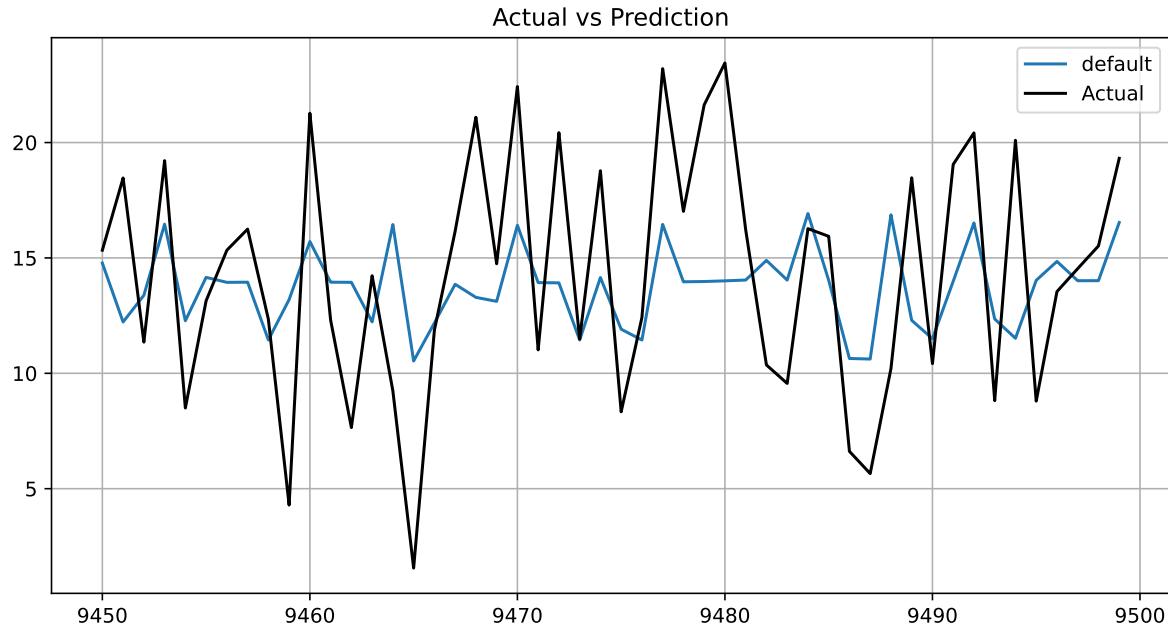
### 11.10.1 Show Predictions

- 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_col)
```



## 11.11 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)

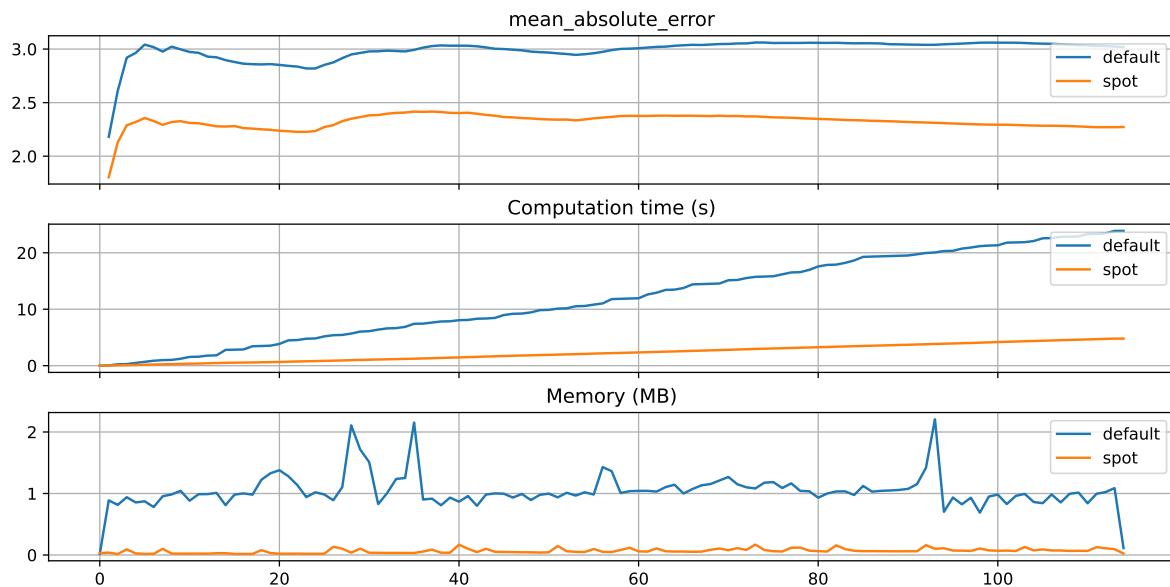
df_eval_spot, df_true_spot = eval_oml_horizon(
    model=model_spot,
    train=fun_control["train"],
    test=fun_control["test"],
```

```

        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_la

```

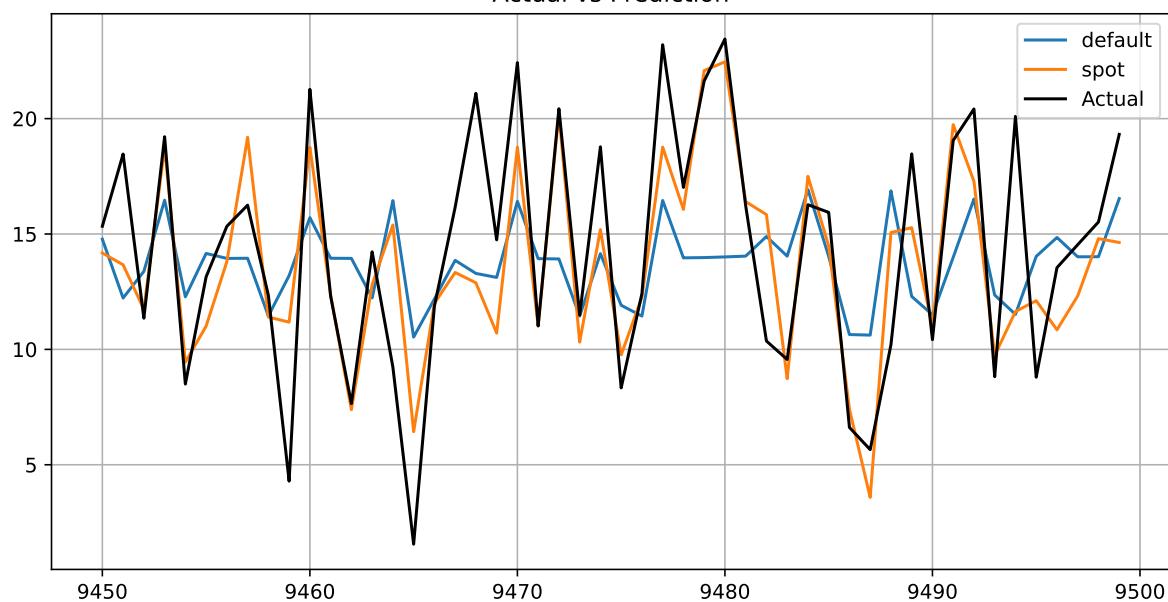


```

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

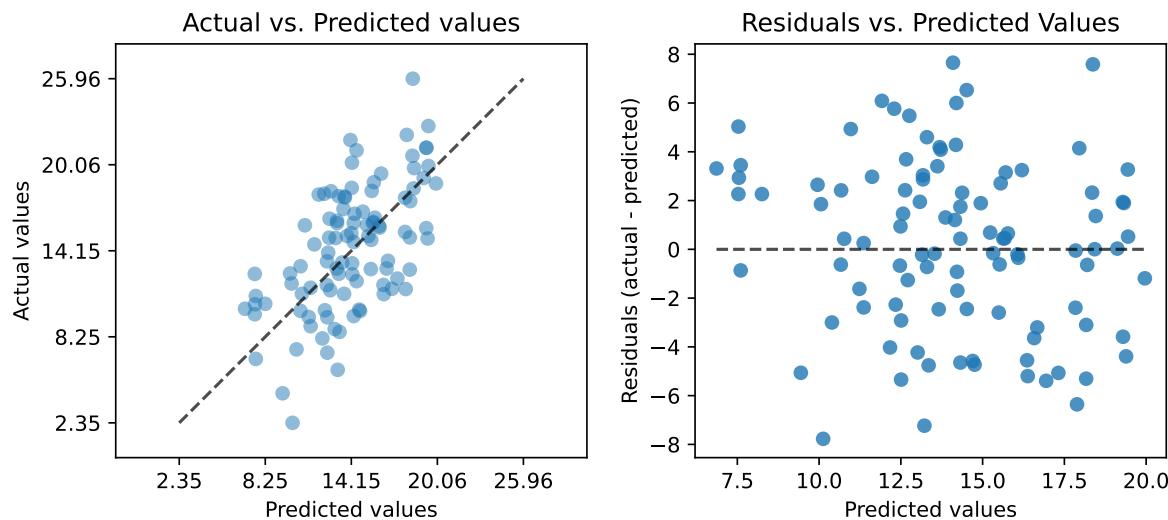
```

Actual vs Prediction

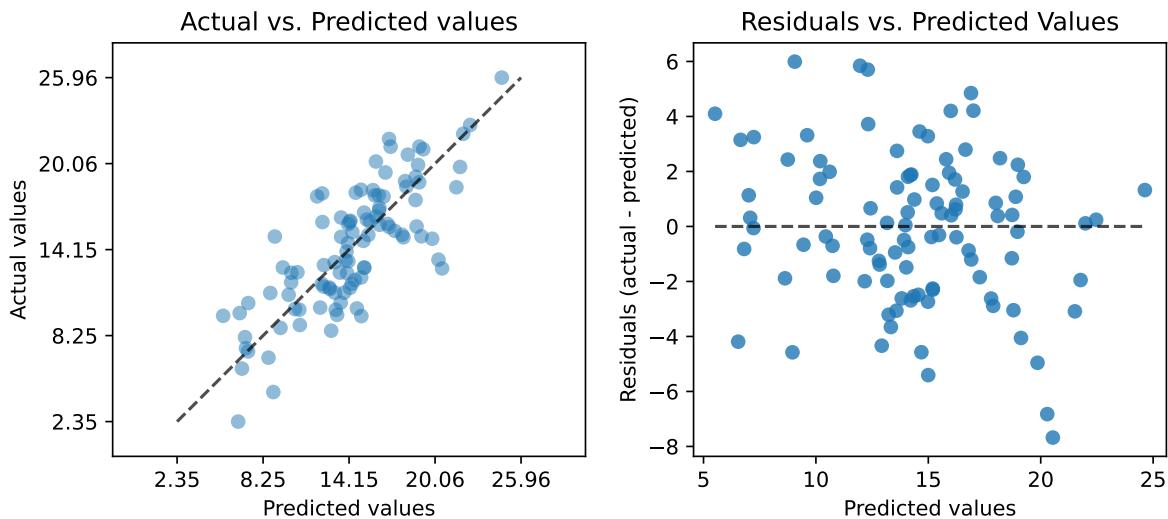


```
from spotPython.plot.validation import plot_actual_vs_predicted
plot_actual_vs_predicted(y_test=df_true_default[target_column], y_pred=df_true_default["Predicted"])
plot_actual_vs_predicted(y_test=df_true_spot[target_column], y_pred=df_true_spot["Predicted"])
```

Default



## SPOT



## 11.12 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}
```

### 11.12.1 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': 81,  
'n_branches': 40,  
'n_leaves': 41,  
'n_active_leaves': 30,  
'n_inactive_leaves': 0,  
'height': 13,  
'total_observed_weight': 39002.0,  
'n_alternate_trees': 31,  
'n_pruned_alternate_trees': 27,  
'n_switch_alternate_trees': 3}
```

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

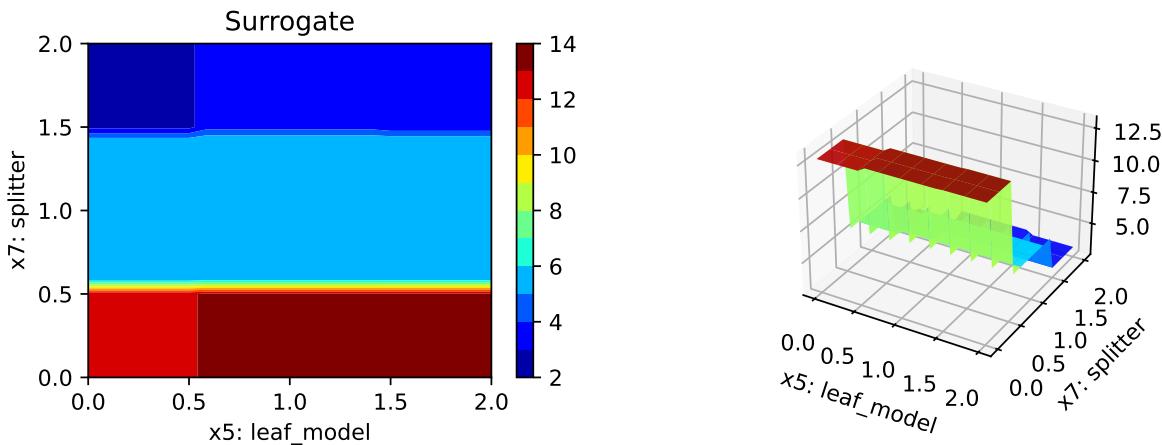
Parameter	Default	Spot
n_nodes	35	81
n_branches	17	40

n_leaves	18	41
n_active_leaves	96	30
n_inactive_leaves	0	0
height	6	13
total_observed_weight	39002	39002
n_alternate_trees	21	31
n_pruned_alternate_trees	6	27
n_switch_alternate_trees	2	3

## 11.13 Detailed Hyperparameter Plots

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

```
leaf_model: 1.9245880384765832
splitter: 100.0
```



## 11.14 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

## 11.15 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)
```

# 12 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.

## 12.1 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\_000samples 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="51-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 `AMFRegressor` functions, see: <https://riverml.xyz/0.19.0/api/forest/AMFRegressor/>.

## 12.2 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
from spotPython.utils.file import get_spot_tensorboard_path
import os
from spotPython.utils.file import get_experiment_name
experiment_name = get_experiment_name(prefix=PREFIX)
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    TENSORBOARD_CLEAN=True)
print(experiment_name)
```

51-river\_maans14\_2023-10-30\_23-38-06

### 💡 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 13.8.4 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.

## 12.3 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.

```
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})
```

## 12.4 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()
fun_control.update({"prep_model": prep_model})
```

## 12.5 SelectModel (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)
```

## 12.6 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])
```

::: {.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	

## 12.7 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 hyperparameter 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  $(\text{step}/n_{\text{steps}})^{\text{weight\_coeff}}$ , 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  
})
```

## 12.8 Calling the SPOT Function

### 12.8.1 Prepare the SPOT Parameters

The hyperparameter tuning configuration is stored in the `fun_control` dictionary. Since Spot can be used as an optimization algorithm with a similar interface as optimization algorithms from `scipy.optimize` [LINK], the bounds and variable types have to be specified explicitly. The `get_var_type`, `get_var_name`, and `get_bound_values` functions [SOURCE] implement the required functionality.

- Get types and variable names as well as lower and upper bounds for the hyperparameters, so that they can be passed to the Spot function.

```
from spotPython.hyperparameters.values import (  
    get_var_type,  
    get_var_name,  
    get_bound_values  
)  
var_type = get_var_type(fun_control)  
var_name = get_var_name(fun_control)  
lower = get_bound_values(fun_control, "lower")  
upper = get_bound_values(fun_control, "upper")
```

### 12.8.2 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)
```

### 12.8.3 Run the Spot Optimizer

The class `Spot` [SOURCE] is the hyperparameter tuning workhorse. It is initialized with the following parameters:

- `fun`: the objective function
- `lower`: lower bounds of the hyperparameters
- `upper`: upper bounds of the hyperparameters
- `fun_evals`: number of function evaluations
- `max_time`: maximum time in seconds
- `tolerance_x`: tolerance for the hyperparameters
- `var_type`: variable types of the hyperparameters
- `var_name`: variable names of the hyperparameters
- `show_progress`: show progress bar
- `fun_control`: dictionary with control parameters for the objective function
- `design_control`: dictionary with control parameters for the initial design
- `surrogate_control`: dictionary with control parameters for the surrogate model

 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.spot import spot
from math import inf
spot_tuner = spot.Spot(fun=fun,
                       lower = lower,
                       upper = upper,
                       fun_evals = inf,
                       max_time = MAX_TIME,
                       tolerance_x = np.sqrt(np.spacing(1)),
                       var_type = var_type,
                       var_name = var_name,
                       show_progress= True,
                       fun_control = fun_control,
                       design_control={"init_size": INIT_SIZE},
                       surrogate_control={"noise": False,
                                         "cod_type": "norm",
                                         "min_theta": -4,
                                         "max_theta": 3,
                                         "n_theta": len(var_name),
```

```
"model_fun_evals": 10_000})  
spot_tuner.run(X_start=X_start)  
  
spotPython tuning: 2.6449122564697376 [#####] 100.00% Done...  
  
<spotPython.spot.spot.Spot at 0x2b9c82350>
```

#### 12.8.4 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.file 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  $\theta$  of the Kriging surrogate [\[SOURCE\]](#) is plotted against the number of optimization steps.

#### 12.8.5 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  
save_pickle(spot_tuner, experiment_name)
```

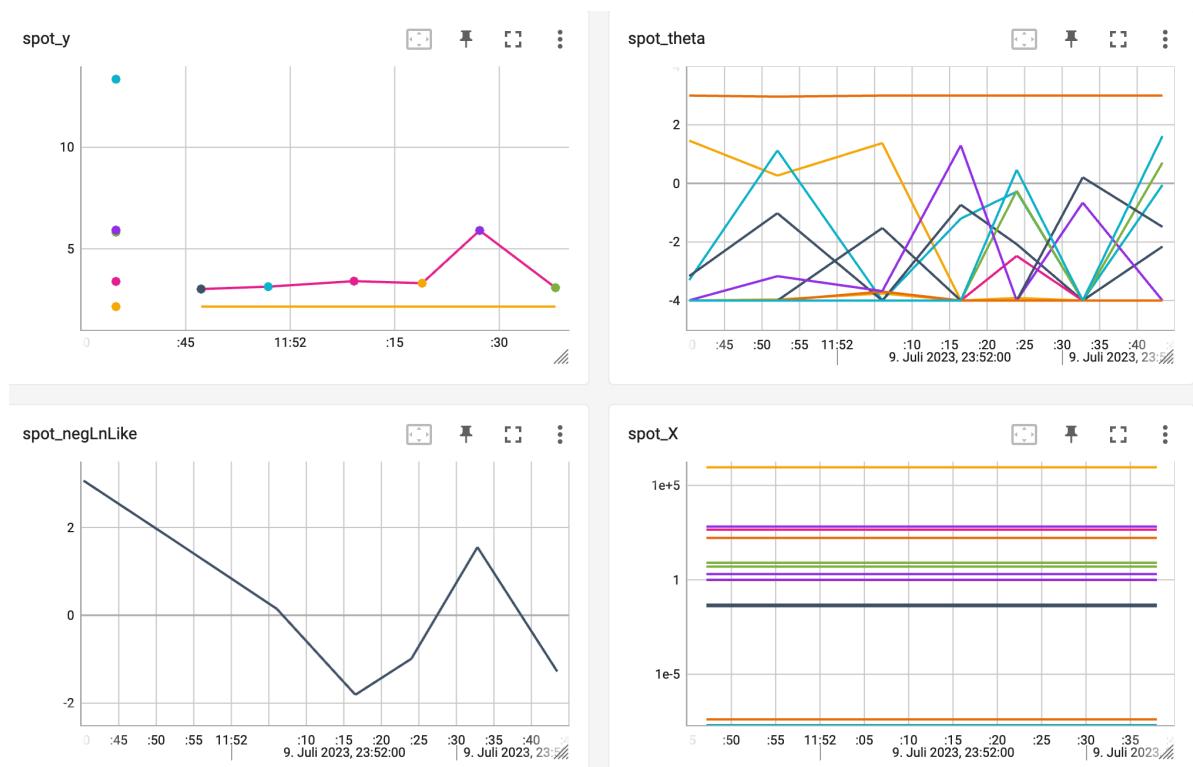
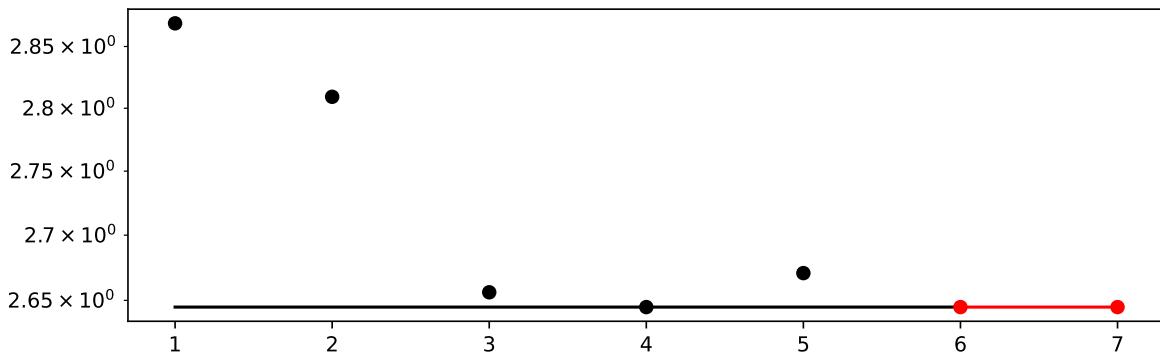


Figure 12.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

```
from spotPython.utils.file import load_pickle  
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 performance values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress(log_y=True, filename=".//figures/" + experiment_name+_progress.pdf
```



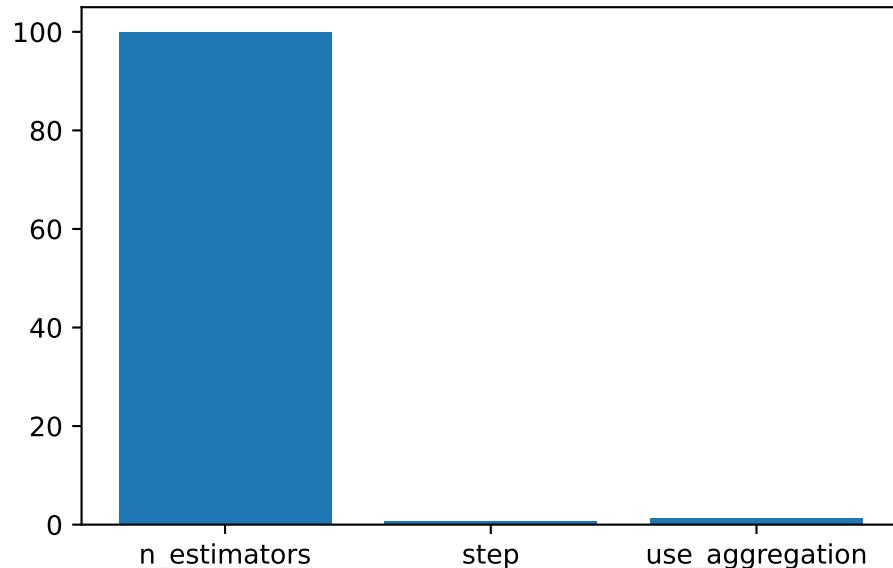
Results can also be printed in tabular form.

```
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	82.0	None
step	float	1.0	0.1	10	1.5323826923617296	None
use_aggregation	factor	1.0	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+_imp
```



## 12.9 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})
```

## 12.10 Get Default Hyperparameters

The default hyperparameters, which will be used for a comparison 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)
```

**i** 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:

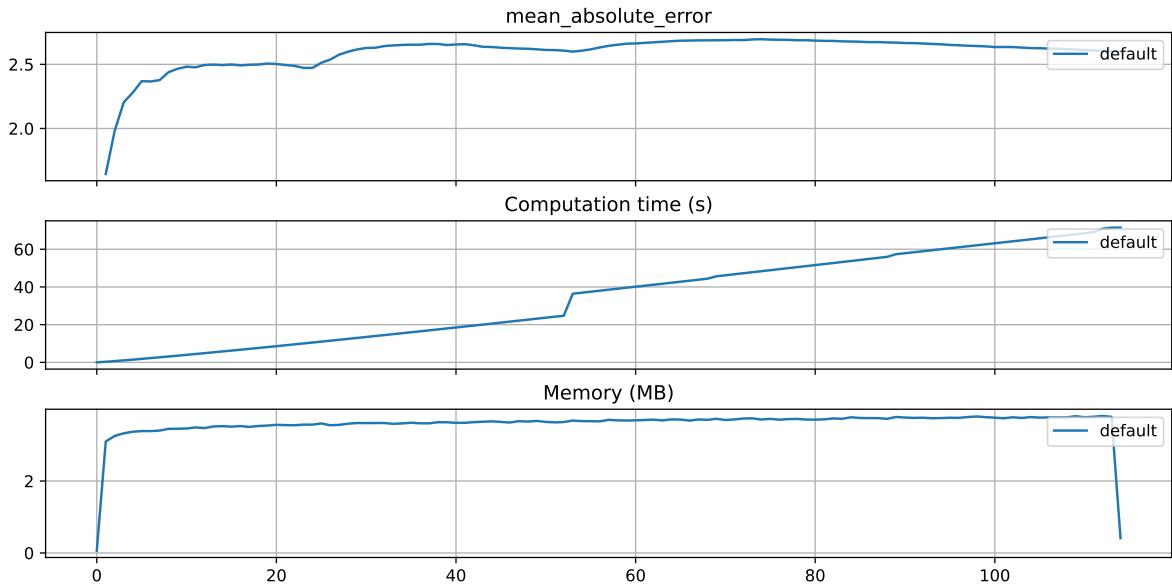
```
from spotRiver.evaluation.eval_bml import eval_oml_horizon

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., `scae` (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_predictions
df_labels=["default"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default], log_y=False, df_labels=df_labels)
```

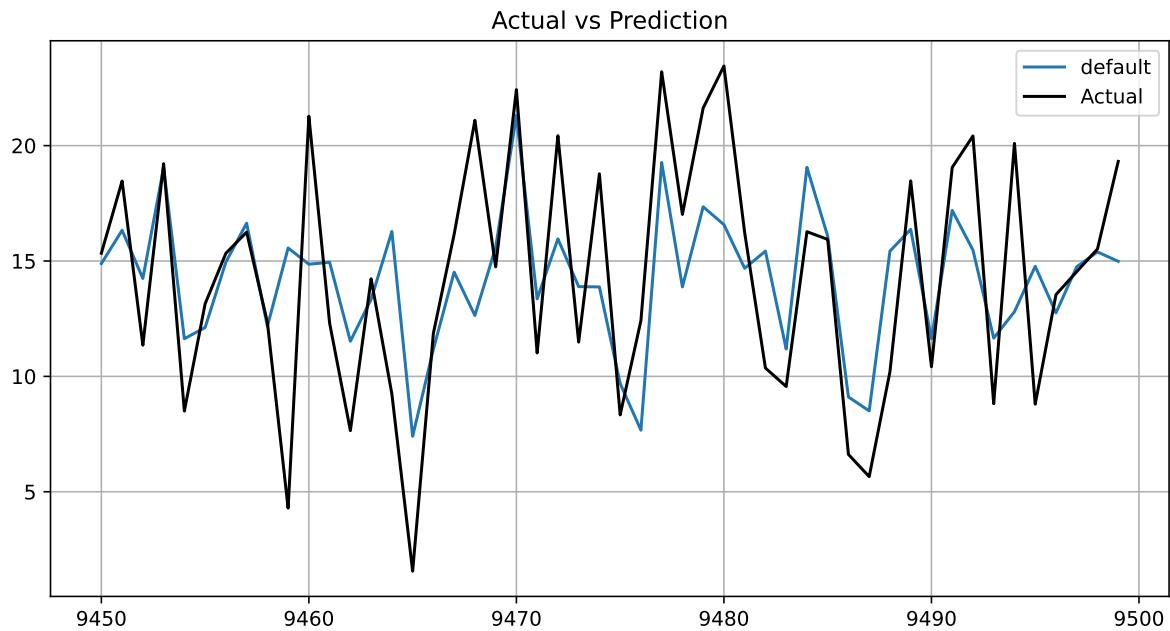


### 12.10.1 Show Predictions

- 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_col)
```



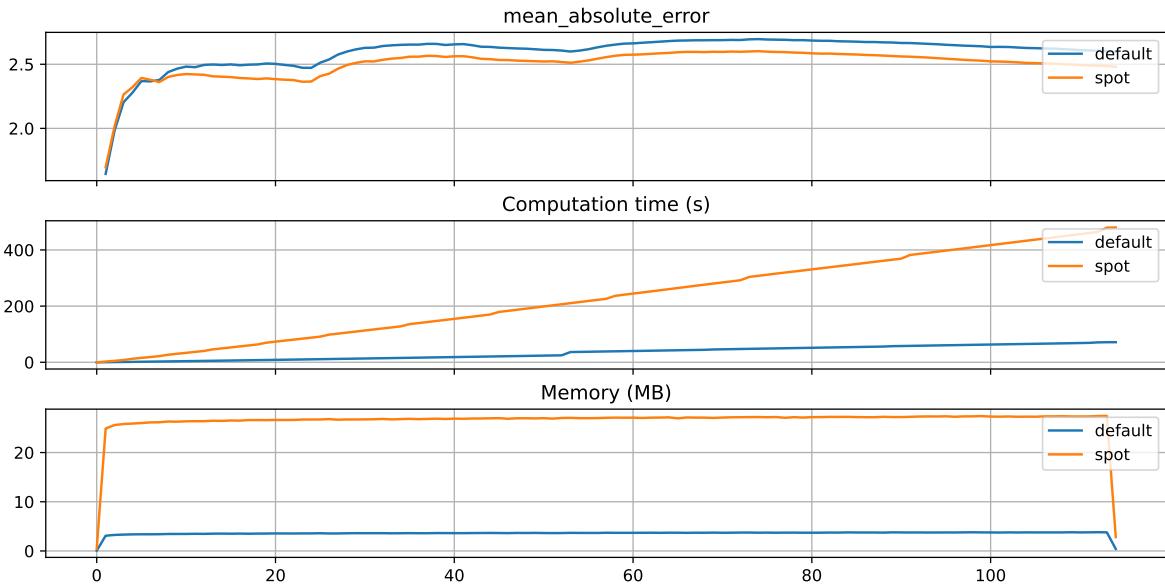
## 12.11 Get SPOT Results

In a similar way, we can obtain the hyperparameters found by `spotPython`.

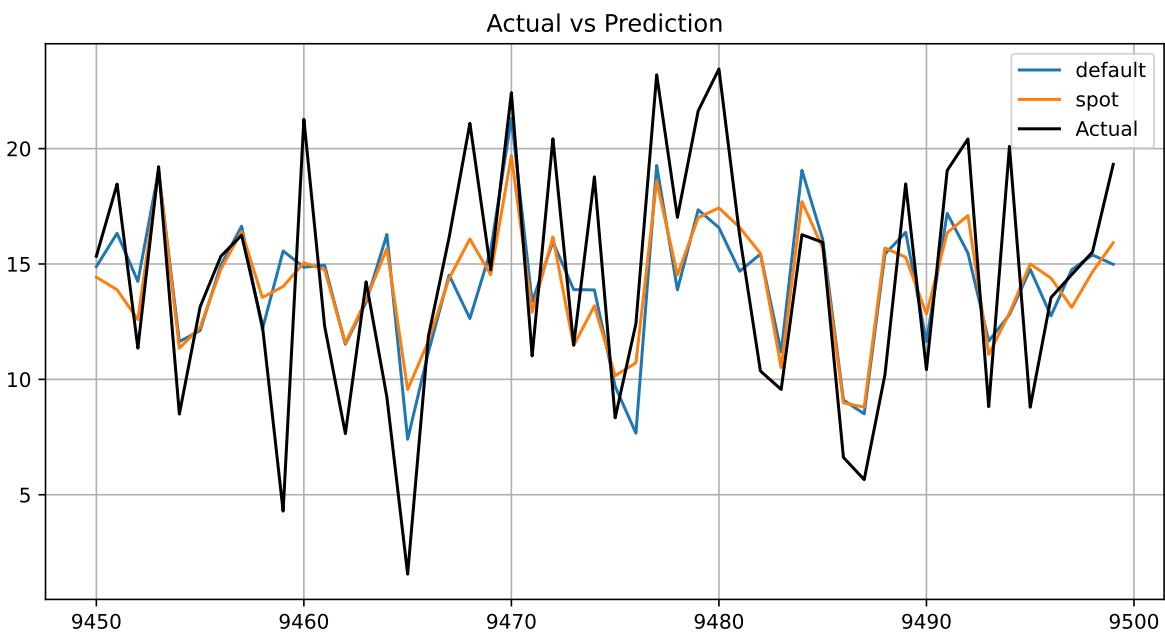
```
from spotPython.hyperparameters.values import get_one_core_model_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
model_spot = get_one_core_model_from_X(X, fun_control)

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_la
```



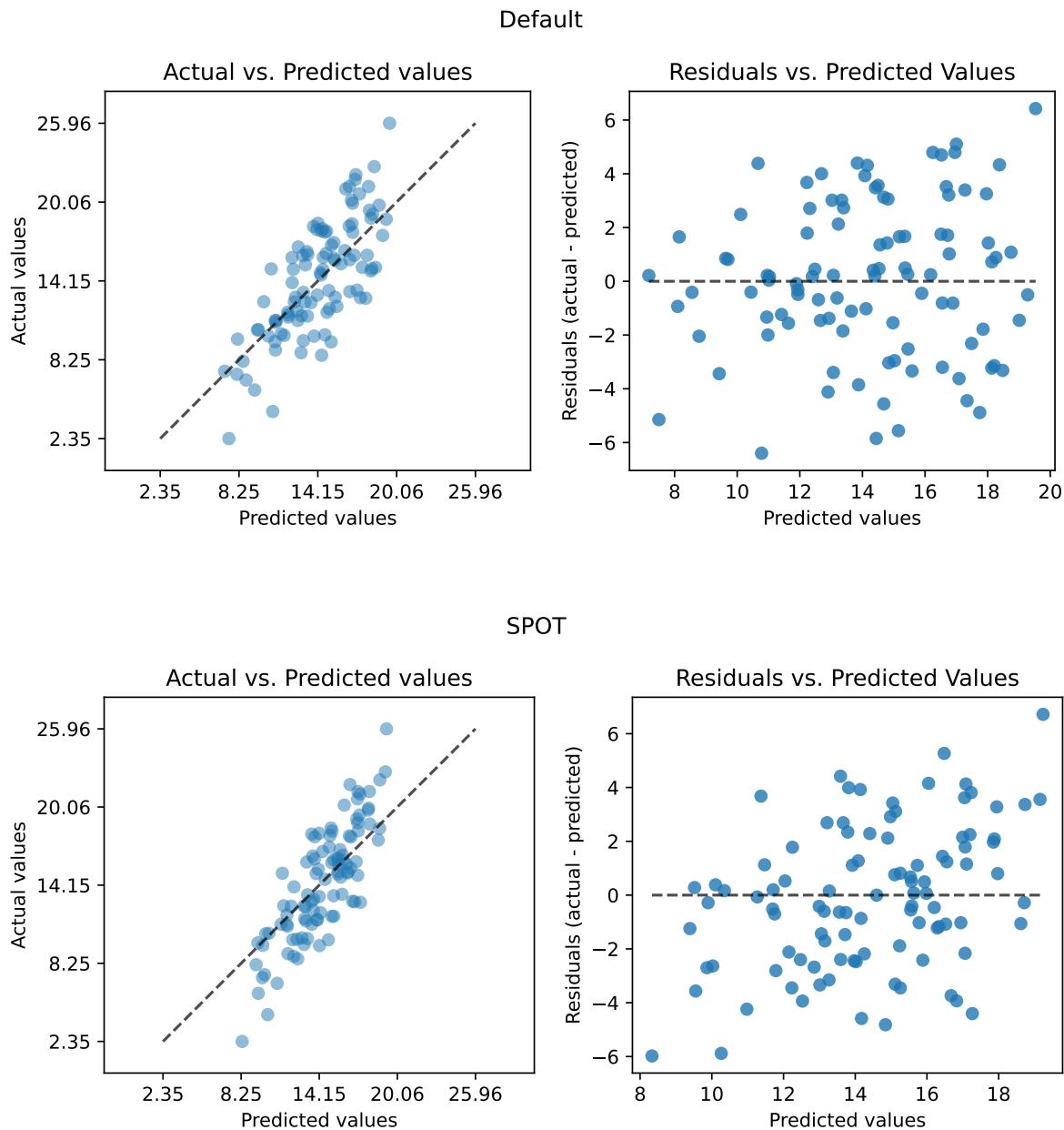
```
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b], df_true_spot[a:b]], targ
```



```

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["Predictions"])
plot_actual_vs_predicted(y_test=df_true_spot[target_column], y_pred=df_true_spot["Predictions"])

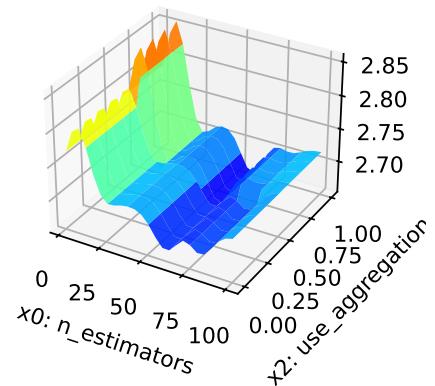
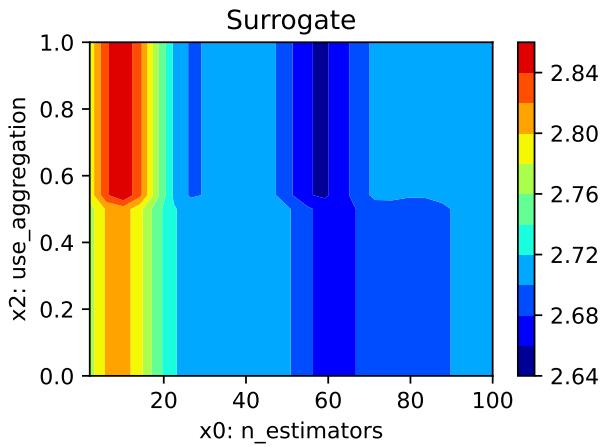
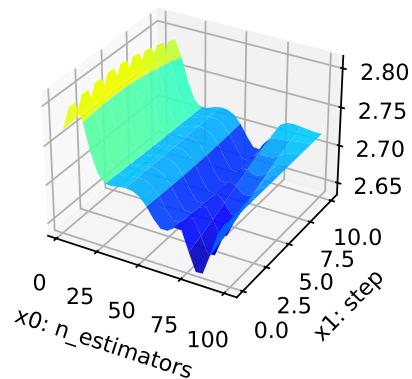
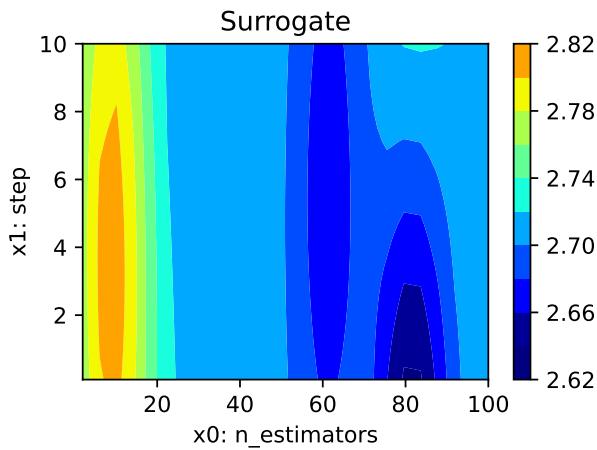
```

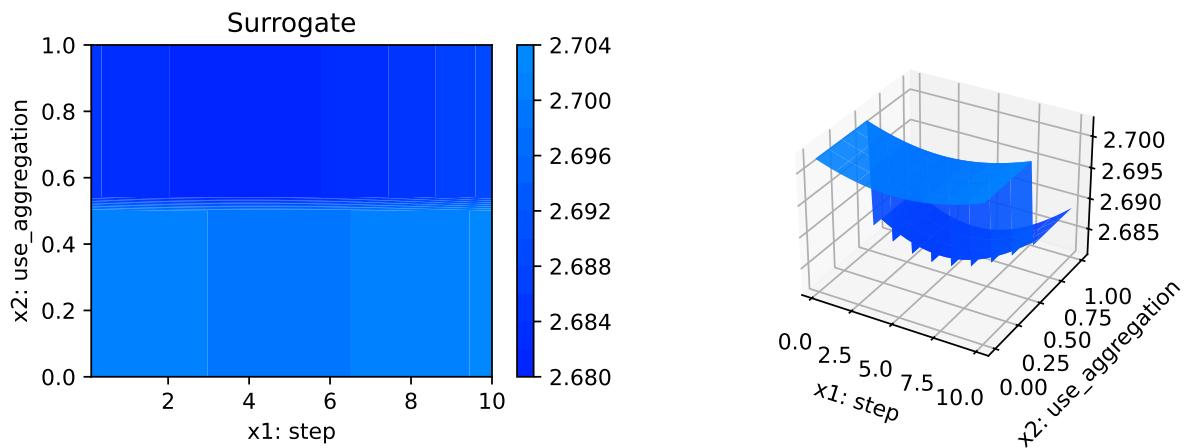


## 12.12 Detailed Hyperparameter Plots

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

```
n_estimators: 100.0
step: 0.6932535182593229
use_aggregation: 1.4139986387491685
```





## 12.13 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

## 12.14 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)
```

# 13 river Hyperparameter Tuning: Mondrian Tree Classifier with Bananas Data

This chapter demonstrates hyperparameter tuning for `river`'s Mondrian Tree Classifier with the bananas data set [\[SOURCE\]](#).

## 13.1 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\_000samples 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="52-river"
```

## 13.2 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
from spotPython.utils.file import get_spot_tensorboard_path
import os
from spotPython.utils.file import get_experiment_name
experiment_name = get_experiment_name(prefix=PREFIX)
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    TENSORBOARD_CLEAN=True)
print(experiment_name)
```

52-river\_maans14\_2023-10-31\_00-06-01

💡 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 13.8.4 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.

## 13.3 Load Data: The Bananas Dataset

We will use the Bananas dataset [SOURCE].

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.
- `n_train`: The number of samples used for training.

```

horizon = 7*24
n_samples = 5300
n_train = 1_000

from river.datasets import Bananas
import pandas as pd
dataset = Bananas(
)

dataset.n_features

```

2

- 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)
df.describe(include="all")

```

	1	2	y
count	5.300000e+03	5.300000e+03	5300
unique	NaN	NaN	2
top	NaN	NaN	False
freq	NaN	NaN	2924
mean	-4.150943e-09	-1.886792e-10	NaN
std	1.000000e+00	1.000000e+00	NaN
min	-3.089839e+00	-2.385937e+00	NaN
25%	-7.533490e-01	-9.139027e-01	NaN
50%	-1.523150e-02	-3.721500e-02	NaN
75%	7.818312e-01	8.221040e-01	NaN
max	2.813360e+00	3.194302e+00	NaN

- Add column names `x1` until `x2` to the first 2 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.

```

df.columns = [f"x{i}" for i in range(1, dataset.n_features+1)] + ["y"]
# map the target from False to integer 0 and from True to integer 1
df["y"] = df["y"].astype(int)

fun_control.update({"train": df[:n_train],
                     "test": df[n_train:],
                     "n_samples": n_samples,
                     "target_column": target_column})

```

## 13.4 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()
prep_model = None
fun_control.update({"prep_model": prep_model})

```

## 13.5 SelectModel (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 AMFClassifier
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=AMFClassifier,
                               fun_control=fun_control,
                               hyper_dict=RiverHyperDict,
                               filename=None)

```

## 13.6 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=[2,20])
modify_hyper_parameter_bounds(fun_control, "step", bounds=[0.5,2])

```

::: {.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	20	None
step	float	1	0.5	2	None
use_aggregation	factor	1	0	1	None
dirichlet	float	0.5	0.01	1	None

split_pure	factor	0	0	1	None	
------------	--------	---	---	---	------	--

## 13.7 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.

**i** 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 hyperparameter 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.

**i** 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  $(\text{step}/n_{\text{steps}})^{\text{weight\_coeff}}$ , 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 accuracy_score

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": accuracy_score
})
```

## 13.8 Calling the SPOT Function

### 13.8.1 Prepare the SPOT Parameters

The hyperparameter tuning configuration is stored in the `fun_control` dictionary. Since Spot can be used as an optimization algorithm with a similar interface as optimization algorithms from `scipy.optimize` [LINK], the bounds and variable types have to be specified explicitly. The `get_var_type`, `get_var_name`, and `get_bound_values` functions [SOURCE] implement the required functionality.

- Get types and variable names as well as lower and upper bounds for the hyperparameters, so that they can be passed to the Spot function.

```
from spotPython.hyperparameters.values import (
    get_var_type,
    get_var_name,
    get_bound_values
)
var_type = get_var_type(fun_control)
var_name = get_var_name(fun_control)
lower = get_bound_values(fun_control, "lower")
upper = get_bound_values(fun_control, "upper")
```

### 13.8.2 The Objective Function

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

```
from spotRiver.fun.hyperriver import HyperRiver
fun = HyperRiver(log_level=50).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)
```

### 13.8.3 Run the Spot Optimizer

The class `Spot` [SOURCE] is the hyperparameter tuning workhorse. It is initialized with the following parameters:

- `fun`: the objective function
- `lower`: lower bounds of the hyperparameters
- `upper`: upper bounds of the hyperparameters
- `fun_evals`: number of function evaluations
- `max_time`: maximum time in seconds
- `tolerance_x`: tolerance for the hyperparameters
- `var_type`: variable types of the hyperparameters
- `var_name`: variable names of the hyperparameters
- `show_progress`: show progress bar
- `fun_control`: dictionary with control parameters for the objective function
- `design_control`: dictionary with control parameters for the initial design
- `surrogate_control`: dictionary with control parameters for the surrogate model

 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.spot import spot
from math import inf
spot_tuner = spot.Spot(fun=fun,
                       lower = lower,
                       upper = upper,
                       fun_evals = inf,
                       max_time = MAX_TIME,
                       tolerance_x = np.sqrt(np.spacing(1)),
                       var_type = var_type,
                       var_name = var_name,
                       show_progress= True,
                       fun_control = fun_control,
                       design_control={"init_size": INIT_SIZE},
```

```

surrogate_control={"noise": False,
                   "cod_type": "norm",
                   "min_theta": -4,
                   "max_theta": 3,
                   "n_theta": len(var_name),
                   "model_fun_evals": 10_000},
                   log_level=50)

spot_tuner.run(X_start=X_start)

spotPython tuning: -1.6745422448173608 [##-----] 19.92%
spotPython tuning: -1.6745422448173608 [#####----] 46.53%
spotPython tuning: -1.6745422448173608 [#####----] 52.14%
spotPython tuning: -1.6745422448173608 [#####----] 59.89%
spotPython tuning: -1.6745422448173608 [#####----] 81.57%
spotPython tuning: -1.6745422448173608 [#####----] 100.00% Done...
<spotPython.spot.spot at 0x29a8c2f80>

```

### 13.8.4 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.file 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  $\theta$  of the Kriging surrogate [SOURCE] is plotted against the number of optimization steps.

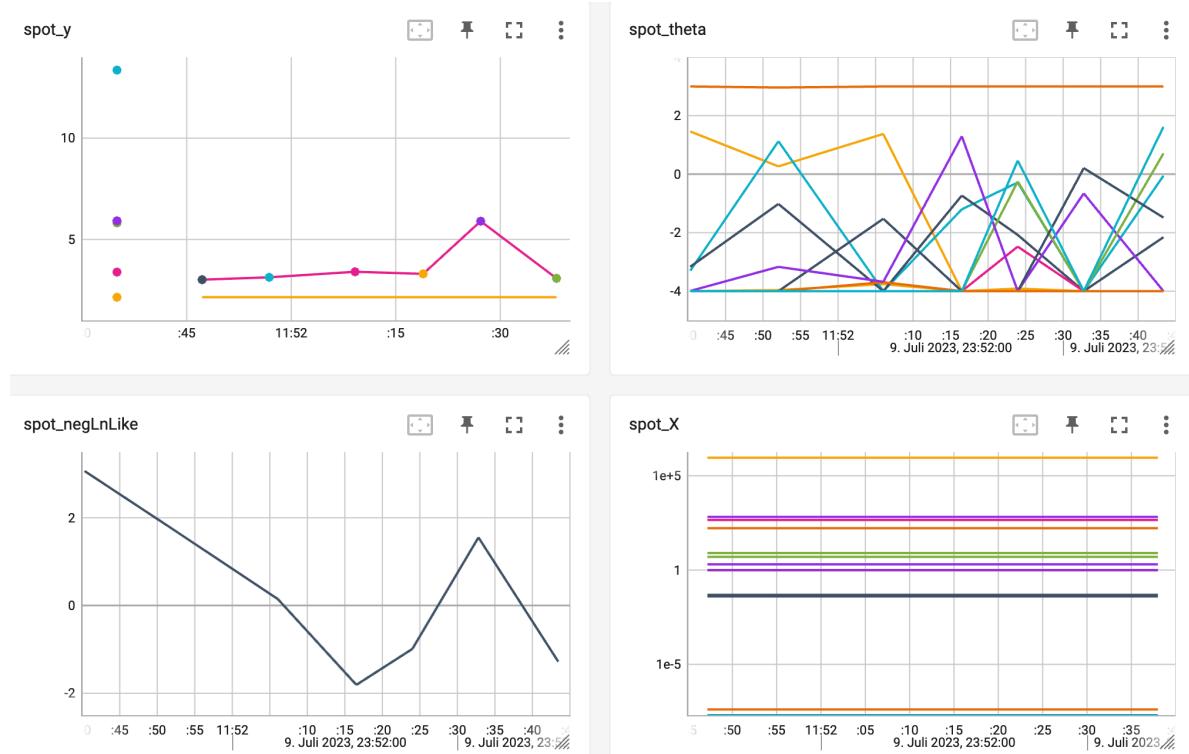


Figure 13.1: TensorBoard visualization of the `spotPython` optimization process and the surrogate model.

### 13.8.5 Results

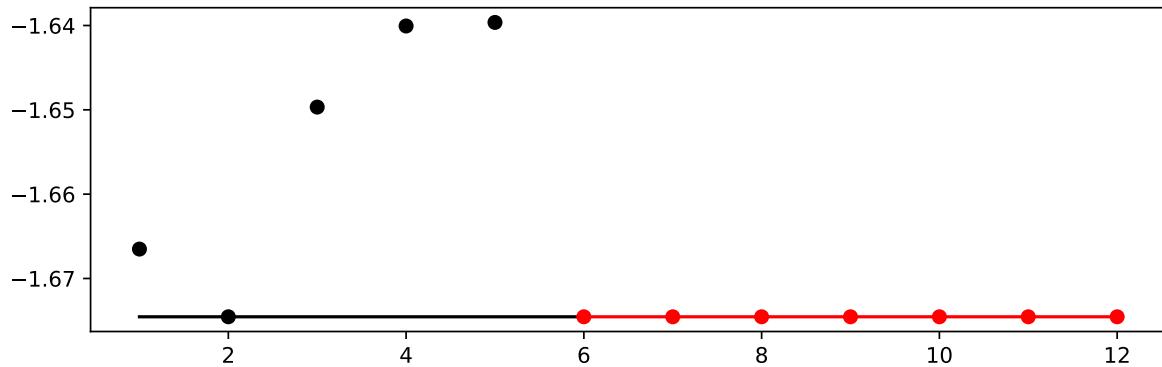
After the hyperparameter tuning run is finished, the results can be saved and reloaded with the following commands:

```
from spotPython.utils.file import save_pickle  
save_pickle(spot_tuner, experiment_name)
```

```
from spotPython.utils.file import load_pickle
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 performance values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress(log_y=False, filename=".//figures/" + experiment_name+"_progress.p
```



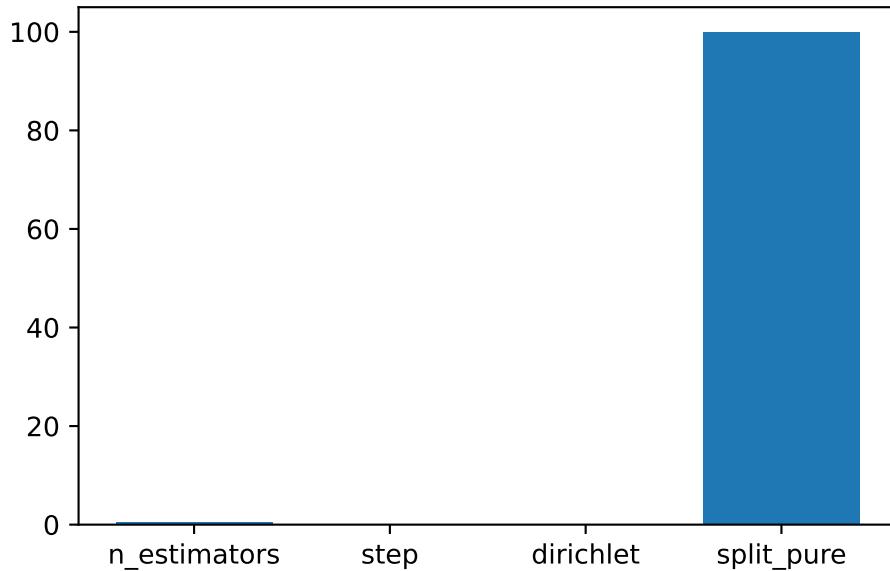
Results can also be printed in tabular form.

```
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	20.0	14.0	None
step	float	1.0	0.5	2.0	0.7838536943593333	None
use_aggregation	factor	1.0	0.0	1.0	1.0	None
dirichlet	float	0.5	0.01	1.0	0.7654943814816635	None
split_pure	factor	0.0	0.0	1.0	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+"_imp
```



## 13.9 Get Default Hyperparameters

The default hyperparameters, which will be used for a comparison 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)
```

**i** 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"],
)

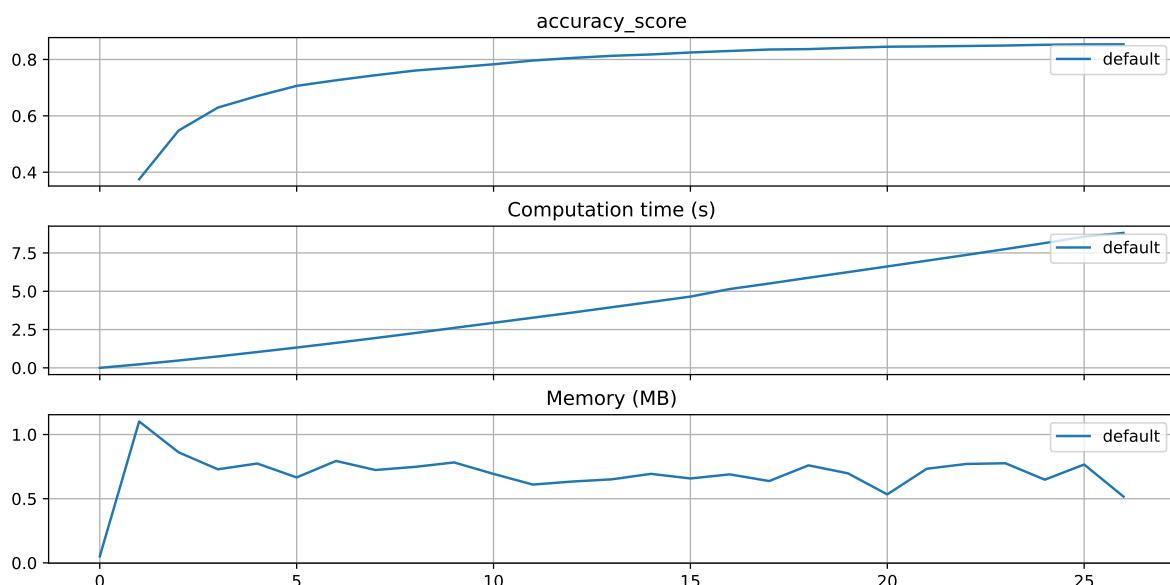
```

The three performance criteria, i.e., space (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)

```



### 13.9.1 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

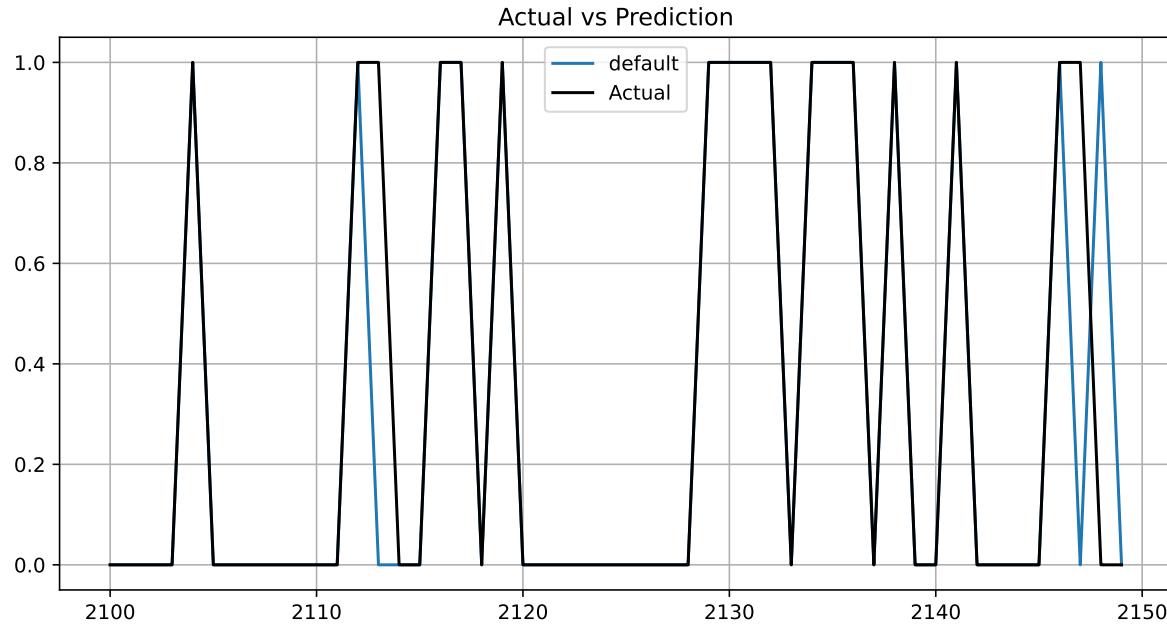
```

```

b = int(m/2)

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

```



### 13.10 Get SPOT Results

In a similar way, we can obtain the hyperparameters found by `spotPython`.

```

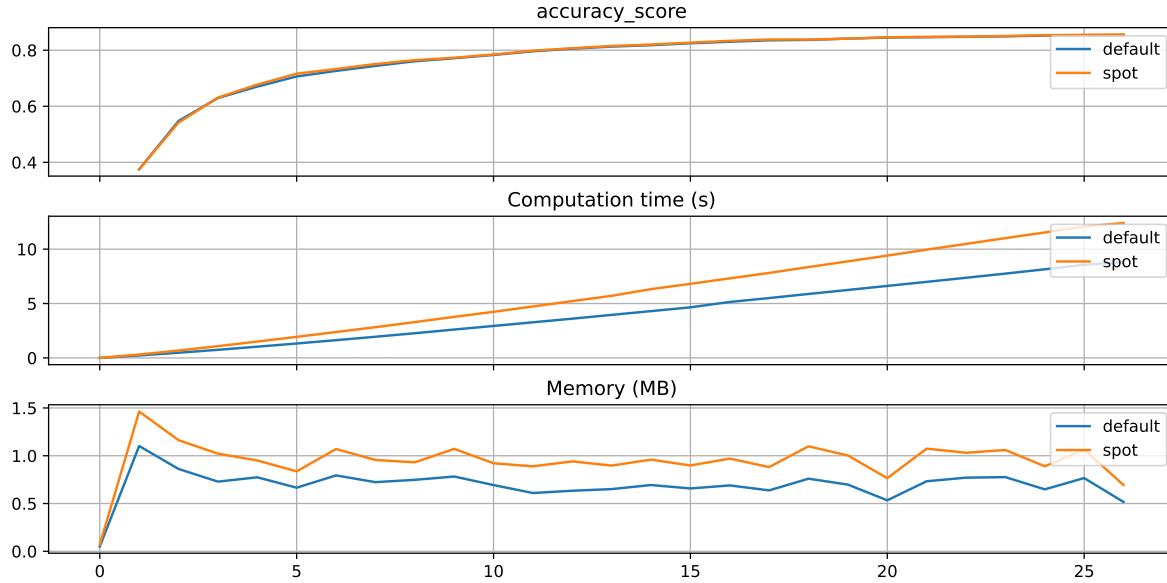
from spotPython.hyperparameters.values import get_one_core_model_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
model_spot = get_one_core_model_from_X(X, fun_control)

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_la
```



```
# plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b], df_true_spot[a:b]], ta
# 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["P
# plot_actual_vs_predicted(y_test=df_true_spot[target_column], y_pred=df_true_spot["Predic
```

## 13.11 Compare Predictions

```
from sklearn.model_selection import cross_val_predict
import matplotlib.pyplot as plt
from sklearn.base import BaseEstimator
from sklearn.metrics import PredictionErrorDisplay
from sklearn.metrics import RocCurveDisplay
from sklearn.metrics import ConfusionMatrixDisplay
from spotPython.utils.convert import get_Xy_from_df
```

```

from typing import Any, Dict, List, Union
import pandas as pd

def plot_roc(
    model_list: List[pd.DataFrame],
    alpha: float = 0.8,
    model_names: List[str] = None,
) -> None:
    """
    Plot ROC curve for a list of models.

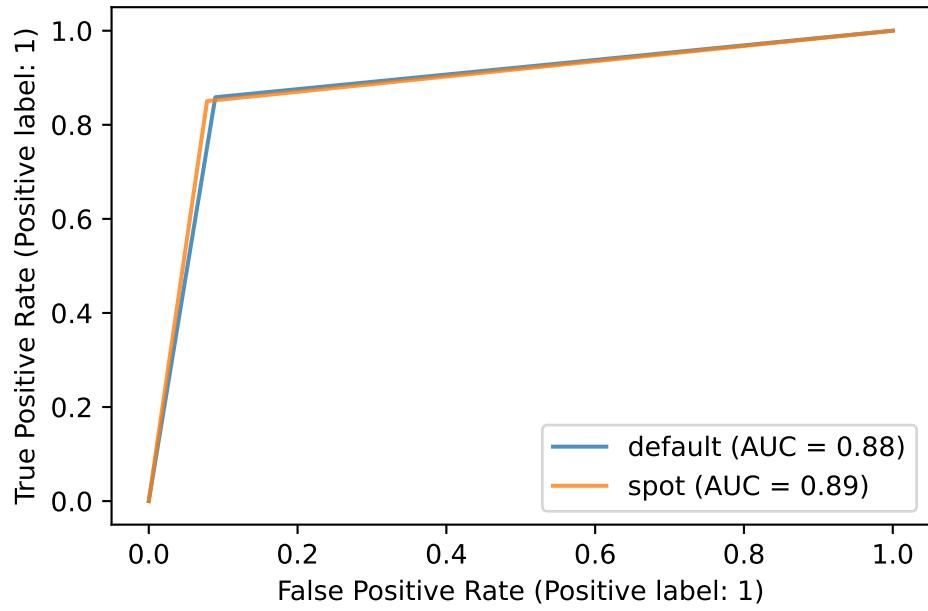
    Args:
        model_list: List of models.
        alpha: Transparency of the plotted lines.
        model_names: List of model names.

    Returns:
        None
    """

    Examples:
        >>> from spotPython.utils.file import load_pickle
        >>> spot_tuner = load_pickle("spot_tuner")
        >>> plot_roc(model_list=[spot_tuner.df_eval], model_names=["SPOT"])
    """
    ax = plt.gca()
    for i, df in enumerate(model_list):
        y_test=df[target_column]
        y_pred=df["Prediction"]
        if model_names is not None:
            model_name = model_names[i]
        else:
            model_name = None
        RocCurveDisplay.from_predictions(y_test, y_pred, ax=ax, alpha=alpha, name=model_name)
    plt.show()

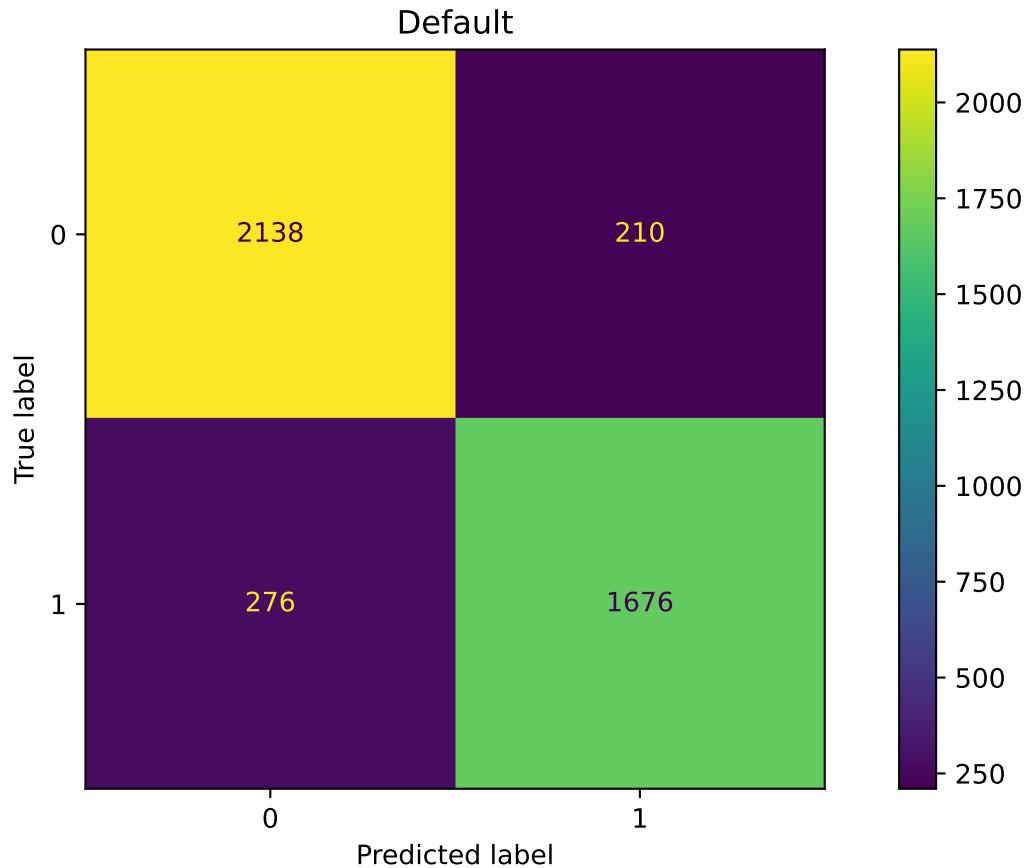
    plot_roc([df_true_default, df_true_spot], model_names=["default", "spot"])

```

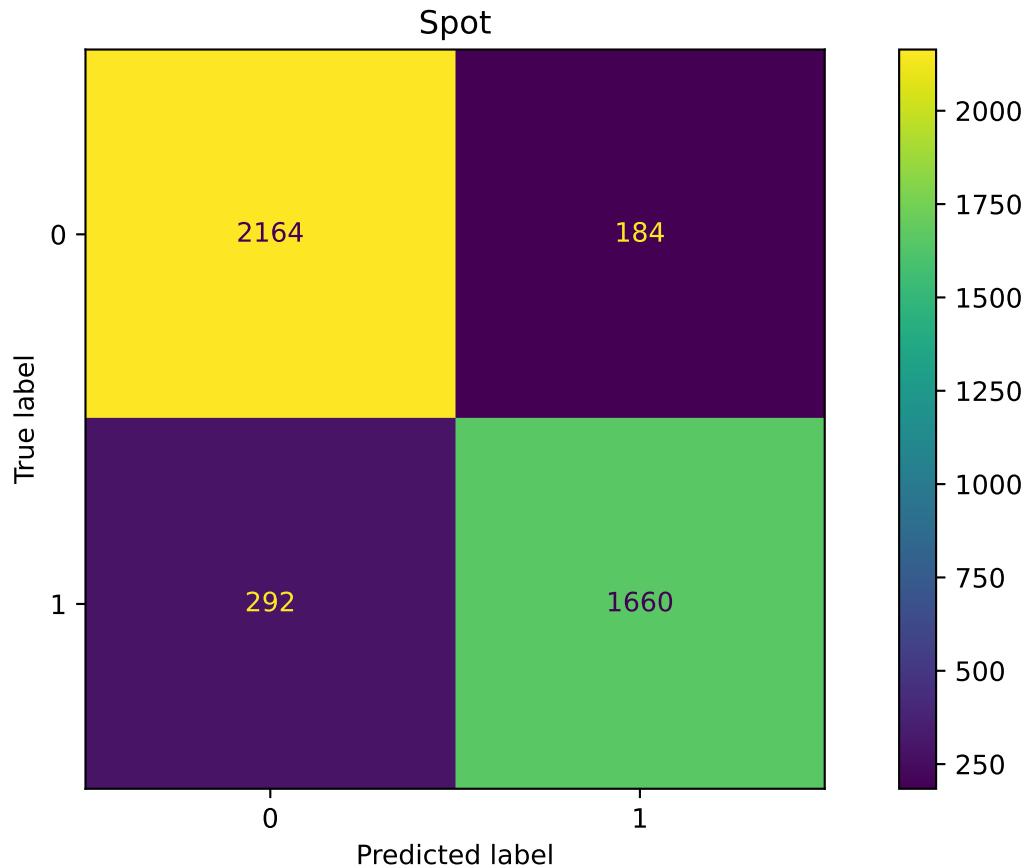


```
def plot_confusion_matrix(df, target_names=None, title=None):
    """
    Plotting a confusion matrix
    """
    y_test=df[target_column]
    pred=df["Prediction"]
    fig, ax = plt.subplots(figsize=(10, 5))
    ConfusionMatrixDisplay.from_predictions(y_test, pred, ax=ax)
    if target_names is not None:
        ax.xaxis.set_ticklabels(target_names)
        ax.yaxis.set_ticklabels(target_names)
    if title is not None:
        _ = ax.set_title(title)

plot_confusion_matrix(df=df_true_default, title="Default")
```



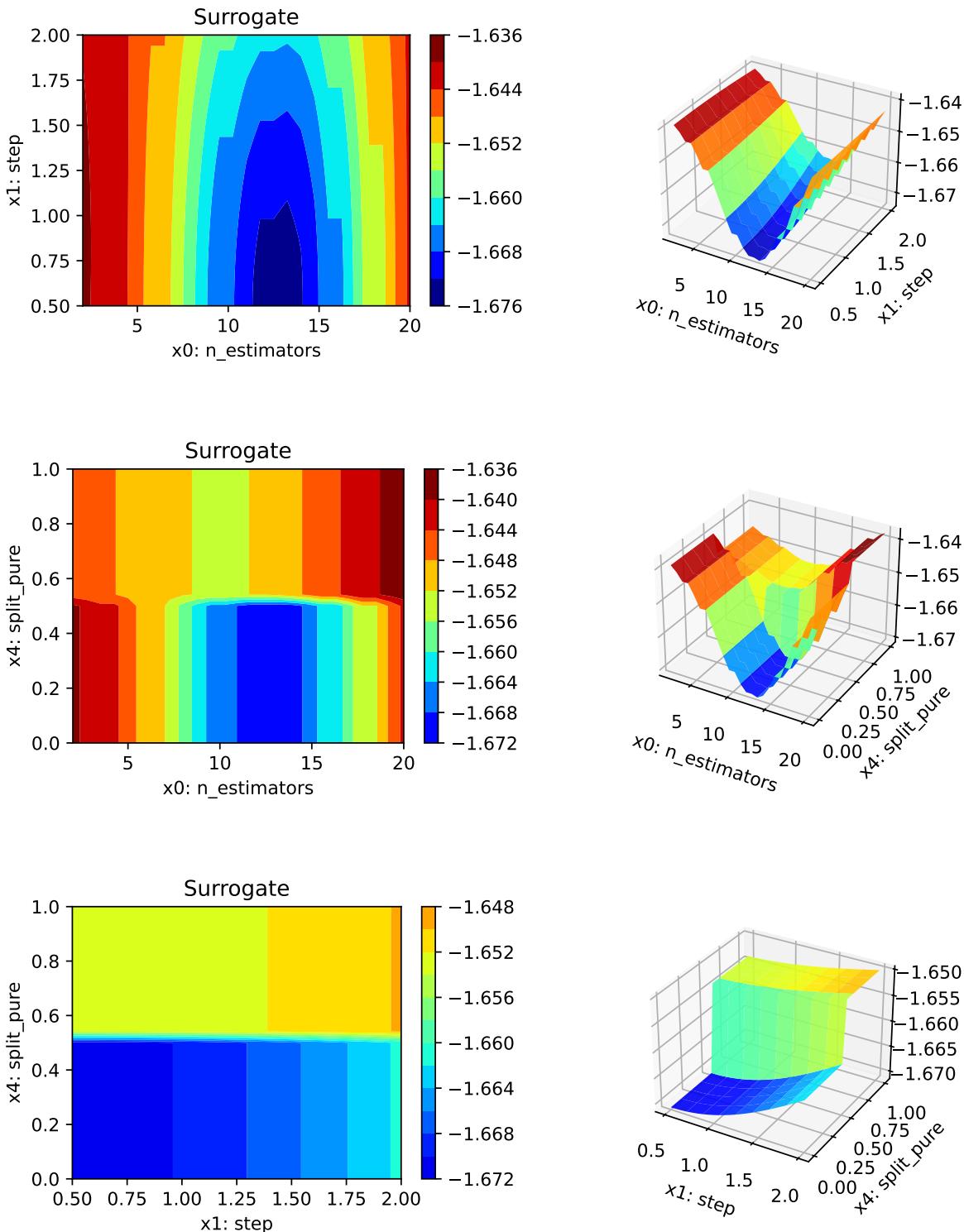
```
plot_confusion_matrix(df=df_true_spot, title="Spot")
```



### 13.12 Detailed Hyperparameter Plots

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

```
n_estimators: 0.5530313002509699
step: 0.031184757672555328
split_pure: 100.0
```



### 13.13 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

### 13.14 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)
```

# 14 HPT: PyTorch With spotPython and Ray Tune on CIFAR10

In this tutorial, we will show how `spotPython` can be integrated into the PyTorch training workflow. It is based on the tutorial “Hyperparameter Tuning with Ray Tune” from the PyTorch documentation (PyTorch 2023a), which is an extension of the tutorial “Training a Classifier” (PyTorch 2023b) for training a CIFAR10 image classifier.

 Note: PyTorch and Lightning

Instead of using the PyTorch interface directly as explained in this chapter, we recommend using the PyTorch Lightning interface. The PyTorch Lightning interface is explained in Chapter 19

A typical hyperparameter tuning process with `spotPython` consists of the following steps:

1. Loading the data (training and test datasets), see Section 14.3.
2. Specification of the preprocessing model, see Section 14.4. This model is called `prep_model` (“preparation” or pre-processing). The information required for the hyperparameter tuning is stored in the dictionary `fun_control`. Thus, the information needed for the execution of the hyperparameter tuning is available in a readable form.
3. Selection of the machine learning or deep learning model to be tuned, see Section 14.5. This is called the `core_model`. Once the `core_model` is defined, then the associated hyperparameters are stored in the `fun_control` dictionary. First, the hyperparameters of the `core_model` are initialized with the default values of the `core_model`. As default values we use the default values contained in the `spotPython` package for the algorithms of the `torch` package.
4. Modification of the default values for the hyperparameters used in `core_model`, see Section 14.6.0.1. This step is optional.
  1. numeric parameters are modified by changing the bounds.
  2. categorical parameters are modified by changing the categories (“levels”).
5. Selection of target function (loss function) for the optimizer, see Section 14.7.5.
6. Calling SPOT with the corresponding parameters, see Section 14.8.4. The results are stored in a dictionary and are available for further analysis.
7. Presentation, visualization and interpretation of the results, see Section 14.10.

`spotPython` can be installed via pip<sup>1</sup>.

```
!pip install spotPython
```

- Uncomment the following lines if you want to for (re-)installation the latest version of `spotPython` from GitHub.

```
# import sys
# !{sys.executable} -m pip install --upgrade build
# !{sys.executable} -m pip install --upgrade --force-reinstall spotPython
```

Results that refer to the `Ray Tune` package are taken from [https://PyTorch.org/tutorials/beginner/hyperparameter\\_tuning\\_tutorial.html](https://PyTorch.org/tutorials/beginner/hyperparameter_tuning_tutorial.html)<sup>2</sup>.

## 14.1 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.

 Note: Device selection

- The device can be selected by setting the variable `DEVICE`.
- Since we are using a simple neural net, the setting "`cpu`" is preferred (on Mac).
- If you have a GPU, you can use "`cuda:0`" instead.
- If `DEVICE` is set to "`auto`" or `None`, `spotPython` will automatically select the device.
  - This might result in "`mps`" on Macs, which is not the best choice for simple neural nets.

<sup>1</sup>Alternatively, the source code can be downloaded from GitHub: <https://github.com/sequential-parameter-optimization/spotPython>.

<sup>2</sup>We were not able to install `Ray Tune` on our system. Therefore, we used the results from the PyTorch tutorial.

```

MAX_TIME = 1
INIT_SIZE = 5
DEVICE = "auto" # "cpu"
PREFIX = "14-torch"

from spotPython.utils.device import getDevice
DEVICE = getDevice(DEVICE)
print(DEVICE)

```

mps

```

import warnings
warnings.filterwarnings("ignore")

```

## 14.2 Step 2: Initialization of the fun\_control Dictionary

spotPython uses a Python dictionary for storing the information required for the hyperparameter tuning process. This dictionary is called `fun_control` and is initialized with the function `fun_control_init`. The function `fun_control_init` returns a skeleton dictionary. The dictionary is filled with the required information for the hyperparameter tuning process. It stores the hyperparameter tuning settings, e.g., the deep learning network architecture that should be tuned, the classification (or regression) problem, and the data that is used for the tuning. The dictionary is used as an input for the SPOT function.

```

from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_experiment_name, get_spot_tensorboard_path
from spotPython.utils.device import getDevice

experiment_name = get_experiment_name(prefix=PREFIX)

fun_control = fun_control_init(
    task="classification",
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    device=DEVICE,
)

```

## 14.3 Step 3: PyTorch Data Loading

The data loading process is implemented in the same manner as described in the Section “Data loaders” in PyTorch (2023a). The data loaders are wrapped into the function `load_data_cifar10` which is identical to the function `load_data` in PyTorch (2023a). A global data directory is used, which allows sharing the data directory between different trials. The method `load_data_cifar10` is part of the `spotPython` package and can be imported from `spotPython.data.torchdata`.

In the following step, the test and train data are added to the dictionary `fun_control`.

```
from spotPython.data.torchdata import load_data_cifar10
train, test = load_data_cifar10()
n_samples = len(train)
# add the dataset to the fun_control
fun_control.update({
    "train": train,
    "test": test,
    "n_samples": n_samples})
```

Files already downloaded and verified

Files already downloaded and verified

## 14.4 Step 4: Specification of the Preprocessing Model

After the training and test data are specified and added to the `fun_control` dictionary, `spotPython` allows the specification of a data preprocessing pipeline, e.g., for the scaling of the data or for the one-hot encoding of categorical variables. The preprocessing model is called `prep_model` (“preparation” or pre-processing) and includes steps that are not subject to the hyperparameter tuning process. The preprocessing model is specified in the `fun_control` dictionary. The preprocessing model can be implemented as a `sklearn` pipeline. The following code shows a typical preprocessing pipeline:

```
categorical_columns = ["cities", "colors"]
one_hot_encoder = OneHotEncoder(handle_unknown="ignore",
                                sparse_output=False)
prep_model = ColumnTransformer(
    transformers=[("categorical", one_hot_encoder, categorical_columns),
```

```
    ] ,  
    remainder=StandardScaler() ,  
)  
)
```

Because the Ray Tune (`ray[tune]`) hyperparameter tuning as described in PyTorch (2023a) does not use a preprocessing model, the preprocessing model is set to `None` here.

```
prep_model = None  
fun_control.update({"prep_model": prep_model})
```

## 14.5 Step 5: Select Model (algorithm) and core\_model\_hyper\_dict

The same neural network model as implemented in the section “Configurable neural network” of the PyTorch tutorial (PyTorch 2023a) is used here. We will show the implementation from PyTorch (2023a) in Section 14.5.0.1 first, before the extended implementation with `spotPython` is shown in Section 14.5.0.2.

### 14.5.0.1 Implementing a Configurable Neural Network With Ray Tune

We used the same hyperparameters that are implemented as configurable in the PyTorch tutorial. We specify the layer sizes, namely 11 and 12, of the fully connected layers:

```
class Net(nn.Module):  
    def __init__(self, l1=120, l2=84):  
        super(Net, self).__init__()  
        self.conv1 = nn.Conv2d(3, 6, 5)  
        self.pool = nn.MaxPool2d(2, 2)  
        self.conv2 = nn.Conv2d(6, 16, 5)  
        self.fc1 = nn.Linear(16 * 5 * 5, l1)  
        self.fc2 = nn.Linear(l1, l2)  
        self.fc3 = nn.Linear(l2, 10)  
  
    def forward(self, x):  
        x = self.pool(F.relu(self.conv1(x)))  
        x = self.pool(F.relu(self.conv2(x)))  
        x = x.view(-1, 16 * 5 * 5)  
        x = F.relu(self.fc1(x))  
        x = F.relu(self.fc2(x))
```

```

    x = self.fc3(x)
    return x

```

The learning rate, i.e., lr, of the optimizer is made configurable, too:

```
optimizer = optim.SGD(net.parameters(), lr=config["lr"], momentum=0.9)
```

#### 14.5.0.2 Implementing a Configurable Neural Network With spotPython

spotPython implements a class which is similar to the class described in the PyTorch tutorial. The class is called `Net_CIFAR10` and is implemented in the file `netcifar10.py`.

```

from torch import nn
import torch.nn.functional as F
import spotPython.torch.netcore as netcore

class Net_CIFAR10(netcore.Net_Core):
    def __init__(self, l1, l2, lr_mult, batch_size, epochs, k_folds, patience,
                 optimizer, sgd_momentum):
        super(Net_CIFAR10, self).__init__(
            lr_mult=lr_mult,
            batch_size=batch_size,
            epochs=epochs,
            k_folds=k_folds,
            patience=patience,
            optimizer=optimizer,
            sgd_momentum=sgd_momentum,
        )
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 11)
        self.fc2 = nn.Linear(l1, l2)
        self.fc3 = nn.Linear(l2, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))

```

```
x = F.relu(self.fc2(x))
x = self.fc3(x)
return x
```

#### 14.5.1 The Net\_Core class

Net\_CIFAR10 inherits from the class `Net_Core` which is implemented in the file `netcore.py`. It implements the additional attributes that are common to all neural network models. The `Net_Core` class is implemented in the file `netcore.py`. It implements hyperparameters as attributes, that are not used by the `core_model`, e.g.:

- optimizer (`optimizer`),
- learning rate (`lr`),
- batch size (`batch_size`),
- epochs (`epochs`),
- k\_folds (`k_folds`), and
- early stopping criterion “patience” (`patience`).

Users can add further attributes to the class. The class `Net_Core` is shown below.

```
from torch import nn

class Net_Core(nn.Module):
    def __init__(self, lr_mult, batch_size, epochs, k_folds, patience,
                 optimizer, sgd_momentum):
        super(Net_Core, self).__init__()
        self.lr_mult = lr_mult
        self.batch_size = batch_size
        self.epochs = epochs
        self.k_folds = k_folds
        self.patience = patience
        self.optimizer = optimizer
        self.sgd_momentum = sgd_momentum
```

#### 14.5.2 Comparison of the Approach Described in the PyTorch Tutorial With spotPython

Comparing the class `Net` from the PyTorch tutorial and the class `Net_CIFAR10` from `spotPython`, we see that the class `Net_CIFAR10` has additional attributes and does not inherit from `nn` directly. It adds an additional class, `Net_core`, that takes care of additional

attributes that are common to all neural network models, e.g., the learning rate multiplier `lr_mult` or the batch size `batch_size`.

`spotPython`'s `core_model` implements an instance of the `Net_CIFAR10` class. In addition to the basic neural network model, the `core_model` can use these additional attributes. `spotPython` provides methods for handling these additional attributes to guarantee 100% compatibility with the PyTorch classes. The method `add_core_model_to_fun_control` adds the hyperparameters and additional attributes to the `fun_control` dictionary. The method is shown below.

```
from spotPython.torch.netcifar10 import Net_CIFAR10
from spotPython.data.torch_hyper_dict import TorchHyperDict
from spotPython.hyperparameters.values import add_core_model_to_fun_control
core_model = Net_CIFAR10
add_core_model_to_fun_control(core_model=core_model,
                             fun_control=fun_control,
                             hyper_dict=TorchHyperDict,
                             filename=None)
```

### 14.5.3 The Search Space: Hyperparameters

In Section 14.5.4, we first describe how to configure the search space with `ray[tune]` (as shown in PyTorch (2023a)) and then how to configure the search space with `spotPython` in -14.

### 14.5.4 Configuring the Search Space With Ray Tune

Ray Tune's search space can be configured as follows (PyTorch 2023a):

```
config = {
    "l1": tune.sample_from(lambda _: 2**np.random.randint(2, 9)),
    "l2": tune.sample_from(lambda _: 2**np.random.randint(2, 9)),
    "lr": tune.loguniform(1e-4, 1e-1),
    "batch_size": tune.choice([2, 4, 8, 16])
}
```

The `tune.sample_from()` function enables the user to define sample methods to obtain hyperparameters. In this example, the `l1` and `l2` parameters should be powers of 2 between 4 and 256, so either 4, 8, 16, 32, 64, 128, or 256. The `lr` (learning rate) should be uniformly sampled between 0.0001 and 0.1. Lastly, the batch size is a choice between 2, 4, 8, and 16.

At each trial, `ray[tune]` will randomly sample a combination of parameters from these search spaces. It will then train a number of models in parallel and find the best performing one

among these. `ray[tune]` uses the `ASHAScheduler` which will terminate bad performing trials early.

## 14.5.5 Configuring the Search Space With `spotPython`

### 14.5.5.1 The `hyper_dict` Hyperparameters for the Selected Algorithm

`spotPython` uses JSON files for the specification of the hyperparameters. Users can specify their individual JSON files, or they can use the JSON files provided by `spotPython`. The JSON file for the `core_model` is called `torch_hyper_dict.json`.

In contrast to `ray[tune]`, `spotPython` can handle numerical, boolean, and categorical hyperparameters. They can be specified in the JSON file in a similar way as the numerical hyperparameters as shown below. Each entry in the JSON file represents one hyperparameter with the following structure: `type`, `default`, `transform`, `lower`, and `upper`.

```
"factor_hyperparameter": {  
    "levels": ["A", "B", "C"],  
    "type": "factor",  
    "default": "B",  
    "transform": "None",  
    "core_model_parameter_type": "str",  
    "lower": 0,  
    "upper": 2},
```

The corresponding entries for the `core_model` class are shown below.

```
fun_control['core_model_hyper_dict']  
  
{'l1': {'type': 'int',  
        'default': 5,  
        'transform': 'transform_power_2_int',  
        'lower': 2,  
        'upper': 9},  
 'l2': {'type': 'int',  
        'default': 5,  
        'transform': 'transform_power_2_int',  
        'lower': 2,  
        'upper': 9},  
 'lr_mult': {'type': 'float',  
        'default': 1.0,  
        'transform': 'None',
```

```

'lower': 0.1,
'upper': 10.0},
'batch_size': {'type': 'int',
'default': 4,
'transform': 'transform_power_2_int',
'lower': 1,
'upper': 4},
'epochs': {'type': 'int',
'default': 3,
'transform': 'transform_power_2_int',
'lower': 3,
'upper': 4},
'k_folds': {'type': 'int',
'default': 1,
'transform': 'None',
'lower': 1,
'upper': 1},
'patience': {'type': 'int',
'default': 5,
'transform': 'None',
'lower': 2,
'upper': 10},
'optimizer': {'levels': ['Adadelta',
'Adagrad',
'Adam',
'AdamW',
'SparseAdam',
'Adamax',
'ASGD',
'NAdam',
'RAdam',
'RMSprop',
'Rprop',
'SGD'],
'type': 'factor',
'default': 'SGD',
'transform': 'None',
'class_name': 'torch.optim',
'core_model_parameter_type': 'str',
'lower': 0,
'upper': 12},
'sgd_momentum': {'type': 'float',
'default': 0.0,

```

```
'transform': 'None',
'lower': 0.0,
'upper': 1.0}]}
```

## 14.6 Step 6: Modify hyper\_dict Hyperparameters for the Selected Algorithm aka core\_model

Ray tune (PyTorch 2023a) does not provide a way to change the specified hyperparameters without re-compilation. However, `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 are described in the following.

### 14.6.0.1 Modify hyper\_dict Hyperparameters for the Selected Algorithm aka core\_model

After specifying the model, the corresponding hyperparameters, their types and bounds are loaded from the JSON file `torch_hyper_dict.json`. After loading, the user can modify the hyperparameters, e.g., the bounds. `spotPython` provides a simple rule for de-activating hyperparameters: If the lower and the upper bound are set to identical values, the hyperparameter is de-activated. This is useful for the hyperparameter tuning, because it allows to specify a hyperparameter in the JSON file, but to de-activate it in the `fun_control` dictionary. This is done in the next step.

### 14.6.0.2 Modify Hyperparameters of Type numeric and integer (boolean)

Since the hyperparameter `k_folds` is not used in the PyTorch tutorial, it is de-activated here by setting the lower and upper bound to the same value. Note, `k_folds` is of type “integer”.

```
from spotPython.hyperparameters.values import modify_hyper_parameter_bounds
modify_hyper_parameter_bounds(fun_control,
    "batch_size", bounds=[1, 5])
modify_hyper_parameter_bounds(fun_control,
    "k_folds", bounds=[0, 0])
modify_hyper_parameter_bounds(fun_control,
    "patience", bounds=[3, 3])
```

#### 14.6.0.3 Modify Hyperparameter of Type factor

In a similar manner as for the numerical hyperparameters, the categorical hyperparameters can be modified. New configurations can be chosen by adding or deleting levels. For example, the hyperparameter `optimizer` can be re-configured as follows:

In the following setting, two optimizers ("SGD" and "Adam") will be compared during the `spotPython` hyperparameter tuning. The hyperparameter `optimizer` is active.

```
from spotPython.hyperparameters.values import modify_hyper_parameter_levels
modify_hyper_parameter_levels(fun_control,
    "optimizer", ["SGD", "Adam"])
```

The hyperparameter `optimizer` can be de-activated by choosing only one value (level), here: "SGD".

```
modify_hyper_parameter_levels(fun_control, "optimizer", ["SGD"])
```

As discussed in Section 14.6.1, there are some issues with the LBFGS optimizer. Therefore, the usage of the LBFGS optimizer is not deactivated in `spotPython` by default. However, the LBFGS optimizer can be activated by adding it to the list of optimizers. `Rprop` was removed, because it does perform very poorly (as some pre-tests have shown). However, it can also be activated by adding it to the list of optimizers. Since `SparseAdam` does not support dense gradients, `Adam` was used instead. Therefore, there are 10 default optimizers:

```
modify_hyper_parameter_levels(fun_control, "optimizer",
    ["Adadelta", "Adagrad", "Adam", "AdamW", "Adamax", "ASGD",
     "NAdam", "RAdam", "RMSprop", "SGD"])
```

### 14.6.1 Optimizers

Table 14.1 shows some of the optimizers available in PyTorch:

*a* denotes (0.9,0.999), *b* (0.5,1.2), and *c* (1e-6, 50), respectively. *R* denotes required, but unspecified. "m" denotes momentum, "w\_d" weight\_decay, "d" dampening, "n" nesterov, "r" rho, "l\_s" learning rate for scaling delta, "l\_d" lr\_decay, "b" betas, "l" lambd, "a" alpha, "m\_d" for momentum\_decay, "e" etas, and "s\_s" for step\_sizes.

Table 14.1: Optimizers available in PyTorch (selection). The default values are shown in the table.

Optimizer	lr	m	w_d	d	n	r	l_s	l_d	b	l	a	m_d e	s_s
Adadelta	-	-	0.	-	-	0.9	1.	-	-	-	-	-	-
Adagrad	1e-2	-	0.	-	-	-	-	0.	-	-	-	-	-
Adam	1e-3	-	0.	-	-	-	-	-	a	-	-	-	-
AdamW	1e-3	-	1e-2	-	-	-	-	-	a	-	-	-	-
SparseAdam	1e-3	-	-	-	-	-	-	-	a	-	-	-	-
Adamax	2e-3	-	0.	-	-	-	-	-	a	-	-	-	-
ASGD	1e-2	.9	0.	-	F	-	-	-	-	1e-4	.75	-	-
LBFGS	1.	-	-	-	-	-	-	-	-	-	-	-	-
NAdam	2e-3	-	0.	-	-	-	-	-	a	-	-	0	-
RAdam	1e-3	-	0.	-	-	-	-	-	a	-	-	-	-
RMSprop	1e-2	0.	0.	-	-	-	-	-	a	-	-	-	-
Rprop	1e-2	-	-	-	-	-	-	-	-	-	b	c	-
SGD	R	0.	0.	F	-	-	-	-	-	-	-	-	-

`spotPython` implements an optimization handler that maps the optimizer names to the corresponding PyTorch optimizers.

#### A note on LBFGS

We recommend deactivating PyTorch's LBFGS optimizer, because it does not perform very well. The PyTorch documentation, see <https://pytorch.org/docs/stable/generated/torch.optim.LBFGS.html#torch.optim.LBFGS>, states:

This is a very memory intensive optimizer (it requires additional `param_bytes * (history_size + 1)` bytes). If it doesn't fit in memory try reducing the history size, or use a different algorithm.

Furthermore, the LBFGS optimizer is not compatible with the PyTorch tutorial. The reason is that the LBFGS optimizer requires the `closure` function, which is not implemented in the PyTorch tutorial. Therefore, the LBFGS optimizer is recommended here. Since there are ten optimizers in the portfolio, it is not recommended tuning the hyperparameters that effect one single optimizer only.

#### A note on the learning rate

`spotPython` provides a multiplier for the default learning rates, `lr_mult`, because optimizers use different learning rates. Using a multiplier for the learning rates might enable

a simultaneous tuning of the learning rates for all optimizers. However, this is not recommended, because the learning rates are not comparable across optimizers. Therefore, we recommend fixing the learning rate for all optimizers if multiple optimizers are used. This can be done by setting the lower and upper bounds of the learning rate multiplier to the same value as shown below.

Thus, the learning rate, which affects the `SGD` optimizer, will be set to a fixed value. We choose the default value of `1e-3` for the learning rate, because it is used in other PyTorch examples (it is also the default value used by `spotPython` as defined in the `optimizer_handler()` method). We recommend tuning the learning rate later, when a reduced set of optimizers is fixed. Here, we will demonstrate how to select in a screening phase the optimizers that should be used for the hyperparameter tuning.

For the same reason, we will fix the `sgd_momentum` to 0.9.

```
modify_hyper_parameter_bounds(fun_control,  
    "lr_mult", bounds=[1.0, 1.0])  
modify_hyper_parameter_bounds(fun_control,  
    "sgd_momentum", bounds=[0.9, 0.9])
```

## 14.7 Step 7: Selection of the Objective (Loss) Function

### 14.7.1 Evaluation: Data Splitting

The evaluation procedure requires the specification of the way how the data is split into a train and a test set and the loss function (and a metric). As a default, `spotPython` provides a standard hold-out data split and cross validation.

### 14.7.2 Hold-out Data Split

If a hold-out data split is used, the data will be partitioned into a training, a validation, and a test data set. The split depends on the setting of the `eval` parameter. If `eval` is set to `train_hold_out`, one data set, usually the original training data set, is split into a new training and a validation data set. The training data set is used for training the model. The validation data set is used for the evaluation of the hyperparameter configuration and early stopping to prevent overfitting. In this case, the original test data set is not used.

### Note

`spotPython` returns the hyperparameters of the machine learning and deep learning models, e.g., number of layers, learning rate, or optimizer, but not the model weights. Therefore, after the SPOT run is finished, the corresponding model with the optimized architecture has to be trained again with the best hyperparameter configuration. The training is performed on the training data set. The test data set is used for the final evaluation of the model.

Summarizing, the following splits are performed in the hold-out setting:

1. Run `spotPython` with `eval` set to `train_hold_out` to determine the best hyperparameter configuration.
2. Train the model with the best hyperparameter configuration (“architecture”) on the training data set: `train_tuned(model_spot, train, "model_spot.pt")`.
3. Test the model on the test data: `test_tuned(model_spot, test, "model_spot.pt")`

These steps will be exemplified in the following sections.

In addition to this `hold-out` setting, `spotPython` provides another hold-out setting, where an explicit test data is specified by the user that will be used as the validation set. To choose this option, the `eval` parameter is set to `test_hold_out`. In this case, the training data set is used for the model training. Then, the explicitly defined test data set is used for the evaluation of the hyperparameter configuration (the validation).

### 14.7.3 Cross-Validation

The cross validation setting is used by setting the `eval` parameter to `train_cv` or `test_cv`. In both cases, the data set is split into  $k$  folds. The model is trained on  $k - 1$  folds and evaluated on the remaining fold. This is repeated  $k$  times, so that each fold is used exactly once for evaluation. The final evaluation is performed on the test data set. The cross validation setting is useful for small data sets, because it allows to use all data for training and evaluation. However, it is computationally expensive, because the model has to be trained  $k$  times.

### Note

Combinations of the above settings are possible, e.g., cross validation can be used for training and hold-out for evaluation or *vice versa*. Also, cross validation can be used for training and testing. Because cross validation is not used in the PyTorch tutorial (PyTorch 2023a), it is not considered further here.

## 14.7.4 Overview of the Evaluation Settings

### 14.7.4.1 Settings for the Hyperparameter Tuning

An overview of the training evaluations is shown in Table 14.2. "train\_cv" and "test\_cv" use `sklearn.model_selection.KFold()` internally. More details on the data splitting are provided in Section G.15 (in the Appendix).

Table 14.2: Overview of the evaluation settings.

eval	train	test	function	comment
"train_hold_out"	✓		<code>train_one_epoch()</code> , <code>validate_one_epoch()</code> for early stopping	splits the <code>train</code> data set internally
"test_hold_out"	✓	✓	<code>train_one_epoch()</code> , <code>validate_one_epoch()</code> for early stopping	use the <code>test</code> data set for <code>validate_one_epoch()</code>
"train_cv"	✓		<code>evaluate_cv(net,</code> <code>train)</code>	CV using the <code>train</code> data set
"test_cv"		✓	<code>evaluate_cv(net,</code> <code>test)</code>	CV using the <code>test</code> data set . Identical to "train_cv", uses only test data.

### 14.7.4.2 Settings for the Final Evaluation of the Tuned Architecture

#### 14.7.4.2.1 Training of the Tuned Architecture

`train_tuned(model, train)`: train the model with the best hyperparameter configuration (or simply the default) on the training data set. It splits the `traindata` into new `train` and `validation` sets using `create_train_val_data_loaders()`, which calls `torch.utils.data.random_split()` internally. Currently, 60% of the data is used for training and 40% for validation. The `train` data is used for training the model with `train_hold_out()`. The `validation` data is used for early stopping using `validate_fold_or_hold_out()` on the `validation` data set.

#### 14.7.4.2.2 Testing of the Tuned Architecture

`test_tuned(model, test)`: test the model on the `test` data set. No data splitting is performed. The (trained) model is evaluated using the `validate_fold_or_hold_out()` function. Note: During training, "shuffle" is set to `True`, whereas during testing, "shuffle" is set to `False`.

Section G.15.1.4 describes the final evaluation of the tuned architecture.

```
fun_control.update({
    "eval": "train_hold_out",
    "path": "torch_model.pt",
    "shuffle": True})
```

#### 14.7.5 Evaluation: Loss Functions and Metrics

The key "loss\_function" specifies the loss function which is used during the optimization. There are several different loss functions under PyTorch's `nn` package. For example, a simple loss is `MSELoss`, which computes the mean-squared error between the output and the target. In this tutorial we will use `CrossEntropyLoss`, because it is also used in the PyTorch tutorial.

```
from torch.nn import CrossEntropyLoss
loss_function = CrossEntropyLoss()
fun_control.update({"loss_function": loss_function})
```

In addition to the loss functions, `spotPython` provides access to a large number of metrics.

- The key "metric\_sklearn" is used for metrics that follow the `scikit-learn` conventions.
- The key "river\_metric" is used for the river based evaluation (Montiel et al. 2021) via `eval_oml_iter_progressive`, and
- the key "metric\_torch" is used for the metrics from `TorchMetrics`.

`TorchMetrics` is a collection of more than 90 PyTorch metrics, see <https://torchmetrics.readthedocs.io/en/latest/>. Because the PyTorch tutorial uses the accuracy as metric, we use the same metric here. Currently, accuracy is computed in the tutorial's example code. We will use `TorchMetrics` instead, because it offers more flexibility, e.g., it can be used for regression and classification. Furthermore, `TorchMetrics` offers the following advantages:

- \* A standardized interface to increase reproducibility
- \* Reduces Boilerplate
- \* Distributed-training compatible
- \* Rigorously tested
- \* Automatic accumulation over batches
- \* Automatic synchronization between multiple devices

Therefore, we set

```

import torchmetrics
metric_torch = torchmetrics.Accuracy(task="multiclass", num_classes=10).to(fun_control["device"])
fun_control.update({"metric_torch": metric_torch})

```

## 14.8 Step 8: Calling the SPOT Function

### 14.8.1 Preparing the SPOT Call

The following code passes the information about the parameter ranges and bounds to `spot`.

```

from spotPython.hyperparameters.values import (
    get_var_type,
    get_var_name,
    get_bound_values
)
var_type = get_var_type(fun_control)
var_name = get_var_name(fun_control)

lower = get_bound_values(fun_control, "lower")
upper = get_bound_values(fun_control, "upper")

```

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` 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	5	2	9	transform_power_2_int
l2	int	5	2	9	transform_power_2_int
lr_mult	float	1.0	1	1	None
batch_size	int	4	1	5	transform_power_2_int
epochs	int	3	3	4	transform_power_2_int
k_folds	int	1	0	0	None
patience	int	5	3	3	None
optimizer	factor	SGD	0	9	None
sgd_momentum	float	0.0	0.9	0.9	None

This allows to check if all information is available and if the information is correct. `?@tbl-design` shows the experimental design for the hyperparameter tuning. The table shows the hyperparameters, their types, default values, lower and upper bounds, and the transformation function. The transformation function is used to transform the hyperparameter values from the unit hypercube to the original domain. The transformation function is applied to the hyperparameter values before the evaluation of the objective function. Hyperparameter transformations are shown in the column “transform”, e.g., the 11 default is 5, which results in the value  $2^5 = 32$  for the network, because the transformation `transform_power_2_int` was selected in the JSON file. The default value of the `batch_size` is set to 4, which results in a batch size of  $2^4 = 16$ .

#### 14.8.2 The Objective Function `fun_torch`

The objective function `fun_torch` is selected next. It implements an interface from PyTorch’s training, validation, and testing methods to `spotPython`.

```
from spotPython.fun.hypertorch import HyperTorch
fun = HyperTorch().fun_torch
```

#### 14.8.3 Using Default Hyperparameters or Results from Previous Runs

We add the default setting to the initial design:

```
from spotPython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
```

#### 14.8.4 Starting the Hyperparameter Tuning

The `spotPython` hyperparameter tuning is started by calling the `Spot` function. Here, we will run the tuner for approximately 30 minutes (`max_time`). Note: the initial design is always evaluated in the `spotPython` run. As a consequence, the run may take longer than specified by `max_time`, because the evaluation time of initial design (here: `init_size`, 10 points) is performed independently of `max_time`. During the run, results from the training is shown. These results can be visualized with Tensorboard as will be shown in Section 14.9.

```
from spotPython.spot import spot
from math import inf
import numpy as np
spot_tuner = spot.Spot(fun=fun,
```

```

        lower = lower,
        upper = upper,
        fun_evals = inf,
        max_time = MAX_TIME,
        tolerance_x = np.sqrt(np.spacing(1)),
        var_type = var_type,
        var_name = var_name,
        show_progress= True,
        fun_control = fun_control,
        design_control={"init_size": INIT_SIZE},
        surrogate_control={"noise": True,
                           "cod_type": "norm",
                           "min_theta": -4,
                           "max_theta": 3,
                           "n_theta": len(var_name),
                           "model_fun_evals": 10_000
                          })
spot_tuner.run(X_start=X_start)

```

```

config: {'l1': 128, 'l2': 8, 'lr_mult': 1.0, 'batch_size': 32, 'epochs': 16, 'k_folds': 0, 'p': 0.5}
Epoch: 1 | MulticlassAccuracy: 0.3838500082492828 | Loss: 1.6562047554016113 | Acc: 0.3838500000000000
Epoch: 2 | MulticlassAccuracy: 0.4587000012397766 | Loss: 1.4801345227241516 | Acc: 0.4587000000000000
Epoch: 3 | MulticlassAccuracy: 0.5033000111579895 | Loss: 1.370735721112976 | Acc: 0.5033000000000000
Epoch: 4 | MulticlassAccuracy: 0.5238500237464905 | Loss: 1.3132523689270019 | Acc: 0.5238500000000000
Epoch: 5 | MulticlassAccuracy: 0.5549499988555908 | Loss: 1.2375486762046815 | Acc: 0.5549500000000001
Epoch: 6 | MulticlassAccuracy: 0.5734999775886536 | Loss: 1.2021079837799071 | Acc: 0.5735000000000000
Epoch: 7 |

```

```
MulticlassAccuracy: 0.5772500038146973 | Loss: 1.1980135177612306 | Acc: 0.5772500000000000
Epoch: 8 |

MulticlassAccuracy: 0.5938000082969666 | Loss: 1.1599345854759215 | Acc: 0.5938000000000000
Epoch: 9 |

MulticlassAccuracy: 0.6009500026702881 | Loss: 1.1328843992233277 | Acc: 0.6009500000000000
Epoch: 10 |

MulticlassAccuracy: 0.6024000048637390 | Loss: 1.1520775830268859 | Acc: 0.6024000000000000
Epoch: 11 |

MulticlassAccuracy: 0.6004999876022339 | Loss: 1.1797576458930970 | Acc: 0.6005000000000000
Epoch: 12 |

MulticlassAccuracy: 0.6086500287055969 | Loss: 1.1477203011512755 | Acc: 0.6086500000000000
Early stopping at epoch 11
Returned to Spot: Validation loss: 1.1477203011512755

config: {'l1': 16, 'l2': 16, 'lr_mult': 1.0, 'batch_size': 8, 'epochs': 8, 'k_folds': 0, 'pa
Epoch: 1 |

MulticlassAccuracy: 0.4344500005245209 | Loss: 1.5370126734256744 | Acc: 0.4344500000000000
Epoch: 2 |

MulticlassAccuracy: 0.4992499947547913 | Loss: 1.3821711903572083 | Acc: 0.4992500000000000
Epoch: 3 |

MulticlassAccuracy: 0.5210499763488770 | Loss: 1.3544268280982972 | Acc: 0.5210500000000000
Epoch: 4 |

MulticlassAccuracy: 0.5225499868392944 | Loss: 1.3431447335362434 | Acc: 0.5225500000000000
Epoch: 5 |

MulticlassAccuracy: 0.5267999768257141 | Loss: 1.3126116187691688 | Acc: 0.5268000000000000
Epoch: 6 |

MulticlassAccuracy: 0.5393000245094299 | Loss: 1.3071551247596740 | Acc: 0.5393000000000000
Epoch: 7 |
```

```
MulticlassAccuracy: 0.5383499860763550 | Loss: 1.3307678449213505 | Acc: 0.5383500000000000
Epoch: 8 |

MulticlassAccuracy: 0.5449500083923340 | Loss: 1.2984912836074829 | Acc: 0.5449500000000000
Returned to Spot: Validation loss: 1.298491283607483

config: {'l1': 256, 'l2': 128, 'lr_mult': 1.0, 'batch_size': 2, 'epochs': 16, 'k_folds': 0,
Epoch: 1 |

MulticlassAccuracy: 0.1054999977350235 | Loss: 2.3146803474247455 | Acc: 0.1055000000000000
Epoch: 2 |

MulticlassAccuracy: 0.1067499965429306 | Loss: 2.3664452521443367 | Acc: 0.1067500000000000
Epoch: 3 |

MulticlassAccuracy: 0.0958999991416931 | Loss: 2.3053415326595306 | Acc: 0.0959000000000000
Epoch: 4 |

MulticlassAccuracy: 0.1058500036597252 | Loss: 2.3030879536867142 | Acc: 0.1058500000000000
Epoch: 5 |

MulticlassAccuracy: 0.1409499943256378 | Loss: 2.5993600514886870 | Acc: 0.1409500000000000
Epoch: 6 |

MulticlassAccuracy: 0.0986500009894371 | Loss: 2.3089555664300918 | Acc: 0.0986500000000000
Epoch: 7 |

MulticlassAccuracy: 0.0958499982953072 | Loss: 2.3065662771463393 | Acc: 0.0958500000000000
Early stopping at epoch 6
Returned to Spot: Validation loss: 2.3065662771463393

config: {'l1': 8, 'l2': 32, 'lr_mult': 1.0, 'batch_size': 4, 'epochs': 8, 'k_folds': 0, 'pati
Epoch: 1 |

MulticlassAccuracy: 0.3878000080585480 | Loss: 1.6677710754990578 | Acc: 0.3878000000000000
Epoch: 2 |

MulticlassAccuracy: 0.4491499960422516 | Loss: 1.5083286433458327 | Acc: 0.4491500000000000
Epoch: 3 |
```

MulticlassAccuracy: 0.4769999980926514 | Loss: 1.4328012544035911 | Acc: 0.4770000000000000  
Epoch: 4 |

MulticlassAccuracy: 0.5037000179290771 | Loss: 1.3618832346677781 | Acc: 0.5037000000000000  
Epoch: 5 |

MulticlassAccuracy: 0.5237500071525574 | Loss: 1.3237895696029067 | Acc: 0.5237500000000000  
Epoch: 6 |

MulticlassAccuracy: 0.5248000025749207 | Loss: 1.3229136003062130 | Acc: 0.5248000000000000  
Epoch: 7 |

MulticlassAccuracy: 0.5267000198364258 | Loss: 1.3368793054573238 | Acc: 0.5266999999999999  
Epoch: 8 |

MulticlassAccuracy: 0.5468500256538391 | Loss: 1.3025472436670213 | Acc: 0.5468499999999999  
Returned to Spot: Validation loss: 1.3025472436670213

config: {'l1': 64, 'l2': 512, 'lr\_mult': 1.0, 'batch\_size': 16, 'epochs': 16, 'k\_folds': 0,  
Epoch: 1 |

MulticlassAccuracy: 0.4503999948501587 | Loss: 1.5053875568389892 | Acc: 0.4504000000000000  
Epoch: 2 |

MulticlassAccuracy: 0.4817500114440918 | Loss: 1.4276890246868132 | Acc: 0.4817500000000000  
Epoch: 3 |

MulticlassAccuracy: 0.4971500039100647 | Loss: 1.3758006053924561 | Acc: 0.4971500000000000  
Epoch: 4 |

MulticlassAccuracy: 0.5052999854087830 | Loss: 1.3617287966728211 | Acc: 0.5053000000000000  
Epoch: 5 |

MulticlassAccuracy: 0.5246499776840210 | Loss: 1.3144499421119691 | Acc: 0.5246499999999999  
Epoch: 6 |

MulticlassAccuracy: 0.5333999991416931 | Loss: 1.2918155289649964 | Acc: 0.5334000000000000  
Epoch: 7 |

MulticlassAccuracy: 0.5371000170707703 | Loss: 1.2804483198165895 | Acc: 0.5371000000000000  
Epoch: 8 |

MulticlassAccuracy: 0.5475000143051147 | Loss: 1.2682600169181824 | Acc: 0.5475000000000000  
Epoch: 9 |

MulticlassAccuracy: 0.5515000224113464 | Loss: 1.2482085966110230 | Acc: 0.5515000000000000  
Epoch: 10 |

MulticlassAccuracy: 0.5543000102043152 | Loss: 1.2493398176908492 | Acc: 0.5543000000000000  
Epoch: 11 |

MulticlassAccuracy: 0.5570499897003174 | Loss: 1.2347032400369644 | Acc: 0.5570500000000000  
Epoch: 12 |

MulticlassAccuracy: 0.5618000030517578 | Loss: 1.2204034348726274 | Acc: 0.5618000000000000  
Epoch: 13 |

MulticlassAccuracy: 0.5636000037193298 | Loss: 1.2188638792991637 | Acc: 0.5636000000000000  
Epoch: 14 |

MulticlassAccuracy: 0.5701000094413757 | Loss: 1.2036190267801286 | Acc: 0.5701000000000001  
Epoch: 15 |

MulticlassAccuracy: 0.5735499858856201 | Loss: 1.2017679643630981 | Acc: 0.5735500000000000  
Epoch: 16 |

MulticlassAccuracy: 0.574000009536743 | Loss: 1.1927833241224288 | Acc: 0.5740000000000000  
Returned to Spot: Validation loss: 1.1927833241224288

config: {'l1': 128, 'l2': 16, 'lr\_mult': 1.0, 'batch\_size': 32, 'epochs': 16, 'k\_folds': 0,  
Epoch: 1 |

MulticlassAccuracy: 0.4125500023365021 | Loss: 1.5910222095489501 | Acc: 0.4125500000000000  
Epoch: 2 |

MulticlassAccuracy: 0.4715499877929688 | Loss: 1.4488346719741820 | Acc: 0.4715500000000000  
Epoch: 3 |

MulticlassAccuracy: 0.5084999799728394 | Loss: 1.3695043066978454 | Acc: 0.5085000000000000  
Epoch: 4 |

MulticlassAccuracy: 0.5306500196456909 | Loss: 1.3158891970634461 | Acc: 0.5306500000000000  
Epoch: 5 |

MulticlassAccuracy: 0.5375000238418579 | Loss: 1.3249270143508911 | Acc: 0.5375000000000000  
Epoch: 6 |

MulticlassAccuracy: 0.5588499903678894 | Loss: 1.2504237797737121 | Acc: 0.5588500000000000  
Epoch: 7 |

MulticlassAccuracy: 0.5788999795913696 | Loss: 1.2066600365638733 | Acc: 0.5789000000000000  
Epoch: 8 |

MulticlassAccuracy: 0.5720999836921692 | Loss: 1.2174073032379151 | Acc: 0.5721000000000001  
Epoch: 9 |

MulticlassAccuracy: 0.5911499857902527 | Loss: 1.1804030963897705 | Acc: 0.5911500000000000  
Epoch: 10 |

MulticlassAccuracy: 0.590099903678894 | Loss: 1.1776824831962585 | Acc: 0.5901000000000000  
Epoch: 11 |

MulticlassAccuracy: 0.5999000072479248 | Loss: 1.1676074934005738 | Acc: 0.5999000000000000  
Epoch: 12 |

MulticlassAccuracy: 0.5935000181198120 | Loss: 1.1752061429977416 | Acc: 0.5935000000000000  
Epoch: 13 |

MulticlassAccuracy: 0.6051499843597412 | Loss: 1.1578538383483887 | Acc: 0.6051500000000000  
Epoch: 14 |

MulticlassAccuracy: 0.6034500002861023 | Loss: 1.1724235271453858 | Acc: 0.6034500000000000  
Epoch: 15 |

MulticlassAccuracy: 0.6014999747276306 | Loss: 1.1903479053497314 | Acc: 0.6015000000000000  
Epoch: 16 |

```
MulticlassAccuracy: 0.6006500124931335 | Loss: 1.1868701674461364 | Acc: 0.6006500000000000
Early stopping at epoch 15
Returned to Spot: Validation loss: 1.1868701674461364
spotPython tuning: 1.1477203011512755 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2df383250>
```

## 14.9 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard.

### 14.9.1 Tensorboard: Start Tensorboard

Start TensorBoard through the command line to visualize data you logged. Specify the root log directory as used in `fun_control = fun_control_init(task="regression", tensorboard_path="runs/24_spot_torch_regression")` as the `tensorboard_path`. The argument `logdir` points to directory where TensorBoard will look to find event files that it can display. TensorBoard will recursively walk the directory structure rooted at `logdir`, looking for `.tfevents` files.

```
tensorboard --logdir=runs
```

Go to the URL it provides or to <http://localhost:6006/>. The following figures show some screenshots of Tensorboard.

### 14.9.2 Saving the State of the Notebook

The state of the notebook can be saved and reloaded as follows:

```
import pickle
SAVE = False
LOAD = False

if SAVE:
    result_file_name = "res_" + experiment_name + ".pkl"
    with open(result_file_name, 'wb') as f:
        pickle.dump(spot_tuner, f)

if LOAD:
```

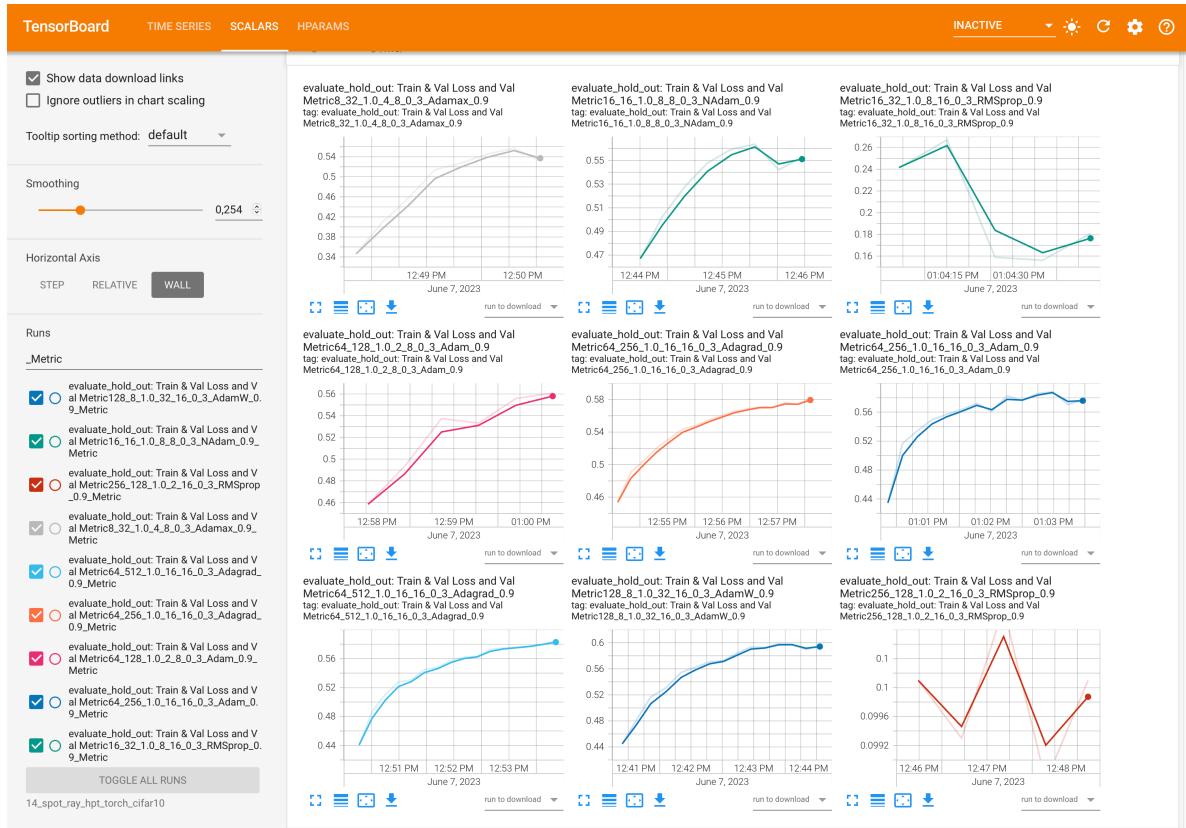


Figure 14.1: Tensorboard



Figure 14.2: Tensorboard

```

result_file_name = "add_the_name_of_the_result_file_here.pkl"
with open(result_file_name, 'rb') as f:
    spot_tuner = pickle.load(f)

```

## 14.10 Step 10: Results

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized. The following code generates the progress plot from `?@fig-progress`.

```

spot_tuner.plot_progress(log_y=False,
    filename="./figures/" + experiment_name+"_progress.png")

```

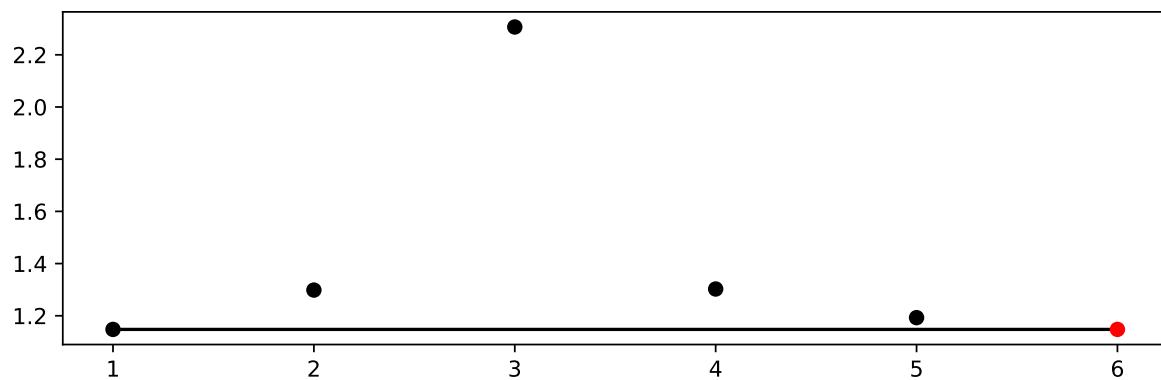


Figure 14.3: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

`?@fig-progress` shows a typical behaviour that can be observed in many hyperparameter studies (Bartz et al. 2022): the largest improvement is obtained during the evaluation of the initial design. The surrogate model based optimization-optimization with the surrogate refines the results. `?@fig-progress` also illustrates one major difference between `ray[tune]` as used in PyTorch (2023a) and `spotPython`: the `ray[tune]` uses a random search and will generate results similar to the *black* dots, whereas `spotPython` uses a surrogate model based optimization and presents results represented by *red* dots in `?@fig-progress`. The surrogate model based optimization is considered to be more efficient than a random search, because the surrogate model guides the search towards promising regions in the hyperparameter space.

In addition to the improved (“optimized”) hyperparameter values, `spotPython` allows a statistical analysis, e.g., a sensitivity analysis, of the results. We can print the results of the hyperparameter tuning, see `?@tbl-results`. The table shows the hyperparameters, their types, default

values, lower and upper bounds, and the transformation function. The column “tuned” shows the tuned values. The column “importance” shows the importance of the hyperparameters. The column “stars” shows the importance of the hyperparameters in stars. The importance is computed by the SPOT software.

```
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	5	2.0	9.0	7.0	transform_power_2_int
l2	int	5	2.0	9.0	3.0	transform_power_2_int
lr_mult	float	1.0	1.0	1.0	1.0	None
batch_size	int	4	1.0	5.0	5.0	transform_power_2_int
epochs	int	3	3.0	4.0	4.0	transform_power_2_int
k_folds	int	1	0.0	0.0	0.0	None
patience	int	5	3.0	3.0	3.0	None
optimizer	factor	SGD	0.0	9.0	3.0	None
sgd_momentum	float	0.0	0.9	0.9	0.9	None

To visualize the most important hyperparameters, `spotPython` provides the function `plot_importance`. The following code generates the importance plot from `?@fig-importance`.

```
spot_tuner.plot_importance(threshold=0.025,
                           filename="./figures/" + experiment_name+"_importance.png")
```

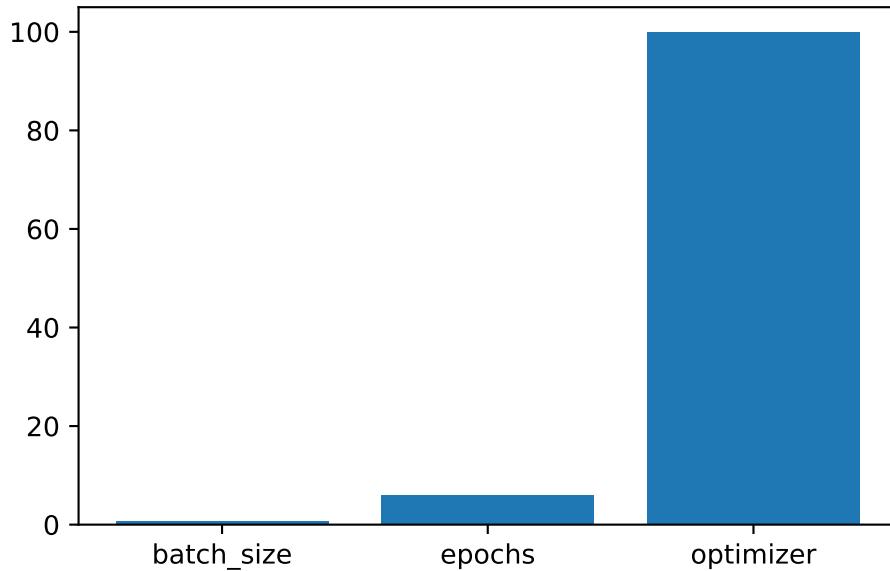


Figure 14.4: Variable importance plot, threshold 0.025.

#### 14.10.1 Get the Tuned Architecture (SPOT Results)

The architecture of the `spotPython` model can be obtained as follows. First, the numerical representation of the hyperparameters are obtained, i.e., the numpy array `X` is generated. This array is then used to generate the model `model_spot` by the function `get_one_core_model_from_X`. The model `model_spot` has the following architecture:

```
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)
model_spot

Net_CIFAR10(
    (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
    (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (fc1): Linear(in_features=400, out_features=128, bias=True)
    (fc2): Linear(in_features=128, out_features=8, bias=True)
    (fc3): Linear(in_features=8, out_features=10, bias=True)
)
```

### 14.10.2 Get Default Hyperparameters

In a similar manner as in Section 14.10.1, the default hyperparameters can be obtained.

```
# fun_control was modified, we generate a new one with the original
# default hyperparameters
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

Net_CIFAR10(
    (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
    (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (fc1): Linear(in_features=400, out_features=32, bias=True)
    (fc2): Linear(in_features=32, out_features=32, bias=True)
    (fc3): Linear(in_features=32, out_features=10, bias=True)
)
```

### 14.10.3 Evaluation of the Default Architecture

The method `train_tuned` takes a model architecture without trained weights and trains this model with the train data. The train data is split into train and validation data. The validation data is used for early stopping. The trained model weights are saved as a dictionary.

This evaluation is similar to the final evaluation in PyTorch (2023a).

```
from spotPython.torch.traintest import (
    train_tuned,
    test_tuned,
)
train_tuned(net=model_default, train_dataset=train, shuffle=True,
            loss_function=fun_control["loss_function"],
            metric=fun_control["metric_torch"],
            device = fun_control["device"], show_batch_interval=1_000_000,
            path=None,
            task=fun_control["task"],)

test_tuned(net=model_default, test_dataset=test,
```

```
    loss_function=fun_control["loss_function"],
    metric=fun_control["metric_torch"],
    shuffle=False,
    device = fun_control["device"],
    task=fun_control["task"],)
```

Epoch: 1 |

MulticlassAccuracy: 0.0995000004768372 | Loss: 2.2999522211074828 | Acc: 0.0995000000000000  
Epoch: 2 |

MulticlassAccuracy: 0.1305000036954880 | Loss: 2.2869610927581787 | Acc: 0.1305000000000000  
Epoch: 3 |

MulticlassAccuracy: 0.1363500058650970 | Loss: 2.2477050289154055 | Acc: 0.1363500000000000  
Epoch: 4 |

MulticlassAccuracy: 0.2006500065326691 | Loss: 2.1781324967384337 | Acc: 0.2006500000000000  
Epoch: 5 |

MulticlassAccuracy: 0.2182500064373016 | Loss: 2.1153550539016726 | Acc: 0.2182500000000000  
Epoch: 6 |

MulticlassAccuracy: 0.2264499962329865 | Loss: 2.0708753735542298 | Acc: 0.2264500000000000  
Epoch: 7 |

MulticlassAccuracy: 0.2435999959707260 | Loss: 2.0414400042533876 | Acc: 0.2436000000000000  
Epoch: 8 |

MulticlassAccuracy: 0.2531499862670898 | Loss: 2.0214681243896484 | Acc: 0.2531500000000000  
Returned to Spot: Validation loss: 2.0214681243896484

MulticlassAccuracy: 0.2574999928474426 | Loss: 2.0162517908096311 | Acc: 0.2575000000000000  
Final evaluation: Validation loss: 2.016251790809631  
Final evaluation: Validation metric: 0.2574999928474426

---

(2.016251790809631, nan, tensor(0.2575, device='mps:0'))

#### 14.10.4 Evaluation of the Tuned Architecture

The following code trains the model `model_spot`.

If `path` is set to a filename, e.g., `path = "model_spot_trained.pt"`, the weights of the trained model will be saved to this file.

If `path` is set to a filename, e.g., `path = "model_spot_trained.pt"`, the weights of the trained model will be loaded from this file.

```
train_tuned(net=model_spot, train_dataset=train,
            loss_function=fun_control["loss_function"],
            metric=fun_control["metric_torch"],
            shuffle=True,
            device = fun_control["device"],
            path=None,
            task=fun_control["task"],)
test_tuned(net=model_spot, test_dataset=test,
           shuffle=False,
           loss_function=fun_control["loss_function"],
           metric=fun_control["metric_torch"],
           device = fun_control["device"],
           task=fun_control["task"],)
```

Epoch: 1 |

MulticlassAccuracy: 0.4187499880790710 | Loss: 1.5769843736648559 | Acc: 0.4187500000000000.  
Epoch: 2 |

MulticlassAccuracy: 0.4675000011920929 | Loss: 1.4508240690231324 | Acc: 0.4675000000000000.  
Epoch: 3 |

MulticlassAccuracy: 0.5137000083923340 | Loss: 1.3569832351684570 | Acc: 0.5137000000000000.  
Epoch: 4 |

MulticlassAccuracy: 0.5307499766349792 | Loss: 1.3070075393676759 | Acc: 0.5307500000000001.  
Epoch: 5 |

MulticlassAccuracy: 0.5393499732017517 | Loss: 1.2841068645477296 | Acc: 0.5393500000000000.  
Epoch: 6 |

```
MulticlassAccuracy: 0.5460500121116638 | Loss: 1.2693293420791627 | Acc: 0.5460500000000000.  
Epoch: 7 |  
  
MulticlassAccuracy: 0.5558999776840210 | Loss: 1.2405844126701355 | Acc: 0.5558999999999999.  
Epoch: 8 |  
  
MulticlassAccuracy: 0.5686500072479248 | Loss: 1.2219975400924683 | Acc: 0.5686500000000000.  
Epoch: 9 |  
  
MulticlassAccuracy: 0.5616499781608582 | Loss: 1.2282655703544616 | Acc: 0.5616500000000000.  
Epoch: 10 |  
  
MulticlassAccuracy: 0.5766999721527100 | Loss: 1.2177041532516479 | Acc: 0.5767000000000000.  
Epoch: 11 |  
  
MulticlassAccuracy: 0.5799999833106995 | Loss: 1.2102932430267335 | Acc: 0.5800000000000000.  
Epoch: 12 |  
  
MulticlassAccuracy: 0.5819000005722046 | Loss: 1.2291380684852600 | Acc: 0.5819000000000000.  
Epoch: 13 |  
  
MulticlassAccuracy: 0.5869500041007996 | Loss: 1.2013124509811401 | Acc: 0.5869500000000000.  
Epoch: 14 |  
  
MulticlassAccuracy: 0.5739499926567078 | Loss: 1.2463197503089904 | Acc: 0.5739500000000000.  
Epoch: 15 |  
  
MulticlassAccuracy: 0.5835999846458435 | Loss: 1.2343276533126830 | Acc: 0.5836000000000000.  
Epoch: 16 |  
  
MulticlassAccuracy: 0.5806499719619751 | Loss: 1.2479942989349366 | Acc: 0.5806500000000000.  
Early stopping at epoch 15  
Returned to Spot: Validation loss: 1.2479942989349366  
  
MulticlassAccuracy: 0.5892999768257141 | Loss: 1.2312500318780113 | Acc: 0.5893000000000000.  
Final evaluation: Validation loss: 1.2312500318780113  
Final evaluation: Validation metric: 0.5892999768257141  
-----  
(1.2312500318780113, nan, tensor(0.5893, device='mps:0'))
```

### 14.10.5 Detailed Hyperparameter Plots

The contour plots in this section visualize the interactions of the three most important hyperparameters. Since some of these hyperparameters take factorial or integer values, sometimes step-like fitness landscapes (or response surfaces) are generated. SPOT draws the interactions of the main hyperparameters by default. It is also possible to visualize all interactions.

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

```
batch_size: 0.808580970629426  
epochs: 6.061007441106652  
optimizer: 100.0
```

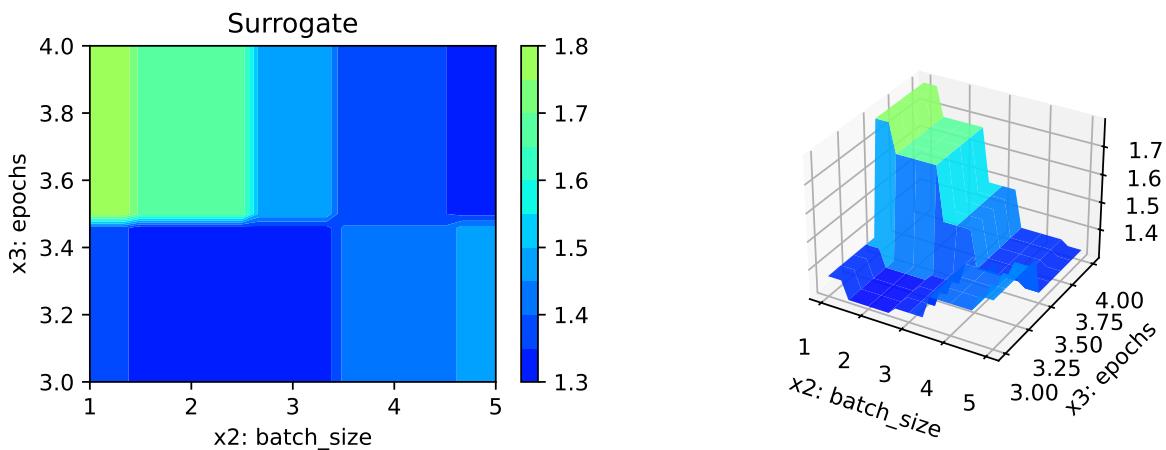
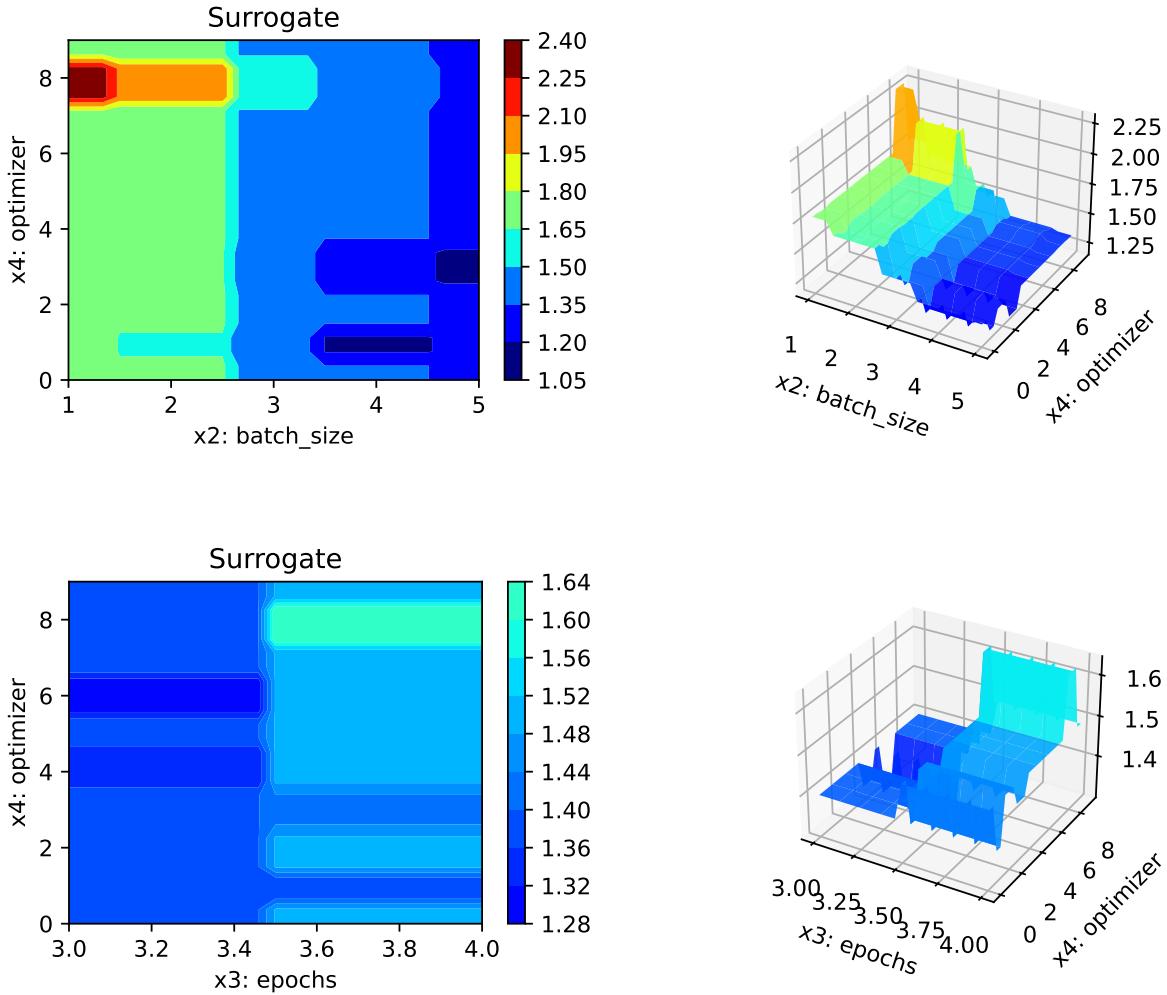


Figure 14.5: Contour plots.



The figures (**?@fig-contour**) show the contour plots of the loss as a function of the hyperparameters. These plots are very helpful for benchmark studies and for understanding neural networks. `spotPython` provides additional tools for a visual inspection of the results and give valuable insights into the hyperparameter tuning process. This is especially useful for model explainability, transparency, and trustworthiness. In addition to the contour plots, **?@fig-parallel** shows the parallel plot of the hyperparameters.

```
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

## 14.11 Summary and Outlook

This tutorial presents the hyperparameter tuning open source software `spotPython` for PyTorch. To show its basic features, a comparison with the “official” PyTorch hyperparameter tuning tutorial (PyTorch 2023a) is presented. Some of the advantages of `spotPython` are:

- Numerical and categorical hyperparameters.
- Powerful surrogate models.
- Flexible approach and easy to use.
- Simple JSON files for the specification of the hyperparameters.
- Extension of default and user specified network classes.
- Noise handling techniques.
- Interaction with `tensorboard`.

Currently, only rudimentary parallel and distributed neural network training is possible, but these capabilities will be extended in the future. The next version of `spotPython` will also include a more detailed documentation and more examples.

### ! Important

Important: This tutorial does not present a complete benchmarking study (Bartz-Beielstein et al. 2020). The results are only preliminary and highly dependent on the local configuration (hard- and software). Our goal is to provide a first impression of the performance of the hyperparameter tuning package `spotPython`. To demonstrate its capabilities, a quick comparison with `ray[tune]` was performed. `ray[tune]` was chosen, because it is presented as “an industry standard tool for distributed hyperparameter tuning.” The results should be interpreted with care.

## 14.12 Appendix

### 14.12.1 Sample Output From Ray Tune’s Run

The output from `ray[tune]` could look like this (PyTorch 2023b):

```
Number of trials: 10 (10 TERMINATED)
-----+-----+-----+-----+-----+-----+
|    11 |    12 |      lr |  batch_size |     loss |   accuracy | training_iteration |
```

64	4	0.00011629	2	1.87273	0.244	2
32	64	0.000339763	8	1.23603	0.567	8
8	16	0.00276249	16	1.1815	0.5836	10
4	64	0.000648721	4	1.31131	0.5224	8
32	16	0.000340753	8	1.26454	0.5444	8
8	4	0.000699775	8	1.99594	0.1983	2
256	8	0.0839654	16	2.3119	0.0993	1
16	128	0.0758154	16	2.33575	0.1327	1
16	8	0.0763312	16	2.31129	0.1042	4
128	16	0.000124903	4	2.26917	0.1945	1

# 15 HPT: sklearn RandomForestClassifier VBDP Data

This chapter describes the hyperparameter tuning of a `RandomForestClassifier` on the Vector Borne Disease Prediction (VBDP) data set.

## ! Vector Borne Disease Prediction Data Set

This chapter uses the Vector Borne Disease Prediction data set from Kaggle. It is a categorical dataset for eleven Vector Borne Diseases with associated symptoms.

The person who associated a work with this deed has dedicated the work to the public domain by waiving all of his or her rights to the work worldwide under copyright law, including all related and neighboring rights, to the extent allowed by law. You can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission. See Other Information below, see <https://creativecommons.org/publicdomain/zero/1.0/>.

The data set is available at: <https://www.kaggle.com/datasets/richardbernat/vector-borne-disease-prediction>,

The data should be downloaded and stored in the `data/VBDP` subfolder. The data set is not available as a part of the `spotPython` package.

## 15.1 Step 1: Setup

Before we consider the detailed experimental setup, we select the parameters that affect run time and the initial design size.

```
MAX_TIME = 1
INIT_SIZE = 5
ORIGINAL = True
PREFIX = "16"
```

```
import warnings
warnings.filterwarnings("ignore")
```

## 15.2 Step 2: Initialization of the Empty fun\_control Dictionary

```
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_experiment_name, get_spot_tensorboard_path
from spotPython.utils.device import getDevice

experiment_name = get_experiment_name(prefix=PREFIX)

fun_control = fun_control_init(
    task="classification",
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
```

## 15.3 Step 3: PyTorch Data Loading

### 15.3.1 Load Data: Classification VBDP

```
import pandas as pd
if ORIGINAL == True:
    train_df = pd.read_csv('./data/VBDP/trainn.csv')
    test_df = pd.read_csv('./data/VBDP/testt.csv')
else:
    train_df = pd.read_csv('./data/VBDP/train.csv')
    # remove the id column
    train_df = train_df.drop(columns=['id'])

from sklearn.preprocessing import OrdinalEncoder
n_samples = train_df.shape[0]
n_features = train_df.shape[1] - 1
target_column = "prognosis"
# Encoder our prognosis labels as integers for easier decoding later
enc = OrdinalEncoder()
train_df[target_column] = enc.fit_transform(train_df[[target_column]])
train_df.columns = [f"x{i}" for i in range(1, n_features+1)] + [target_column]
print(train_df.shape)
```

```
train_df.head()
```

(252, 65)

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	...	x56	x57	x58	x59	x60	x61	x62	x63	x64
0	0	1	1	1	1	0	1	0	0	0	...	0	0	0	0	0	0	0	0	
1	1	1	1	1	1	0	1	1	1	0	...	0	0	0	0	0	0	0	0	
2	0	1	0	1	0	0	1	1	0	0	...	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	1	1	1	0	0	...	0	0	0	0	0	0	0	0	
4	1	0	0	0	1	1	1	1	0	0	...	0	0	0	0	0	0	0	0	

The full data set `train_df` 64 features. The target column is labeled as `prognosis`.

### 15.3.2 Holdout Train and Test Data

We split out a hold-out test set (25% of the data) so we can calculate an example MAP@K

```
import numpy as np
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(train_df.drop(target_column, axis=1),
                                                    random_state=42,
                                                    test_size=0.25,
                                                    stratify=train_df[target_column])
train = pd.DataFrame(np.hstack((X_train, np.array(y_train).reshape(-1, 1))))
test = pd.DataFrame(np.hstack((X_test, np.array(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]
print(train.shape)
print(test.shape)
train.head()
```

(189, 65)

(63, 65)

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	...	x56	x57	x58	x59	x60	x61	x62	x63
0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	...	0.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	...	x56	x57	x58	x59	x60	x61	x62	x63
1	1.0	0.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	...	0.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0
2	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	1.0	1.0	1.0	1.0	1.0	1.0	0.0	0.0	1.0	1.0	...	1.0	0.0	1.0	1.0	1.0	0.0	0.0	1.0
4	1.0	1.0	1.0	0.0	1.0	1.0	0.0	1.0	1.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

```
# add the dataset to the fun_control
fun_control.update({"data": train_df, # full dataset,
                    "train": train,
                    "test": test,
                    "n_samples": n_samples,
                    "target_column": target_column})
```

## 15.4 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:

```
# 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(),
# )
```

## 15.5 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:

```
add_core_model_to_fun_control(SVC, fun_control, SklearnHyperDict)
```

Other core\_models are, e.g.,:

- RidgeCV
- GradientBoostingRegressor
- ElasticNet
- RandomForestClassifier
- LogisticRegression
- KNeighborsClassifier
- RandomForestClassifier
- GradientBoostingClassifier
- HistGradientBoostingClassifier

We will use the `RandomForestClassifier` classifier in this example.

```
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
from spotPython.hyperparameters.values import add_core_model_to_fun_control
from spotPython.data.sklearn_hyper_dict import SklearnHyperDict
from spotPython.fun.hypersklearn import HyperSklearn

# core_model = RidgeCV
# core_model = GradientBoostingRegressor
# core_model = ElasticNet
core_model = RandomForestClassifier
# core_model = SVC
# core_model = LogisticRegression
# core_model = KNeighborsClassifier
```

```

# core_model = GradientBoostingClassifier
add_core_model_to_fun_control(core_model=core_model,
                             fun_control=fun_control,
                             hyper_dict=SklearnHyperDict,
                             filename=None)

```

Now `fun_control` has the information from the JSON file. The available hyperparameters are:

```

print(*fun_control["core_model_hyper_dict"].keys(), sep="\n")

n_estimators
criterion
max_depth
min_samples_split
min_samples_leaf
min_weight_fraction_leaf
max_features
max_leaf_nodes
min_impurity_decrease
bootstrap
oob_score

```

## 15.6 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model`

### 15.6.1 Modify hyperparameter of type numeric and integer (boolean)

Numeric and boolean values can be modified using the `modify_hyper_parameter_bounds` method. For example, to change the `tol` hyperparameter of the SVC model to the interval [1e-3, 1e-2], the following code can be used:

```

modify_hyper_parameter_bounds(fun_control, "tol", bounds=[1e-3, 1e-2])

from spotPython.hyperparameters.values import modify_hyper_parameter_bounds
# modify_hyper_parameter_bounds(fun_control, "tol", bounds=[1e-3, 1e-2])

```

### 15.6.2 Modify hyperparameter of type factor

spotPython provides functions for modifying the hyperparameters, their bounds and factors as well as for activating and de-activating hyperparameters without re-compilation of the Python source code. These functions were described in Section 14.6.

Factors can be modified with the `modify_hyper_parameter_levels` function. For example, to exclude the `sigmoid` kernel from the tuning, the `kernel` hyperparameter of the SVC model can be modified as follows:

```
modify_hyper_parameter_levels(fun_control, "kernel", ["linear", "rbf"])
```

The new setting can be controlled via:

```
fun_control["core_model_hyper_dict"]["kernel"]
```

```
from spotPython.hyperparameters.values import modify_hyper_parameter_levels
# XGBoost:
# modify_hyper_parameter_levels(fun_control, "loss", ["log_loss"])
```

**i** Note: RandomForestClassifier and Out-of-bag Estimation

Since `oob_score` requires the `bootstrap` hyperparameter to `True`, we set the `oob_score` parameter to `False`. The `oob_score` is later discussed in Section 15.7.3.

```
modify_hyper_parameter_bounds(fun_control, "bootstrap", bounds=[0, 1])
modify_hyper_parameter_bounds(fun_control, "oob_score", bounds=[0, 0])
```

### 15.6.3 Optimizers

Optimizers are described in Section 14.6.1.

### 15.6.4 Selection of the Objective: Metric and Loss Functions

- Machine learning models are optimized with respect to a metric, for example, the `accuracy` function.
- Deep learning, e.g., neural networks are optimized with respect to a loss function, for example, the `cross_entropy` function and evaluated with respect to a metric, for example, the `accuracy` function.

## 15.7 Step 7: Selection of the Objective (Loss) Function

The loss function, that is usually used in deep learning for optimizing the weights of the net, is stored in the `fun_control` dictionary as "loss\_function".

### 15.7.1 Metric Function

There are two different types of metrics in `spotPython`:

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.

We will consider multi-class classification metrics, e.g., `mapk_score` and `top_k_accuracy_score`.

#### Predict Probabilities

In this multi-class classification example the machine learning algorithm should return the probabilities of the specific classes ("`predict_proba`") instead of the predicted values.

We set "`predict_proba`" to True in the `fun_control` dictionary.

#### 15.7.1.1 The MAPK Metric

To select the MAPK metric, the following two entries can be added to the `fun_control` dictionary:

```
"metric_sklearn": mapk_score"  
"metric_params": {"k": 3}.
```

#### 15.7.1.2 Other Metrics

Alternatively, other metrics for multi-class classification can be used, e.g.: \* `top_k_accuracy_score` or \* `roc_auc_score`

The metric `roc_auc_score` requires the parameter "`multi_class`", e.g.,

```
"multi_class": "ovr".
```

This is set in the `fun_control` dictionary.

### **i** Weights

spotPython performs a minimization, therefore, metrics that should be maximized have to be multiplied by -1. This is done by setting "weights" to -1.

- The complete setup for the metric in our example is:

```
from spotPython.utils.metrics import mapk_score
fun_control.update({
    "weights": -1,
    "metric_sklearn": mapk_score,
    "predict_proba": True,
    "metric_params": {"k": 3},
})
```

### 15.7.2 Evaluation on Hold-out Data

- The default method for computing the performance is "eval\_holdout".
- Alternatively, cross-validation can be used for every machine learning model.
- Specifically for RandomForests, the OOB-score can be used.

```
fun_control.update({
    "eval": "train_hold_out",
})
```

### 15.7.3 OOB Score

Using the OOB-Score is a very efficient way to estimate the performance of a random forest classifier. The OOB-Score is calculated on the training data and does not require a hold-out test set. If the OOB-Score is used, the key "eval" in the `fun_control` dictionary should be set to "oob\_score" as shown below.

### **i** OOB-Score

In addition to setting the key "eval" in the `fun_control` dictionary to "oob\_score", the keys "oob\_score" and "bootstrap" have to be set to `True`, because the OOB-Score requires the bootstrap method.

- Uncomment the following lines to use the OOB-Score:

```

fun_control.update({
    "eval": "eval_oob_score",
})
modify_hyper_parameter_bounds(fun_control, "bootstrap", bounds=[1, 1])
modify_hyper_parameter_bounds(fun_control, "oob_score", bounds=[1, 1])

```

### 15.7.3.1 Cross Validation

Instead of using the OOB-score, the classical cross validation can be used. The number of folds is set by the key "k\_folds". For example, to use 5-fold cross validation, the key "k\_folds" is set to 5. Uncomment the following line to use cross validation:

```

# fun_control.update({
#     "eval": "train_cv",
#     "k_folds": 10,
# })

```

## 15.8 Step 8: Calling the SPOT Function

### 15.8.1 Preparing the SPOT Call

- Get types and variable names as well as lower and upper bounds for the hyperparameters.

```

# extract the variable types, names, and bounds
from spotPython.hyperparameters.values import (get_bound_values,
                                                get_var_name,
                                                get_var_type)
var_type = get_var_type(fun_control)
var_name = get_var_name(fun_control)
lower = get_bound_values(fun_control, "lower")
upper = get_bound_values(fun_control, "upper")

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

```

name	type	default	lower	upper	transform
n_estimators	int	7	5	10	transform_power_2_int

criterion	factor	gini	0	2	None
max_depth	int	10	1	20	transform_power_2_int
min_samples_split	int	2	2	100	None
min_samples_leaf	int	1	1	25	None
min_weight_fraction_leaf	float	0.0	0	0.01	None
max_features	factor	sqrt	0	1	transform_none_to_None
max_leaf_nodes	int	10	7	12	transform_power_2_int
min_impurity_decrease	float	0.0	0	0.01	None
bootstrap	factor	1	1	1	None
oob_score	factor	0	1	1	None

### 15.8.2 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
```

### 15.8.3 Run the Spot Optimizer

- Run SPOT for approx. x mins (`max_time`).
- Note: the run takes longer, because the evaluation time of initial design (here: `initi_size`, 20 points) is not considered.

```
from spotPython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
X_start

array([[ 7.,  0., 10.,  2.,  1.,  0.,  0., 10.,  0.,  1.,  0.]])
```

```
import numpy as np
from spotPython.spot import spot
from math import inf
spot_tuner = spot.Spot(fun=fun,
                      lower = lower,
                      upper = upper,
                      fun_evals = inf,
                      fun_repeats = 1,
```

```
    max_time = MAX_TIME,
    noise = False,
    tolerance_x = np.sqrt(np.spacing(1)),
    var_type = var_type,
    var_name = var_name,
    infill_criterion = "y",
    n_points = 1,
    seed=123,
    log_level = 50,
    show_models= False,
    show_progress= True,
    fun_control = fun_control,
    design_control={"init_size": INIT_SIZE,
                    "repeats": 1},
    surrogate_control={"noise": True,
                      "cod_type": "norm",
                      "min_theta": -4,
                      "max_theta": 3,
                      "n_theta": len(var_name),
                      "model_fun_evals": 10_000,
                      "log_level": 50
                     })
spot_tuner.run(X_start=X_start)
```

```
spotPython tuning: -0.8544973544973545 [-----] 1.57%
```

```
spotPython tuning: -0.8544973544973545 [-----] 2.59%
```

```
spotPython tuning: -0.8544973544973545 [-----] 3.78%
```

```
spotPython tuning: -0.8544973544973545 [#-----] 5.52%
```

```
spotPython tuning: -0.8544973544973545 [#-----] 7.55%
```

```
spotPython tuning: -0.8544973544973545 [#-----] 9.65%
```

```
spotPython tuning: -0.8544973544973545 [#-----] 11.85%
```

```
spotPython tuning: -0.8544973544973545 [#-----] 14.51%
```

```
spotPython tuning: -0.8544973544973545 [##-----] 16.51%
spotPython tuning: -0.8544973544973545 [##-----] 19.39%
spotPython tuning: -0.8544973544973545 [##-----] 21.32%
spotPython tuning: -0.8544973544973545 [##-----] 23.42%
spotPython tuning: -0.8544973544973545 [###-----] 26.48%
spotPython tuning: -0.8544973544973545 [###-----] 29.06%
spotPython tuning: -0.8544973544973545 [###-----] 31.42%
spotPython tuning: -0.8544973544973545 [#####----] 37.52%
spotPython tuning: -0.8544973544973545 [#####----] 43.36%
spotPython tuning: -0.8544973544973545 [#####----] 47.46%
spotPython tuning: -0.8544973544973545 [#####----] 51.15%
spotPython tuning: -0.8580246913580246 [#####----] 54.20%
spotPython tuning: -0.8580246913580246 [#####----] 62.96%
spotPython tuning: -0.8580246913580246 [#####----] 70.38%
spotPython tuning: -0.8580246913580246 [#####---] 78.44%
spotPython tuning: -0.8580246913580246 [#####---] 90.05%
spotPython tuning: -0.8580246913580246 [#####---] 100.00% Done...
<spotPython.spot.spot at 0x2d264ac20>
```

## 15.9 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard as described in Section 14.9, see also the description in the documentation: [Tensorboard](#).

## 15.10 Step 10: Results

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized. The following code generates the progress plot from `?@fig-progress`.

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

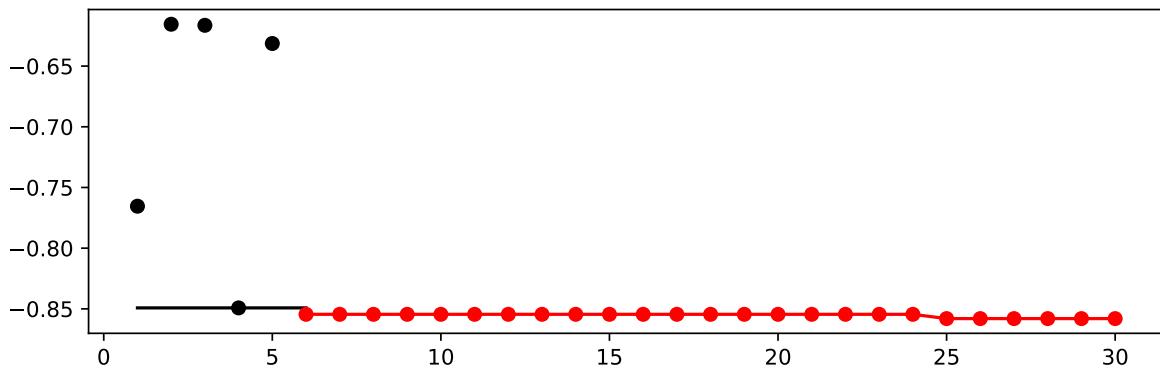


Figure 15.1: Progress plot. *Black dots* denote results from the initial design. *Red dots* illustrate the improvement found by the surrogate model based optimization.

- Print the results

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

name	type	default	lower	upper	tuned
n_estimators	int	7	5.0	10.0	7.0
criterion	factor	gini	0.0	2.0	1.0
max_depth	int	10	1.0	20.0	4.0
min_samples_split	int	2	2.0	100.0	9.0
min_samples_leaf	int	1	1.0	25.0	1.0

min_weight_fraction_leaf	float	0.0	0.0	0.01	0.0	0.0
max_features	factor	sqrt	0.0	1.0	0.0	0.0
max_leaf_nodes	int	10	7.0	12.0	12.0	12.0
min_impurity_decrease	float	0.0	0.0	0.01	0.003665798756399038	0.003665798756399038
bootstrap	factor	1	1.0	1.0	1.0	1.0
oob_score	factor	0	1.0	1.0	1.0	1.0

### 15.10.1 Show variable importance

```
spot_tuner.plot_importance(threshold=0.025, filename="./figures/" + experiment_name+"_importance")
```

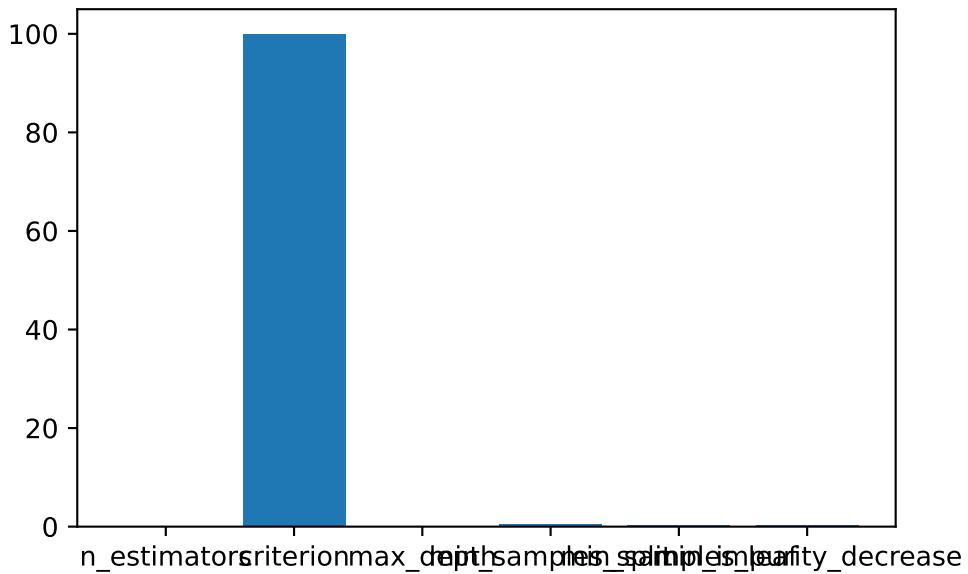


Figure 15.2: Variable importance plot, threshold 0.025.

### 15.10.2 Get Default Hyperparameters

```
from spotPython.hyperparameters.values import get_default_values, transform_hyper_parameters
values_default = get_default_values(fun_control)
values_default = transform_hyper_parameter_values(fun_control=fun_control, hyper_parameter_values_default)
```

```
{'n_estimators': 128,
```

```

'criterion': 'gini',
'max_depth': 1024,
'min_samples_split': 2,
'min_samples_leaf': 1,
'min_weight_fraction_leaf': 0.0,
'max_features': 'sqrt',
'max_leaf_nodes': 1024,
'min_impurity_decrease': 0.0,
'bootstrap': 1,
'oob_score': 0}

from sklearn.pipeline import make_pipeline
model_default = make_pipeline(fun_control["prep_model"], fun_control["core_model"](**value)
model_default

Pipeline(steps=[('nonetype', None),
               ('randomforestclassifier',
                RandomForestClassifier(bootstrap=1, max_depth=1024,
                                      max_leaf_nodes=1024, n_estimators=128,
                                      oob_score=0))])

```

### 15.10.3 Get SPOT Results

```

X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
print(X)

[[7.0000000e+00 1.0000000e+00 4.0000000e+00 9.0000000e+00
 1.0000000e+00 0.0000000e+00 0.0000000e+00 1.2000000e+01
 3.66579876e-03 1.0000000e+00 1.0000000e+00]]

from spotPython.hyperparameters.values import assign_values, return_conf_list_from_var_dict
v_dict = assign_values(X, fun_control["var_name"])
return_conf_list_from_var_dict(var_dict=v_dict, fun_control=fun_control)

[{'n_estimators': 128,
 'criterion': 'entropy',
 'max_depth': 16,
 'min_samples_split': 9,

```

```

'min_samples_leaf': 1,
'min_weight_fraction_leaf': 0.0,
'max_features': 'sqrt',
'max_leaf_nodes': 4096,
'min_impurity_decrease': 0.003665798756399038,
'bootstrap': 1,
'oob_score': 1}]

from spotPython.hyperparameters.values import get_one_sklearn_model_from_X
model_spot = get_one_sklearn_model_from_X(X, fun_control)
model_spot

RandomForestClassifier(bootstrap=1, criterion='entropy', max_depth=16,
                      max_leaf_nodes=4096,
                      min_impurity_decrease=0.003665798756399038,
                      min_samples_split=9, n_estimators=128, oob_score=1)

```

#### 15.10.4 Evaluate SPOT Results

- Fetch the data.

```

from spotPython.utils.convert import get_Xy_from_df
X_train, y_train = get_Xy_from_df(fun_control["train"], fun_control["target_column"])
X_test, y_test = get_Xy_from_df(fun_control["test"], fun_control["target_column"])
X_test.shape, y_test.shape

```

((63, 64), (63,))

- Fit the model with the tuned hyperparameters. This gives one result:

```

model_spot.fit(X_train, y_train)
y_pred = model_spot.predict_proba(X_test)
res = mapk_score(y_true=y_test, y_pred=y_pred, k=3)
res

```

0.8465608465608466

```

def repeated_eval(n, model):
    res_values = []
    for i in range(n):
        model.fit(X_train, y_train)
        y_pred = model.predict_proba(X_test)
        res = mapk_score(y_true=y_test, y_pred=y_pred, k=3)
        res_values.append(res)
    mean_res = np.mean(res_values)
    print(f"mean_res: {mean_res}")
    std_res = np.std(res_values)
    print(f"std_res: {std_res}")
    min_res = np.min(res_values)
    print(f"min_res: {min_res}")
    max_res = np.max(res_values)
    print(f"max_res: {max_res}")
    median_res = np.median(res_values)
    print(f"median_res: {median_res}")
    return mean_res, std_res, min_res, max_res, median_res

```

### 15.10.5 Handling Non-deterministic Results

- Because the model is non-deterministic, we perform  $n = 30$  runs and calculate the mean and standard deviation of the performance metric.

```
_ = repeated_eval(30, model_spot)
```

```

mean_res: 0.8451499118165785
std_res: 0.011239149182613631
min_res: 0.8174603174603174
max_res: 0.8650793650793651
median_res: 0.8439153439153438

```

### 15.10.6 Evaluation of the Default Hyperparameters

```
model_default.fit(X_train, y_train) ["randomforestclassifier"]
```

```

RandomForestClassifier(bootstrap=1, max_depth=1024, max_leaf_nodes=1024,
                      n_estimators=128, oob_score=0)

```

- One evaluation of the default hyperparameters is performed on the hold-out test set.

```
y_pred = model_default.predict_proba(X_test)
mapk_score(y_true=y_test, y_pred=y_pred, k=3)
```

0.8650793650793651

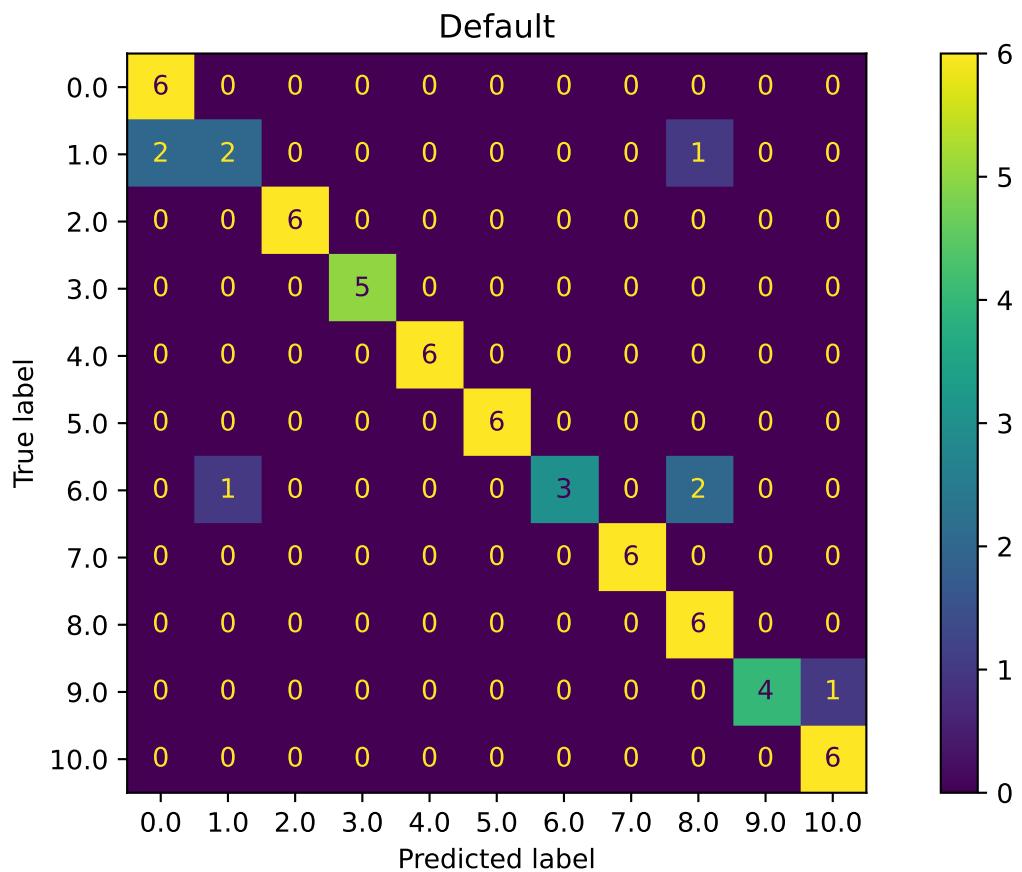
Since one single evaluation is not meaningful, we perform, similar to the evaluation of the SPOT results,  $n = 30$  runs of the default setting and calculate the mean and standard deviation of the performance metric.

```
_ = repeated_eval(30, model_default)
```

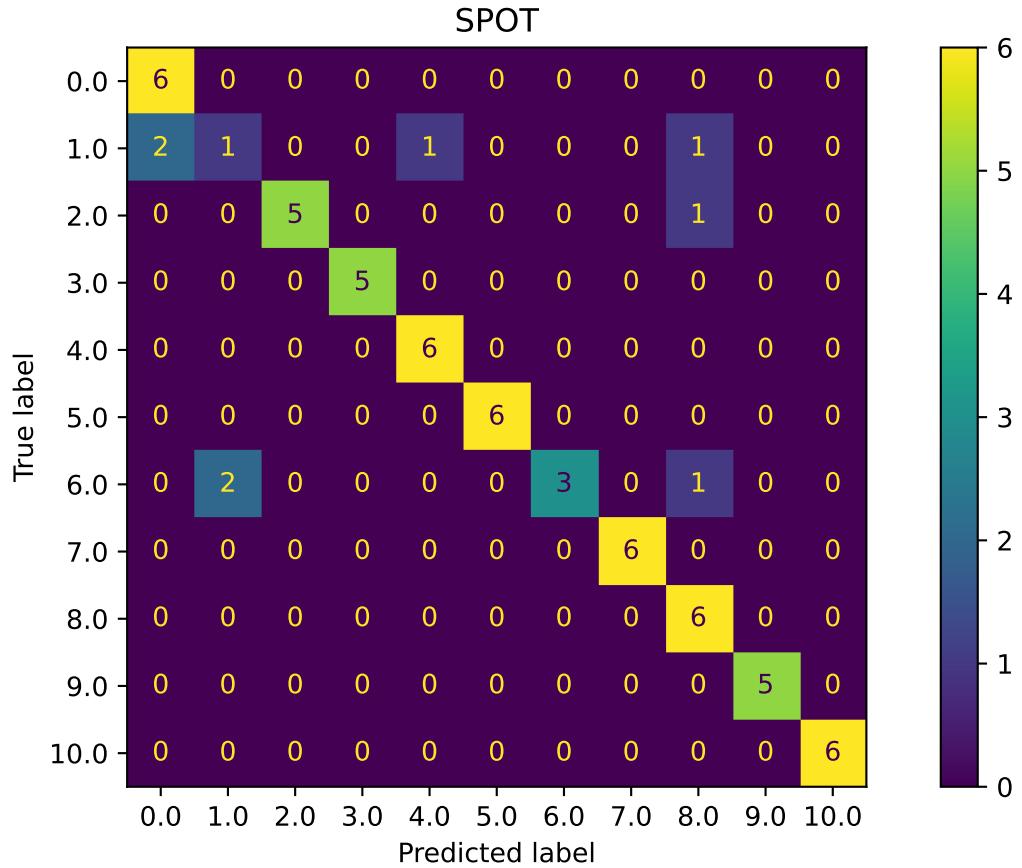
```
mean_res: 0.8536155202821869
std_res: 0.009273944850183373
min_res: 0.8253968253968254
max_res: 0.8703703703703705
median_res: 0.8544973544973544
```

### 15.10.7 Plot: Compare Predictions

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



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



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

```
(-0.8580246913580246, -0.6155202821869489)
```

### 15.10.8 Cross-validated Evaluations

```
from spotPython.sklearn.traintest import evaluate_cv
fun_control.update({
    "eval": "train_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

```
(0.8622319688109161, None)
```

```
fun_control.update({
    "eval": "test_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

```
Error in fun_sklearn(). Call to evaluate_cv failed. err=ValueError('n_splits=10 cannot be gr
(nan, None)
```

- This is the evaluation that will be used in the comparison:

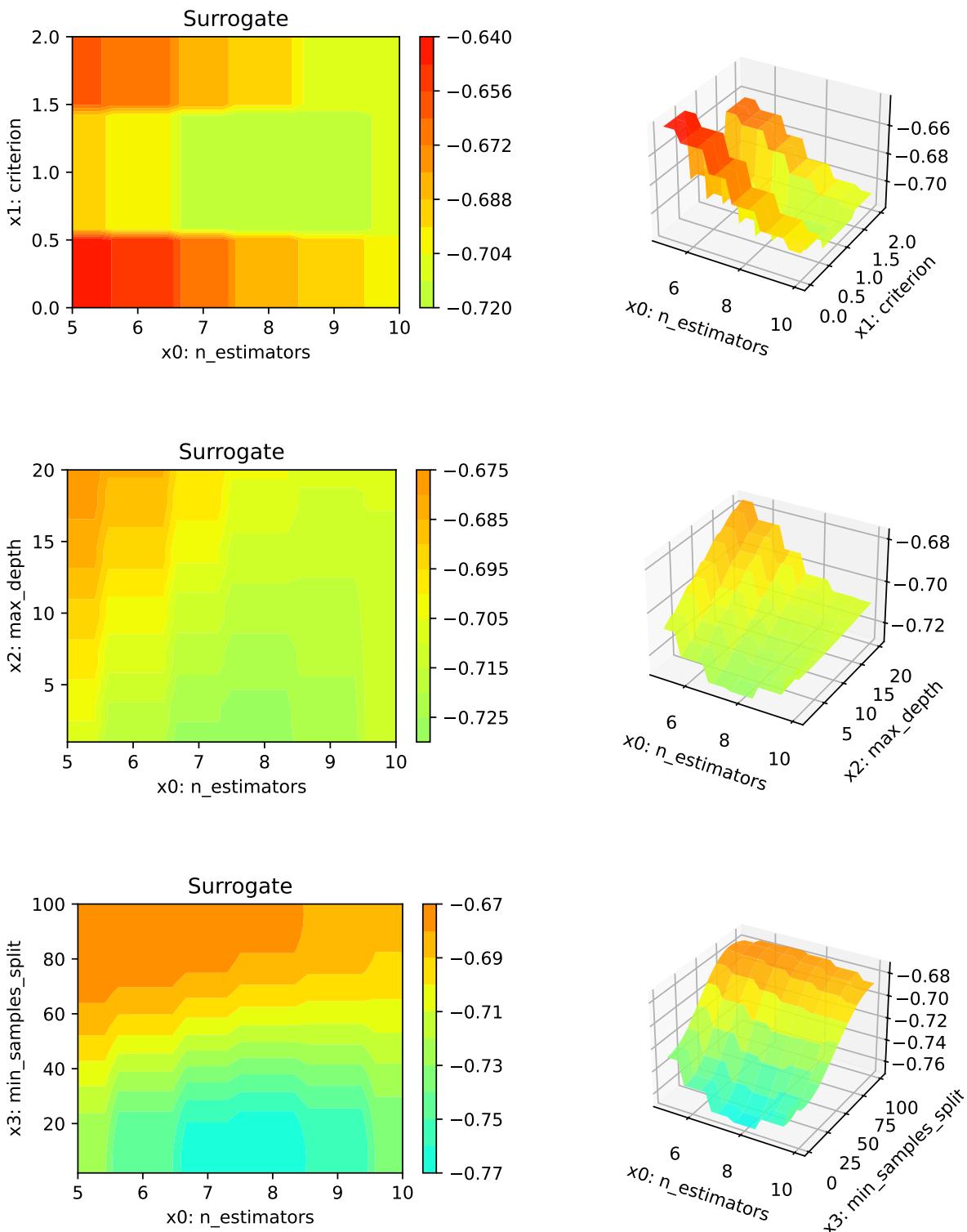
```
fun_control.update({
    "eval": "data_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

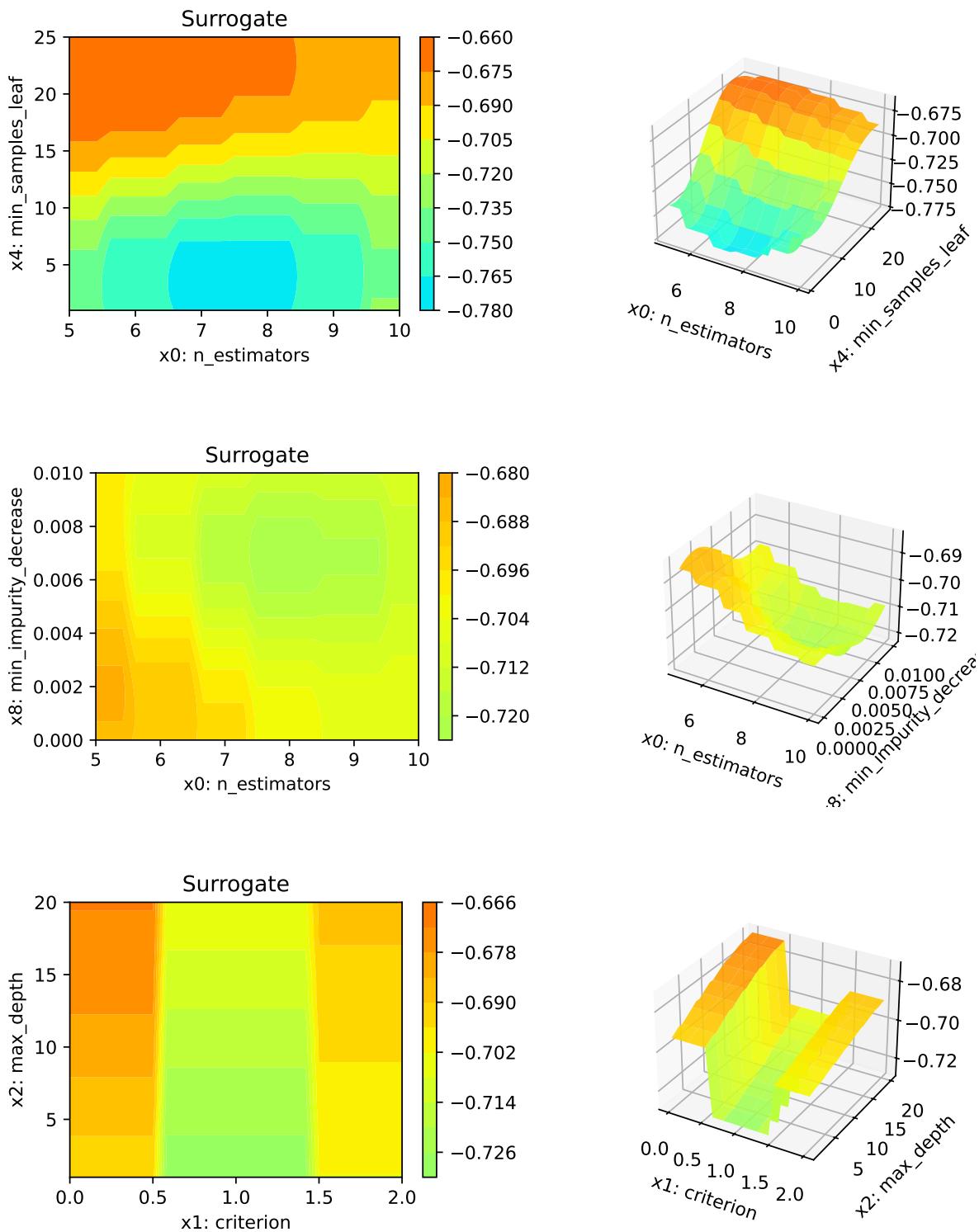
```
(0.8753076923076923, None)
```

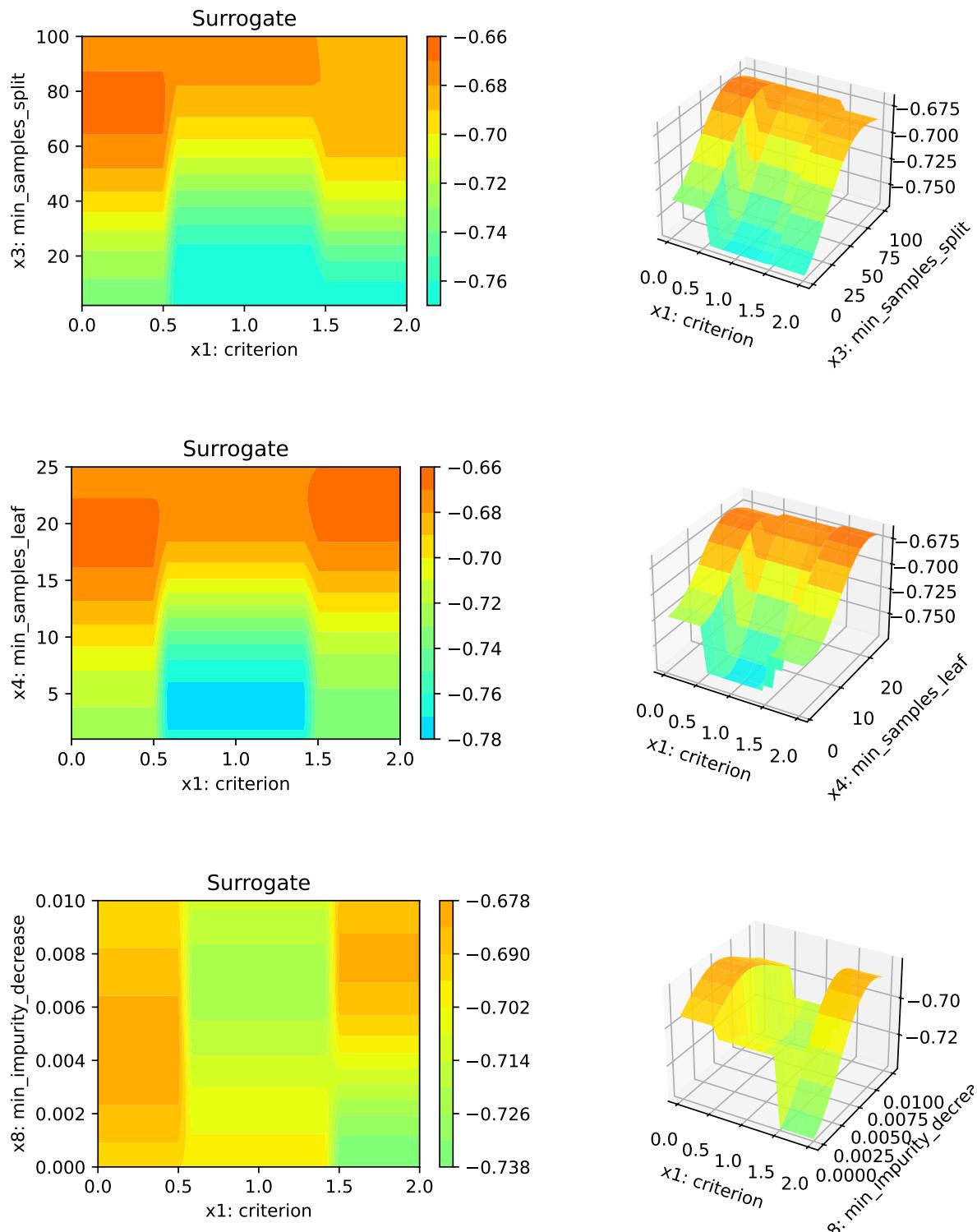
### 15.10.9 Detailed Hyperparameter Plots

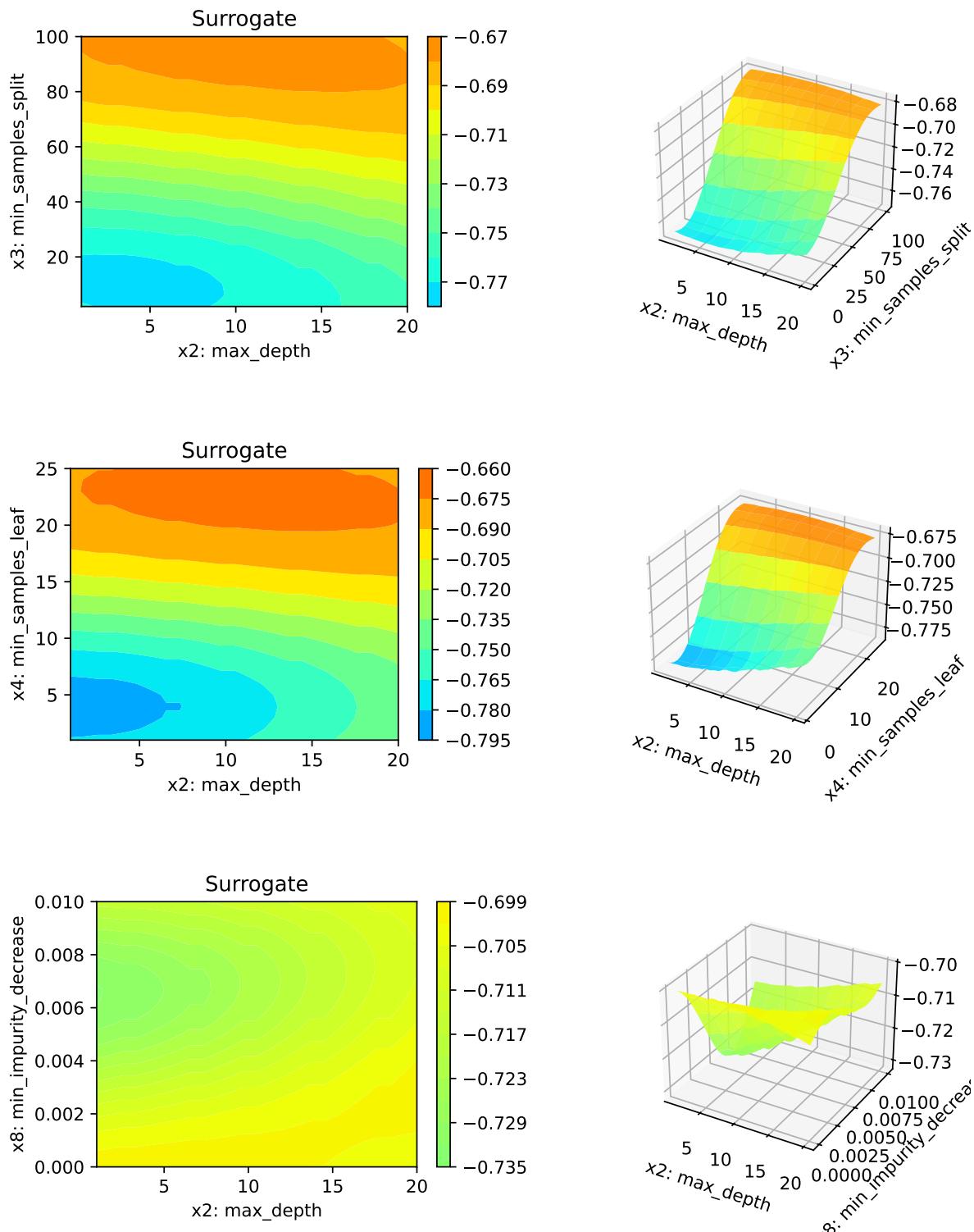
```
filename = "./figures/" + experiment_name
spot_tuner.plot_important_hyperparameter_contour(filename)

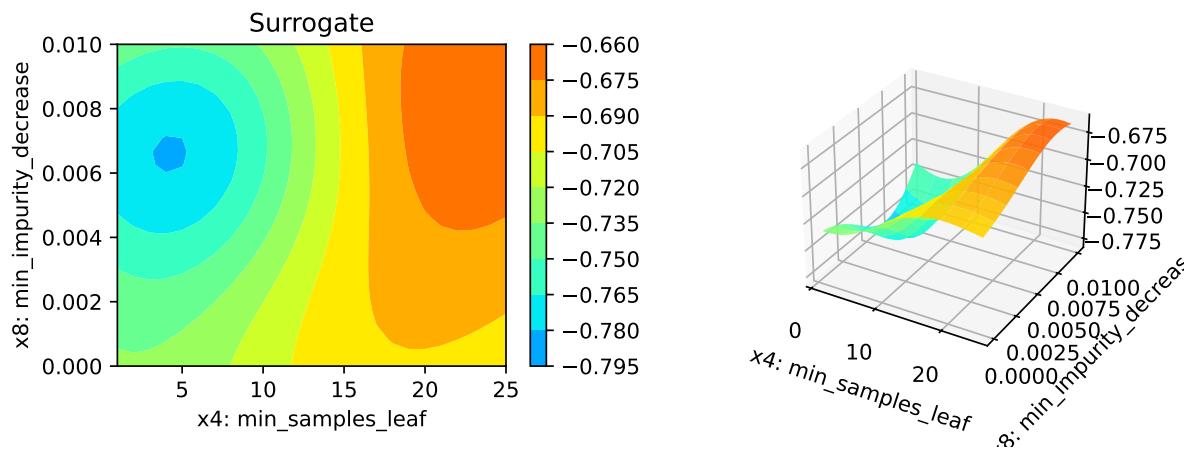
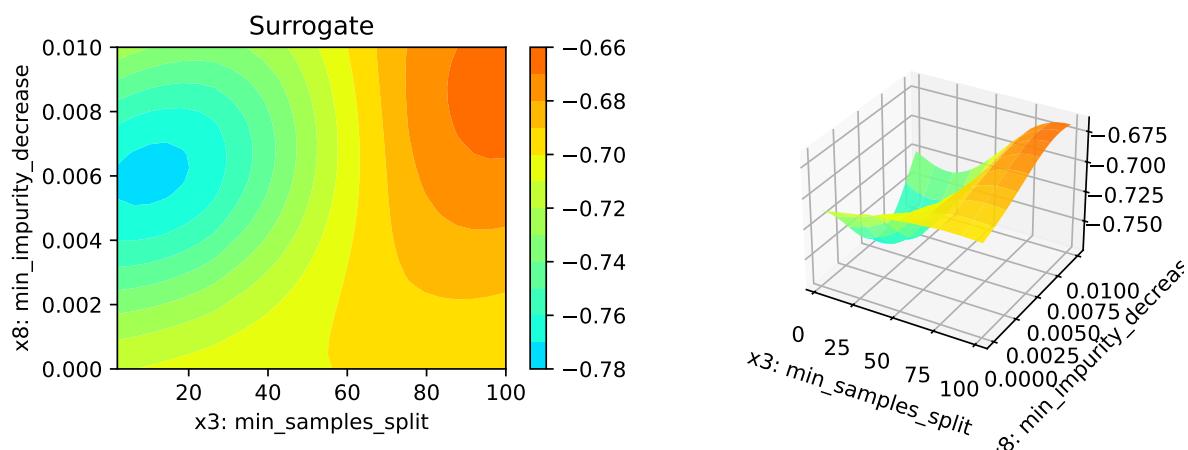
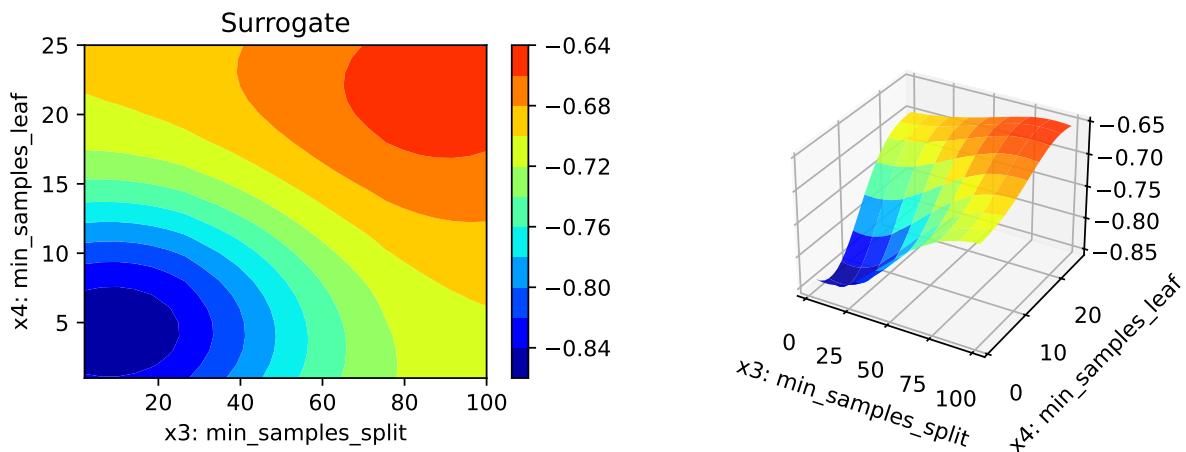
n_estimators: 0.18598329755189402
criterion: 100.0
max_depth: 0.05150030996613426
min_samples_split: 0.4128237333449712
min_samples_leaf: 0.3648379064824644
min_impurity_decrease: 0.316135400232626
```











### 15.10.10 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

### 15.10.11 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)
```

# 16 HPT: sklearn XGB Classifier VBDP Data

This chapter describes the hyperparameter tuning of a `HistGradientBoostingClassifier` on the Vector Borne Disease Prediction (VBDP) data set.

## ! Vector Borne Disease Prediction Data Set

This chapter uses the Vector Borne Disease Prediction data set from Kaggle. It is a categorical dataset for eleven Vector Borne Diseases with associated symptoms.

The person who associated a work with this deed has dedicated the work to the public domain by waiving all of his or her rights to the work worldwide under copyright law, including all related and neighboring rights, to the extent allowed by law. You can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission. See Other Information below, see <https://creativecommons.org/publicdomain/zero/1.0/>.

The data set is available at: <https://www.kaggle.com/datasets/richardbernat/vector-borne-disease-prediction>,

The data should be downloaded and stored in the `data/VBDP` subfolder. The data set is not available as a part of the `spotPython` package.

## 16.1 Step 1: Setup

Before we consider the detailed experimental setup, we select the parameters that affect run time and the initial design size.

```
MAX_TIME = 1
INIT_SIZE = 5
ORIGINAL = True
PREFIX = "17"

import warnings
warnings.filterwarnings("ignore")
```

## 16.2 Step 2: Initialization of the Empty fun\_control Dictionary

```
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_experiment_name, get_spot_tensorboard_path
from spotPython.utils.device import getDevice

experiment_name = get_experiment_name(prefix=PREFIX)

fun_control = fun_control_init(
    task="classification",
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
```

## 16.3 Step 3: PyTorch Data Loading

### 16.3.1 1. Load Data: Classification VBDP

```
import pandas as pd
if ORIGINAL == True:
    train_df = pd.read_csv('./data/VBDP/trainn.csv')
    test_df = pd.read_csv('./data/VBDP/testt.csv')
else:
    train_df = pd.read_csv('./data/VBDP/train.csv')
    # remove the id column
    train_df = train_df.drop(columns=['id'])

from sklearn.preprocessing import OrdinalEncoder
n_samples = train_df.shape[0]
n_features = train_df.shape[1] - 1
target_column = "prognosis"
# Encoder our prognosis labels as integers for easier decoding later
enc = OrdinalEncoder()
train_df[target_column] = enc.fit_transform(train_df[[target_column]])
train_df.columns = [f"x{i}" for i in range(1, n_features+1)] + [target_column]
print(train_df.shape)
train_df.head()
```

(252, 65)

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	...	x56	x57	x58	x59	x60	x61	x62	x63	x64
0	0	1	1	1	1	0	1	0	0	0	...	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	0	1	1	1	0	...	0	0	0	0	0	0	0	0	0
2	0	1	0	1	0	0	1	1	0	0	...	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	1	1	1	0	0	...	0	0	0	0	0	0	0	0	0
4	1	0	0	0	1	1	1	1	0	0	...	0	0	0	0	0	0	0	0	0

The full data set `train_df` 64 features. The target column is labeled as `prognosis`.

### 16.3.2 Holdout Train and Test Data

We split out a hold-out test set (25% of the data) so we can calculate an example MAP@K

```
import numpy as np
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(train_df.drop(target_column, axis=1),
                                                    random_state=42,
                                                    test_size=0.25,
                                                    stratify=train_df[target_column])
train = pd.DataFrame(np.hstack((X_train, np.array(y_train).reshape(-1, 1))))
test = pd.DataFrame(np.hstack((X_test, np.array(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]
print(train.shape)
print(test.shape)
train.head()
```

(189, 65)  
(63, 65)

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	...	x56	x57	x58	x59	x60	x61	x62	x63
0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	...	0.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0
1	1.0	0.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	...	0.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0
2	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	1.0	1.0	1.0	1.0	1.0	1.0	0.0	0.0	1.0	1.0	...	1.0	0.0	1.0	1.0	1.0	0.0	0.0	1.0
4	1.0	1.0	1.0	0.0	1.0	1.0	0.0	1.0	1.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

```

# add the dataset to the fun_control
fun_control.update({"data": train_df, # full dataset,
                    "train": train,
                    "test": test,
                    "n_samples": n_samples,
                    "target_column": target_column})

```

## 16.4 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:

```

# 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(),
# )

```

## 16.5 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:

```
add_core_model_to_fun_control(SVC, fun_control, SklearnHyperDict)
```

Other core\_models are, e.g.,:

- RidgeCV
- GradientBoostingRegressor
- ElasticNet
- RandomForestClassifier
- LogisticRegression
- KNeighborsClassifier
- RandomForestClassifier
- GradientBoostingClassifier
- HistGradientBoostingClassifier

We will use the `RandomForestClassifier` classifier in this example.

```
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.ensemble import HistGradientBoostingClassifier
from sklearn.linear_model import ElasticNet
from spotPython.hyperparameters.values import add_core_model_to_fun_control
from spotPython.data.sklearn_hyper_dict import SklearnHyperDict
from spotPython.fun.hypersklearn import HyperSklearn

# core_model = RidgeCV
# core_model = GradientBoostingRegressor
# core_model = ElasticNet
core_model = RandomForestClassifier
# core_model = SVC
# core_model = LogisticRegression
# core_model = KNeighborsClassifier
# core_model = GradientBoostingClassifier
core_model = HistGradientBoostingClassifier
add_core_model_to_fun_control(core_model=core_model,
                             fun_control=fun_control,
                             hyper_dict=SklearnHyperDict,
                             filename=None)
```

Now `fun_control` has the information from the JSON file. The available hyperparameters are:

```
print(*fun_control["core_model_hyper_dict"].keys(), sep="\n")  
  
loss  
learning_rate  
max_iter  
max_leaf_nodes  
max_depth  
min_samples_leaf  
l2_regularization  
max_bins  
early_stopping  
n_iter_no_change  
tol
```

## 16.6 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model`

### 16.6.1 Modify hyperparameter of type numeric and integer (boolean)

Numeric and boolean values can be modified using the `modify_hyper_parameter_bounds` method. For example, to change the `tol` hyperparameter of the SVC model to the interval [1e-3, 1e-2], the following code can be used:

```
modify_hyper_parameter_bounds(fun_control, "tol", bounds=[1e-3, 1e-2])  
  
from spotPython.hyperparameters.values import modify_hyper_parameter_bounds  
# modify_hyper_parameter_bounds(fun_control, "tol", bounds=[1e-3, 1e-2])  
# modify_hyper_parameter_bounds(fun_control, "min_samples_split", bounds=[3, 20])  
# modify_hyper_parameter_bounds(fun_control, "dual", bounds=[0, 0])  
# modify_hyper_parameter_bounds(fun_control, "probability", bounds=[1, 1])  
# fun_control["core_model_hyper_dict"]["tol"]  
# modify_hyper_parameter_bounds(fun_control, "min_samples_leaf", bounds=[1, 25])  
# modify_hyper_parameter_bounds(fun_control, "n_estimators", bounds=[5, 10])
```

### 16.6.2 Modify hyperparameter of type factor

`spotPython` provides functions for modifying the hyperparameters, their bounds and factors as well as for activating and de-activating hyperparameters without re-compilation of the Python source code. These functions were described in Section 14.6.

Factors can be modified with the `modify_hyper_parameter_levels` function. For example, to exclude the `sigmoid` kernel from the tuning, the `kernel` hyperparameter of the SVC model can be modified as follows:

```
modify_hyper_parameter_levels(fun_control, "kernel", ["linear", "rbf"])
```

The new setting can be controlled via:

```
fun_control["core_model_hyper_dict"]["kernel"]
```

```
from spotPython.hyperparameters.values import modify_hyper_parameter_levels
# XGBoost:
modify_hyper_parameter_levels(fun_control, "loss", ["log_loss"])
```

### 16.6.3 Optimizers

Optimizers are described in Section 14.6.1.

## 16.7 Step 7: Selection of the Objective (Loss) Function

### 16.7.1 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 and
2. the loss function (and a metric).

### 16.7.2 Selection of the Objective: Metric and Loss Functions

- Machine learning models are optimized with respect to a metric, for example, the `accuracy` function.
- Deep learning, e.g., neural networks are optimized with respect to a loss function, for example, the `cross_entropy` function and evaluated with respect to a metric, for example, the `accuracy` function.

### 16.7.3 Loss Function

The loss function, that is usually used in deep learning for optimizing the weights of the net, is stored in the `fun_control` dictionary as "`loss_function`".

### 16.7.4 Metric Function

There are two different types of metrics in `spotPython`:

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.

We will consider multi-class classification metrics, e.g., `mapk_score` and `top_k_accuracy_score`.

#### Predict Probabilities

In this multi-class classification example the machine learning algorithm should return the probabilities of the specific classes ("`predict_proba`") instead of the predicted values.

We set "`predict_proba`" to `True` in the `fun_control` dictionary.

#### 16.7.4.1 The MAPK Metric

To select the MAPK metric, the following two entries can be added to the `fun_control` dictionary:

```
"metric_sklearn": mapk_score  
"metric_params": {"k": 3}.
```

#### 16.7.4.2 Other Metrics

Alternatively, other metrics for multi-class classification can be used, e.g.,: \* `top_k_accuracy_score` or \* `roc_auc_score`

The metric `roc_auc_score` requires the parameter "`multi_class`", e.g.,

```
"multi_class": "ovr".
```

This is set in the `fun_control` dictionary.

## **i** Weights

spotPython performs a minimization, therefore, metrics that should be maximized have to be multiplied by -1. This is done by setting "weights" to -1.

- The complete setup for the metric in our example is:

```
from spotPython.utils.metrics import mapk_score
fun_control.update({
    "weights": -1,
    "metric_sklearn": mapk_score,
    "predict_proba": True,
    "metric_params": {"k": 3},
})
```

### 16.7.5 Evaluation on Hold-out Data

- The default method for computing the performance is "eval\_holdout".
- Alternatively, cross-validation can be used for every machine learning model.
- Specifically for RandomForests, the OOB-score can be used.

```
fun_control.update({
    "eval": "train_hold_out",
})
```

#### 16.7.5.1 Cross Validation

Instead of using the OOB-score, the classical cross validation can be used. The number of folds is set by the key "k\_folds". For example, to use 5-fold cross validation, the key "k\_folds" is set to 5. Uncomment the following line to use cross validation:

```
# fun_control.update({
#     "eval": "train_cv",
#     "k_folds": 10,
# })
```

## 16.8 Step 8: Calling the SPOT Function

### 16.8.1 Preparing the SPOT Call

- Get types and variable names as well as lower and upper bounds for the hyperparameters.

```
# extract the variable types, names, and bounds
from spotPython.hyperparameters.values import (get_bound_values,
                                                get_var_name,
                                                get_var_type)
var_type = get_var_type(fun_control)
var_name = get_var_name(fun_control)
lower = get_bound_values(fun_control, "lower")
upper = get_bound_values(fun_control, "upper")

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

name	type	default	lower	upper	transform
loss	factor	log_loss	0	0	None
learning_rate	float	-1.0	-5	0	transform_power_10
max_iter	int	7	3	10	transform_power_2_int
max_leaf_nodes	int	5	1	12	transform_power_2_int
max_depth	int	2	1	20	transform_power_2_int
min_samples_leaf	int	4	2	10	transform_power_2_int
l2_regularization	float	0.0	0	10	None
max_bins	int	255	127	255	None
early_stopping	factor	1	0	1	None
n_iter_no_change	int	10	5	20	None
tol	float	0.0001	1e-05	0.001	None

### 16.8.2 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
```

### 16.8.3 Run the Spot Optimizer

- Run SPOT for approx. x mins (`max_time`).
- Note: the run takes longer, because the evaluation time of initial design (here: `initi_size`, 20 points) is not considered.

```
from spotPython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
X_start

array([[ 0.00e+00, -1.00e+00,  7.00e+00,  5.00e+00,  2.00e+00,  4.00e+00,
       0.00e+00,  2.55e+02,  1.00e+00,  1.00e+01,  1.00e-04]])
```

```
import numpy as np
from spotPython.spot import spot
from math import inf
spot_tuner = spot.Spot(fun=fun,
                       lower = lower,
                       upper = upper,
                       fun_evals = inf,
                       fun_repeats = 1,
                       max_time = MAX_TIME,
                       noise = False,
                       tolerance_x = np.sqrt(np.spacing(1)),
                       var_type = var_type,
                       var_name = var_name,
                       infill_criterion = "y",
                       n_points = 1,
                       seed=123,
                       log_level = 50,
                       show_models= False,
                       show_progress= True,
                       fun_control = fun_control,
                       design_control={"init_size": INIT_SIZE,
                                       "repeats": 1},
                       surrogate_control={"noise": True,
                                         "cod_type": "norm",
                                         "min_theta": -4,
                                         "max_theta": 3,
                                         "n_theta": len(var_name),
                                         "model_fun_evals": 10_000},
```

```
        "log_level": 50
    })
spot_tuner.run(X_start=X_start)

spotPython tuning: -0.84375 [##-----] 21.98%

spotPython tuning: -0.84375 [#####-----] 42.80%

spotPython tuning: -0.84375 [#####----] 45.05%

spotPython tuning: -0.84375 [#####---] 55.75%

spotPython tuning: -0.84375 [#####--] 80.60%

spotPython tuning: -0.864583333333334 [#####---] 94.27%

spotPython tuning: -0.864583333333334 [#####--] 98.50%

spotPython tuning: -0.864583333333334 [#####-] 100.00% Done...

<spotPython.spot.spot.Spot at 0x29560f5e0>
```

## 16.9 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard as described in Section 14.9, see also the description in the documentation: [Tensorboard](#).

## 16.10 Step 10: Results

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized. The following code generates the progress plot from `?@fig-progress`.

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

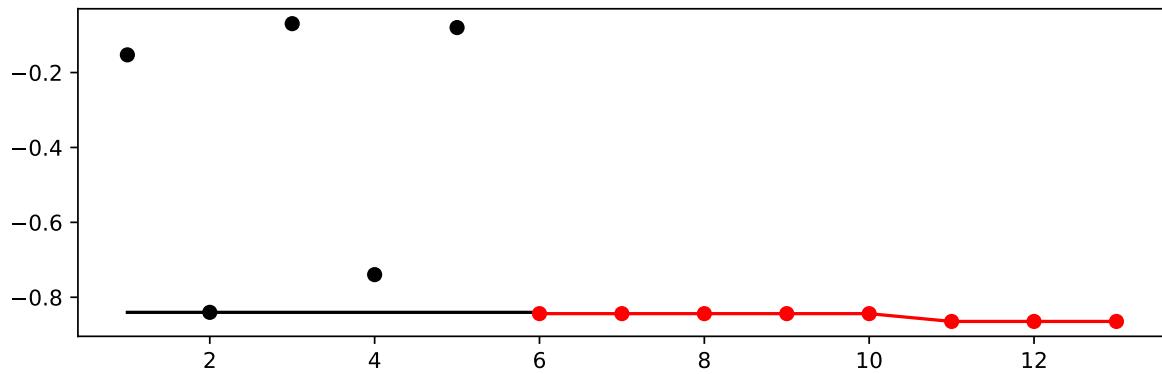


Figure 16.1: Progress plot. *Black dots* denote results from the initial design. *Red dots* illustrate the improvement found by the surrogate model based optimization.

- Print the results

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

name	type	default	lower	upper	tuned	transform
loss	factor	log_loss	0.0	0.0	0.0	None
learning_rate	float	-1.0	-5.0	0.0	-0.744606058195489	transform
max_iter	int	7	3.0	10.0	8.0	transform
max_leaf_nodes	int	5	1.0	12.0	12.0	transform
max_depth	int	2	1.0	20.0	2.0	transform
min_samples_leaf	int	4	2.0	10.0	2.0	transform
l2_regularization	float	0.0	0.0	10.0	0.0	None
max_bins	int	255	127.0	255.0	204.0	None
early_stopping	factor	1	0.0	1.0	1.0	None
n_iter_no_change	int	10	5.0	20.0	20.0	None
tol	float	0.0001	1e-05	0.001	0.001	None

### 16.10.1 Show variable importance

```
spot_tuner.plot_importance(threshold=0.025, filename=".//figures/" + experiment_name+"_imp
```

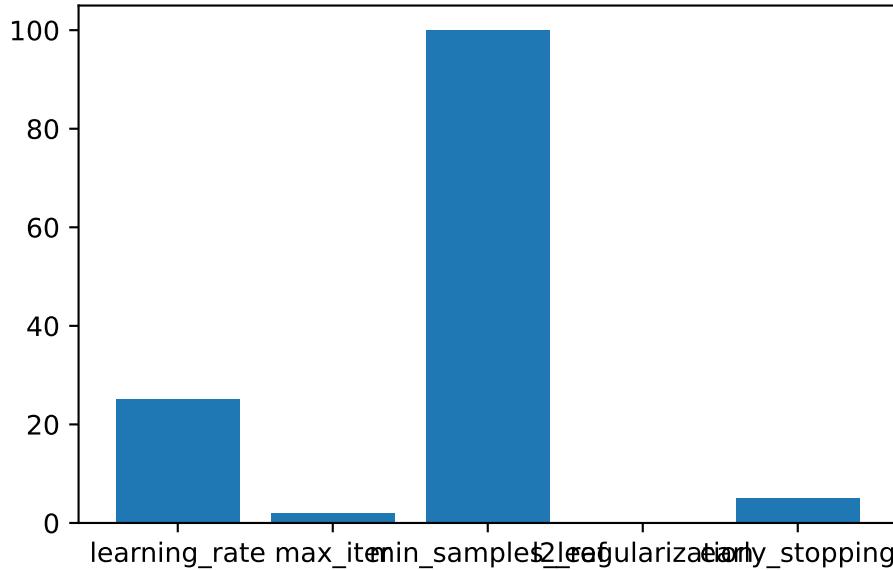


Figure 16.2: Variable importance plot, threshold 0.025.

### 16.10.2 Get Default Hyperparameters

```
from spotPython.hyperparameters.values import get_default_values, transform_hyper_parameters
values_default = get_default_values(fun_control)
values_default = transform_hyper_parameter_values(fun_control=fun_control, hyper_parameter=hyper_parameter)
values_default
```

```
{'loss': 'log_loss',
 'learning_rate': 0.1,
 'max_iter': 128,
 'max_leaf_nodes': 32,
 'max_depth': 4,
 'min_samples_leaf': 16,
 'l2_regularization': 0.0,
 'max_bins': 255,
 'early_stopping': 1,
```

```

'n_iter_no_change': 10,
'tol': 0.0001}

from sklearn.pipeline import make_pipeline
model_default = make_pipeline(fun_control["prep_model"], fun_control["core_model"](**value
model_default

Pipeline(steps=[('nonetype', None),
                ('histgradientboostingclassifier',
                 HistGradientBoostingClassifier(early_stopping=1, max_depth=4,
                                                max_iter=128, max_leaf_nodes=32,
                                                min_samples_leaf=16,
                                                tol=0.0001))])

```

### 16.10.3 Get SPOT Results

```

X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
print(X)

[[ 0.0000000e+00 -7.44606058e-01  8.0000000e+00  1.2000000e+01
  2.0000000e+00  2.0000000e+00  0.0000000e+00  2.0400000e+02
  1.0000000e+00  2.0000000e+01  1.0000000e-03]]

from spotPython.hyperparameters.values import assign_values, return_conf_list_from_var_dict
v_dict = assign_values(X, fun_control["var_name"])
return_conf_list_from_var_dict(var_dict=v_dict, fun_control=fun_control)

[{'loss': 'log_loss',
 'learning_rate': 0.1800503383382619,
 'max_iter': 256,
 'max_leaf_nodes': 4096,
 'max_depth': 4,
 'min_samples_leaf': 4,
 'l2_regularization': 0.0,
 'max_bins': 204,
 'early_stopping': 1,
 'n_iter_no_change': 20,
 'tol': 0.001}]

```

```

from spotPython.hyperparameters.values import get_one_sklearn_model_from_X
model_spot = get_one_sklearn_model_from_X(X, fun_control)
model_spot

HistGradientBoostingClassifier(early_stopping=1,
                               learning_rate=0.1800503383382619, max_bins=204,
                               max_depth=4, max_iter=256, max_leaf_nodes=4096,
                               min_samples_leaf=4, n_iter_no_change=20,
                               tol=0.001)

```

#### 16.10.4 Evaluate SPOT Results

- Fetch the data.

```

from spotPython.utils.convert import get_Xy_from_df
X_train, y_train = get_Xy_from_df(fun_control["train"], fun_control["target_column"])
X_test, y_test = get_Xy_from_df(fun_control["test"], fun_control["target_column"])
X_test.shape, y_test.shape

((63, 64), (63,))

```

- Fit the model with the tuned hyperparameters. This gives one result:

```

model_spot.fit(X_train, y_train)
y_pred = model_spot.predict_proba(X_test)
res = mapk_score(y_true=y_test, y_pred=y_pred, k=3)
res

```

0.7883597883597884

```

def repeated_eval(n, model):
    res_values = []
    for i in range(n):
        model.fit(X_train, y_train)
        y_pred = model.predict_proba(X_test)
        res = mapk_score(y_true=y_test, y_pred=y_pred, k=3)
        res_values.append(res)
    mean_res = np.mean(res_values)
    print(f"mean_res: {mean_res}")

```

```
    std_res = np.std(res_values)
    print(f"std_res: {std_res}")
    min_res = np.min(res_values)
    print(f"min_res: {min_res}")
    max_res = np.max(res_values)
    print(f"max_res: {max_res}")
    median_res = np.median(res_values)
    print(f"median_res: {median_res}")
    return mean_res, std_res, min_res, max_res, median_res
```

### 16.10.5 Handling Non-deterministic Results

- Because the model is non-deterministic, we perform  $n = 30$  runs and calculate the mean and standard deviation of the performance metric.

```
_ = repeated_eval(30, model_spot)
```

```
mean_res: 0.7916225749559083
std_res: 0.012918495037590494
min_res: 0.7671957671957672
max_res: 0.8148148148148149
median_res: 0.791005291005291
```

### 16.10.6 Evaluation of the Default Hyperparameters

```
model_default.fit(X_train, y_train) ["histgradientboostingclassifier"]
```

```
HistGradientBoostingClassifier(early_stopping=1, max_depth=4, max_iter=128,
                               max_leaf_nodes=32, min_samples_leaf=16,
                               tol=0.0001)
```

- One evaluation of the default hyperparameters is performed on the hold-out test set.

```
y_pred = model_default.predict_proba(X_test)
mapk_score(y_true=y_test, y_pred=y_pred, k=3)
```

```
0.783068783068783
```

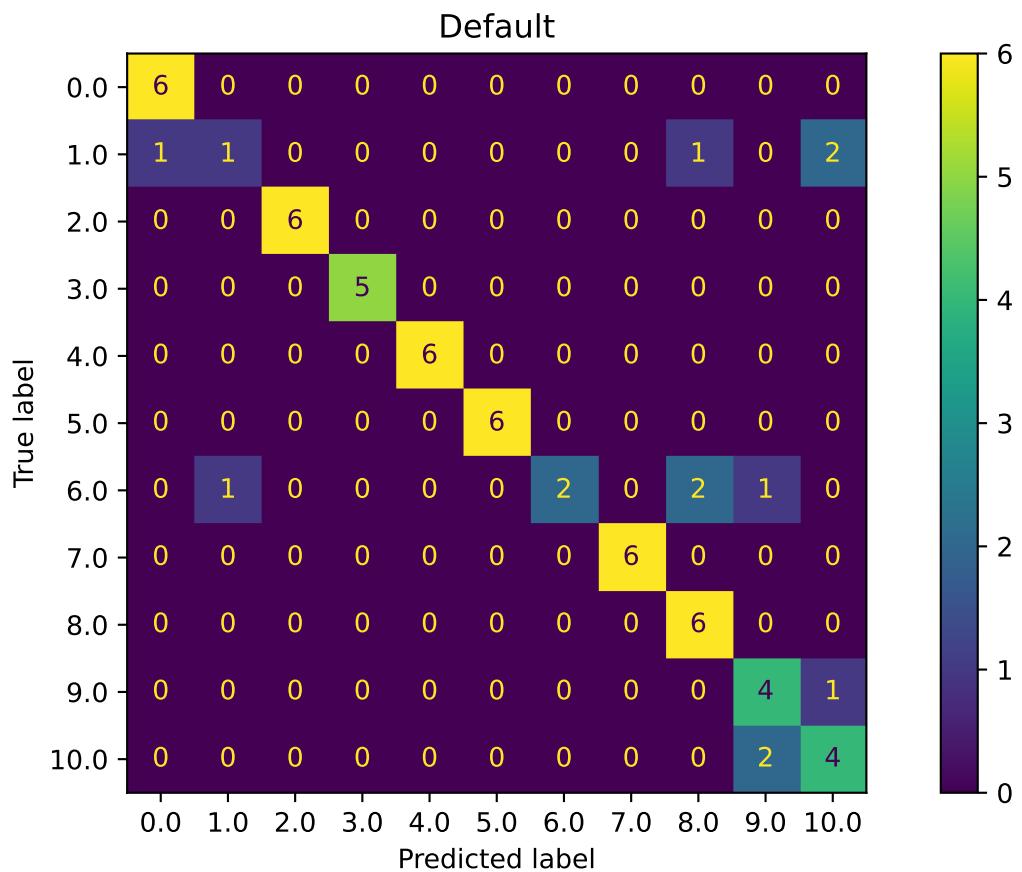
Since one single evaluation is not meaningful, we perform, similar to the evaluation of the SPOT results,  $n = 30$  runs of the default setting and calculate the mean and standard deviation of the performance metric.

```
_ = repeated_eval(30, model_default)

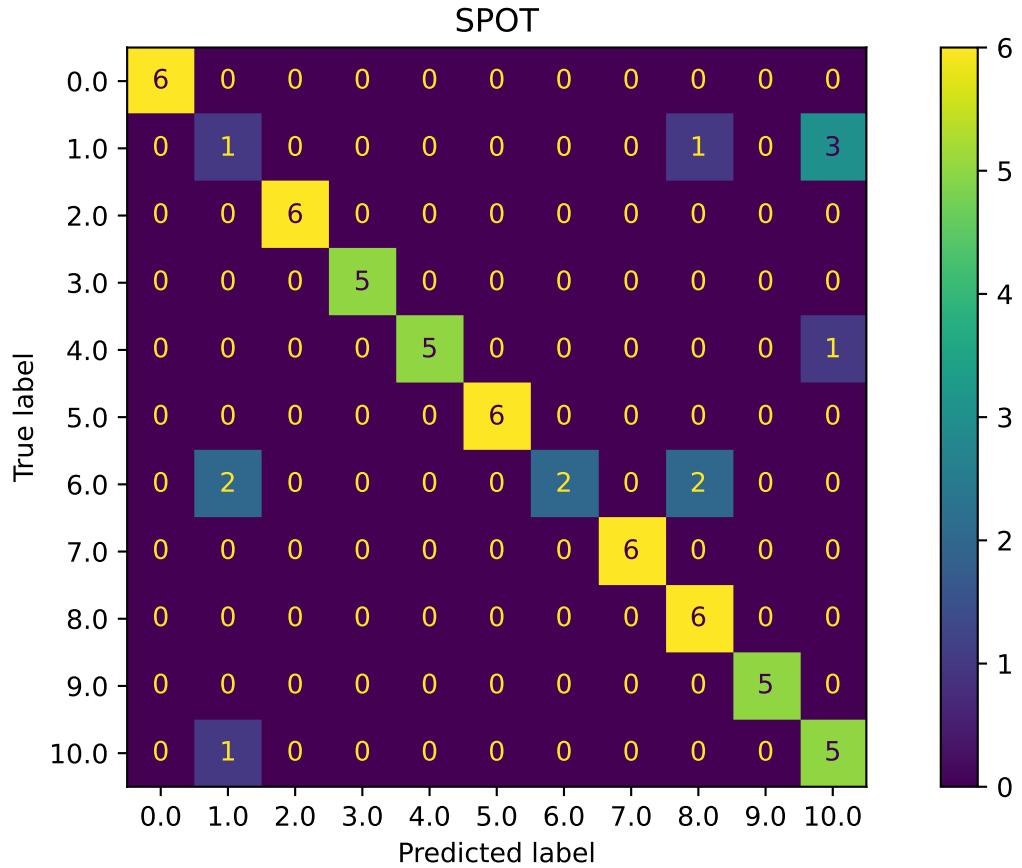
mean_res: 0.7910934744268077
std_res: 0.014188764587985373
min_res: 0.7592592592592592
max_res: 0.8121693121693121
median_res: 0.7936507936507936
```

### 16.10.7 Plot: Compare Predictions

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



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



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

```
(-0.8645833333333334, -0.06944444444444443)
```

### 16.10.8 Cross-validated Evaluations

```
from spotPython.sklearn.traintest import evaluate_cv
fun_control.update({
    "eval": "train_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

```
(0.8142787524366473, None)
```

```
fun_control.update({
    "eval": "test_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

Error in fun\_sklearn(). Call to evaluate\_cv failed. err=ValueError('n\_splits=10 cannot be gr

(nan, None)

- This is the evaluation that will be used in the comparison:

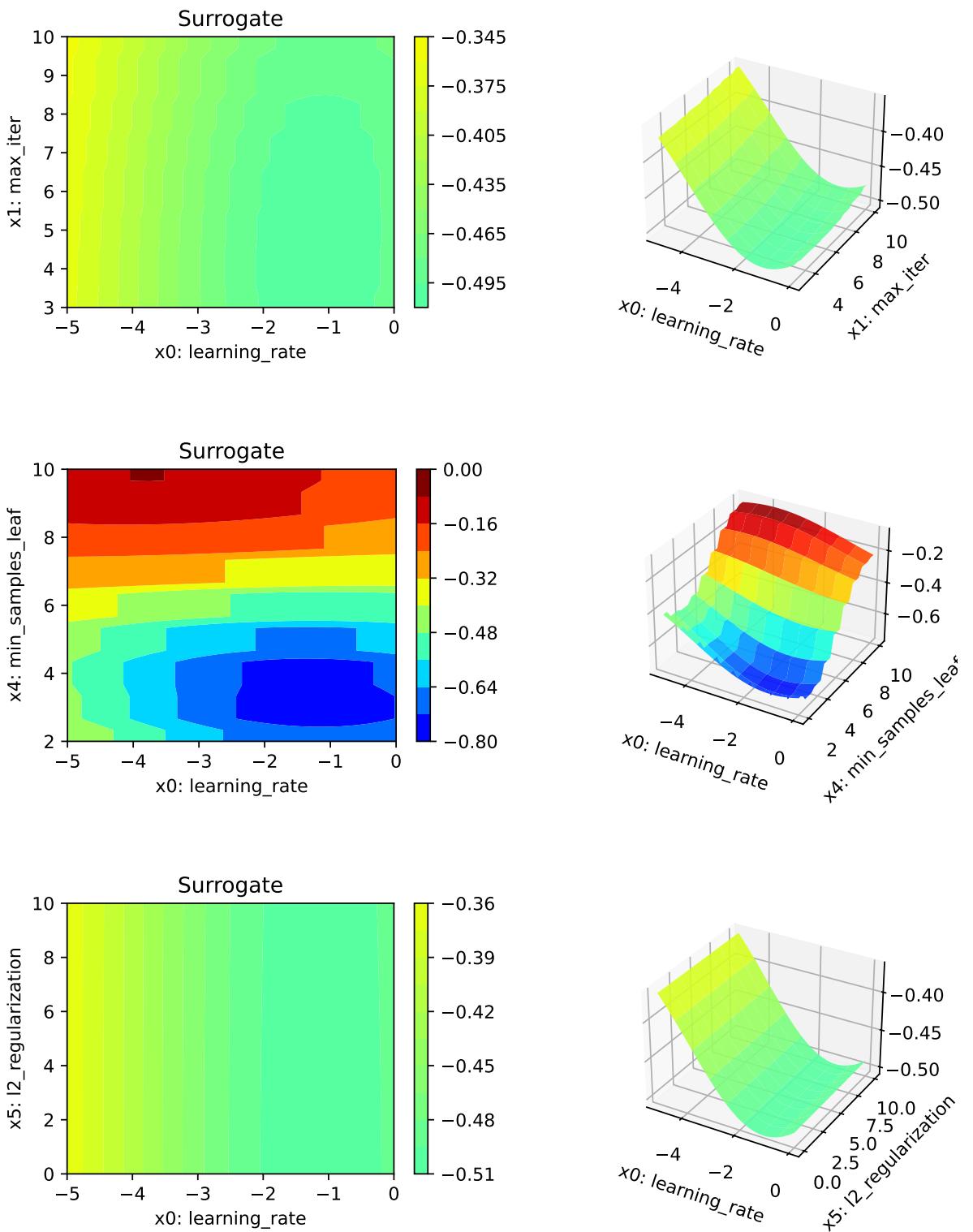
```
fun_control.update({
    "eval": "data_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

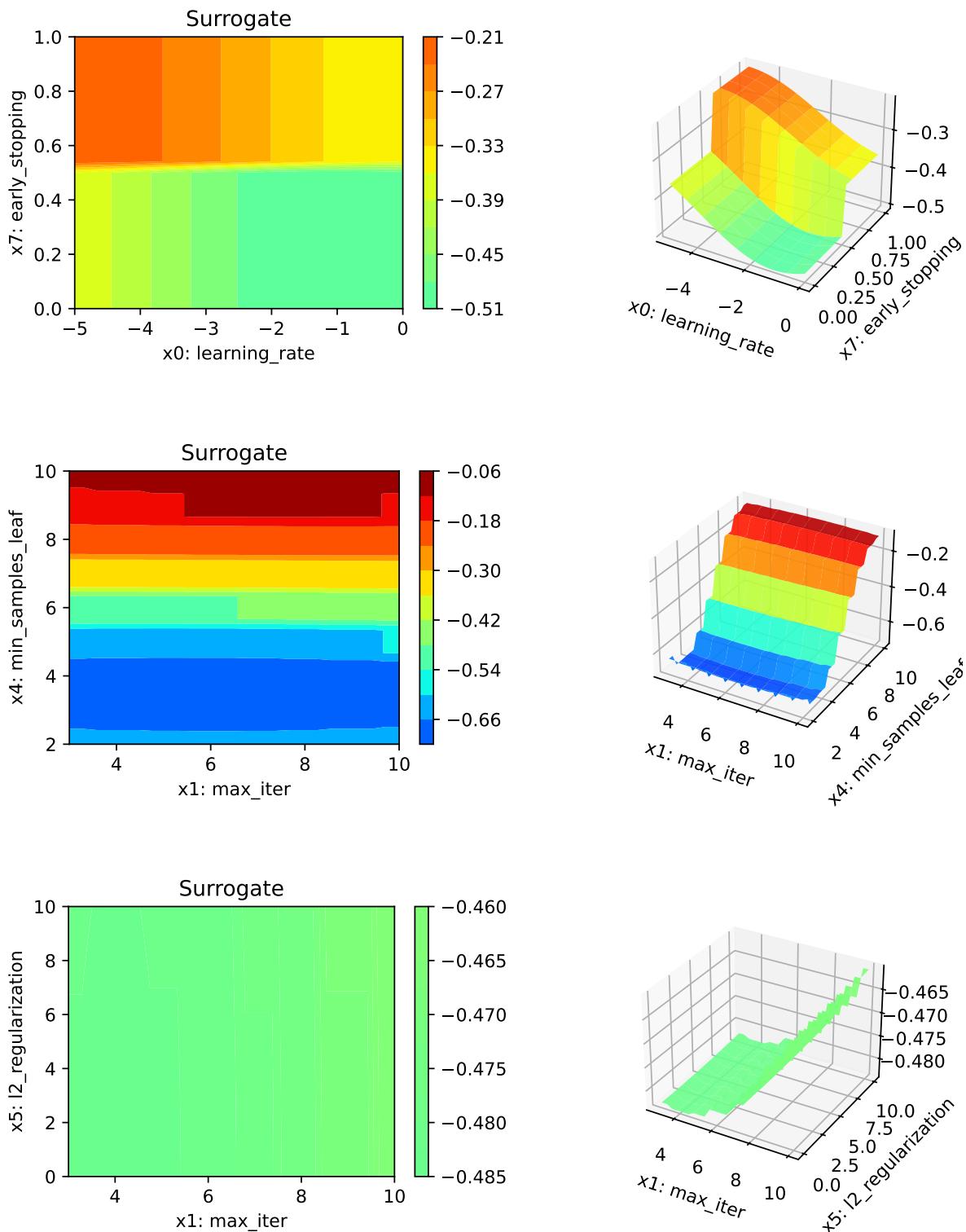
(0.8374615384615384, None)

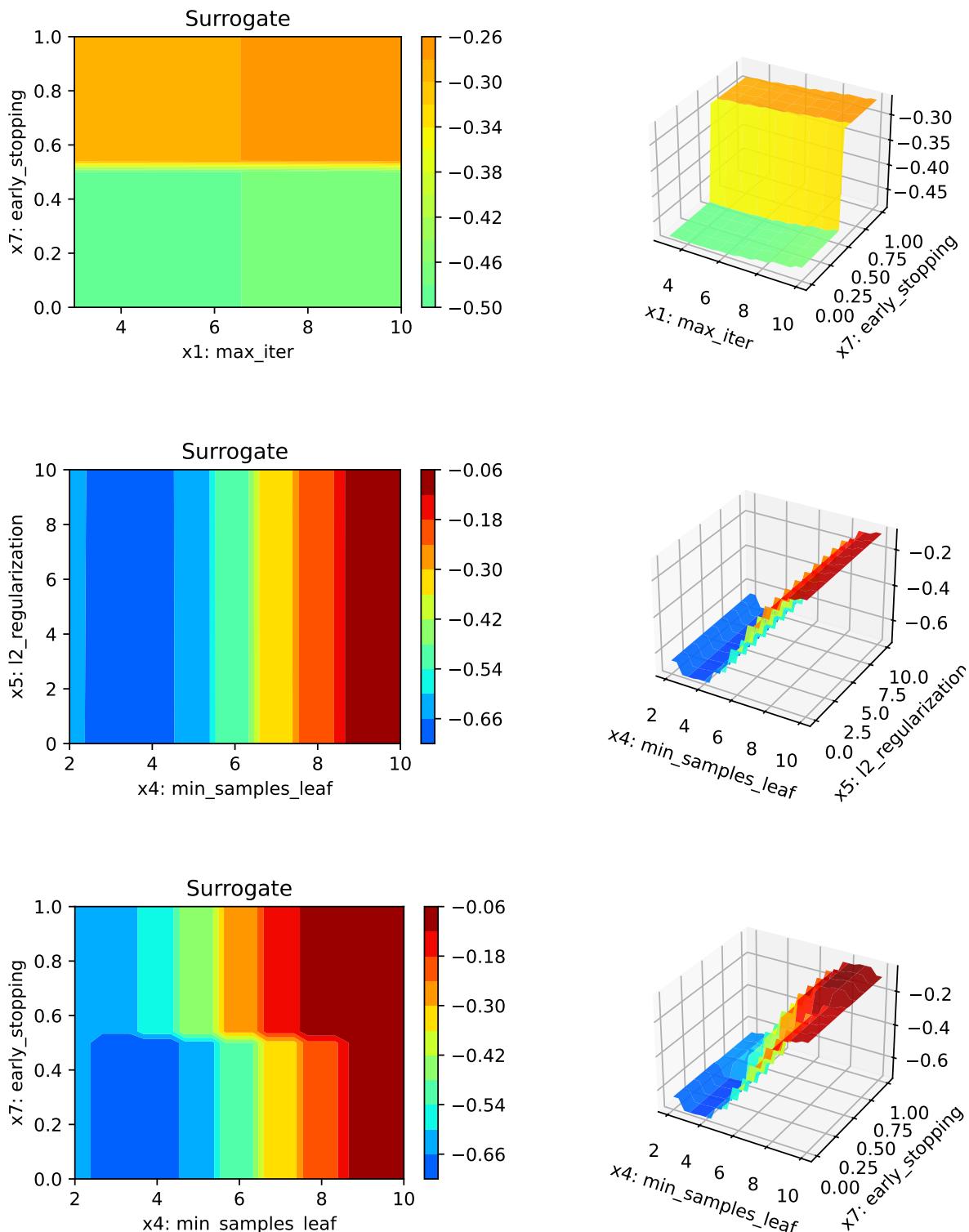
### 16.10.9 Detailed Hyperparameter Plots

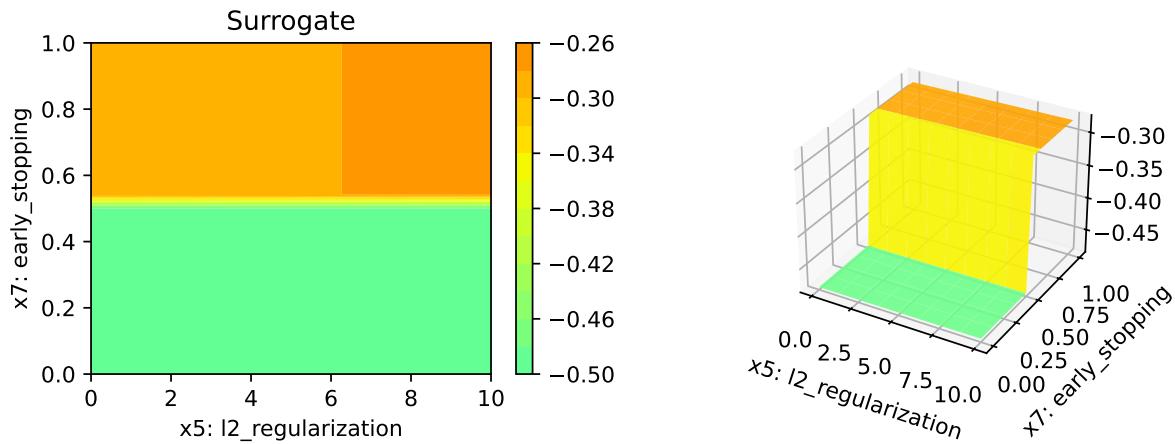
```
filename = "./figures/" + experiment_name
spot_tuner.plot_important_hyperparameter_contour(filename)

learning_rate: 25.038151098863793
max_iter: 2.0061454393150147
min_samples_leaf: 99.99999999999999
l2_regularization: 0.08861356049567441
early_stopping: 5.07201008659328
```









### 16.10.10 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

### 16.10.11 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)
```

# 17 HPT: sklearn SVC VBDP Data

This chapter describes the hyperparameter tuning of a SVC on the Vector Borne Disease Prediction (VBDP) data set.

## ! Vector Borne Disease Prediction Data Set

This chapter uses the Vector Borne Disease Prediction data set from Kaggle. It is a categorical dataset for eleven Vector Borne Diseases with associated symptoms.

The person who associated a work with this deed has dedicated the work to the public domain by waiving all of his or her rights to the work worldwide under copyright law, including all related and neighboring rights, to the extent allowed by law. You can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission. See Other Information below, see <https://creativecommons.org/publicdomain/zero/1.0/>.

The data set is available at: <https://www.kaggle.com/datasets/richardbernat/vector-borne-disease-prediction>,

The data should be downloaded and stored in the `data/VBDP` subfolder. The data set is not available as a part of the `spotPython` package.

## 17.1 Step 1: Setup

Before we consider the detailed experimental setup, we select the parameters that affect run time and the initial design size.

```
MAX_TIME = 1
INIT_SIZE = 5
ORIGINAL = True
PREFIX = "18"

import warnings
warnings.filterwarnings("ignore")
```

## 17.2 Step 2: Initialization of the Empty fun\_control Dictionary

```
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_experiment_name, get_spot_tensorboard_path
from spotPython.utils.device import getDevice

experiment_name = get_experiment_name(prefix=PREFIX)

fun_control = fun_control_init(
    task="classification",
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
```

## 17.3 Step 3: PyTorch Data Loading

### 17.3.1 1. Load Data: Classification VBDP

```
import pandas as pd
if ORIGINAL == True:
    train_df = pd.read_csv('./data/VBDP/trainn.csv')
    test_df = pd.read_csv('./data/VBDP/testt.csv')
else:
    train_df = pd.read_csv('./data/VBDP/train.csv')
    # remove the id column
    train_df = train_df.drop(columns=['id'])

from sklearn.preprocessing import OrdinalEncoder
n_samples = train_df.shape[0]
n_features = train_df.shape[1] - 1
target_column = "prognosis"
# Encoder our prognosis labels as integers for easier decoding later
enc = OrdinalEncoder()
train_df[target_column] = enc.fit_transform(train_df[[target_column]])
train_df.columns = [f"x{i}" for i in range(1, n_features+1)] + [target_column]
print(train_df.shape)
train_df.head()
```

(252, 65)

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	...	x56	x57	x58	x59	x60	x61	x62	x63	x64
0	0	1	1	1	1	0	1	0	0	0	...	0	0	0	0	0	0	0	0	
1	1	1	1	1	1	0	1	1	1	0	...	0	0	0	0	0	0	0	0	
2	0	1	0	1	0	0	1	1	0	0	...	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	1	1	1	0	0	...	0	0	0	0	0	0	0	0	
4	1	0	0	0	1	1	1	1	0	0	...	0	0	0	0	0	0	0	0	

The full data set `train_df` 64 features. The target column is labeled as `prognosis`.

### 17.3.2 Holdout Train and Test Data

We split out a hold-out test set (25% of the data) so we can calculate an example MAP@K

```
import numpy as np
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(train_df.drop(target_column, axis=1),
                                                    random_state=42,
                                                    test_size=0.25,
                                                    stratify=train_df[target_column])
train = pd.DataFrame(np.hstack((X_train, np.array(y_train).reshape(-1, 1))))
test = pd.DataFrame(np.hstack((X_test, np.array(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]
print(train.shape)
print(test.shape)
train.head()
```

(189, 65)  
(63, 65)

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	...	x56	x57	x58	x59	x60	x61	x62	x63
0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	...	0.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0
1	1.0	0.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	...	0.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0
2	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	1.0	1.0	1.0	1.0	1.0	1.0	0.0	0.0	1.0	1.0	...	1.0	0.0	1.0	1.0	1.0	0.0	0.0	1.0
4	1.0	1.0	1.0	0.0	1.0	1.0	0.0	1.0	1.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

```
# add the dataset to the fun_control
fun_control.update({"data": train_df, # full 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:

```
# 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:

```
add_core_model_to_fun_control(SVC, fun_control, SklearnHyperDict)
```

Other core\_models are, e.g.,:

- RidgeCV
- GradientBoostingRegressor
- ElasticNet
- RandomForestClassifier
- LogisticRegression
- KNeighborsClassifier
- RandomForestClassifier
- GradientBoostingClassifier
- HistGradientBoostingClassifier

We will use the `RandomForestClassifier` classifier in this example.

```
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.ensemble import HistGradientBoostingClassifier
from sklearn.linear_model import ElasticNet
from spotPython.hyperparameters.values import add_core_model_to_fun_control
from spotPython.data.sklearn_hyper_dict import SklearnHyperDict
from spotPython.fun.hypersklearn import HyperSklearn

# core_model = RidgeCV
# core_model = GradientBoostingRegressor
# core_model = ElasticNet
# core_model = RandomForestClassifier
core_model = SVC
# core_model = LogisticRegression
# core_model = KNeighborsClassifier
# core_model = GradientBoostingClassifier
# core_model = HistGradientBoostingClassifier
add_core_model_to_fun_control(core_model=core_model,
                             fun_control=fun_control,
                             hyper_dict=SklearnHyperDict,
                             filename=None)
```

Now `fun_control` has the information from the JSON file. The available hyperparameters are:

```
print(*fun_control["core_model_hyper_dict"].keys(), sep="\n")
```

```
C
kernel
degree
gamma
coef0
shrinking
probability
tol
cache_size
break_ties
```

## 17.6 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model`

### 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. For example, to change the `tol` hyperparameter of the SVC model to the interval [1e-3, 1e-2], the following code can be used:

```
modify_hyper_parameter_bounds(fun_control, "tol", bounds=[1e-3, 1e-2])
```

```
from spotPython.hyperparameters.values import modify_hyper_parameter_bounds
modify_hyper_parameter_bounds(fun_control, "probability", bounds=[1, 1])
```

### 17.6.2 Modify hyperparameter of type factor

`spotPython` provides functions for modifying the hyperparameters, their bounds and factors as well as for activating and de-activating hyperparameters without re-compilation of the Python source code. These functions were described in Section 14.6.

Factors can be modified with the `modify_hyper_parameter_levels` function. For example, to exclude the `sigmoid` kernel from the tuning, the `kernel` hyperparameter of the SVC model can be modified as follows:

```
modify_hyper_parameter_levels(fun_control, "kernel", ["linear", "rbf"])
```

The new setting can be controlled via:

```
fun_control["core_model_hyper_dict"]["kernel"]
```

```
from spotPython.hyperparameters.values import modify_hyper_parameter_levels
modify_hyper_parameter_levels(fun_control, "kernel", ["rbf"])
```

### 17.6.3 Optimizers

Optimizers are described in Section 14.6.1.

### 17.6.4 Selection of the Objective: Metric and Loss Functions

- Machine learning models are optimized with respect to a metric, for example, the `accuracy` function.
- Deep learning, e.g., neural networks are optimized with respect to a loss function, for example, the `cross_entropy` function and evaluated with respect to a metric, for example, the `accuracy` function.

## 17.7 Step 7: Selection of the Objective (Loss) Function

The loss function, that is usually used in deep learning for optimizing the weights of the net, is stored in the `fun_control` dictionary as "`loss_function`".

### 17.7.1 Metric Function

There are two different types of metrics in `spotPython`:

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.

We will consider multi-class classification metrics, e.g., `mapk_score` and `top_k_accuracy_score`.

#### Predict Probabilities

In this multi-class classification example the machine learning algorithm should return the probabilities of the specific classes ("`predict_proba`") instead of the predicted values.

We set "`predict_proba`" to `True` in the `fun_control` dictionary.

### 17.7.1.1 The MAPK Metric

To select the MAPK metric, the following two entries can be added to the `fun_control` dictionary:

```
"metric_sklearn": mapk_score  
"metric_params": {"k": 3}.
```

### 17.7.1.2 Other Metrics

Alternatively, other metrics for multi-class classification can be used, e.g., `* top_k_accuracy_score` or `* roc_auc_score`

The metric `roc_auc_score` requires the parameter `"multi_class"`, e.g.,

```
"multi_class": "ovr".
```

This is set in the `fun_control` dictionary.

#### i Weights

`spotPython` performs a minimization, therefore, metrics that should be maximized have to be multiplied by -1. This is done by setting `"weights"` to -1.

- The complete setup for the metric in our example is:

```
from spotPython.utils.metrics import mapk_score  
fun_control.update({  
    "weights": -1,  
    "metric_sklearn": mapk_score,  
    "predict_proba": True,  
    "metric_params": {"k": 3},  
})
```

### 17.7.2 Evaluation on Hold-out Data

- The default method for computing the performance is `"eval_holdout"`.
- Alternatively, cross-validation can be used for every machine learning model.
- Specifically for RandomForests, the OOB-score can be used.

```

fun_control.update({
    "eval": "train_hold_out",
})

```

### 17.7.2.1 Cross Validation

Instead of using the OOB-score, the classical cross validation can be used. The number of folds is set by the key "k\_folds". For example, to use 5-fold cross validation, the key "k\_folds" is set to 5. Uncomment the following line to use cross validation:

```

# fun_control.update({
#     "eval": "train_cv",
#     "k_folds": 10,
# })

```

## 17.8 Step 8: Calling the SPOT Function

### 17.8.1 Preparing the SPOT Call

- Get types and variable names as well as lower and upper bounds for the hyperparameters.

```

# extract the variable types, names, and bounds
from spotPython.hyperparameters.values import (get_bound_values,
                                                get_var_name,
                                                get_var_type)
var_type = get_var_type(fun_control)
var_name = get_var_name(fun_control)
lower = get_bound_values(fun_control, "lower")
upper = get_bound_values(fun_control, "upper")

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

```

name	type	default	lower	upper	transform
C	float	1.0	0.1	10	None
kernel	factor	rbf	0	0	None
degree	int	3	3	3	None

gamma	factor	scale	0	1	None	
coef0	float	0.0	0	0	None	
shrinking	factor	0	0	1	None	
probability	factor	0	1	1	None	
tol	float	0.001	0.0001	0.01	None	
cache_size	float	200.0	100	400	None	
break_ties	factor	0	0	1	None	

## 17.8.2 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
```

## 17.8.3 Run the Spot Optimizer

- Run SPOT for approx. x mins (`max_time`).
- Note: the run takes longer, because the evaluation time of initial design (here: `initi_size`, 20 points) is not considered.

```
from spotPython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
X_start

array([[1.e+00, 0.e+00, 3.e+00, 0.e+00, 0.e+00, 0.e+00, 0.e+00, 1.e-03,
       2.e+02, 0.e+00]])

import numpy as np
from spotPython.spot import spot
from math import inf
spot_tuner = spot.Spot(fun=fun,
                       lower = lower,
                       upper = upper,
                       fun_evals = inf,
                       fun_repeats = 1,
                       max_time = MAX_TIME,
                       noise = False,
```

```
tolerance_x = np.sqrt(np.spacing(1)),
var_type = var_type,
var_name = var_name,
infill_criterion = "y",
n_points = 1,
seed=123,
log_level = 50,
show_models= False,
show_progress= True,
fun_control = fun_control,
design_control={"init_size": INIT_SIZE,
                "repeats": 1},
surrogate_control={"noise": True,
                    "cod_type": "norm",
                    "min_theta": -4,
                    "max_theta": 3,
                    "n_theta": len(var_name),
                    "model_fun_evals": 10_000,
                    "log_level": 50
                })
spot_tuner.run(X_start=X_start)
```

```
spotPython tuning: -0.875 [-----] 0.55%
```

```
spotPython tuning: -0.875 [-----] 1.09%
```

```
spotPython tuning: -0.875 [-----] 1.53%
```

```
spotPython tuning: -0.875 [-----] 1.92%
```

```
spotPython tuning: -0.875 [-----] 2.81%
```

```
spotPython tuning: -0.875 [-----] 3.34%
```

```
spotPython tuning: -0.875 [-----] 4.22%
```

```
spotPython tuning: -0.875 [#-----] 5.15%
```

```
spotPython tuning: -0.875 [#-----] 6.17%
```

spotPython tuning: -0.875 [#-----] 7.84%

spotPython tuning: -0.875 [#-----] 9.61%

spotPython tuning: -0.875 [#-----] 10.37%

spotPython tuning: -0.875 [#-----] 12.02%

spotPython tuning: -0.875 [#-----] 13.93%

spotPython tuning: -0.875 [##-----] 17.28%

spotPython tuning: -0.875 [##-----] 20.89%

spotPython tuning: -0.875 [##-----] 23.04%

spotPython tuning: -0.875 [###-----] 25.02%

spotPython tuning: -0.875 [###-----] 26.52%

spotPython tuning: -0.875 [###-----] 27.89%

spotPython tuning: -0.875 [###-----] 31.36%

spotPython tuning: -0.875 [###-----] 34.95%

spotPython tuning: -0.875 [####-----] 38.78%

spotPython tuning: -0.875 [####-----] 42.42%

spotPython tuning: -0.875 [#####-----] 46.86%

spotPython tuning: -0.875 [#####-----] 50.07%

spotPython tuning: -0.875 [#####-----] 52.21%

spotPython tuning: -0.875 [#####-----] 54.17%

```
spotPython tuning: -0.875 [#####----] 59.93%
spotPython tuning: -0.875 [#####---] 67.34%
spotPython tuning: -0.875 [#####---] 74.47%
spotPython tuning: -0.875 [#####---] 81.98%
spotPython tuning: -0.875 [#####---] 87.58%
spotPython tuning: -0.875 [#####---] 90.95%
spotPython tuning: -0.875 [#####---] 93.53%
spotPython tuning: -0.875 [#####---] 96.24%
spotPython tuning: -0.875 [#####---] 100.00% Done...
<spotPython.spot.spot.Spot at 0x2e746a260>
```

## 17.9 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard as described in Section 14.9, see also the description in the documentation: [Tensorboard](#).

## 17.10 Step 10: Results

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized. The following code generates the progress plot from `?@fig-progress`.

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

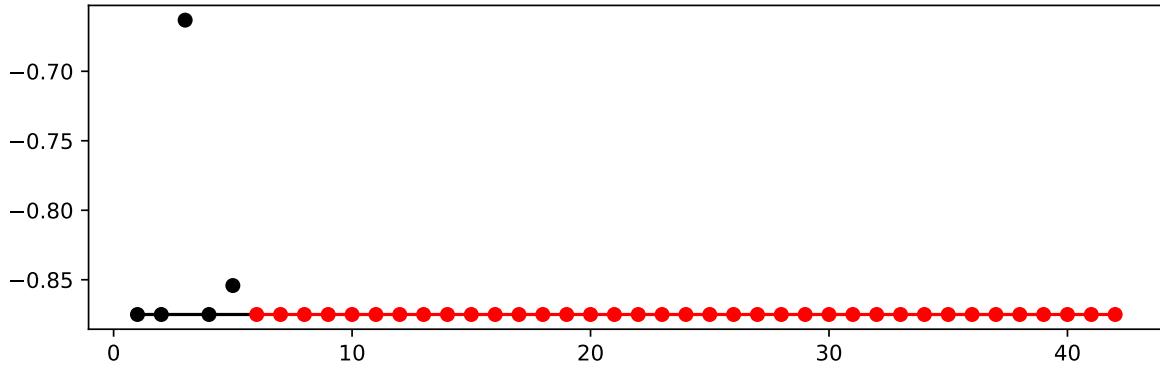


Figure 17.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

- Print the results

```
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	8.648943310768674	None
kernel	factor	rbf	0.0	0.0	0.0	None
degree	int	3	3.0	3.0	3.0	None
gamma	factor	scale	0.0	1.0	1.0	None
coef0	float	0.0	0.0	0.0	0.0	None
shrinking	factor	0	0.0	1.0	0.0	None
probability	factor	0	1.0	1.0	1.0	None
tol	float	0.001	0.0001	0.01	0.0036949438148166343	None
cache_size	float	200.0	100.0	400.0	389.44564593489815	None
break_ties	factor	0	0.0	1.0	0.0	None

### 17.10.1 Show variable importance

```
spot_tuner.plot_importance(threshold=0.025, filename=".//figures/" + experiment_name+"_imp
```

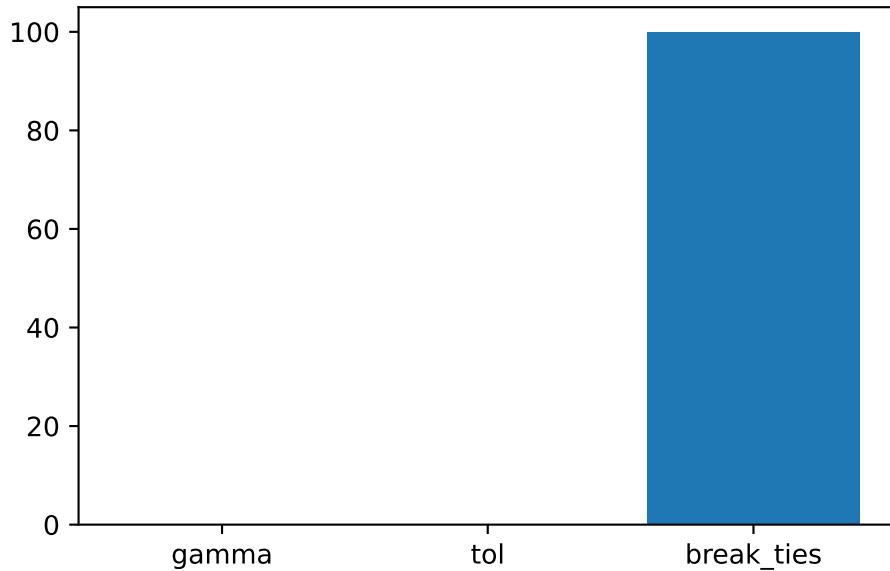


Figure 17.2: Variable importance plot, threshold 0.025.

### 17.10.2 Get Default Hyperparameters

```

from spotPython.hyperparameters.values import get_default_values, transform_hyper_parameters
values_default = get_default_values(fun_control)
values_default = transform_hyper_parameter_values(fun_control=fun_control, hyper_parameter_values_default

{'C': 1.0,
'kernel': 'rbf',
'degree': 3,
'gamma': 'scale',
'coef0': 0.0,
'shrinking': 0,
'probability': 0,
'tol': 0.001,
'cache_size': 200.0,
'break_ties': 0}

from sklearn.pipeline import make_pipeline
model_default = make_pipeline(fun_control["prep_model"], fun_control["core_model"](**values_default)
model_default

```

```
Pipeline(steps=[('nonetype', None),
               ('svc',
                SVC(break_ties=0, cache_size=200.0, probability=0,
                     shrinking=0))])
```

**i Note**

- Default value for “probability” is False, but we need it to be True for the metric “mapk\_score”.

```
values_default.update({"probability": 1})
```

### 17.10.3 Get SPOT Results

```
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
print(X)
```

```
[[8.64894331e+00 0.0000000e+00 3.0000000e+00 1.0000000e+00
 0.0000000e+00 0.0000000e+00 1.0000000e+00 3.69494381e-03
 3.89445646e+02 0.0000000e+00]]
```

```
from spotPython.hyperparameters.values import assign_values, return_conf_list_from_var_dict
v_dict = assign_values(X, fun_control["var_name"])
return_conf_list_from_var_dict(var_dict=v_dict, fun_control=fun_control)
```

```
[{'C': 8.648943310768674,
 'kernel': 'rbf',
 'degree': 3,
 'gamma': 'auto',
 'coef0': 0.0,
 'shrinking': 0,
 'probability': 1,
 'tol': 0.0036949438148166343,
 'cache_size': 389.44564593489815,
 'break_ties': 0}]
```

```

from spotPython.hyperparameters.values import get_one_sklearn_model_from_X
model_spot = get_one_sklearn_model_from_X(X, fun_control)
model_spot

SVC(C=8.648943310768674, break_ties=0, cache_size=389.44564593489815,
     gamma='auto', probability=1, shrinking=0, tol=0.0036949438148166343)

```

#### 17.10.4 Evaluate SPOT Results

- Fetch the data.

```

from spotPython.utils.convert import get_Xy_from_df
X_train, y_train = get_Xy_from_df(fun_control["train"], fun_control["target_column"])
X_test, y_test = get_Xy_from_df(fun_control["test"], fun_control["target_column"])
X_test.shape, y_test.shape

((63, 64), (63,))

```

- Fit the model with the tuned hyperparameters. This gives one result:

```

model_spot.fit(X_train, y_train)
y_pred = model_spot.predict_proba(X_test)
res = mapk_score(y_true=y_test, y_pred=y_pred, k=3)
res

```

0.8465608465608466

```

def repeated_eval(n, model):
    res_values = []
    for i in range(n):
        model.fit(X_train, y_train)
        y_pred = model.predict_proba(X_test)
        res = mapk_score(y_true=y_test, y_pred=y_pred, k=3)
        res_values.append(res)
    mean_res = np.mean(res_values)
    print(f"mean_res: {mean_res}")
    std_res = np.std(res_values)
    print(f"std_res: {std_res}")
    min_res = np.min(res_values)

```

```

print(f"min_res: {min_res}")
max_res = np.max(res_values)
print(f"max_res: {max_res}")
median_res = np.median(res_values)
print(f"median_res: {median_res}")
return mean_res, std_res, min_res, max_res, median_res

```

### 17.10.5 Handling Non-deterministic Results

- Because the model is non-deterministic, we perform  $n = 30$  runs and calculate the mean and standard deviation of the performance metric.

```
_ = repeated_eval(30, model_spot)
```

```

mean_res: 0.8485890652557321
std_res: 0.0039050422972635584
min_res: 0.8386243386243387
max_res: 0.8571428571428571
median_res: 0.8465608465608466

```

### 17.10.6 Evaluation of the Default Hyperparameters

```

model_default["svc"].probability = True
model_default.fit(X_train, y_train)["svc"]

```

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

- One evaluation of the default hyperparameters is performed on the hold-out test set.

```

y_pred = model_default.predict_proba(X_test)
mapk_score(y_true=y_test, y_pred=y_pred, k=3)

```

```
0.8571428571428571
```

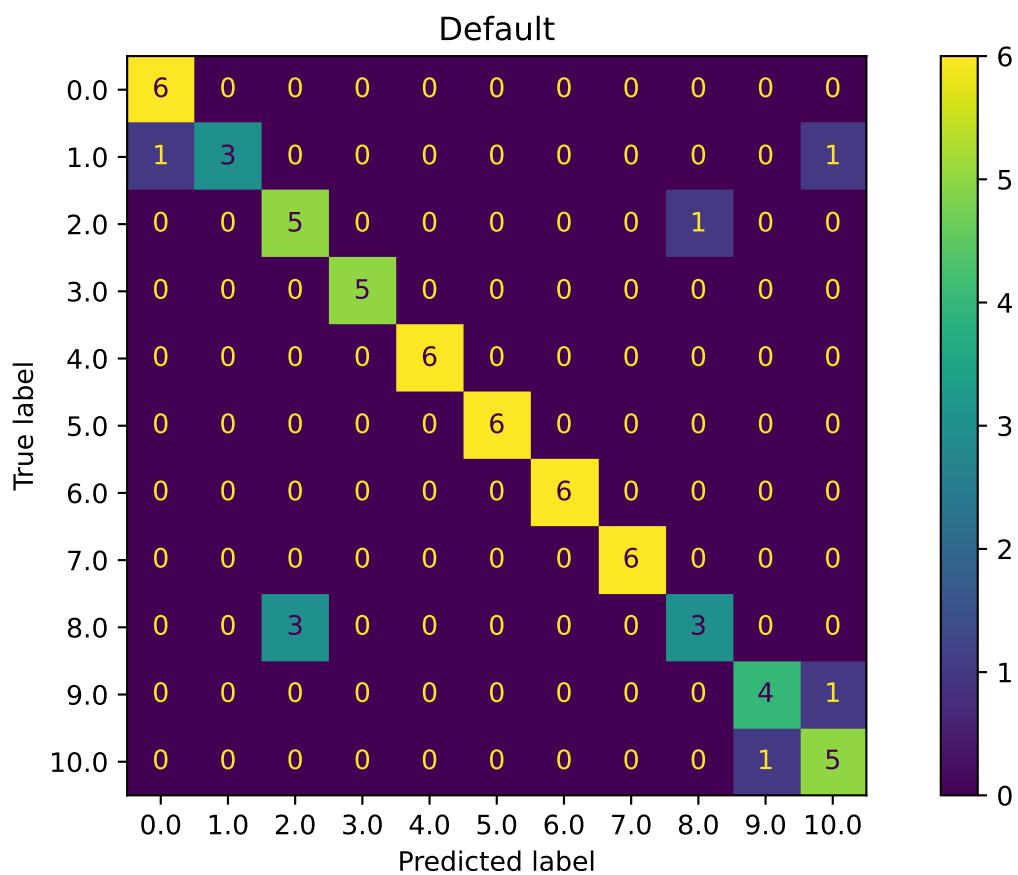
Since one single evaluation is not meaningful, we perform, similar to the evaluation of the SPOT results,  $n = 30$  runs of the default setting and calculate the mean and standard deviation of the performance metric.

```
_ = repeated_eval(30, model_default)
```

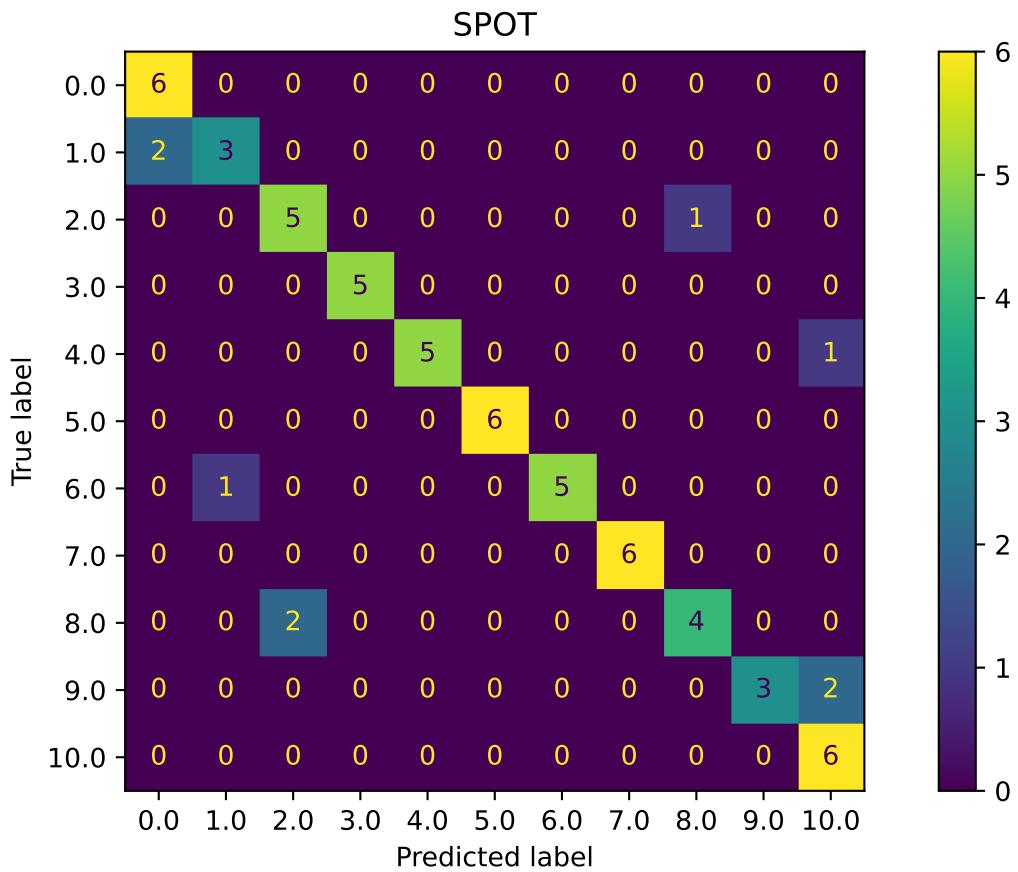
```
mean_res: 0.854320987654321
std_res: 0.004207005446870034
min_res: 0.8492063492063492
max_res: 0.8650793650793651
median_res: 0.8571428571428571
```

### 17.10.7 Plot: Compare Predictions

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



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



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

```
(-0.875, -0.6631944444444444)
```

### 17.10.8 Cross-validated Evaluations

```
from spotPython.sklearn.traintest import evaluate_cv
fun_control.update({
    "eval": "train_cv",
    "k_folds": 10,
})
```

```
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

```
(0.8513645224171539, None)
```

```
fun_control.update({
    "eval": "test_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

```
Error in fun_sklearn(). Call to evaluate_cv failed. err=ValueError('n_splits=10 cannot be gr
```

```
(nan, None)
```

- This is the evaluation that will be used in the comparison:

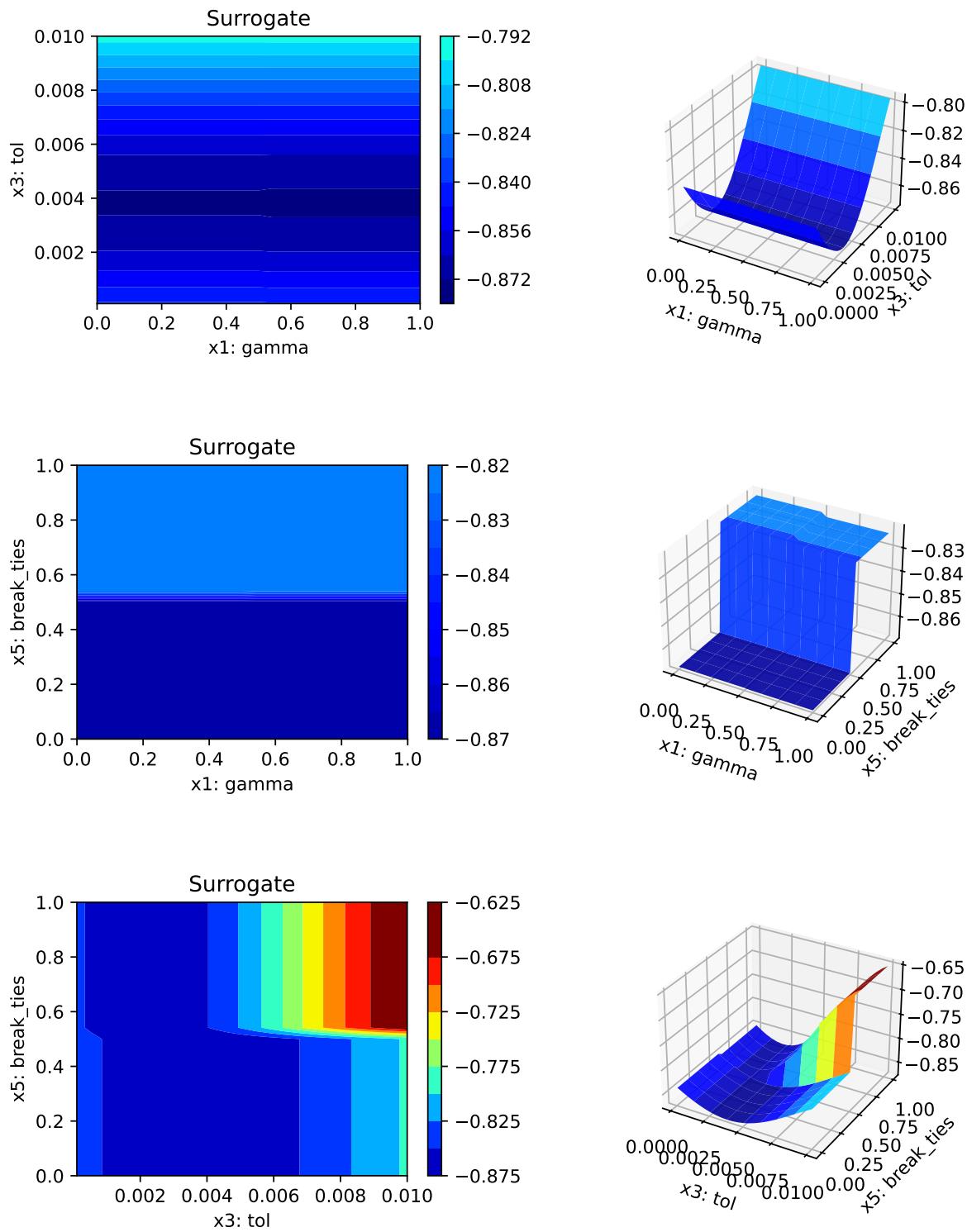
```
fun_control.update({
    "eval": "data_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

```
(0.8807179487179487, None)
```

### 17.10.9 Detailed Hyperparameter Plots

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

```
gamma: 0.047342720750464695
tol: 0.1821713271282166
break_ties: 100.0
```



### 17.10.10 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

### 17.10.11 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)
```

# 18 HPT: sklearn KNN Classifier VBDP Data

This chapter describes the hyperparameter tuning of a `KNeighborsClassifier` on the Vector Borne Disease Prediction (VBDP) data set.

## ! Vector Borne Disease Prediction Data Set

This chapter uses the Vector Borne Disease Prediction data set from Kaggle. It is a categorical dataset for eleven Vector Borne Diseases with associated symptoms.

The person who associated a work with this deed has dedicated the work to the public domain by waiving all of his or her rights to the work worldwide under copyright law, including all related and neighboring rights, to the extent allowed by law. You can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission. See Other Information below, see <https://creativecommons.org/publicdomain/zero/1.0/>.

The data set is available at: <https://www.kaggle.com/datasets/richardbernat/vector-borne-disease-prediction>,

The data should be downloaded and stored in the `data/VBDP` subfolder. The data set is not available as a part of the `spotPython` package.

## 18.1 Step 1: Setup

Before we consider the detailed experimental setup, we select the parameters that affect run time and the initial design size.

```
MAX_TIME = 1
INIT_SIZE = 5
ORIGINAL = True
PREFIX = "19"

import warnings
warnings.filterwarnings("ignore")
```

## 18.2 Step 2: Initialization of the Empty fun\_control Dictionary

```
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_experiment_name, get_spot_tensorboard_path
from spotPython.utils.device import getDevice

experiment_name = get_experiment_name(prefix=PREFIX)

fun_control = fun_control_init(
    task="classification",
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name))
```

### 18.2.1 Load Data: Classification VBDP

```
import pandas as pd
if ORIGINAL == True:
    train_df = pd.read_csv('./data/VBDP/trainn.csv')
    test_df = pd.read_csv('./data/VBDP/testt.csv')
else:
    train_df = pd.read_csv('./data/VBDP/train.csv')
    # remove the id column
    train_df = train_df.drop(columns=['id'])

from sklearn.preprocessing import OrdinalEncoder
n_samples = train_df.shape[0]
n_features = train_df.shape[1] - 1
target_column = "prognosis"
# Encoder our prognosis labels as integers for easier decoding later
enc = OrdinalEncoder()
train_df[target_column] = enc.fit_transform(train_df[[target_column]])
train_df.columns = [f"x{i}" for i in range(1, n_features+1)] + [target_column]
print(train_df.shape)
train_df.head()
```

(252, 65)

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	...	x56	x57	x58	x59	x60	x61	x62	x63	x64
0	0	1	1	1	1	0	1	0	0	0	...	0	0	0	0	0	0	0	0	
1	1	1	1	1	1	0	1	1	1	0	...	0	0	0	0	0	0	0	0	
2	0	1	0	1	0	0	1	1	0	0	...	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	1	1	1	0	0	...	0	0	0	0	0	0	0	0	
4	1	0	0	0	1	1	1	1	0	0	...	0	0	0	0	0	0	0	0	

The full data set `train_df` 64 features. The target column is labeled as `prognosis`.

### 18.2.2 Holdout Train and Test Data

We split out a hold-out test set (25% of the data) so we can calculate an example MAP@K

```
import numpy as np
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(train_df.drop(target_column, axis=1),
                                                    random_state=42,
                                                    test_size=0.25,
                                                    stratify=train_df[target_column])
train = pd.DataFrame(np.hstack((X_train, np.array(y_train).reshape(-1, 1))))
test = pd.DataFrame(np.hstack((X_test, np.array(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]
print(train.shape)
print(test.shape)
train.head()
```

(189, 65)  
(63, 65)

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	...	x56	x57	x58	x59	x60	x61	x62	x63
0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	...	0.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0
1	1.0	0.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	...	0.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0
2	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	1.0	1.0	1.0	1.0	1.0	1.0	0.0	0.0	1.0	1.0	...	1.0	0.0	1.0	1.0	1.0	0.0	0.0	1.0
4	1.0	1.0	1.0	0.0	1.0	1.0	0.0	1.0	1.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

```

# add the dataset to the fun_control
fun_control.update({"data": train_df, # full dataset,
                    "train": train,
                    "test": test,
                    "n_samples": n_samples,
                    "target_column": target_column})

```

## 18.3 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:

```

# 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(),
# )

```

## 18.4 Step 5: Select Model (algorithm) and core\_model\_hyper\_dict

The selection of the algorithm (ML model) that should be tuned is done by specifying the its name from the `sklearn` implementation. For example, the `SVC` support vector machine classifier is selected as follows:

```
add_core_model_to_fun_control(SVC, fun_control, SklearnHyperDict)
```

Other core\_models are, e.g.,:

- RidgeCV
- GradientBoostingRegressor
- ElasticNet
- RandomForestClassifier
- LogisticRegression
- KNeighborsClassifier
- RandomForestClassifier
- GradientBoostingClassifier
- HistGradientBoostingClassifier

We will use the `RandomForestClassifier` classifier in this example.

```
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.ensemble import HistGradientBoostingClassifier
from sklearn.linear_model import ElasticNet
from spotPython.hyperparameters.values import add_core_model_to_fun_control
from spotPython.data.sklearn_hyper_dict import SklearnHyperDict
from spotPython.fun.hypersklearn import HyperSklearn

# core_model = RidgeCV
# core_model = GradientBoostingRegressor
# core_model = ElasticNet
# core_model = RandomForestClassifier
core_model = KNeighborsClassifier
# core_model = LogisticRegression
# core_model = KNeighborsClassifier
# core_model = GradientBoostingClassifier
# core_model = HistGradientBoostingClassifier
add_core_model_to_fun_control(core_model=core_model,
                             fun_control=fun_control,
                             hyper_dict=SklearnHyperDict,
                             filename=None)
```

Now `fun_control` has the information from the JSON file. The available hyperparameters are:

```
print(*fun_control["core_model_hyper_dict"].keys(), sep="\n")  
  
n_neighbors  
weights  
algorithm  
leaf_size  
p
```

## 18.5 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model`

### 18.5.1 Modify hyperparameter of type numeric and integer (boolean)

Numeric and boolean values can be modified using the `modify_hyper_parameter_bounds` method. For example, to change the `tol` hyperparameter of the SVC model to the interval [1e-3, 1e-2], the following code can be used:

```
modify_hyper_parameter_bounds(fun_control, "tol", bounds=[1e-3, 1e-2])  
  
# from spotPython.hyperparameters.values import modify_hyper_parameter_bounds  
# modify_hyper_parameter_bounds(fun_control, "probability", bounds=[1, 1])
```

### 18.5.2 Modify hyperparameter of type factor

`spotPython` provides functions for modifying the hyperparameters, their bounds and factors as well as for activating and de-activating hyperparameters without re-compilation of the Python source code. These functions were described in Section 14.6.

Factors can be modified with the `modify_hyper_parameter_levels` function. For example, to exclude the `sigmoid` kernel from the tuning, the `kernel` hyperparameter of the SVC model can be modified as follows:

```
modify_hyper_parameter_levels(fun_control, "kernel", ["linear", "rbf"])
```

The new setting can be controlled via:

```
fun_control["core_model_hyper_dict"]["kernel"]
```

```
# from spotPython.hyperparameters.values import modify_hyper_parameter_levels
# modify_hyper_parameter_levels(fun_control, "kernel", ["rbf"])
```

### 18.5.3 Optimizers

Optimizers are described in Section 14.6.1.

### 18.5.4 Selection of the Objective: Metric and Loss Functions

- Machine learning models are optimized with respect to a metric, for example, the `accuracy` function.
- Deep learning, e.g., neural networks are optimized with respect to a loss function, for example, the `cross_entropy` function and evaluated with respect to a metric, for example, the `accuracy` function.

## 18.6 Step 7: Selection of the Objective (Loss) Function

The loss function, that is usually used in deep learning for optimizing the weights of the net, is stored in the `fun_control` dictionary as "`loss_function`".

### 18.6.1 Metric Function

There are two different types of metrics in `spotPython`:

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.

We will consider multi-class classification metrics, e.g., `mapk_score` and `top_k_accuracy_score`.

#### Predict Probabilities

In this multi-class classification example the machine learning algorithm should return the probabilities of the specific classes ("`predict_proba`") instead of the predicted values.

We set "`predict_proba`" to `True` in the `fun_control` dictionary.

### 18.6.1.1 The MAPK Metric

To select the MAPK metric, the following two entries can be added to the `fun_control` dictionary:

```
"metric_sklearn": mapk_score  
"metric_params": {"k": 3}.
```

### 18.6.1.2 Other Metrics

Alternatively, other metrics for multi-class classification can be used, e.g., `* top_k_accuracy_score` or `* roc_auc_score`

The metric `roc_auc_score` requires the parameter `"multi_class"`, e.g.,

```
"multi_class": "ovr".
```

This is set in the `fun_control` dictionary.

#### i Weights

`spotPython` performs a minimization, therefore, metrics that should be maximized have to be multiplied by -1. This is done by setting `"weights"` to -1.

- The complete setup for the metric in our example is:

```
from spotPython.utils.metrics import mapk_score  
fun_control.update({  
    "weights": -1,  
    "metric_sklearn": mapk_score,  
    "predict_proba": True,  
    "metric_params": {"k": 3},  
})
```

### 18.6.2 Evaluation on Hold-out Data

- The default method for computing the performance is `"eval_holdout"`.
- Alternatively, cross-validation can be used for every machine learning model.
- Specifically for RandomForests, the OOB-score can be used.

```

fun_control.update({
    "eval": "train_hold_out",
})

```

### 18.6.2.1 Cross Validation

Instead of using the OOB-score, the classical cross validation can be used. The number of folds is set by the key "k\_folds". For example, to use 5-fold cross validation, the key "k\_folds" is set to 5. Uncomment the following line to use cross validation:

```

# fun_control.update({
#     "eval": "train_cv",
#     "k_folds": 10,
# })

```

## 18.7 Step 8: Calling the SPOT Function

### 18.7.1 Preparing the SPOT Call

- Get types and variable names as well as lower and upper bounds for the hyperparameters.

```

# extract the variable types, names, and bounds
from spotPython.hyperparameters.values import (get_bound_values,
                                                get_var_name,
                                                get_var_type)
var_type = get_var_type(fun_control)
var_name = get_var_name(fun_control)
lower = get_bound_values(fun_control, "lower")
upper = get_bound_values(fun_control, "upper")

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

```

name	type	default	lower	upper	transform
n_neighbors	int	2	1	7	transform_power_2_int
weights	factor	uniform	0	1	None
algorithm	factor	auto	0	3	None

leaf_size   int   5	2   7   transform_power_2_int
p   int   2	1   2   None

### 18.7.2 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
```

### 18.7.3 Run the Spot Optimizer

- Run SPOT for approx. x mins (`max_time`).
- Note: the run takes longer, because the evaluation time of initial design (here: `initi_size`, 20 points) is not considered.

```
from spotPython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
X_start

array([[2, 0, 0, 5, 2]])

import numpy as np
from spotPython.spot import spot
from math import inf
spot_tuner = spot.Spot(fun=fun,
                       lower = lower,
                       upper = upper,
                       fun_evals = inf,
                       fun_repeats = 1,
                       max_time = MAX_TIME,
                       noise = False,
                       tolerance_x = np.sqrt(np.spacing(1)),
                       var_type = var_type,
                       var_name = var_name,
                       infill_criterion = "y",
                       n_points = 1,
                       seed=123,
```

```
    log_level = 50,
    show_models= False,
    show_progress= True,
    fun_control = fun_control,
    design_control={"init_size": INIT_SIZE,
                    "repeats": 1},
    surrogate_control={"noise": True,
                       "cod_type": "norm",
                       "min_theta": -4,
                       "max_theta": 3,
                       "n_theta": len(var_name),
                       "model_fun_evals": 10_000,
                       "log_level": 50
                     })
spot_tuner.run(X_start=X_start)
```

spotPython tuning: -0.71875 [-----] 0.77%

spotPython tuning: -0.71875 [-----] 1.56%

spotPython tuning: -0.7326388888888888 [-----] 2.39%

spotPython tuning: -0.7326388888888888 [-----] 2.82%

spotPython tuning: -0.7326388888888888 [-----] 3.30%

spotPython tuning: -0.7326388888888888 [-----] 3.85%

spotPython tuning: -0.7326388888888888 [-----] 4.40%

spotPython tuning: -0.7326388888888888 [#-----] 5.16%

spotPython tuning: -0.7326388888888888 [#-----] 5.67%

spotPython tuning: -0.7326388888888888 [#-----] 6.20%

spotPython tuning: -0.7326388888888888 [#-----] 6.74%

spotPython tuning: -0.7326388888888888 [#-----] 7.96%

spotPython tuning: -0.7326388888888888 [#-----] 9.20%

spotPython tuning: -0.7326388888888888 [#-----] 10.53%

spotPython tuning: -0.7326388888888888 [#-----] 11.72%

spotPython tuning: -0.7326388888888888 [#-----] 13.35%

spotPython tuning: -0.7326388888888888 [#-----] 14.90%

spotPython tuning: -0.7326388888888888 [##-----] 15.74%

spotPython tuning: -0.7326388888888888 [##-----] 16.67%

spotPython tuning: -0.7465277777777777 [##-----] 18.83%

spotPython tuning: -0.7465277777777777 [##-----] 21.31%

spotPython tuning: -0.7465277777777777 [##-----] 23.52%

spotPython tuning: -0.7465277777777777 [###-----] 26.22%

spotPython tuning: -0.7465277777777777 [###-----] 29.07%

spotPython tuning: -0.7465277777777777 [###-----] 31.38%

spotPython tuning: -0.7465277777777777 [###-----] 32.59%

spotPython tuning: -0.7465277777777777 [###-----] 33.56%

spotPython tuning: -0.7465277777777777 [####-----] 37.69%

spotPython tuning: -0.7465277777777777 [####-----] 41.46%

spotPython tuning: -0.7465277777777777 [#####-----] 45.30%

```
spotPython tuning: -0.7465277777777777 [#####----] 49.91%
spotPython tuning: -0.7465277777777777 [#####----] 53.72%
spotPython tuning: -0.7465277777777777 [#####----] 56.97%
spotPython tuning: -0.7465277777777777 [#####----] 58.61%
spotPython tuning: -0.7465277777777777 [#####----] 60.54%
spotPython tuning: -0.7465277777777777 [#####----] 71.12%
spotPython tuning: -0.7465277777777777 [#####----] 88.99%
spotPython tuning: -0.7465277777777777 [#####----] 100.00% Done...
<spotPython.spot.spot.Spot at 0x2a99d36d0>
```

## 18.8 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard as described in Section 14.9, see also the description in the documentation: [Tensorboard](#).

## 18.9 Step 10: Results

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized. The following code generates the progress plot from `?@fig-progress`.

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

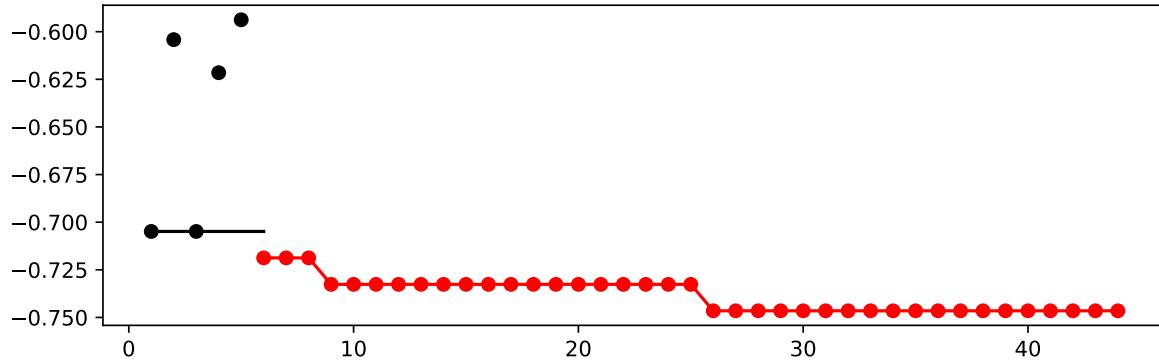


Figure 18.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

- Print the results

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

name	type	default	lower	upper	tuned	transform
n_neighbors	int	2	1	7	3.0	transform_power_2_int
weights	factor	uniform	0	1	0.0	None
algorithm	factor	auto	0	3	1.0	None
leaf_size	int	5	2	7	4.0	transform_power_2_int
p	int	2	1	2	2.0	None

### 18.9.1 Show variable importance

```
spot_tuner.plot_importance(threshold=0.025, filename=".//figures/" + experiment_name+"_importance")
```

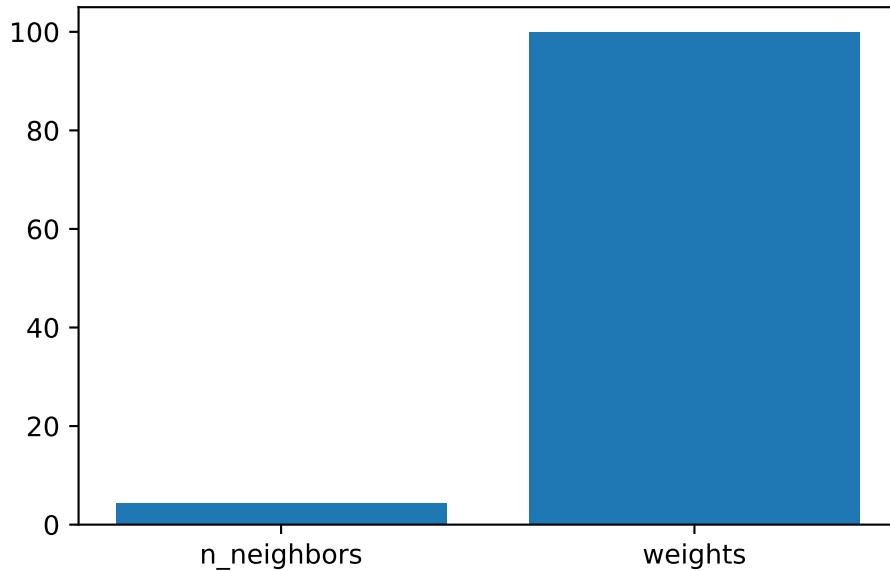


Figure 18.2: Variable importance plot, threshold 0.025.

### 18.9.2 Get Default Hyperparameters

```

from spotPython.hyperparameters.values import get_default_values, transform_hyper_parameter
values_default = get_default_values(fun_control)
values_default = transform_hyper_parameter_values(fun_control=fun_control, hyper_parameter
values_default

{'n_neighbors': 4,
 'weights': 'uniform',
 'algorithm': 'auto',
 'leaf_size': 32,
 'p': 2}

from sklearn.pipeline import make_pipeline
model_default = make_pipeline(fun_control["prep_model"], fun_control["core_model"](**value
model_default

Pipeline(steps=[('nonetype', None),
                 ('kneighborsclassifier',
                  KNeighborsClassifier(leaf_size=32, n_neighbors=4))])

```

### 18.9.3 Get SPOT Results

```
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
print(X)

[[3. 0. 1. 4. 2.]]

from spotPython.hyperparameters.values import assign_values, return_conf_list_from_var_dict
v_dict = assign_values(X, fun_control["var_name"])
return_conf_list_from_var_dict(var_dict=v_dict, fun_control=fun_control)

[{'n_neighbors': 8,
 'weights': 'uniform',
 'algorithm': 'ball_tree',
 'leaf_size': 16,
 'p': 2}]

from spotPython.hyperparameters.values import get_one_sklearn_model_from_X
model_spot = get_one_sklearn_model_from_X(X, fun_control)
model_spot

KNeighborsClassifier(algorithm='ball_tree', leaf_size=16, n_neighbors=8)
```

### 18.9.4 Evaluate SPOT Results

- Fetch the data.

```
from spotPython.utils.convert import get_Xy_from_df
X_train, y_train = get_Xy_from_df(fun_control["train"], fun_control["target_column"])
X_test, y_test = get_Xy_from_df(fun_control["test"], fun_control["target_column"])
X_test.shape, y_test.shape

((63, 64), (63,))
```

- Fit the model with the tuned hyperparameters. This gives one result:

```

model_spot.fit(X_train, y_train)
y_pred = model_spot.predict_proba(X_test)
res = mapk_score(y_true=y_test, y_pred=y_pred, k=3)
res

```

0.7010582010582012

```

def repeated_eval(n, model):
    res_values = []
    for i in range(n):
        model.fit(X_train, y_train)
        y_pred = model.predict_proba(X_test)
        res = mapk_score(y_true=y_test, y_pred=y_pred, k=3)
        res_values.append(res)
    mean_res = np.mean(res_values)
    print(f"mean_res: {mean_res}")
    std_res = np.std(res_values)
    print(f"std_res: {std_res}")
    min_res = np.min(res_values)
    print(f"min_res: {min_res}")
    max_res = np.max(res_values)
    print(f"max_res: {max_res}")
    median_res = np.median(res_values)
    print(f"median_res: {median_res}")
    return mean_res, std_res, min_res, max_res, median_res

```

### 18.9.5 Handling Non-deterministic Results

- Because the model is non-deterministic, we perform  $n = 30$  runs and calculate the mean and standard deviation of the performance metric.

```
_ = repeated_eval(30, model_spot)
```

```

mean_res: 0.7010582010582015
std_res: 3.3306690738754696e-16
min_res: 0.7010582010582012
max_res: 0.7010582010582012
median_res: 0.7010582010582012

```

### 18.9.6 Evaluation of the Default Hyperparameters

```
model_default.fit(X_train, y_train) ["kneighborsclassifier"]
```

```
KNeighborsClassifier(leaf_size=32, n_neighbors=4)
```

- One evaluation of the default hyperparameters is performed on the hold-out test set.

```
y_pred = model_default.predict_proba(X_test)
mapk_score(y_true=y_test, y_pred=y_pred, k=3)
```

```
0.6878306878306879
```

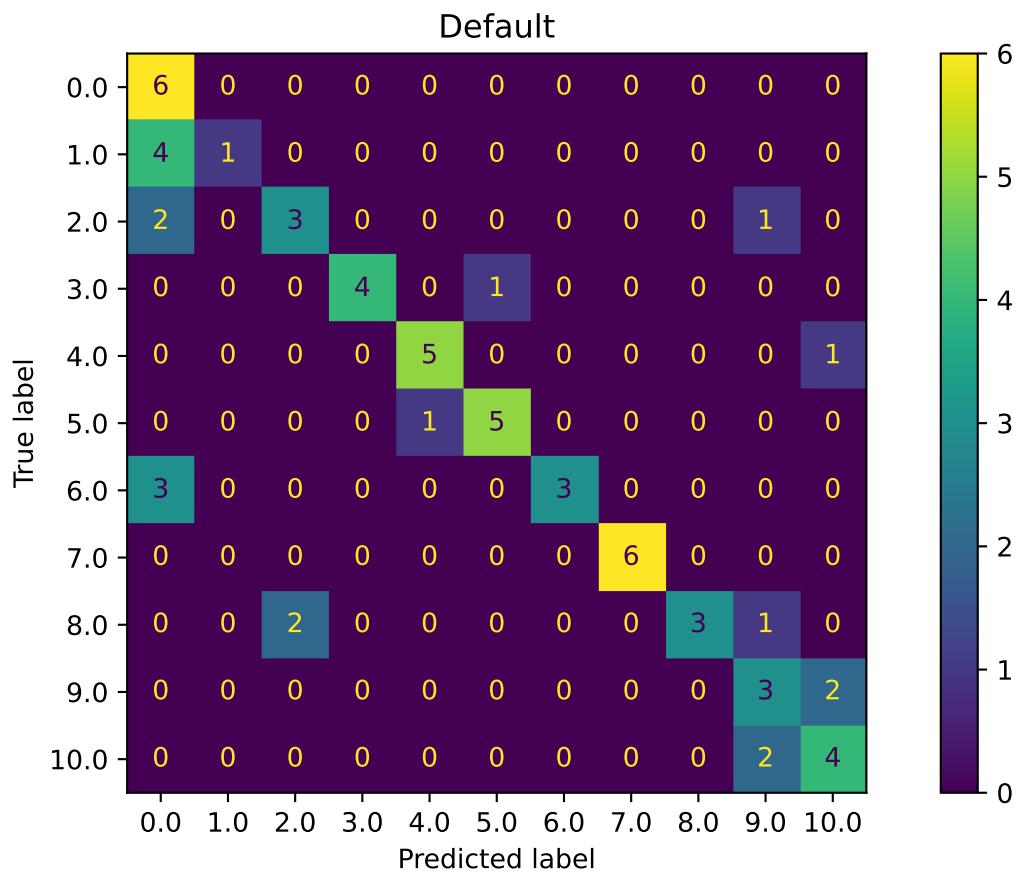
Since one single evaluation is not meaningful, we perform, similar to the evaluation of the SPOT results,  $n = 30$  runs of the default setting and calculate the mean and standard deviation of the performance metric.

```
_ = repeated_eval(30, model_default)
```

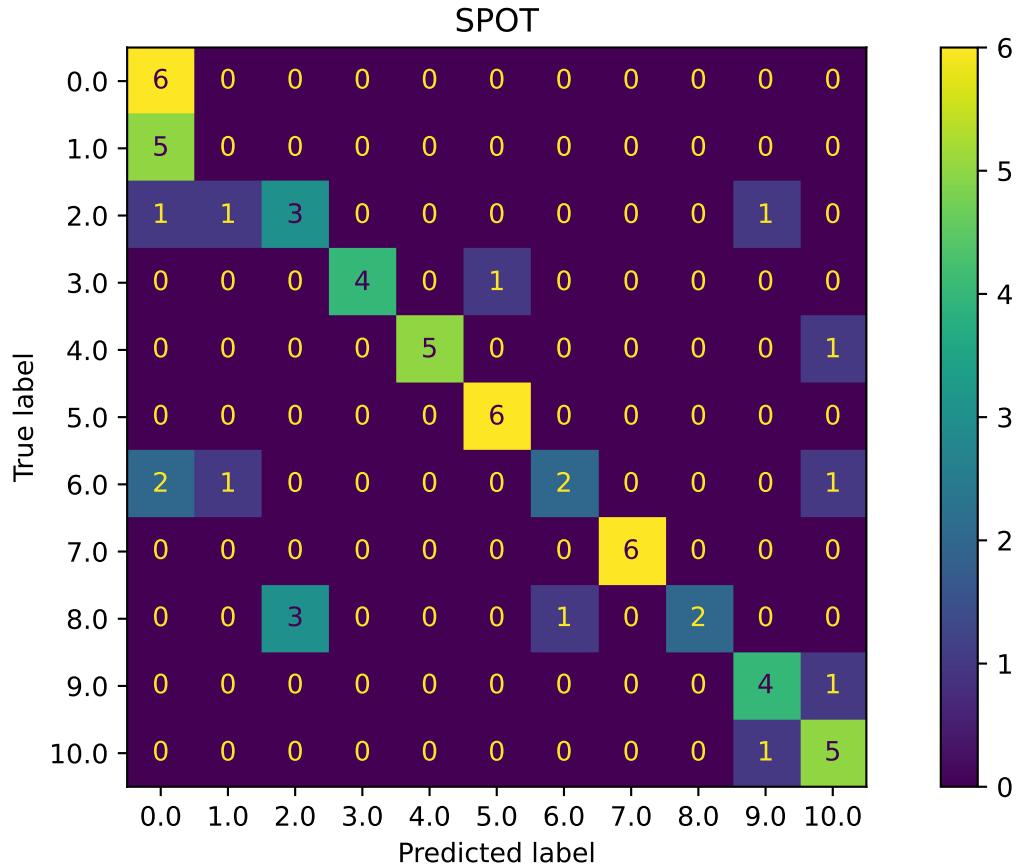
```
mean_res: 0.6878306878306877
std_res: 2.220446049250313e-16
min_res: 0.6878306878306879
max_res: 0.6878306878306879
median_res: 0.6878306878306879
```

### 18.9.7 Plot: Compare Predictions

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



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



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

```
(-0.7465277777777777, -0.59375)
```

### 18.9.8 Cross-validated Evaluations

```
from spotPython.sklearn.traintest import evaluate_cv
fun_control.update({
    "eval": "train_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

```
(0.7156920077972708, None)
```

```
fun_control.update({
    "eval": "test_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

```
Error in fun_sklearn(). Call to evaluate_cv failed. err=ValueError('n_splits=10 cannot be gr
(nan, None)
```

- This is the evaluation that will be used in the comparison:

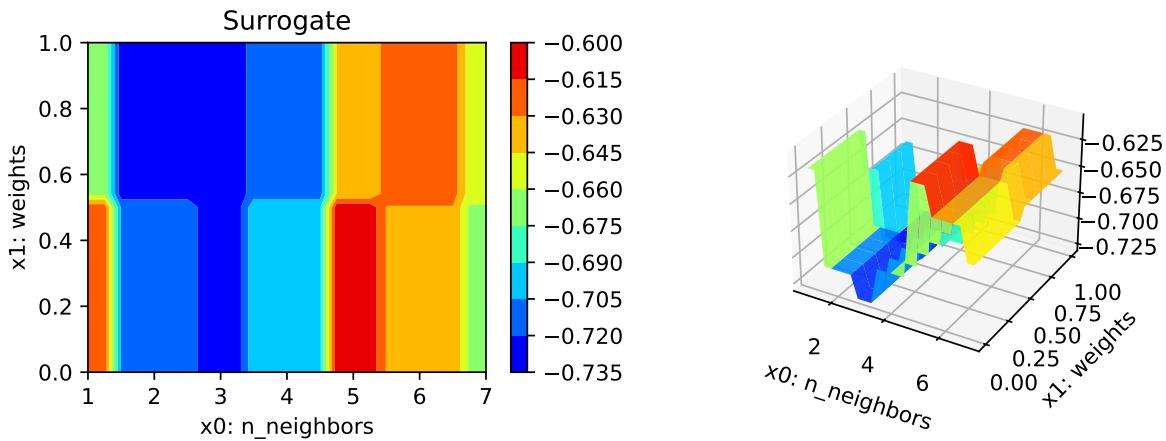
```
fun_control.update({
    "eval": "data_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

```
(0.7089487179487179, None)
```

### 18.9.9 Detailed Hyperparameter Plots

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

n_neighbors: 4.382057528269729
weights: 100.0
```



### 18.9.10 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

### 18.9.11 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)
```

# 19 HPT PyTorch Lightning: VBDP

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

This chapter describes the hyperparameter tuning of a PyTorch `Lightning` network on the Vector Borne Disease Prediction (VBDP) data set.

## ! Vector Borne Disease Prediction Data Set

This chapter uses the Vector Borne Disease Prediction data set from Kaggle. It is a categorical dataset for eleven Vector Borne Diseases with associated symptoms.

The person who associated a work with this deed has dedicated the work to the public domain by waiving all of his or her rights to the work worldwide under copyright law, including all related and neighboring rights, to the extent allowed by law. You can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission. See Other Information below, see <https://creativecommons.org/publicdomain/zero/1.0/>.

The data set is available at: <https://www.kaggle.com/datasets/richardbernat/vector-borne-disease-prediction>,

The data should be downloaded and stored in the `data/VBDP` subfolder. The data set is not available as a part of the `spotPython` package.

This document refers to the latest `spotPython` version, which can be installed via pip. Alternatively, the source code can be downloaded from GitHub: <https://github.com/sequential-parameter-optimization/spotPython>.

- Uncomment the following lines if you want to for (re-)installation the latest version of `spotPython` from GitHub.

```
# import sys
# !{sys.executable} -m pip install --upgrade build
# !{sys.executable} -m pip install --upgrade --force-reinstall spotPython
```

## 19.1 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.

```
MAX_TIME = 1  
INIT_SIZE = 5  
WORKERS = 0  
PREFIX="31"
```

 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. Therefore, we imitate the LightningModule behaviour which selects the highest device.
- The method `spotPython.utils.device.getDevice()` returns the device that is used by Lightning.

## 19.2 Step 2: Initialization of the `fun_control` Dictionary

`spotPython` uses a Python dictionary for storing the information required for the hyperparameter tuning process, which was described in Section [14.2](#), see [Initialization of the `fun\_control` Dictionary](#) in the documentation.

```
from spotPython.utils.init import fun_control_init
from spotPython.utils.file import get_experiment_name, get_spot_tensorboard_path
from spotPython.utils.device import getDevice

experiment_name = get_experiment_name(prefix=PREFIX)
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    num_workers=WORKERS,
    device=getDevice(),
    _L_in=64,
    _L_out=11,
    TENSORBOARD_CLEAN=True)

fun_control["device"]

'mps'
```

## 19.3 Step 3: PyTorch Data Loading

### 19.3.1 Lightning Dataset and DataModule

The data loading and preprocessing is handled by Lightning and PyTorch. It comprehends the following classes:

- **CSVDataset**: A class that loads the data from a CSV file. [\[SOURCE\]](#)
- **CSVDataModule**: A class that prepares the data for training and testing. [\[SOURCE\]](#)

Section [19.12.2](#) illustrates how to access the data.

## 19.4 Step 4: Preprocessing

Preprocessing is handled by Lightning and PyTorch. It can be implemented in the **CSVDataModule** class [\[SOURCE\]](#) and is described in the [LIGHTNINGDATAMODULE](#) documentation. Here you can find information about the **transforms** methods.

## 19.5 Step 5: Select the NN Model (algorithm) and core\_model\_hyper\_dict

spotPython includes the `NetLightBase` 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.netlightbase import NetLightBase
from spotPython.data.light_hyper_dict import LightHyperDict
from spotPython.hyperparameters.values import add_core_model_to_fun_control
add_core_model_to_fun_control(core_model=NetLightBase,
                               fun_control=fun_control,
                               hyper_dict= LightHyperDict)
```

The `NetLightBase` is a configurable neural network. The hyperparameters of the model are specified in the `core_model_hyper_dict` dictionary [SOURCE].

## 19.6 Step 6: Modify hyper\_dict Hyperparameters for the Selected Algorithm aka core\_model

spotPython provides functions for modifying the hyperparameters, their bounds and factors as well as for activating and de-activating hyperparameters without re-compilation of the Python source code. These functions were described in Section 14.6.



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.:
  - `modify_hyper_parameter_bounds(fun_control, "epochs", bounds=[7, 9])` and
  - `modify_hyper_parameter_bounds(fun_control, "patience", bounds=[2, 7])`

```

from spotPython.hyperparameters.values import modify_hyper_parameter_bounds

modify_hyper_parameter_bounds(fun_control, "l1", bounds=[5,8])
modify_hyper_parameter_bounds(fun_control, "epochs", bounds=[6,13])
modify_hyper_parameter_bounds(fun_control, "batch_size", bounds=[2, 8])

from spotPython.hyperparameters.values import modify_hyper_parameter_levels
modify_hyper_parameter_levels(fun_control, "optimizer", ["Adam", "AdamW", "Adamax", "NAdam"])
# modify_hyper_parameter_levels(fun_control, "optimizer", ["Adam"])

```

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	5	8	transform_power_2_int
epochs	int	4	6	13	transform_power_2_int
batch_size	int	4	2	8	transform_power_2_int
act_fn	factor	ReLU	0	5	None
optimizer	factor	SGD	0	3	None
dropout_prob	float	0.01	0	0.25	None
lr_mult	float	1.0	0.1	10	None
patience	int	2	2	6	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.

**i** 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"]`.

## 19.7 Step 7: Data Splitting, the Objective (Loss) Function and the Metric

### 19.7.1 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 (see Section [14.7.1](#))
2. the loss function (and a metric).

 Caution: Data Splitting in Lightning

- The data splitting is handled by `Lightning`.

### 19.7.2 Loss Functions and Metrics

The loss function is specified in the configurable network class [\[SOURCE\]](#). We will use CrossEntropy loss for the multiclass-classification task.

### 19.7.3 Metric

- We will use the MAP@k metric [\[SOURCE\]](#) for the evaluation of the model.
- An example, how this metric works, is shown in the Appendix, see Section {Section [19.12.3](#)}.

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`.

## 19.8 Step 8: Calling the SPOT Function

### 19.8.1 Preparing the SPOT Call

The following code passes the information about the parameter ranges and bounds to `spot`. It extracts the variable types, names, and bounds

```
from spotPython.hyperparameters.values import (get_bound_values,
                                                get_var_name,
                                                get_var_type)
var_type = get_var_type(fun_control)
var_name = get_var_name(fun_control)
lower = get_bound_values(fun_control, "lower")
upper = get_bound_values(fun_control, "upper")
```

### 19.8.2 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().fun
```

### 19.8.3 Starting the Hyperparameter Tuning

The `spotPython` hyperparameter tuning is started by calling the `Spot` function [SOURCE] as described in Section 14.8.4.

```
import numpy as np
from spotPython.spot import spot
from math import inf
spot_tuner = spot.Spot(fun=fun,
                      lower = lower,
                      upper = upper,
                      fun_evals = inf,
                      max_time = MAX_TIME,
                      tolerance_x = np.sqrt(np.spacing(1)),
                      var_type = var_type,
                      var_name = var_name,
```

```
        show_progress= True,
        fun_control = fun_control,
        design_control={"init_size": INIT_SIZE},
        surrogate_control={"noise": True,
                           "min_theta": -4,
                           "max_theta": 3,
                           "n_theta": len(var_name),
                           "model_fun_evals": 10_000,
                           })
spot_tuner.run()
```

fun: Calling train\_model

Validate metric	DataLoader 0
hp_metric	2.2744216918945312
val_acc	0.24381625652313232
val_loss	2.2744216918945312
valid_mapk	0.3699631094932556

fun: train\_model returned

fun: Calling train\_model

Validate metric	DataLoader 0
hp_metric	2.2821764945983887
val_acc	0.268551230430603
val_loss	2.2821764945983887
valid_mapk	0.3694058656692505

fun: train\_model returned

fun: Calling train\_model

Validate metric	DataLoader 0
hp_metric	2.40523099899292
val_acc	0.13780918717384338
val_loss	2.40523099899292

```
valid_mapk          0.1892361044883728

fun: train_model returned
fun: Calling train_model
```

```
Validate metric      DataLoader 0

hp_metric           2.301100015640259
val_acc             0.23674911260604858
val_loss             2.301100015640259
valid_mapk          0.3090277910232544
```

```
fun: train_model returned
fun: Calling train_model
```

```
Validate metric      DataLoader 0

hp_metric           2.2979023456573486
val_acc             0.22968198359012604
val_loss             2.2979023456573486
valid_mapk          0.3073495328426361
```

```
fun: train_model returned
fun: Calling train_model
```

```
Validate metric      DataLoader 0

hp_metric           2.285515546798706
val_acc             0.24028268456459045
val_loss             2.285515546798706
valid_mapk          0.3344908058643341
```

```
fun: train_model returned
```

```
spotPython tuning: 2.2744216918945312 [#####----] 64.83%
```

```
fun: Calling train_model
```

```
Validate metric           DataLoader 0

    hp_metric          2.285055637359619
    val_acc            0.2473498284816742
    val_loss           2.285055637359619
    valid_mapk         0.3562082052230835

fun: train_model returned

spotPython tuning: 2.2744216918945312 [#####---] 69.76%

fun: Calling train_model

Validate metric           DataLoader 0

    hp_metric          2.276766061782837
    val_acc            0.2473498284816742
    val_loss           2.276766061782837
    valid_mapk         0.3588348925113678

fun: train_model returned

spotPython tuning: 2.2744216918945312 [#####---] 92.75%

fun: Calling train_model

Validate metric           DataLoader 0

    hp_metric          2.272672414779663
    val_acc            0.2473498284816742
    val_loss           2.272672414779663
    valid_mapk         0.3421362042427063

fun: train_model returned

spotPython tuning: 2.272672414779663 [#####] 100.00% Done...

<spotPython.spot.spot.Spot at 0x2f7267eb0>
```

## 19.9 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.

## 19.10 Step 10: Results

After the hyperparameter tuning run is finished, the results can be analyzed as described in Section [14.10](#).

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

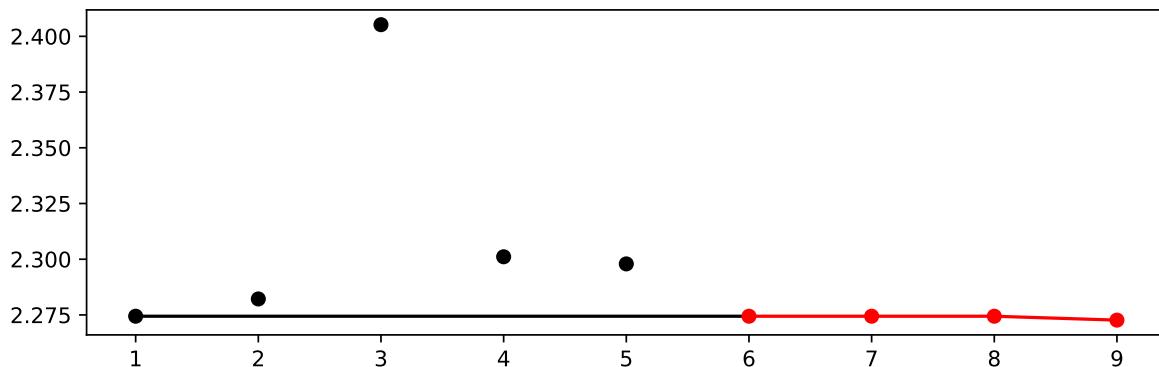


Figure 19.1: Progress plot. *Black dots* denote results from the initial design. *Red dots* illustrate the improvement found by the surrogate model based optimization.

```
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	5.0	8.0	5.0	transform_p
epochs	int	4	6.0	13.0	7.0	transform_p
batch_size	int	4	2.0	8.0	2.0	transform_p

act_fn	factor	ReLU	0.0	5.0	3.0	None
optimizer	factor	SGD	0.0	3.0	2.0	None
dropout_prob	float	0.01	0.0	0.25	0.25	None
lr_mult	float	1.0	0.1	10.0	2.1136416734217627	None
patience	int	2	2.0	6.0	5.0	transform_p
initialization	factor	Default	0.0	2.0	0.0	None

```
spot_tuner.plot_importance(threshold=0.025,
                           filename="./figures/" + experiment_name+"_importance.png")
```

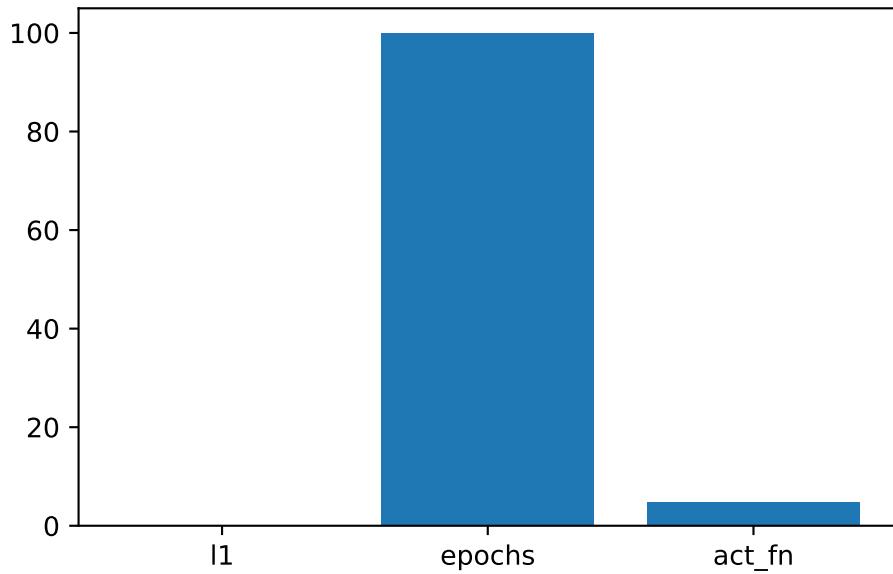


Figure 19.2: Variable importance plot, threshold 0.025.

### 19.10.1 Get the Tuned Architecture

```
from spotPython.light.utils import get_tuned_architecture
config = get_tuned_architecture(spot_tuner, fun_control)
```

- Test on the full data set

```
from spotPython.light.traintest import test_model
test_model(config, fun_control)
```

```
Test metric           DataLoader 0
hp_metric            2.19404935836792
test_mapk_epoch      0.42514118552207947
val_acc              0.3465346395969391
val_loss              2.19404935836792
```

```
(2.19404935836792, 0.3465346395969391)
```

```
from spotPython.light.traintest import load_light_from_checkpoint
model_loaded = load_light_from_checkpoint(config, fun_control)
```

```
Loading model from runs/saved_models/-2182942955981762195_TEST/last.ckpt
```

### 19.10.2 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.traintest import cv_model
# set the number of folds to 10
fun_control["k_folds"] = 10
cv_model(config, fun_control)
```

```
k: 0
Train Dataset Size: 636
Val Dataset Size: 71
```

```
Validate metric       DataLoader 0
hp_metric             2.1368539333343506
```

```
    val_acc          0.4084506928920746
    val_loss         2.1368539333343506
    valid_mapk       0.4606481194496155
```

```
train_model result: {'valid_mapk': 0.4606481194496155, 'val_loss': 2.1368539333343506, 'val_a
k: 1
Train Dataset Size: 636
Val Dataset Size: 71
```

Validate metric	DataLoader 0
hp_metric	2.2264761924743652
val_acc	0.30985915660858154
val_loss	2.2264761924743652
valid_mapk	0.4004629850387573

```
train_model result: {'valid_mapk': 0.4004629850387573, 'val_loss': 2.2264761924743652, 'val_a
k: 2
Train Dataset Size: 636
Val Dataset Size: 71
```

Validate metric	DataLoader 0
hp_metric	2.3198959827423096
val_acc	0.23943662643432617
val_loss	2.3198959827423096
valid_mapk	0.3009259104728699

```
train_model result: {'valid_mapk': 0.3009259104728699, 'val_loss': 2.3198959827423096, 'val_a
k: 3
Train Dataset Size: 636
Val Dataset Size: 71
```

Validate metric	DataLoader 0
hp_metric	2.2523036003112793
val_acc	0.2535211145877838
val_loss	2.2523036003112793
valid_mapk	0.3634259104728699

```
train_model result: {'valid_mapk': 0.3634259104728699, 'val_loss': 2.2523036003112793, 'val_a
k: 4
Train Dataset Size: 636
Val Dataset Size: 71

Validate metric      DataLoader 0

    hp_metric      2.302013635635376
        val_acc      0.22535210847854614
        val_loss      2.302013635635376
        valid_mapk    0.2847222089767456

train_model result: {'valid_mapk': 0.2847222089767456, 'val_loss': 2.302013635635376, 'val_a
k: 5
Train Dataset Size: 636
Val Dataset Size: 71

Validate metric      DataLoader 0

    hp_metric      2.2929389476776123
        val_acc      0.19718310236930847
        val_loss      2.2929389476776123
        valid_mapk    0.3379629850387573

train_model result: {'valid_mapk': 0.3379629850387573, 'val_loss': 2.2929389476776123, 'val_a
k: 6
Train Dataset Size: 636
Val Dataset Size: 71

Validate metric      DataLoader 0

    hp_metric      2.294412851333618
        val_acc      0.22535210847854614
        val_loss      2.294412851333618
        valid_mapk    0.3356481194496155

train_model result: {'valid_mapk': 0.3356481194496155, 'val_loss': 2.294412851333618, 'val_a
```

```
Train Dataset Size: 637
Val Dataset Size: 70
```

```
Validate metric      DataLoader 0

hp_metric           2.260765790939331
val_acc             0.2571428716182709
val_loss            2.260765790939331
valid_mapk          0.37731483578681946
```

```
train_model result: {'valid_mapk': 0.37731483578681946, 'val_loss': 2.260765790939331, 'val_a
k: 8
Train Dataset Size: 637
Val Dataset Size: 70
```

```
Validate metric      DataLoader 0

hp_metric           2.3352291584014893
val_acc             0.17142857611179352
val_loss            2.3352291584014893
valid_mapk          0.3194444477558136
```

```
train_model result: {'valid_mapk': 0.3194444477558136, 'val_loss': 2.3352291584014893, 'val_a
k: 9
Train Dataset Size: 637
Val Dataset Size: 70
```

```
Validate metric      DataLoader 0

hp_metric           2.22326922416687
val_acc             0.3142857253551483
val_loss            2.22326922416687
valid_mapk          0.36574074625968933
```

```
train_model result: {'valid_mapk': 0.36574074625968933, 'val_loss': 2.22326922416687, 'val_a
0.3546296268701553
```

**i** Note: Evaluation for the Final Comaprison

- This is the evaluation that will be used in the comparison.

### 19.10.3 Detailed Hyperparameter Plots

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

```
11: 0.11651205242017379
epochs: 100.0
act_fn: 4.7762099479218705
```

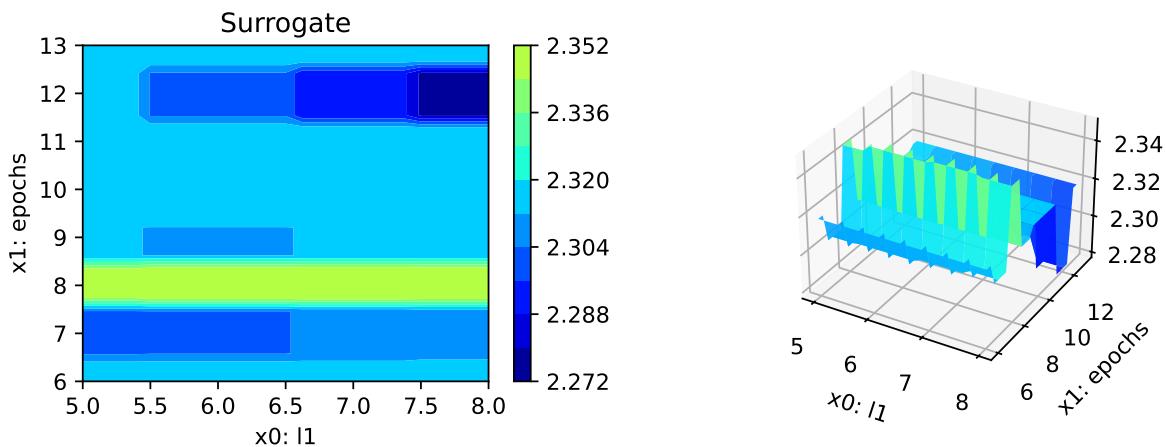
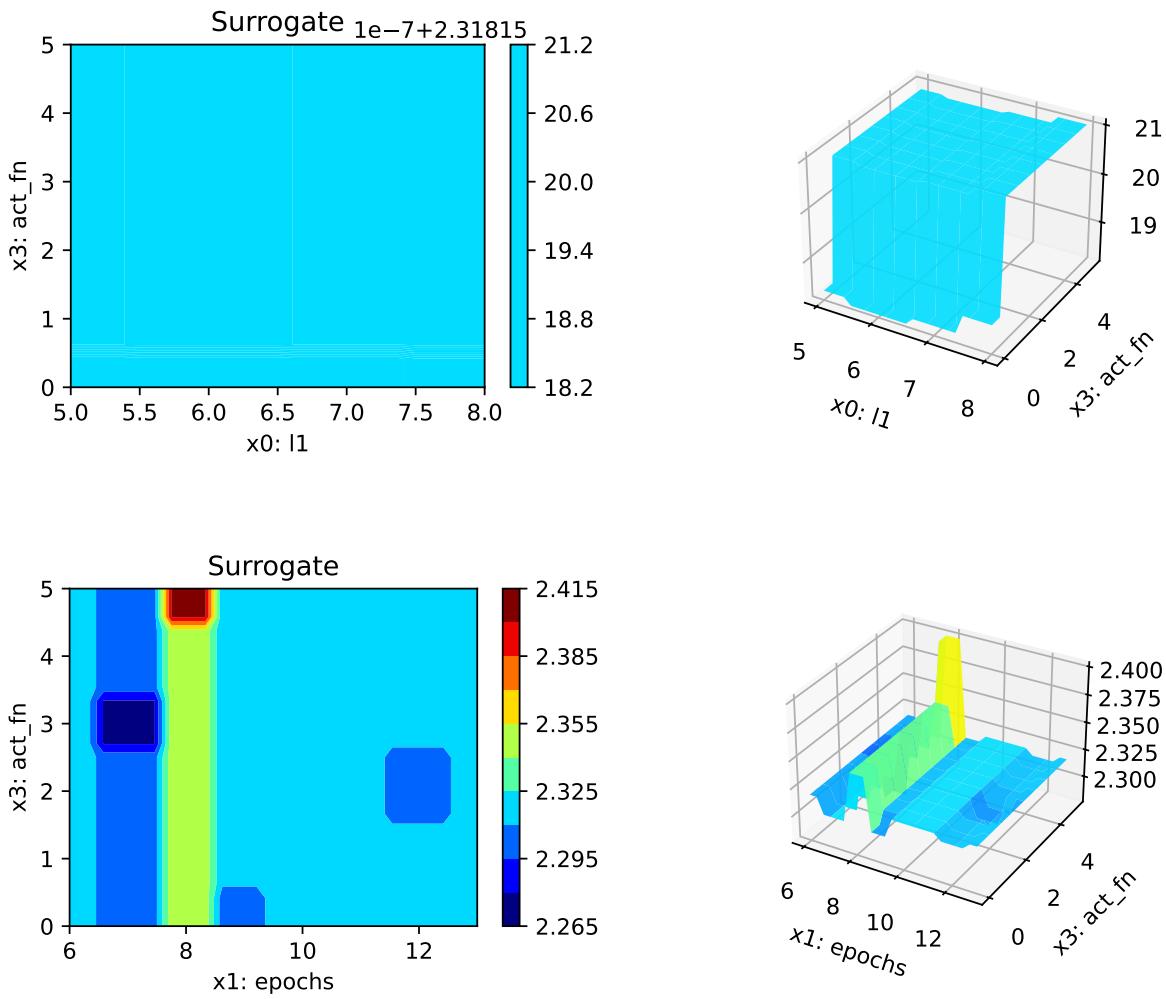


Figure 19.3: Contour plots.



#### 19.10.4 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

### 19.10.5 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)
```

### 19.10.6 Visualizing the Activation Distribution

 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:

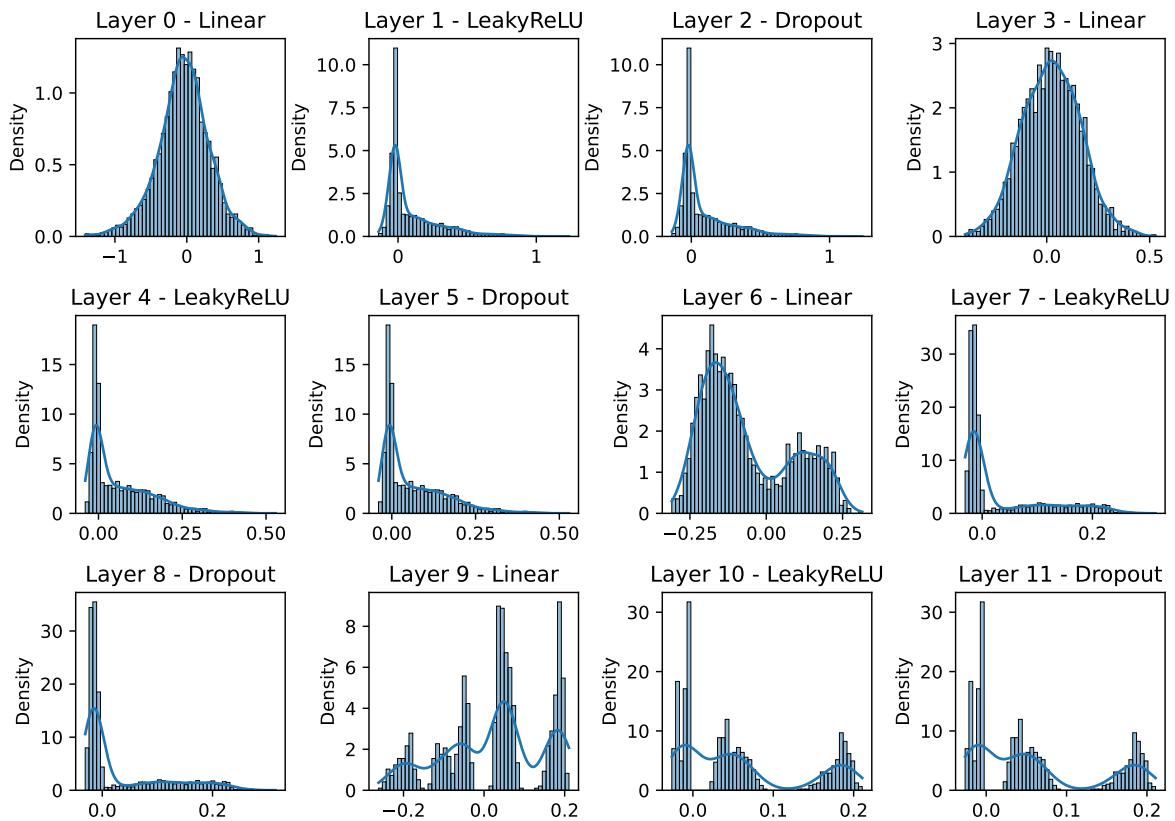
```
from spotPython.torch.activation import Sigmoid, Tanh, ReLU, LeakyReLU, ELU, Swish
act_fn_by_name = {"sigmoid": Sigmoid, "tanh": Tanh, "relu": ReLU, "leakyrelu": LeakyReLU, "elu": ELU, "swish": Swish}

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 = NetLightBase(
    (train_mapk): MAPK(),
    (valid_mapk): MAPK(),
    (test_mapk): MAPK(),
    (layers): Sequential(
        (0): Linear(in_features=64, out_features=32, bias=True)
```

```
(1): LeakyReLU()
(2): Dropout(p=0.25, inplace=False)
(3): Linear(in_features=32, out_features=16, bias=True)
(4): LeakyReLU()
(5): Dropout(p=0.25, inplace=False)
(6): Linear(in_features=16, out_features=16, bias=True)
(7): LeakyReLU()
(8): Dropout(p=0.25, inplace=False)
(9): Linear(in_features=16, out_features=8, bias=True)
(10): LeakyReLU()
(11): Dropout(p=0.25, inplace=False)
(12): Linear(in_features=8, out_features=11, bias=True)
)
)

from spotPython.utils.eda import visualize_activations
visualize_activations(model, color=f"C{0}")
```

Activation distribution for activation function LeakyReLU()



## 19.11 Submission

```

import numpy as np
import pandas as pd
from sklearn.preprocessing import OrdinalEncoder

import pandas as pd
from sklearn.preprocessing import OrdinalEncoder
train_df = pd.read_csv('./data/VBDP/train.csv', index_col=0)
# remove the id column
# train_df = train_df.drop(columns=['id'])
n_samples = train_df.shape[0]
n_features = train_df.shape[1] - 1

```

```

target_column = "prognosis"
# Encode our prognosis labels as integers for easier decoding later
enc = OrdinalEncoder()
y = enc.fit_transform(train_df[[target_column]])
test_df = pd.read_csv('./data/VBDP/test.csv', index_col=0)
test_df

```

	sudden_fever	headache	mouth_bleed	nose_bleed	muscle_pain	joint_pain	vomiting	rash	...
id									
707	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
708	1.0	1.0	0.0	1.0	0.0	1.0	1.0	1.0	1.0
709	1.0	1.0	0.0	1.0	1.0	1.0	1.0	0.0	0.0
710	0.0	1.0	0.0	0.0	0.0	1.0	1.0	1.0	1.0
711	0.0	0.0	1.0	0.0	1.0	1.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...
1005	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1006	1.0	0.0	1.0	0.0	1.0	1.0	0.0	1.0	0.0
1007	1.0	0.0	0.0	1.0	1.0	0.0	1.0	1.0	0.0
1008	1.0	0.0	1.0	1.0	1.0	0.0	1.0	0.0	0.0
1009	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0

```

import torch
X_tensor = torch.Tensor(test_df.values)
X_tensor = X_tensor.to(fun_control["device"])

y = model_loaded(X_tensor)
y.shape

torch.Size([303, 11])

# convert the predictions to a numpy array
y = y.cpu().detach().numpy()
y

array([[9.34182387e-03, 5.37784013e-04, 2.97997333e-03, ...,
       1.40091812e-03, 2.82173569e-04, 6.32511335e-04],
       [9.99848008e-01, 7.51649758e-18, 1.86774578e-25, ...,
       4.66555216e-25, 7.63578790e-28, 1.09261675e-27],
       ...])

```

```
[0.00000000e+00, 7.19368104e-34, 4.33420110e-03, ...,
 3.09993569e-02, 8.19055815e-13, 9.93454279e-08],
...,
[0.00000000e+00, 0.00000000e+00, 8.14403385e-16, ...,
 2.04248618e-09, 1.57991721e-34, 7.37077867e-24],
[0.00000000e+00, 2.22367631e-28, 2.23893952e-02, ...,
 8.89114141e-02, 1.02690148e-10, 2.38016310e-06],
[0.00000000e+00, 4.63367610e-30, 1.01549635e-02, ...,
 5.43198213e-02, 2.20471991e-11, 7.28340353e-07]], dtype=float32)
```

```
test_sorted_prediction_ids = np.argsort(-y, axis=1)
test_top_3_prediction_ids = test_sorted_prediction_ids[:, :3]
original_shape = test_top_3_prediction_ids.shape
test_top_3_prediction = enc.inverse_transform(test_top_3_prediction_ids.reshape(-1, 1))
test_top_3_prediction = test_top_3_prediction.reshape(original_shape)
test_df['prognosis'] = np.apply_along_axis(lambda x: np.array(' '.join(x), dtype="object"),
test_df['prognosis'].reset_index().to_csv('./data/VBDP/submission.csv', index=False)
```

## 19.12 Appendix

### 19.12.1 Differences to the spotPython Approaches for torch, sklearn and river

 Caution: Data Loading in Lightning

- Data loading is handled independently from the `fun_control` dictionary by Lightning and PyTorch.
- In contrast to spotPython with `torch`, `river` and `sklearn`, the data sets are not added to the `fun_control` dictionary.

#### 19.12.1.1 Specification of the Preprocessing Model

The `fun_control` dictionary, the `torch`, `sklearn` and `river` versions of spotPython allow the specification of a data preprocessing pipeline, e.g., for the scaling of the data or for the one-hot encoding of categorical variables, see Section 14.4. This feature is not used in the Lightning version.



## Caution: Data preprocessing in Lightning

Lightning allows the data preprocessing to be specified in the `LightningDataModule` class. It is not considered here, because it should be computed at one location only.

## 19.12.2 Taking a Look at the Data

### 19.12.3 The MAPK Metric

Here is an example how the MAPK metric is calculated.

```
from spotPython.torch.mapk import MAPK
import torch
mapk = MAPK(k=2)
target = torch.tensor([0, 1, 2, 2])
preds = torch.tensor(
    [
        [0.5, 0.2, 0.2], # 0 is in top 2
        [0.3, 0.4, 0.2], # 1 is in top 2
        [0.2, 0.4, 0.3], # 2 is in top 2
        [0.7, 0.2, 0.1], # 2 isn't in top 2
    ]
)
mapk.update(preds, target)
print(mapk.compute()) # tensor(0.6250)

tensor(0.6250)
```

# **Part III**

# **Optimization**

## 20 Introduction to `scipy.optimize`

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>

# 21 Lecture 01: Background

## 21.1 Contents of this Course

- 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

## 21.2 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

## **21.3 Surrogate Example: Data**

## **21.4 Surrogate Example: Cubic Regression Model**

## **21.5 Extrapolation**

## **21.6 New Modeling Approach**

## **21.7 Comparison**

## **21.8 Benefits**

- 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

## **21.9 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.

## **21.10 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

## 21.11 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**

## 21.12 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

## 21.13 Experimentation is Changing

- 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

## 21.14 Examples

- Facebook uses surrogates to tailor its web portal and apps to optimize engagement
- Uber uses surrogates trained to traffic simulations to route pooled ride-shares in real-time, reducing travel and wait time

## 21.15 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–Volterra) 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

## 21.16 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

## 21.17 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**

## 21.18 Approaches: RSM and DACE (GP)

- One perspective is so-called **response surface methods** (RSMs):
- a poster child from industrial statistics' heyday, well before information technology became a dominant industry

## **21.19 Response Surface Methods: Pros and Cons**

- Pros:
  - Literature is rich
  - Methods are tried and tested in practice, especially in manufacturing
  - Careful experimental design paired with a well understood model and humble expectations
  - Can add a lot of value to scientific inquiry, process refinement, optimization, and more
- Cons:
  - Surrogates: simple, crude
  - Hands-on methods
  - Local methods

## **21.20 Response Surface Methods: Related Fields**

- Design of Experiments
- Quality management
- Reliability and
- Productivity

## **21.21 RSM Applications**

- Billions of applications from industry and manufacturing
  - Focused on design, development, and formulation of new products and the improvement of existing products
  - Also from laboratory research
- Main domains:
  - materials science,
  - manufacturing,
  - applied chemistry, and
  - climate science, and many more

## 21.22 What is RSM?

- Collection of statistical and mathematical tools useful for
  - developing,
  - improving, and
  - optimizing processes
- Overarching theme:
  - *study of how input variables controlling a product or process potentially influence a response measuring performance or quality characteristics*

## 21.23 RSM Example

- Relationship between the **response variable** yield ( $y$ ) in a chemical process and two **process variables**:
  - reaction time ( $\xi_1$ ) and
  - reaction temperature ( $\xi_2$ ).

# 22 Lecture 02

## 22.1 RSM: Example

- Code below synthesizes this setting for the benefit of illustration
  - the form is a variation on the so-called *banana function*

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from mpl_toolkits.mplot3d import Axes3D

#%matplotlib inline
#%matplotlib notebook
#%pylab
```

## 22.2 3d Plot

```
import numpy as np
# from mpl_toolkits.mplot3d import Axes3D
# Axes3D import has side effects, it enables using projection='3d' in add_subplot
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)
```

```

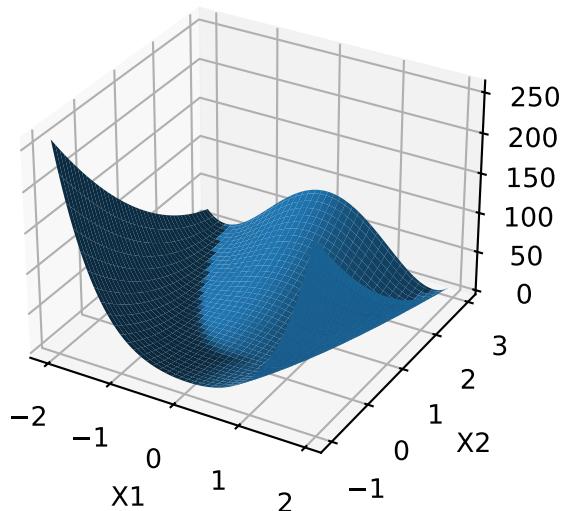
zs = np.array(fun_rosen(np.ravel(X), np.ravel(Y)))
Z = zs.reshape(X.shape)

ax.plot_surface(X, Y, Z)

ax.set_xlabel('X1')
ax.set_ylabel('X2')
ax.set_zlabel('Y')

plt.show()

```



## 22.3 Contour plot

```

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)

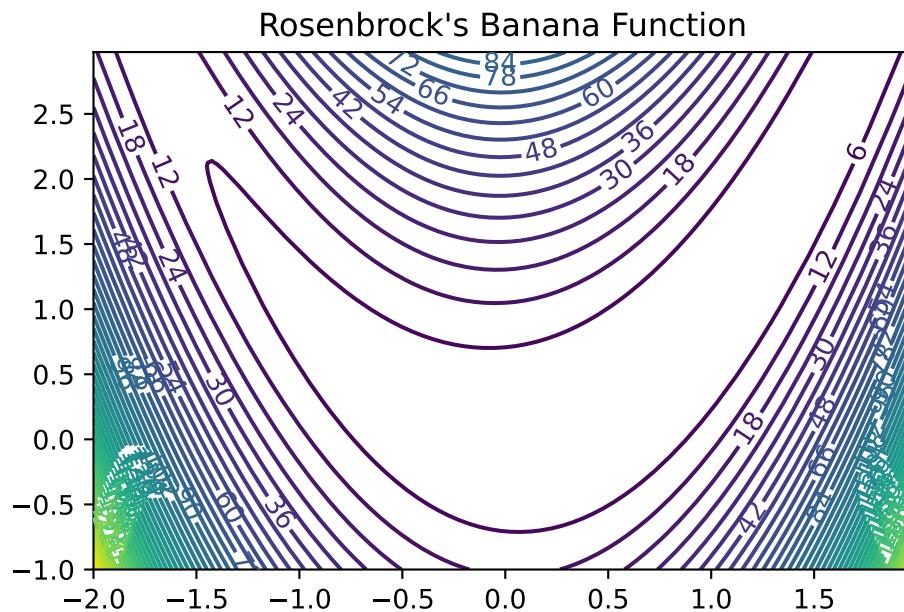
```

```

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  $(\xi_1, \xi_2)$

## 22.4 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

## 22.5 RSM: Strategies

- RSMs consist of experimental strategies for

- **exploring** the space of the process (i.e., independent/input) variables (above  $\xi_1$  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)

## 22.6 RSM: 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  $\xi_1, \xi_2, \dots, \xi_m$

## 22.7 RSM: Equations of the Empirical Model

- $Y = f(\xi_1, \xi_2, \dots, \xi_m) + \epsilon$
- $\mathbb{E}\{Y\} = \eta = f(\xi_1, \xi_2, \dots, \xi_m)$
- $\epsilon$  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

## 22.8 RSM: Noise in the Empirical Model

- Typical simplifying assumption:  $\epsilon \sim N(0, \sigma^2)$
- We seek estimates for  $f$  and  $\sigma^2$  from noisy observations  $Y$  at inputs  $\xi$

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

# 23 Lecture 02: RSM Low-order Polynomials

## 23.1 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**

## 23.2 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

## 23.3 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
```

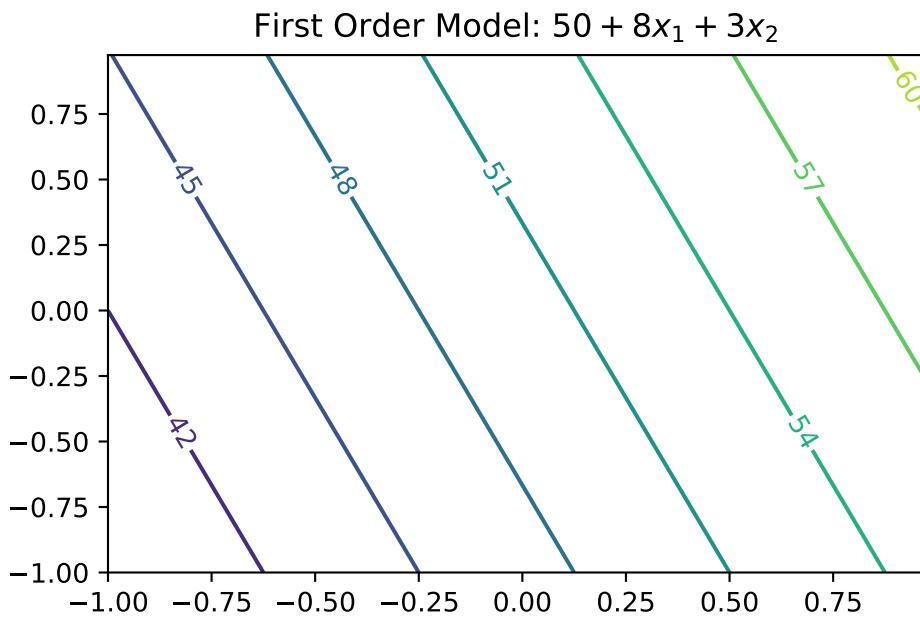
```

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\$')



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

## 23.5 First-order Model with Interactions in python

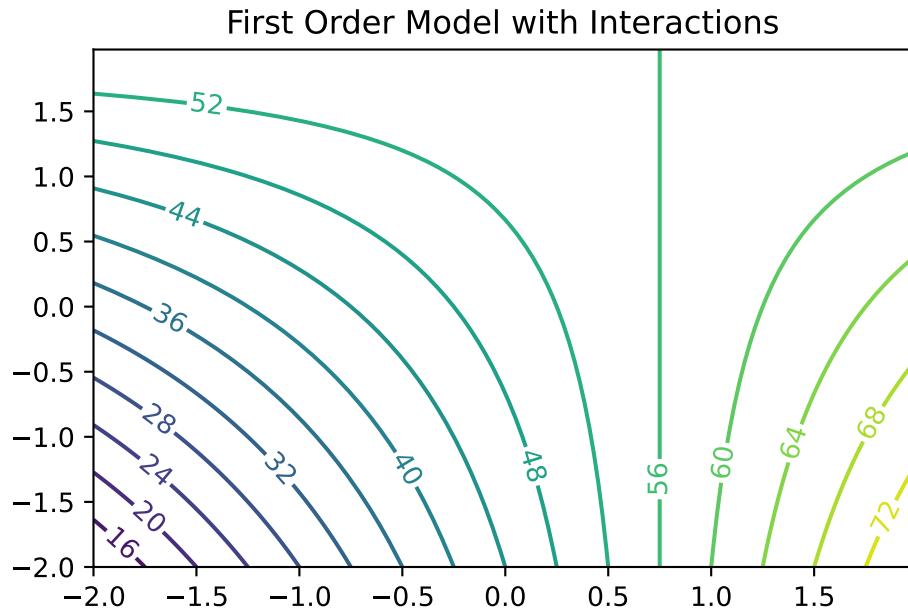
- Code below facilitates evaluations for pairs  $(x_1, x_2)$
- Responses may be observed over a mesh in the same double-unit square

```
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')
```



## 23.6 First-order Model with Interactions

## 23.7 Observations: First-Order Model with Interactions

- Mean response  $\eta$  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

## 23.8 Second-Order Models

- Second-order model may be appropriate near local optima where  $f$  would have substantial curvature:

$$\eta = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{11} x_1^2 + \beta_{22} x_2^2 + \beta_{12} x_1 x_2$$

- For example

$$\eta = 50 + 8x_1 + 3x_2 - 7x_1^2 - 3x_2^2 - 4x_1 x_2$$

- 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

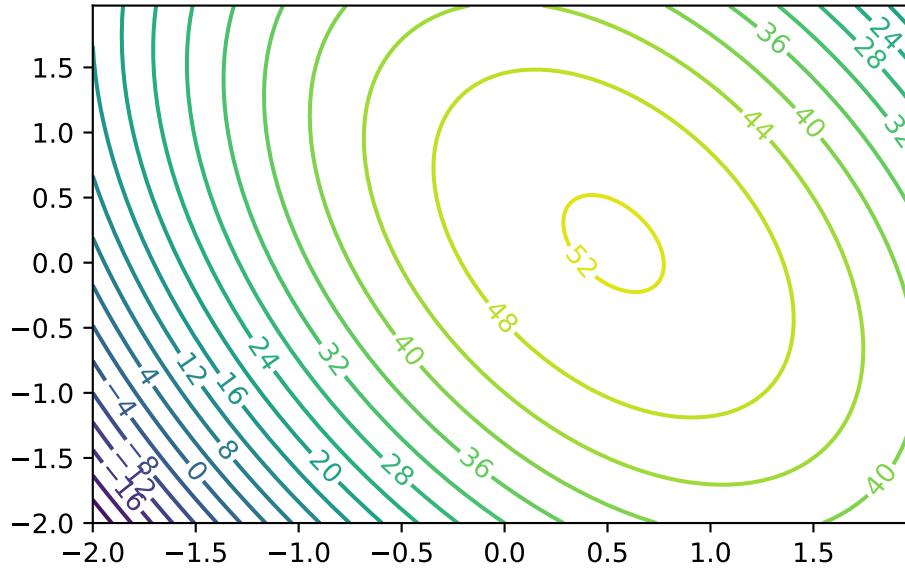
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

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)\$')

Second Order Model with Interactions. Maximum near about (0.6, 0.2)



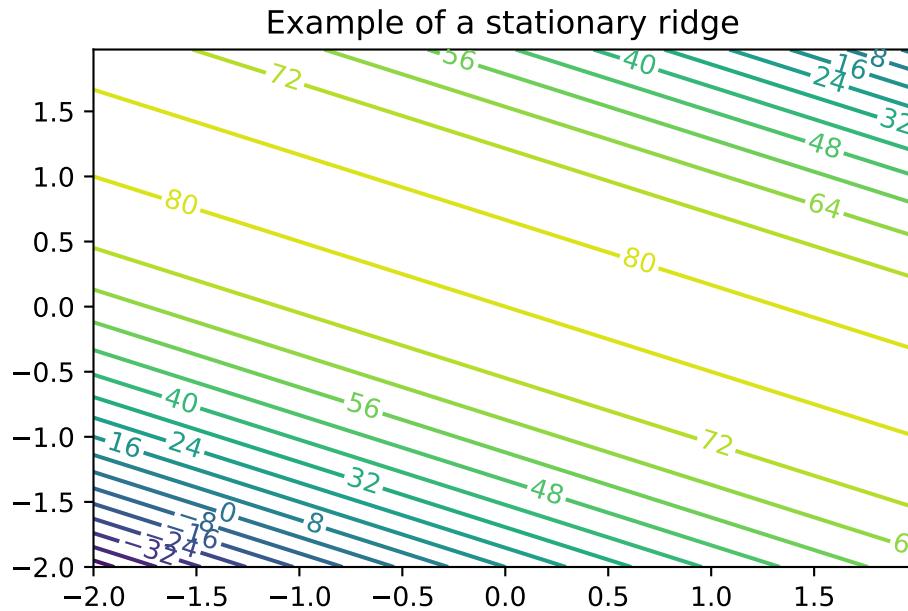
## 23.9 Second-Order Models: Properties

- 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

## 23.10 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')
```



### 23.11 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

### 23.12 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

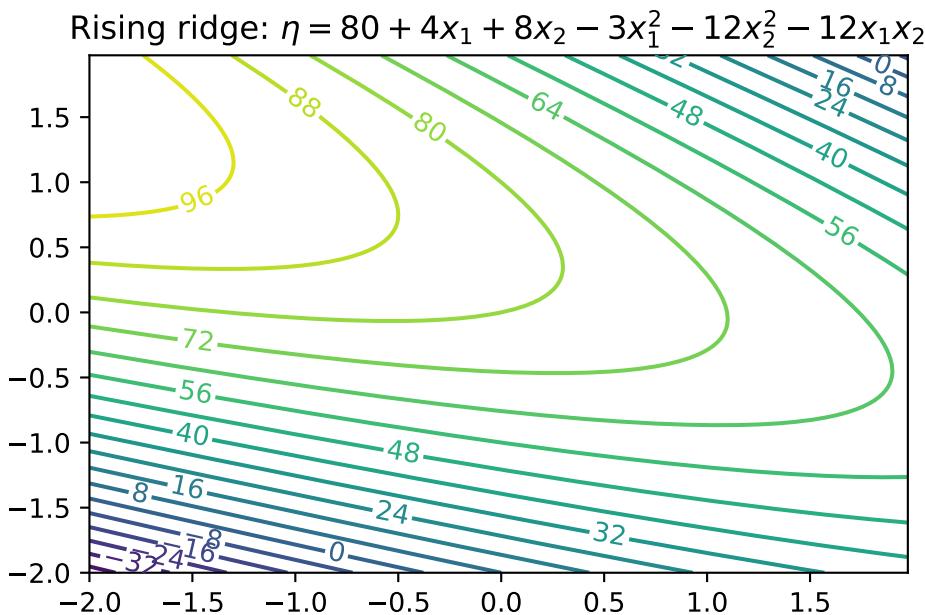
delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
```

```

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\$')



### 23.13 Summary: Rising Ridge

- The stationary point is remote to the study region
- Continuum 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.

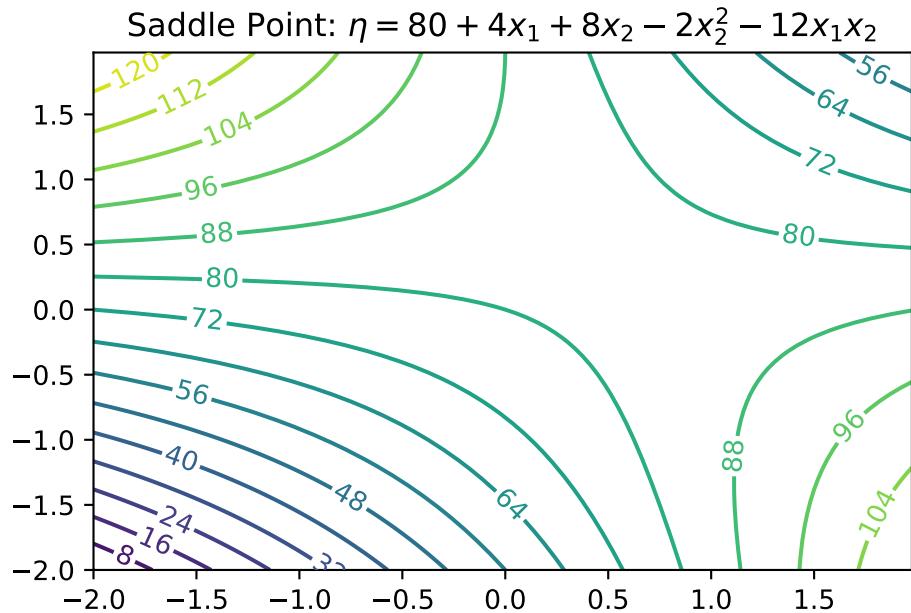
## 23.14 Falling Ridge

- 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.

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



### 23.16 Interpretation: Saddle Points

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

### 23.17 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

## 23.18 General RSM Models

- General **first-order model** on  $m$  process variables  $x_1, x_2, \dots, x_m$  is

$$\eta = \beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m$$

- General **second-order model** on  $m$  process variables

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

## 23.19 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

## 23.20 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

## 23.21 Different Designs

- **Screening designs:** 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

## 23.22 RSM Experimentation: 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

## 23.23 RSM Experimentation: 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

## 23.24 RSM Experimentation: 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

## 24 Example 1 (DOE)

- Example is shown in the notebook [doe1.ipynb](#)

## 25 Example 2 (DOE)

- Example is shown in the notebook [doe2.ipynb](#)

# 26 RSM: Review and General Considerations

## 26.1 First Glimpse

- RSM seems sensible, and pretty straightforward as quantitative statistics-based analysis goes
- 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

## 26.2 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

## 26.3 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

## 26.4 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

## 26.5 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

## 26.6 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

## 26.7 Status Quo

- Nowadays, field experiments and statistical models, designs and optimizations are coupled 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

## 26.8 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

## 26.9 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

## 27 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.

# 28 Lecture 04: DACE and Aircraft Wing Example

## 28.1 Design and Analysis of Computer Experiments

- Mathematical models implemented in computer codes to avoid expensive field data collection
- Highly nonlinear response surfaces, high signal-to-noise ratios (often deterministic evaluations) and global scope: new approach required compared to RSM
- Improved computing power and simulation fidelity result in higher confidence:
  - better understanding of physical, biological, social dynamics
- Expansion of configuration spaces and increasing input dimension yearn for bigger designs
- High performance computing (HPC) allowing thousands of runs where only tens possible before
- Shifts the burden to big models, big training data: new computational challenges

## 28.2 Research Questions for DACE

- Research questions include:
  - how to *design computer experiments* that spend on computation judiciously, and
  - how to *meta-model* computer codes to save on simulation effort
- Choice of surrogate model for computer codes has substantial effect on the optimal design of the experiment
- Depending on your goal, different model-design pairs may be preferred
- Combining computer simulation, design, and modeling with field data from similar, real-world experiments leads to a new class of computer model tuning problems
- Goal: to **automate** to the extent possible so that HPC can be deployed with minimal human intervention

## 28.3 Remaining Differences Between RSM and DACE: Noise

- Replication, which would never feature in a deterministic setting, is used to separate signal from noise
- Traditional RSM is intended for situations in which a substantial proportion of variability in the data is just noise and the number of data values that can be acquired can sometimes be severely limited
- Consequently, RSM is intended for a somewhat different class of problems, and is indeed well-suited for their purposes

## 28.4 DACE Literature

- Two very good texts on computer experiments and surrogate modeling:
  1. The Design and Analysis of Computer Experiments, by Santner, Williams, and Notz (2018) is the canonical reference in the statistics literature
  2. Engineering Design via Surrogate Modeling by Forrester, Sobester, and Keane (2008) is perhaps more popular in engineering
    - We will analyze an example from the latter.

# 29 Aircraft Wing Weight Example

## 29.1 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:

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

## 29.2 AWWE Parameters (Part 1)

Table 29.1: 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
$\Lambda$	Quarter-chord sweep (deg)	0	-10	10
$q$	Dynamic pressure at cruise ( $lb/ft^2$ )	34	16	45
$\lambda$	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

## 29.3 Discussing the AWWE Parameters and Equations

- Baseline Cessna C172 Skyhawk Aircraft
- Effect of wing area and fuel weight on weight is obvious
- Aspect ratio  $A$ : the length of the wing divided by the average chord (thickness of the airfoil)
- Taper ratio  $\lambda$ : the ratio of the maximum to the minimum thickness of the airfoil (or the maximum to minimum chord).
- Note: Eq. is not a computer simulation, although we'll use it as one for the purposes of this illustration
  - Utilizing a true form, but treating it as unknown: helpful tool for synthesizing realistic settings in order to test methodology
  - That functional form was derived by “calibrating” known physical relationships to curves obtained from existing aircraft data (Raymer 2012)
  - It is in a sense itself a surrogate for actual measurements of the weight of aircrafts

## 29.4 Mathematical Properties of the AWWE Equation

- Although we won't presume to know that functional form in any of our analysis below, observe that the response is highly nonlinear in its inputs
- Typical trick: Apply the logarithm to simplify the equation with complicated exponents
  - But: even when modeling the logarithm, which turns powers into slope coefficients and products into sums, the response would still be nonlinear owing to the trigonometric terms
- Considering the nonlinearity and high input dimension, simple linear and quadratic response surface approximations will likely be insufficient

## 29.5 Goals: Understanding and Optimization

1. **Understanding:** The most straightforward might simply be to understand input-output relationships:
  - Given the global purview implied by that context, a fancier model is all but essential
  - For now, let us concentrate on that setting to fix ideas
2. **Optimization:** Another application might be optimization:
  - There might be interest in minimizing weight, but probably not without some constraints
  - Constraints: wings with nonzero area are needed if the airplane is going to fly

- A global perspective, and thus flexible modeling, is essential in (constrained) optimization settings

## 29.6 AWWE: Python Code

- Python code below serves as a genuine computer implementation “solving” a mathematical model
  - It takes arguments coded in the unit cube
  - Defaults are used to encode baseline settings from Table 29.1, also mapped to coded units:
  - To map values from the interval  $[a, b]$  to the interval  $[0, 1]$ , the following formula can be used:

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

- To revert this mapping, we can use:

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

## 29.7 AWWE: Python Code

```
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)
```

## 29.8 Properties of the Python “Solver”

- Compute time required by the `wingwt` “solver” is trivial
- Approximation error is minuscule—essentially machine precision
- Simulation of time consuming evaluation by adding a `sleep(3600)` command to synthesize a one-hour execution time

## 29.9 AWWE Visualization

- Plotting in 2d is lots easier than 9d: code below makes a grid in the unit square to facilitate sliced visuals
- We generate a `meshgrid` as follows:

```
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

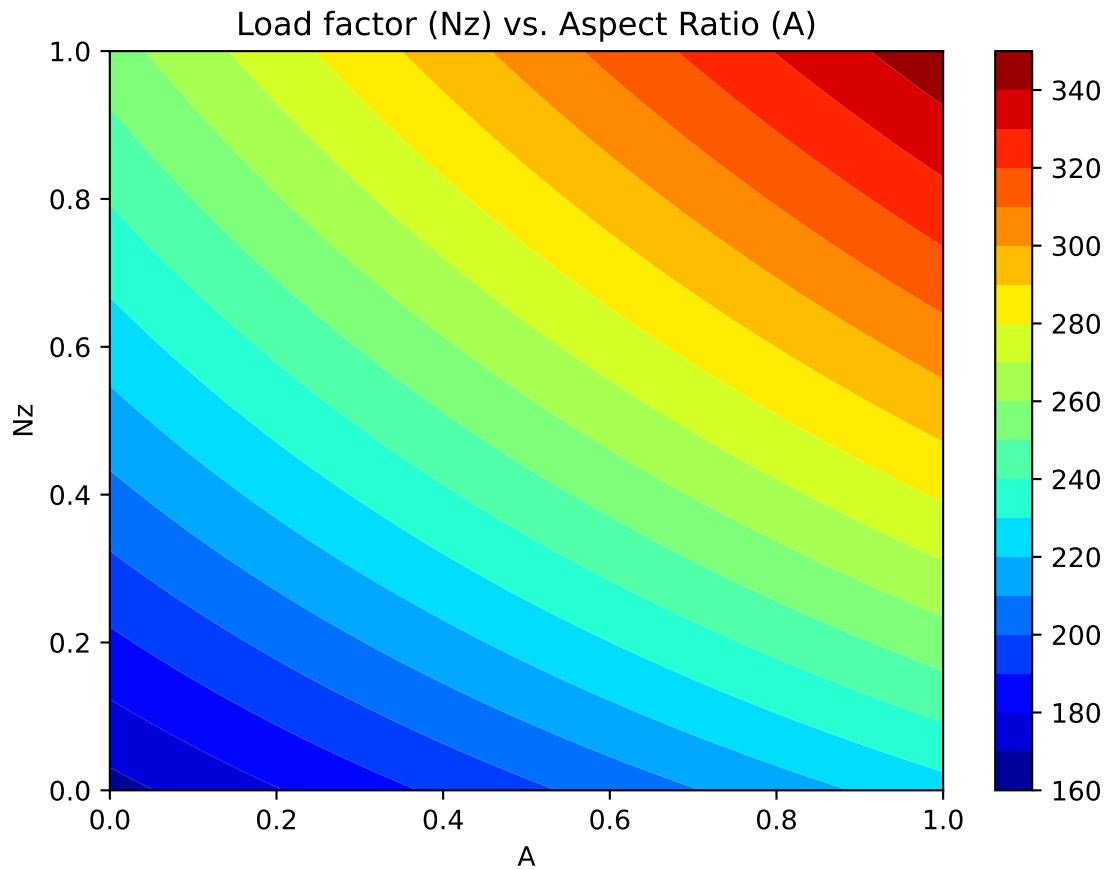
```
%matplotlib inline
import matplotlib.pyplot as plt
# plt.style.use('seaborn-white')
import numpy as np
x = np.linspace(0, 1, 100)
y = np.linspace(0, 1, 100)
X, Y = np.meshgrid(x, y)
```

## 29.10 Plot 1: Load Factor ( $N_z$ ) and Aspect Ratio ( $A$ )

- 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()
```

```
<matplotlib.colorbar.Colorbar at 0x1465fb490>
```



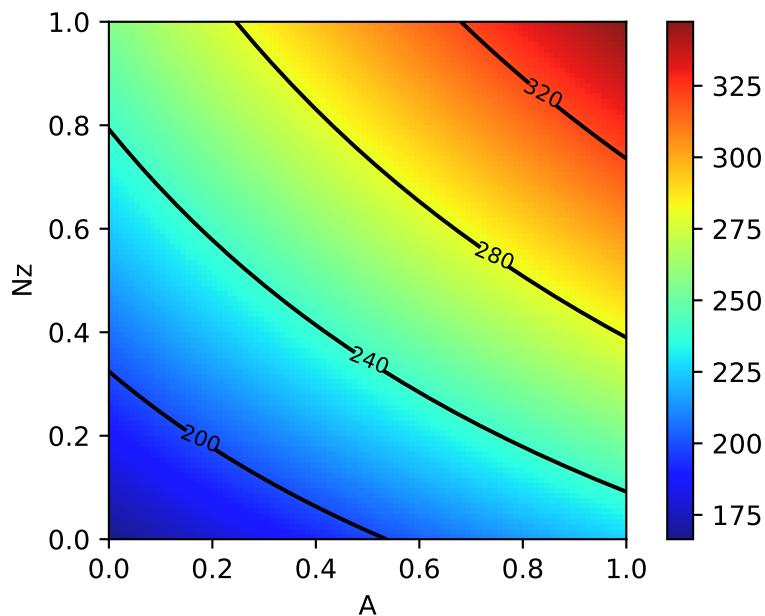
## 29.11 Variations of the Contour Plots

- 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()
```

<matplotlib.colorbar.Colorbar at 0x147a3b550>



## 29.12 Plot 1: Interpretation of the AWWE Plot

- Figure shows the weight response as a function of  $N_z$  and  $A$  with an image-contour plot
- Slight curvature in the contours indicates an interaction between these two variables

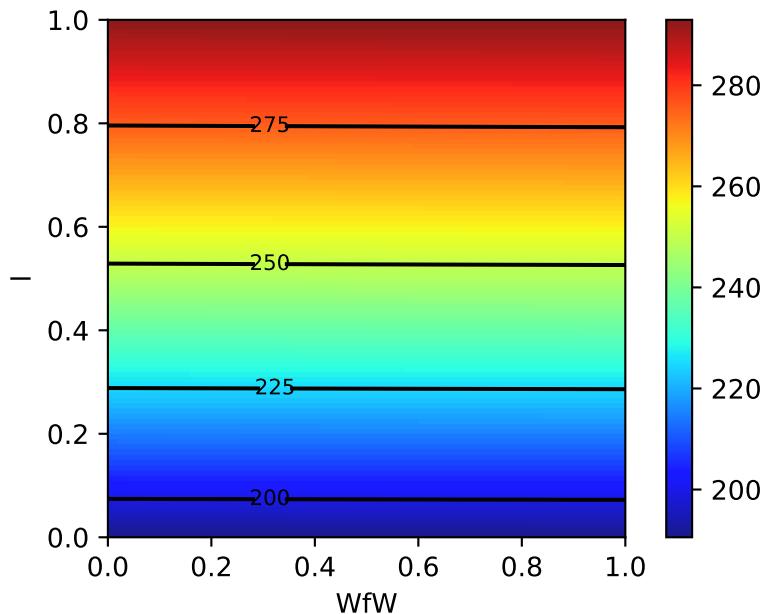
- Actually, this output range (160–320 approximately) nearly covers the entire span of outputs observed from settings of inputs in the full, 9d input space
- Apparently an aircraft wing is heavier when aspect ratios  $A$  are high
- Wings are designed to cope with large  $g$ -forces (large  $N_z$ ), with a compounding effect
- Perhaps this is because fighter jets cannot have efficient (light) glider-like wings

## 29.13 Plot 2: Taper Ratio and Fuel Weight

- The same experiment for two other inputs, e.g., taper ratio  $\lambda$  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("λ")

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



## 29.14 Plot 2: Interpretation of Taper Ratio ( $l$ ) and Fuel Weight ( $W_{fw}$ )

- Apparently, neither input has much effect on wing weight:
  - with  $\lambda$  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  $\lambda \times W_{fw}$

# 30 The Big Picture

## 30.1 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']`

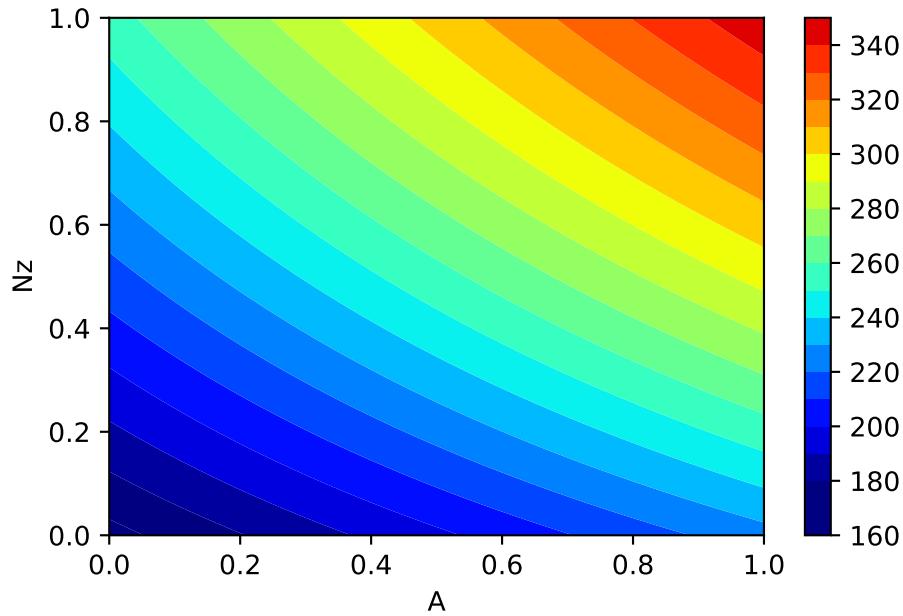
- To help interpret outputs from experiments such as this one—to level the playing field when comparing outputs from other pairs of inputs—code below sets up a color palette that can be re-used from one experiment to the next
  - We use the arguments `vmin=180` and `vmax =360` to implement comparability

```

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()

```

<matplotlib.colorbar.Colorbar at 0x147d434c0>



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

```

p = 11
Zlab[p]

```

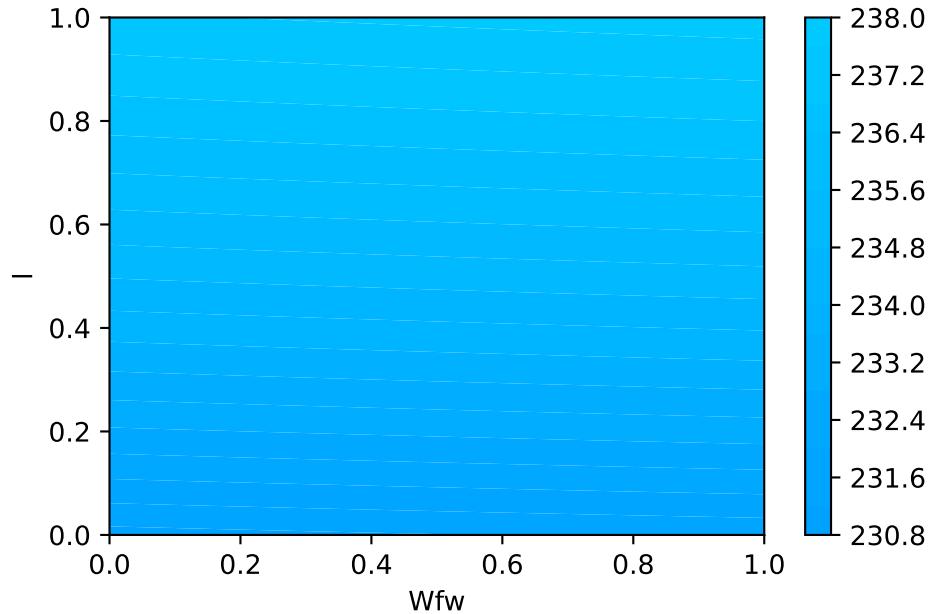
['Wfw', '1']

```

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()

```

```
<matplotlib.colorbar.Colorbar at 0x147df25f0>
```



- Using a global colormap indicates that these variables have minor effects on the wing weight.
- Important factors can be detected by visual inspection

## 30.2 Plotting the Big Picture

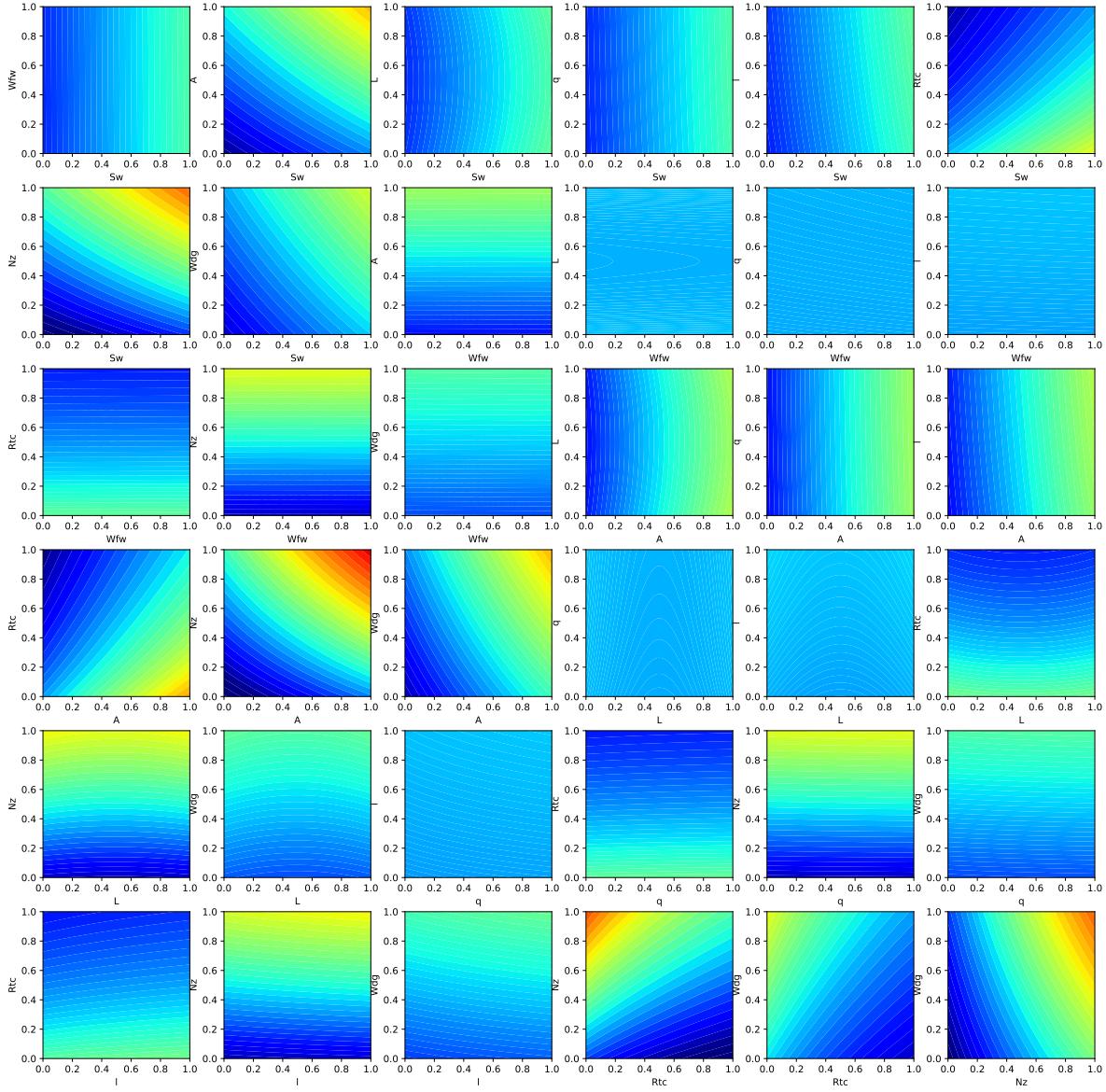
- We can plot all 36 combinations in one figure.

```
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()
```



### 30.3 AWWE Landscape

#### 30.3.1 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 reason 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 weight  $W_{dg}$ .
  4. Little impact: dynamic pressure  $q$ , taper ratio  $l$ , and quarter-chord sweep  $L$ .

### 30.3.2 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.

## 30.4 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

# 31 Exercise 1

## 31.0.1 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.

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

# 32 Lecture 05: Kriging

```
import numpy as np
from numpy import array, zeros, power, ones, exp, multiply, eye, linspace, mat, spacing, s
from numpy.linalg import cholesky, solve
from numpy.random import multivariate_normal
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)
```

## 32.1 Introduction

- GP regression: simple extension of linear modeling
- Many names and acronyms:
  - kriging, comes from geostatistics (Matheron 1963)
  - Gaussian spatial modeling or a Gaussian stochastic process
  - Machine learning (ML) researchers like Gaussian process regression (GPR)
- All of these instances are about regression:
  - training on inputs and outputs with the ultimate goal of prediction and uncertainty quantification (UQ)
- Warning:
  - GPs are no panacea
  - Specialized tools can work better in less generic contexts
  - And GPs have their limitations

# 33 Gaussian Process Basics

## 33.1 Gaussian Process Prior

- Any finite collection of realizations (i.e.,  $n$  observations) is modeled as having a multivariate normal (MVN) distribution
- Characteristics of those realizations are completely described by
  - their mean  $n$ -vector  $\mu$  and
  - their  $n \times n$  covariance matrix  $\Sigma$ .

## 33.2 Covariance

- Covariance function defined by inverse exponentiated squared Euclidean distance:
$$\Sigma(x, x') = \exp\{-||x - x'||^2\}.$$
- Covariance decays exponentially fast as  $x$  and  $x'$  become farther apart
- Observe that
  - $\Sigma(x, x) = 1$  and
  - $\Sigma(x, x') < 1$  for  $x \neq x'$
- The function  $\Sigma(x, x')$  must be positive definite.

## 33.3 Positive Definiteness

- Pos. definite: Covariance matrix  $\Sigma_n$ , based on evaluating  $\Sigma(x_i, x_j)$  at pairs of  $n$   $x$ -values  $x_1, \dots, x_n$ , we must have that

$$x^\top \Sigma_n x > 0 \quad \text{for all } x \neq 0.$$

- We intend to use  $\Sigma_n$  as a covariance matrix in an MVN, and a positive (semi-) definite covariance matrix is required for MVN analysis.
- Positive definiteness is the multivariate extension of requiring that a univariate Gaussian have positive variance parameter,  $\sigma^2$ .

### 33.4 Generating GP Random Functions

- GPs can be used to generate random data following a smooth functional relationship:
  - Take a set of  $x$ -values  $x_1, \dots, x_n$ ,
  - define  $\Sigma_n$  via  $\Sigma_n^{ij} = \Sigma(x_i, x_j)$ , for  $i, j = 1, \dots, n$ ,
  - then draw an  $n$ -variate realization  $Y \sim \mathcal{N}_n(0, \Sigma_n)$ ,
  - and plot the result in the  $x$ - $y$  plane

# 34 Example in 1 Dimension

## 34.1 Input Grid

- Use  $x$ -values in 1d
- First create an input grid with 100 elements

```
n = 100
X = np.linspace(0, 10, n, endpoint=False).reshape(-1, 1)
```

## 34.2 Covariance Matrix

- Build up covariance matrix  $\Sigma_n$  as inverse exponentiated squared Euclidean distances
- Notice: later we will augment the diagonal with a small number  $\text{eps} \equiv \epsilon$ 
  - Although inverse exponentiated distances guarantee a positive definite matrix in theory, sometimes in practice the matrix is numerically ill-conditioned
  - Augmenting the diagonal a tiny bit prevents ill-conditioned matrices
  - $\epsilon$  can be called the jitter in this context.

```
theta = np.array([1.0])
Psi = build_Psi(X, theta)
```

## 34.3 Multivariate Normal Distribution

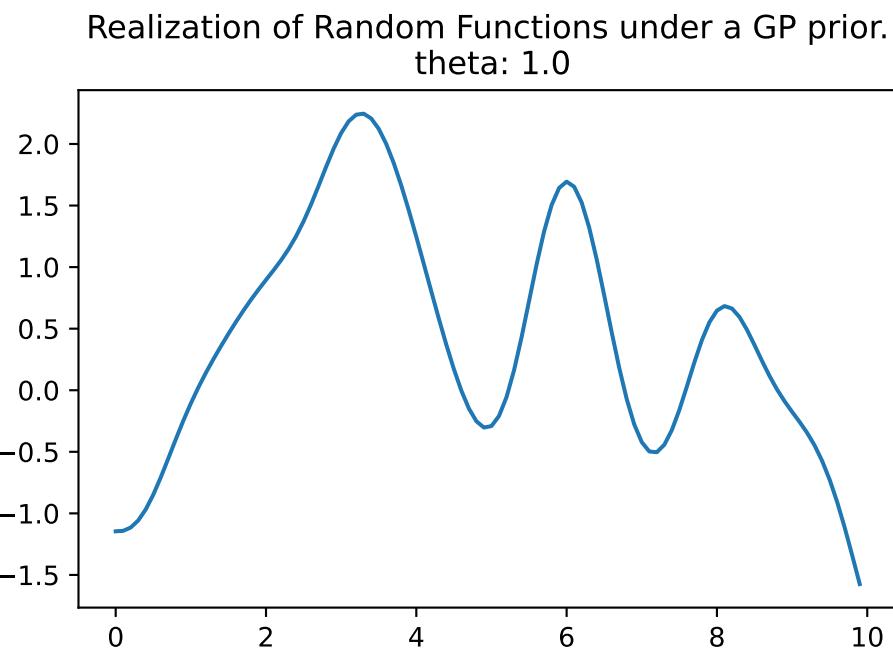
- Plug that covariance matrix into an MVN random generator
- The function `multivariate_normal` provides a random number generator for the multivariate normal distribution with mean equal to `mean` and covariance matrix `Psi`.
- The argument `size` denotes the number of realizations, which is in our case one.
- The default value of the mean vector is zero.
  - We will use the argument `zeros(n)`, where `n` is the number of samples (identical to the number of rows in the `X` and `Psi` matrices).

```
Y = multivariate_normal(zeros(Psi.shape[0]), Psi, size = 1).reshape(-1,1)
```

## 34.4 Plot the Results

- Finite realization of a random function under a GP prior with a particular covariance structure.
- 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.\ntheta: {}".format(theta[0]))
plt.show()
```



# 35 Properties of the 1d Example

## 35.1 Several Bumps

- Observe several bumps in the  $x$ -range of  $[0, 10]$  because short distances are highly correlated and long distances are essentially uncorrelated.

## 35.2 Smooth Look

- Function plotted above is only a finite realization, meaning that we really only have 100 pairs of points.
- Those points look smooth, in a tactile sense, because they're close together and because the plot function can be used for connecting the dots with lines.
- The full surface, which you might conceptually extend to an infinite realization over a compact domain, is extremely smooth in a calculus sense because the covariance function is infinitely differentiable.

## 35.3 Scale of Two

- $Y$  values have range of about  $[-2, 2]$ , with 95% probability, because the scale of the covariance is 1, ignoring the jitter  $\epsilon$  added to the diagonal
- 95% lie within two standard deviations of the mean

## 35.4 Background: Expectation, Mean

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

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

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

- The mean,  $\mu$ , 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.

## 35.5 Example

- Let  $X$  denote the number produced by rolling a fair die. Then

$$E(X) = 1 \times 1/6 + 2 \times 1/6 + 3 \times 1/6 + 4 \times 1/6 + 5 \times 1/6 + 6 \times 1/6 = 3.5$$

## 35.6 Sample Mean

- The sample mean is an important estimate of the population mean.
- The sample mean of a sample  $\{x_i\}$  ( $i = 1, 2, \dots, n$ ) is defined as

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

## 35.7 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.

## 35.8 Variance

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

$$Var(X) = E[(X - \mu)^2]. \quad (35.1)$$

## 35.9 Standard Deviation

- Taking the square root of the variance to get back to the same scale of units as  $X$  gives the standard deviation.
- The standard deviation of  $X$  is the square root of the variance of  $X$ .

$$sd(X) = \sqrt{Var(X)}. \quad (35.2)$$

## 35.10 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 standard deviation is computed for the flattened array by default, otherwise over the specified axis.
  - 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

$$\sqrt{\frac{1}{N} \sum_i (x_i - \bar{x})^2} \quad \text{with} \quad \bar{x} = \sum_{i=1}^N x_i / N. \quad (35.3)$$

- If no `axis` is specified, `std` uses the flattened array (which is the default.)
- 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
```

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

```
array([0., 0., 0.])
```

- 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

np.std(a, axis = 1)

array([0.81649658])

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])
```

### 35.10.1 Single Versus Double Precision

- In single precision, `std()` can be inaccurate:

```
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. ],
       [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)

np.std(a)

```

0.45000002

- Computing the standard deviation in float64 is more accurate (result may vary):

```
np.std(a, dtype=np.float64)
```

0.44999999925494194

## 35.11 Visualization of the Standard Deviation

- 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.

```

import matplotlib
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(1)

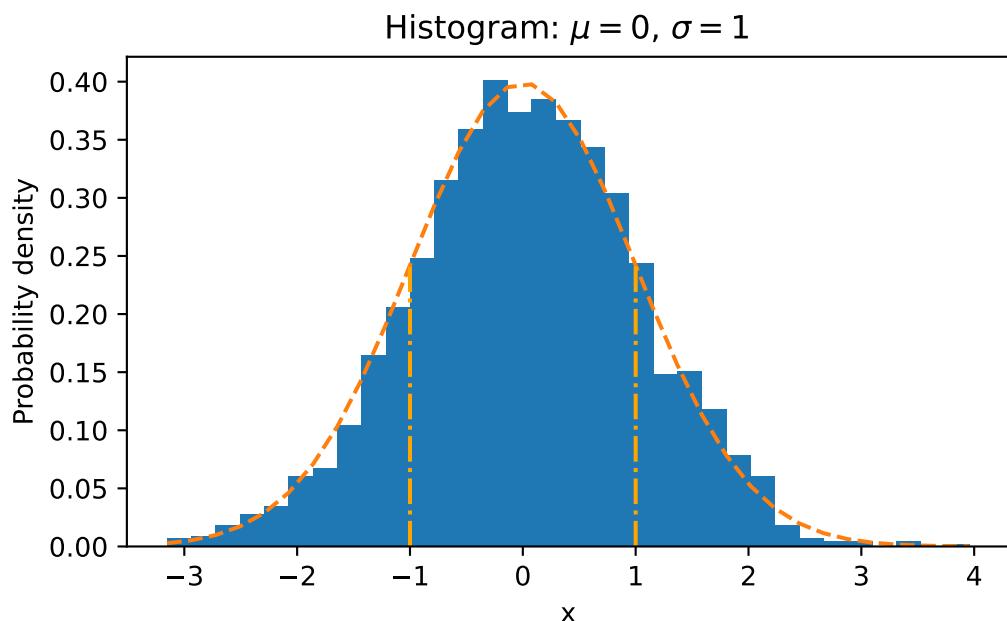
# example data
mu = 0.0 # mean of distribution
sigma = 1 # standard deviation of distribution

```

```

x = mu + sigma * np.random.randn(2000)
num_bins = 33
fig, ax = plt.subplots()
# the histogram of the data
n, bins, patches = ax.hist(x, num_bins, density=1)
# add a 'best fit' line
y = ((1 / (np.sqrt(2 * np.pi) * sigma)) *
      np.exp(-0.5 * (1 / sigma * (bins - mu))**2))
ax.plot(bins, y, '--')
ax.set_xlabel('x')
ax.set_ylabel('Probability density')
ax.set_title(r'Histogram: $\mu=0$, $\sigma=1$')
ax.vlines(-1, ymin=0, ymax = ((1 / (np.sqrt(2 * np.pi) * sigma)) *
      np.exp(-0.5 * (1 / sigma * (-1.0 - mu))**2)), colors="orange", linestyles="-.")
ax.vlines(1, ymin=0, ymax = ((1 / (np.sqrt(2 * np.pi) * sigma)) *
      np.exp(-0.5 * (1 / sigma * (1.0 - mu))**2)), colors="orange", linestyles="-.")
# Tweak spacing to prevent clipping of ylabel
fig.tight_layout()
plt.show()

```



## 35.12 Standardization

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

## 35.13 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.
- To generate ten random numbers from a normal distribution, the following command can be used.

```
import numpy as np
rng = np.random.default_rng()
n = 10
mu, sigma = 2, 0.1
x = rng.normal(mu, sigma, n)
```

- Verify the mean:

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

0.019345510040265967

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

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

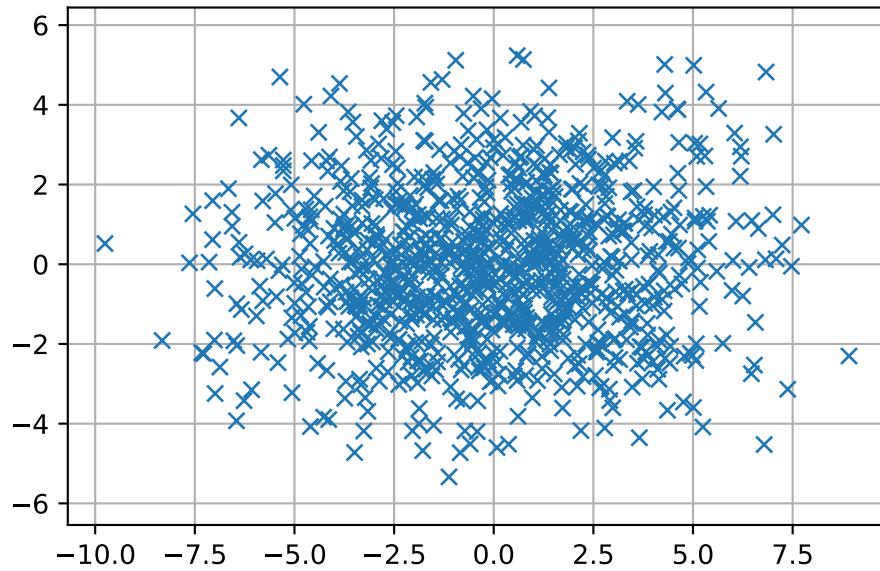
0.01483357389323213

## 35.14 The Multivariate Normal Distribution

- The multivariate normal, multinormal, or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions.
- Such a distribution is specified by its mean and covariance matrix.
  - These parameters are analogous to the mean (average or “center”) and variance (squared standard deviation or “width”) of the one-dimensional normal distribution.
- The mean is a coordinate in  $n$ -dimensional space, which represents the location where samples are most likely to be generated.
  - This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.
- The covariance indicates the level to which two variables vary together.
  - From the multivariate normal distribution, we draw  $n$ -dimensional samples,  $X = [x_1, x_2, \dots, x_n]$ .
- The covariance matrix element  $C_{ij}$  is the covariance of  $x_i$  and  $x_j$ .

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

Bivariate Normal. Mean zero and covariance:  $[[9, 0], [0, 4]]$



## 35.15 Kriging

### 35.15.1 The Kriging Covariance Matrix

- Basis functions of the form

$$\psi^{(i)} = \exp\left(-\sum_{l=1}^k \theta_l |x_l^{(i)} - x_l|^{p_l}\right) \quad (35.4)$$

are used in a method known as Kriging.

- Although the Kriging basis function is related to the Gaussian basis function, there are some differences:
  - Where the Gaussian basis function has  $1/\sigma^2$ , the Kriging basis has a vector  $\theta = [\theta_1, \theta_2, \dots, \theta_k]^T$ .
  - The  $\theta$  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 to vary (typically from 1 to 2).

### 35.15.2 Building the Kriging Model

- Consider a set of  $k$ -dimensional sample data

$$X = [x_1, x_2, \dots, x_n],$$

i.e.,  $X$  is a  $(n, k)$ -matrix, with related responses

$$y = [y_1, y_2, \dots, y_n].$$

- We are interested in finding an expression for the predicted value at a new point  $x$ .
- We are going to interpret the observed responses as if they are generated by a stochastic process, which will be denoted as follows:

$$Y = [Y(x_1), \dots, Y(x_n)]^T. \quad (35.5)$$

- The random process has a mean of  $\mu$  (which is a  $(n, 1)$ -dimensional vector). The random variables are correlated with each other using the basis function expression

$$\text{cor}[Y(x_i), Y(x_j)] = \exp\left(-\sum_{l=1}^k \theta_l |x_{il} - x_{jl}|^{p_l}\right). \quad (35.6)$$

- From this, the  $(n, n)$  correlation matrix  $\Psi$  of the observed data can be computed:

$$\Psi = \begin{pmatrix} \text{cor}[Y(x_1), Y(x_1)] & \dots & \text{cor}[Y(x_1), Y(x_n)] \\ \vdots & \ddots & \vdots \\ \text{cor}[Y(x_n), Y(x_1)] & \dots & \text{cor}[Y(x_n), Y(x_n)] \end{pmatrix}. \quad (35.7)$$

### 35.15.3 Example: Correlation Matrix

- 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  $\Psi$ , the following  $k$   $(n, n)$ -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  $\theta_i$ , 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:

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

- Finally,

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

is computed.

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

```
from numpy import array, zeros, power, ones, exp, multiply, eye, linspace, mat, spacing, s
from numpy.linalg import cholesky, solve
theta = np.array([1,2,3])
X = np.array([[1,0,0], [0,1,0], [0,0,1], [0,0,0]])
X

array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1],
       [0, 0, 0]])

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, 0.01831564, 0.36787944],
       [0.04978707, 1.        , 0.00673795, 0.13533528],
       [0.01831564, 0.00673795, 1.        , 0.04978707],
       [0.36787944, 0.13533528, 0.04978707, 1.        ]])
```

- The same result can be obtained with existing python functions, e.g., from the package `scipy`.
  - Note: A small value, `eps`, can be passed to the function `build_Psi` to improve the condition number.

```
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))) +  multipl

Psi = build_Psi(X, theta, eps=.0)
Psi
```

```
array([[1.        , 0.04978707, 0.01831564, 0.36787944],
       [0.04978707, 1.        , 0.00673795, 0.13533528],
       [0.01831564, 0.00673795, 1.        , 0.04978707],
       [0.36787944, 0.13533528, 0.04978707, 1.        ]])
```

### 35.15.4 Generate sample from $\mathbf{N}(\mu, \Psi)$

- The following block of code generates 5 draws from a bivariate normal distribution with zero mean and covariance matrix  $\Psi$ .
- Because our input space is three dimensional, the results cannot be visualized in a simple way.
- The next examples will use a one-dimensional input space, so that our results can be easily visualized in 2d.

```
import numpy as np
from numpy.random import multivariate_normal
m = 5
rng = np.random.default_rng()
Y = multivariate_normal(zeros(Psi.shape[0]), Psi, size=m)
```

# 36 Kriging in a Nutshell

- We will use the Kriging correlation  $\Psi$  to predict new values based on the observed data.
- The matrix algebra involved in 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 Forr08a and Bart21i.

## 36.0.1 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  $\vec{\mu}$ , which is a  $(n \times 1)$  vector.

## 36.0.2 Correlations

- The random vectors are correlated with each other using the basis function expression

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

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

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

- Note: correlations depend on the absolute distances between sample points  $|x_j^{(n)} - x_j^{(n)}|$  and the parameters  $p_j$  and  $\theta_j$ .
- Correlation is intuitive, because when two points move close together, then  $|x_l^{(i)} - x_l| \rightarrow 0$  and  $\exp(-|x_l^{(i)} - x_l|) \rightarrow 1$ , points show very close correlation and  $Y(x_l^{(i)}) = Y(x_l)$ .
- $\theta$  can be seen as a width parameter:
  - low  $\theta_j$  means that all points will have a high correlation, with  $Y(x_j)$  being similar across the sample
  - high  $\theta_j$  means that there is a significant difference between the  $Y(x_j)$ 's
- $\theta_j$  is a measure of how active the function we are approximating is
- High  $\theta_j$  indicate important parameters

### 36.0.3 MLE to estimate $\theta$ and $p$

- We know what the correlations mean, but how do we estimate the values of  $\theta_j$  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

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

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  $\mu$  and  $\sigma$  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:

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

- Note:
  - The first term 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.

## 36.1 Tuning $\theta$ and $p$

- Optimizers such as Nelder-Mead, Conjugate Gradient, or Simulated Annealing can be used to determine optimal values for  $\theta$  and  $p$ .
- After the optimization, the correlation matrix is build with the optimized  $\theta$  and  $p$  values. This is best (most likely) Kriging model for the given data  $y$ .
- We will skip the optimization step here and use  $\theta = 1$  and  $p = 2$ .

## 36.2 Kriging Prediction

- Main idea for prediction:
  - The new  $Y(x)$  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  $\vec{\tilde{y}} = (\vec{y}^T, \hat{y})^T$ .
- A vector of correlations between the observed data and the new prediction is defined as

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

- The augmented correlation matrix is constructed as

$$\vec{\tilde{\Psi}} = \begin{pmatrix} \vec{\Psi} & \vec{\psi} \\ \vec{\psi}^T & 1 \end{pmatrix}.$$

- The log-likelihood of the augmented data is

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

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

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

### 36.2.1 Properties of the Predictor

- This Equation reveals two important properties of the Kriging predictor:
  - **Basis functions:** The basis function impacts the vector  $\vec{\psi}$ , which contains the  $n$  correlations between the new point  $\vec{x}$  and the observed locations. Values from the  $n$  basis functions are added to a mean base term  $\mu$  with weightings  $\vec{w} = \vec{\Psi}^{(-1)}(\vec{y} - \vec{1}\hat{\mu})$ .
  - **Interpolation:** The predictions interpolate the sample data. When calculating the prediction at the  $i$ th sample point,  $\vec{x}^{(i)}$ , the  $i$ th column of  $\vec{\Psi}^{-1}$  is  $\vec{\psi}$ , and  $\vec{\psi}\vec{\Psi}^{-1}$  is the  $i$ th unit vector. Hence,  $\hat{y}(\vec{x}^{(i)}) = y^{(i)}$ .

## 36.3 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.

## 36.4 Calculating the Correlation Matrix $\Psi$

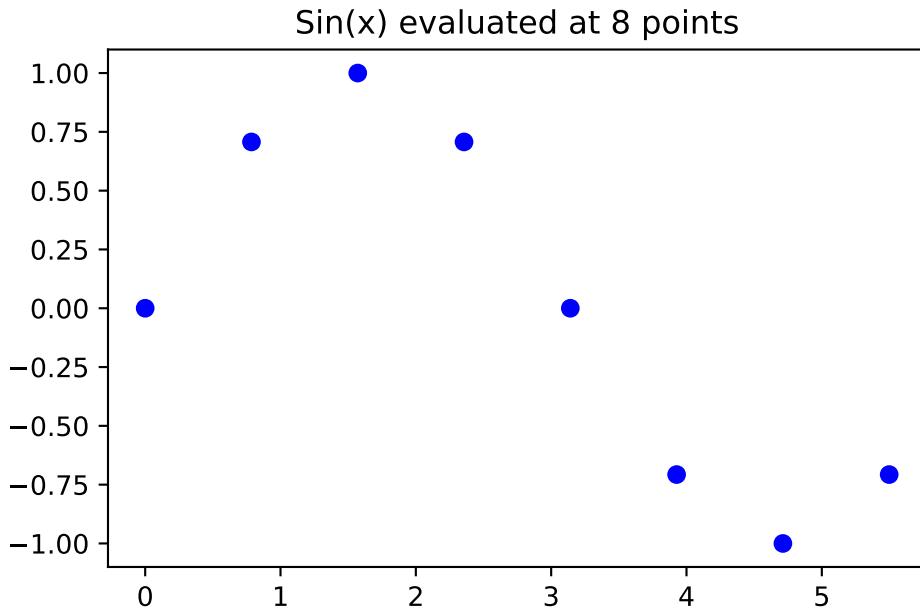
- The correlation matrix  $\Psi$  is based on the pairwise squared distances between the input locations
- Here we will use  $n = 8$  sample locations
- $\theta$  is set to 1.0

```
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("Sin(x) evaluated at {} points".format(n))
plt.show()
```



## 36.5 Computing the $\psi$ Vector

### 36.5.1 Based on Distances Between Testing and Training Data Locations

- 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='sqrEuclidean',
               out=None,
               w=theta)
    print(D.shape)
    psi = exp(-D)
```

```
# return psi transpose to be consistent with the literature
return(psi.T)
```

- We would like to predict at  $m = 100$  new locations:

```
m = 100
x = np.linspace(0, 2*np.pi, m, endpoint=False).reshape(-1,1)
psi = build_psi(X, x, theta)
```

```
(100, 8)
```

## 36.6 Predictive Equations

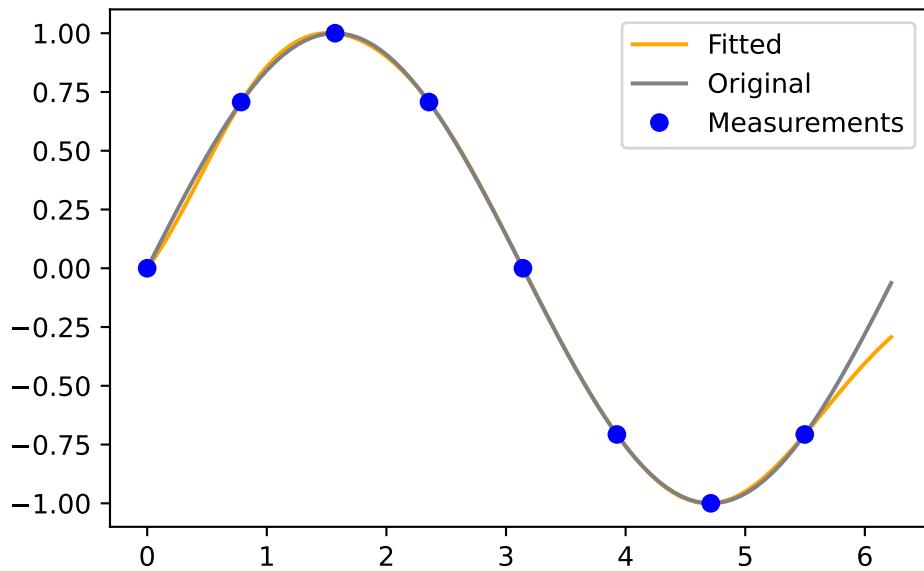
```
U = cholesky(Psi).T

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)))

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.\ntheta: {}".format(n, theta[0]))
plt.legend(loc='upper right')
plt.show()
```

Kriging prediction of  $\sin(x)$  with 8 points.  
theta: 1.0



# 37 Exercises

## 37.1 1 Number of Sample Points

- The example uses  $n = 8$  sample points to fit the sin function.
  - What happens, if less than 8 samples are available?

## 37.2 2 Modified $\theta$ values

- The example uses a  $\theta$  value of 1.0.
  - What happens if  $\theta$  is modified?
  - Can get better predictions with smaller or larger  $\theta$  values?

## 37.3 3 Prediction Interval

- The prediction interval was identical to the measurement interval, i.e., in the range from 0 to  $2\pi$ . This is referred to as “interpolation”.
  - What happens if this interval is increased (which is referred to as “extrapolation”)?

## 38 DOE 1

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels
```

In this article I want to explore the topic of Design of Experiments. First of all we should ask ourselves how do we know more about a system?

The only way to know more or learn something about a system is to disturb it and then observe it. This is the basis of machine learning actually.

Ideally you want your algorithm to learn patterns in your data after all possible disturbances you may have, and with all variables you can find for your experiments.

But if you are to design a new experiment to learn from your system you probably can't afford to wait one year to collect enough data or explore all possible variables combinations in a random manner. There is a scientific way to do it which is proved to give you the best result. This approach or system is called Design of Experiment.

Imagine you want to understand which configuration is better for boiling water, with some parameters which you are free to control:

- 1) Heater Settings (Low/High)
- 2) Pot Type (Metal/Glass)
- 3) LID (with/without)
- 4) Heater Kind (gas/electric)

this very simple system, of which you may already know the answer, has 4 parameters, each with 2 possible states. So the total number of experiments you can do to explore all of them is  $2^4 = 16$  experiments.

But what I mean with “better” ? We have to define the objective which might be:

- A) the faster way for boiling water -> I want to minimize it
- B) the cheaper way for boiling water -> I want to minimize the cost

The outcome of our experiment is therefore either PRICE or TIME.

I said “parameters” above: Parameters, variables, factors, features in data science / statistics / research there are a lot of synonyms. They can be numerical or categorical. We will explore a simple basic example doing the calculation by hand and getting a bit of help with python to display some data, and then we will try to do the same in a more automatic way in a more complicated system. In the second part we will try to apply some powerful statistical tools to explore our data and make the first regressions, starting with OLS and then trying to explore more of our data with PCA/PCR and PLS. Finally we will try to apply some machine learning algorithm to the very same data.

## 38.1 Two factors experiment

```
Light = ['On','Off']
Watering = ['Low','High']
# create combinations for all parameters
experiments = [(x,y) for x in Light for y in Watering]
exp_df = pd.DataFrame(experiments,columns=['A','B'])

exp_df

# map the variable and encode them to ±1
from sklearn.preprocessing import OrdinalEncoder
enc = OrdinalEncoder(categories=[['On','Off'],['Low','High']])

encoded_df = pd.DataFrame(enc.fit_transform(exp_df[['A','B']]),columns=['A','B'])
#define the experiments order which must be random
encoded_df['exp_order'] = np.random.choice(np.arange(4),4,replace=False)
encoded_df['outcome'] = [25,37,55,65]
encoded_df

plt.scatter(encoded_df['A'],encoded_df['B'], marker='o', s=80, c=encoded_df['outcome'], cm
plt.xlabel('A')
plt.ylabel('B')
for i, txt in enumerate(encoded_df['outcome']):
    plt.annotate(txt, (encoded_df['A'].iloc[i], encoded_df['B'].iloc[i]), xytext=(encoded_d

average_response = encoded_df['outcome'].mean()
```

```

def Pars_effect(data = encoded_df, par='A', effect='B'):
    data_1 = data[data[par] == data[par].min()]
    data_2 = data[data[par] == data[par].max()]

    eff_1 = (data_1[data_1[effect] == data_1[effect].max()].outcome.max() - data_1[data_1[effect].max()])
    eff_2 = (data_2[data_2[effect] == data_2[effect].max()].outcome.max() - data_2[data_2[effect].max()])
    return (eff_2+eff_1)/2

def predict_outcome(baseline, A, Ax, B, Bx):
    if A == 0:
        A = -1
    if B == 0:
        B = -1
    print(f'y ~ {baseline} + 1/2 {Ax}*A + 1/2 {Bx}*B')
    pred_y = baseline + 1/2*Ax*A + 1/2*Bx*B
    print(f'{pred_y}')
    return pred_y

Ax = Pars_effect(data = encoded_df, par='B', effect='A')
Bx = Pars_effect(data = encoded_df, par='A', effect='B')

y_pred_11 = predict_outcome(average_response, 1, Ax, 1, Bx)

y_pred_00 = predict_outcome(average_response, 0, Ax, 0, Bx)

average_response

import statsmodels.api as sm
import statsmodels.formula.api as smf

results = smf.ols('outcome ~ A + B', data=encoded_df).fit()

print(results.summary())

sns.lineplot(x='A', y='outcome', hue='B', data=encoded_df)

encoded_df

```

```

A = np.matrix(encoded_df[['A','B']])
A = np.c_[ A, np.ones(len(A)) ]
b = np.array(encoded_df['outcome'])
print(np.linalg.lstsq(A,b, rcond=None))
print((np.linalg.inv(A.T*A)*A.T)*b.reshape(-1,1))

encoded_df['A'].replace(0, -1, inplace=True)
encoded_df['B'].replace(0, -1, inplace=True)

encoded_df

A = np.matrix(encoded_df[['A','B']])
A = np.c_[ A, np.ones(len(A)) ]
b = np.array(encoded_df['outcome'])
print(np.linalg.lstsq(A,b, rcond=None))
print((np.linalg.inv(A.T*A)*A.T)*b.reshape(-1,1))

```

## 38.2 Negative Interaction

```

encoded_df_int = pd.DataFrame(enc.fit_transform(exp_df[['A','B']]),columns=['A','B'])
encoded_df_int['exp_order'] = np.random.choice(np.arange(4),4,replace=False)
encoded_df_int['outcome'] = [21,23,25,44]
average_int_resp = encoded_df_int['outcome'].mean()
Ax = Pars_effect(data = encoded_df_int, par='B', effect='A')
Bx = Pars_effect(data = encoded_df_int, par='A', effect='B')
y_pred_11 = predict_outcome(average_response,1,Ax,1,Bx)
y_pred_00 = predict_outcome(average_response,0,Ax,0,Bx)

results = smf.ols('outcome ~ A + B', data=encoded_df_int).fit()
print(results.summary())

A = np.matrix(encoded_df_int[['A','B']])
A = np.c_[np.ones(len(A)), A ]
b = np.array(encoded_df_int['outcome'])
print(np.linalg.lstsq(A,b, rcond=None))

```

```

# print((np.linalg.inv(A.T*A)*A.T)*b.reshape(-1,1))
# print((np.linalg.inv(A.T*A)*A.T)*b.reshape(-1,1))
encoded_df_int['A*B'] = encoded_df_int['A']*encoded_df_int['B']
encoded_df_int

A = np.matrix(encoded_df_int[['A','B','A*B']])
A = np.c_[np.ones(len(A)), A]
b = np.array(encoded_df_int['outcome'])
print(np.linalg.lstsq(A,b, rcond=None,)[:2])

results = smf.ols('outcome ~ A + B + A*B', data=encoded_df_int).fit()
print(results.summary())

results = smf.ols('outcome ~ A*B', data=encoded_df_int).fit()
print(results.summary())

plt.scatter(encoded_df['A'],encoded_df['B'], marker='o', s=80, c=encoded_df['outcome'], cm
plt.xlabel('A')
plt.ylabel('B')
for i, txt in enumerate(encoded_df['outcome']):
    plt.annotate(txt, (encoded_df['A'].iloc[i], encoded_df['B'].iloc[i]), xytext=(encoded_d
    )

sns.lineplot(x='A',y='outcome',hue='B',data=encoded_df_int)

def Pars_effect_interaction(data = encoded_df, par='A', effect='B', verbose=True):
    data_1 = data[data[par] == data[par].min()].copy()
    data_2 = data[data[par] == data[par].max()].copy()
    if verbose:
        display(data_1)
        display(data_2)

    eff_1 = (data_1[data_1[effect] == data_1[effect].max()].outcome.max() - data_1[data_1[e
    eff_2 = (data_2[data_2[effect] == data_2[effect].max()].outcome.max() - data_2[data_2[e

    return (max(eff_1,eff_2)-min(eff_1,eff_2))/2

def predict_outcome_int(baseline, A, Ax, B, Bx, *args):
    if A == 0:

```

```

        A = -1
if B == 0:
    B = -1

if args:
    ABx = args[0]
    print(f'y ~ {baseline}+1/2 {Ax}*A + 1/2 {Bx}*B + 1/2 {ABx} *A*B')
else:
    ABx=0
    print(f'y ~ {baseline}+1/2 {Ax}*A + 1/2 {Bx}*B')

pred_y = baseline + 1/2*Ax*A + 1/2*Bx*B +1/2 *A*B*ABx
print(f'{pred_y}')
return pred_y

max(2,5)

Ax_int = Pars_effect_interaction(data = encoded_df_int, par='B', effect='A', verbose=False)
Bx_int = Pars_effect_interaction(data = encoded_df_int, par='A', effect='B', verbose=True)
Ax_int,Bx_int

encoded_df_int

predict_outcome_int(average_int_resp,1,Ax,1,Bx,Ax_int)

predict_outcome_int(average_int_resp,0,Ax,0,Bx,Ax_int)

df2 = pd.DataFrame(enc.fit_transform(exp_df[['A','B']]),columns=['A','B'])
df2['exp_order'] = np.random.choice(np.arange(4),4,replace=False)
df2['outcome'] = [20,23.5,25.5,44.5]
df2['A*B'] = df2['A']*df2['B']
df2

df2[df2.A!=df2.B]

df_all = pd.concat([encoded_df_int,df2[df2.A!=df2.B]]).drop('exp_order',1)
df_all

```

```
sns.lineplot(x='A',y='outcome',hue='B',data=df_all)

results = smf.ols('outcome ~ A*B', data=df_all).fit()
print(results.summary())

df_all = pd.concat([encoded_df_int,df2[df2.A==df2.B]]).drop('exp_order',1)
results = smf.ols('outcome ~ A*B', data=df_all).fit()
print(results.summary())
```

## 39 DOE 2

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline

def gaussian_function(x, y, pars=None, add_noise=False):
    # to be changed: generate all data here
    np.random.seed(2)
    if not pars:
        x0 = np.random.randn(1)[0]+0.5
        y0 = x0+0.5
        fwhm = np.random.normal(loc=1.5,scale=3, size=100).mean()
    else:
        x0, y0, fwhm = pars

    func = 10*np.exp(-4*np.log(2) * ((x-x0)**2 + (y-y0)**2) / fwhm**2)

    if add_noise:
        func = add_noise_func(func)

    return func, x0, y0, fwhm

def add_noise_func(input_data, mean_noise=0):
    # incomplete: refactor code to generate data + noise in one function
    avg_data = np.mean(input_data)
    noise_ = np.random.normal(mean_noise, np.sqrt(avg_data), 1)[0]
    output_data = input_data + noise_

    return output_data
```

```

def add_final_points(temp_df):
    mean_x, mean_y, range_x, range_y = normal_to_coded_var(temp_df, output_pars = True)

    temp_df.loc[len(temp_df)] = [mean_x+range_x,mean_y]
    temp_df.loc[len(temp_df)] = [mean_x-range_x,mean_y]
    temp_df.loc[len(temp_df)] = [mean_x,mean_y+range_y]
    temp_df.loc[len(temp_df)] = [mean_x,mean_y-range_y]

    return temp_df

def output_model(x_d1, y_d1, model=None, pars=None, cnt_point=True, final_stage=False) ->
    # inputs 2 arrays of coordinates, outputs a full factorial
    temp_df = pd.DataFrame([(i,j) for i in x_d1 for j in y_d1], columns=['x','y'])

    if cnt_point:
        temp_df.loc[len(temp_df)] = [temp_df['x'].mean(),temp_df['y'].mean()]

    if final_stage:
        temp_df = add_final_points(temp_df)

    temp_df = normal_to_coded_var(temp_df)
    temp_df['outcome'] = temp_df.apply(lambda x: gaussian_function(x['x'], x['y'], pars)[0])

    if model:
        temp_df['predict'] = model.predict(temp_df[['x','y']])
        temp_df['mape'] = 100*(temp_df['outcome'] - temp_df['predict'])/temp_df['outcome']

    return temp_df

def normal_to_coded_var(input_df, output_pars = False):
    # mapping between original and coded variables
    temp_df = input_df.copy()
    mean_x = temp_df['x'].mean()
    mean_y = temp_df['y'].mean()
    range_x = temp_df['x'].max()-temp_df['x'].min()
    range_y = temp_df['y'].max()-temp_df['y'].min()

    temp_df['X'] = (temp_df['x']-mean_x)/(range_x/2)
    temp_df['Y'] = (temp_df['y']-mean_y)/(range_y/2)

    if output_pars:

```

```

        return mean_x, mean_y, range_x, range_y

    return temp_df

def coded_to_normal(pars, x_coded, y_coded):
    # mapping between coded and normal variables
    mean_x, mean_y, range_x, range_y = pars

    real_x = x_coded*(range_x/2) + mean_x
    real_y = y_coded*(range_y/2) + mean_y

    return real_x, real_y

def plot_contour_fromdf(test_df, fill_value=0, model=None, levels=10, plot_normal_vars=True):
    # surface plots
    # if there's no model try to plot a contour plot given the dataframe
    # otherwise use the model to predict values from min to max value of each dimension
    # x, y = 'cX', 'cY'
    x, y = 'x', 'y'

    if not model:
        hdf = test_df.groupby(input_vars).mean()
        hdfreset = hdf.reset_index()
        hdfreset.columns = [x, y, 'outcome']
        hdfpivot= hdfreset.pivot(x, y).fillna(0)

        X=hdfpivot.columns.levels[1].values
        Y=hdfpivot.index.values
        Xi,Yi = np.meshgrid(X, Y)
        Z=hdfpivot.values
    else:
        X = np.linspace(test_df[x].min(),test_df[x].max(),levels)
        Y = np.linspace(test_df[y].min(),test_df[y].max(),levels)
        Xi,Yi = np.meshgrid(X, Y)
        Z = model.predict(np.c_[Xi.ravel(),Yi.ravel()]).reshape(len(X),len(Y))

    if plot_normal_vars:
        temp_df = pd.DataFrame()
        temp_df[x] = Xi.ravel()
        temp_df[y] = Yi.ravel()

```

```

Z = model.predict(temp_df).reshape(len(X),len(Y))
mean_x, mean_y, range_x, range_y = normal_to_coded_var(test_df, output_pars = True)
temp_df['x'] = temp_df[x]*(range_x/2) + mean_x
temp_df['y'] = temp_df[y]*(range_y/2) + mean_y
X = np.linspace(test_df['x'].min(),test_df['x'].max(),levels)
Y = np.linspace(test_df['y'].min(),test_df['y'].max(),levels)

Xi,Yi = np.meshgrid(X, Y)
else:
    mean_x, mean_y = 0, 0

# gradients given Z to plot quiverplot
v, u = np.gradient(Z, 0.1, 0.1)
plt.contourf(Xi, Yi, Z, alpha=0.7)
plt.colorbar()
plt.quiver(Xi,Yi,u,v);

def linear_pipe_fit(degree=2, interaction=True, dataframe=pd.DataFrame(), input_vars=['x', 'y']):
    lr = LinearRegression()
    pol = PolynomialFeatures(degree=degree, interaction_only=interaction, include_bias=False)
    pipe = Pipeline([('pol', pol), ('lr', lr)])
    pipe.fit(dataframe[input_vars], dataframe['outcome'])
    coefs = pipe['lr'].coef_
    intercept = pipe['lr'].intercept_

    return pipe

def next_step(model=None, increment_y = 1, step_x = 0.125, step_y = 0.125, dataframe=pd.DataFrame()):
    mean_x, mean_y, range_x, range_y = normal_to_coded_var(dataframe, output_pars = True)
    coef_x = model['lr'].coef_[0]
    coef_y = model['lr'].coef_[1]
    ratio = coef_x/coef_y

    print(f'coef x {coef_x:.2f} and coef y {coef_y:.2f}')

    increment_x = np.abs(ratio)*increment_y*np.sign(coef_x)
    increment_y = increment_y*np.sign(coef_y)

    print(f'increment x {increment_x:.2f} and increment y {increment_y:.2f} with ratio {ratio:.2f}')

```

```

base_x = increment_x*(range_x/2)+mean_x
base_y = increment_y*(range_y/2)+mean_y

print(f'new x {base_x:.2f} and new y {base_y:.2f}')

next_x = np.round(np.array([base_x-step_x, base_x+step_x]),2)
next_y = np.round(np.array([base_y-step_y, base_y+step_y]),2)

return next_x, next_y

def final_step(model=None, pars=None, dataframe=pd.DataFrame(), final_stage=True):
    # do the last step with an increased precision
    next_x, next_y = next_step(model=model, dataframe=dataframe)
    test_df = output_model(next_x, next_y, pars=pars, cnt_point=True, final_stage=final_stage)
    return test_df

x = np.round(np.linspace(-2,2,101),2)
y = np.round(np.linspace(-2,2,101),2)
X, Y = np.meshgrid(x,y)
Z, x0, y0, fwhm = gaussian_function(X,Y,add_noise=True)
plt.contourf(x,y,Z, levels=10, vmin=0)
plt.colorbar()

```

## 39.1 first full factorial

```

x_d1 = np.array([0.5,1])
y_d1 = np.array([-0.5,-1])
first_doe = output_model(x_d1, y_d1, pars=(x0, y0, fwhm), cnt_point=False)
first_doe

plt.contourf(x,y,Z, levels=10, vmin=0)
plt.colorbar()
plt.scatter(first_doe.x,first_doe.y, c='r')

pipe1 = linear_pipe_fit(degree=2, interaction=True, dataframe=first_doe, input_vars=['x', 'y'])
plot_contour_fromdf(first_doe, fill_value=0, model = pipe1, plot_normal_vars=True)

```

## 39.2 Second Step

```
x_d2, y_d2 = next_step(model=pipe1, increment_y = 2, dataframe=first_doe)
print(x_d2, y_d2)

test_df2 = output_model(x_d2, y_d2, model = pipe1, pars=(x0, y0, fwhm), cnt_point=True)
pipe2 = linear_pipe_fit(degree=1, interaction=True, dataframe=test_df2, input_vars=['x', 'y']
plot_contour_fromdf(test_df2, fill_value=0, model = pipe2)
x_d3, y_d3 = next_step(model=pipe2, increment_y = 2, dataframe=test_df2)
print(x_d3, y_d3)

test_df2

plt.contourf(x,y,Z, levels=10, vmin=0)
plt.colorbar()
plt.scatter(first_doe.x,first_doe.y, c='r')
plt.scatter(test_df2.x,test_df2.y, c='b')
```

## 39.3 Third Step

```
test_df3 = output_model(x_d3, y_d3, pipe2, pars=(x0, y0, fwhm), cnt_point=False)
pipe3 = linear_pipe_fit(degree=1, interaction=True, dataframe=test_df3, input_vars=['x', 'y']
plot_contour_fromdf(test_df3, fill_value=0, model = pipe3)
x_d4, y_d4 = next_step(model=pipe3, increment_y = 2, dataframe=test_df3)
print(x_d4, y_d4)

test_df3

plt.contourf(x,y,Z, levels=10, vmin=0)
plt.colorbar()
plt.scatter(first_doe.x,first_doe.y, c='r')
plt.scatter(test_df2.x,test_df2.y, c='b')
plt.scatter(test_df3.x,test_df3.y, c='k')
```

# 40 Final Implementation

```
steps_df = pd.DataFrame()
x_t = np.array([0.5,1])
y_t = np.array([-0.5,-1])

for i in range(10):
    test_df = output_model(x_t, y_t, pars=(x0, y0, fwhm), cnt_point=False)
    test_df['iteration'] = i
    pipe1 = linear_pipe_fit(degree=1, interaction=True, dataframe=test_df, input_vars=['x']
    x_t, y_t = next_step(pipe1, increment_y = 2, dataframe=test_df)
    #plot_contour_fromdf(test_df, fill_value=0, model = pipe1, plot_normal_vars=True)
    steps_df = steps_df.append(test_df)
    max_outcome = test_df['outcome'].max()

    if max_outcome < steps_df['outcome'].max():
        final_ = final_step(model=pipe1, pars=(x0, y0, fwhm), dataframe=test_df)
        final_['iteration'] = i
        pipe_last = linear_pipe_fit(degree=2, interaction=True, dataframe=final_, input_va
        steps_df = steps_df.append(final_)
        print(f'maximum found at iteration {i}')
        break
    print(x_t, y_t)

plt.contourf(x,y,Z)
plt.colorbar()
sns.scatterplot(x='x',y='y',data=steps_df, hue='iteration', palette='coolwarm', legend='fu

steps_df[steps_df.iteration==6]

pol = PolynomialFeatures(degree=2, interaction_only=False, include_bias=False)
lr = LinearRegression()
pipe_all = Pipeline([('pol',pol), ('lr', lr)])
pipe_all.fit(steps_df[['x','y']], steps_df['outcome'])
```

```
pipe_all['lr'].coef_

plot_contour_fromdf(steps_df, fill_value=0, model = pipe_all, levels=30)

pipe_all.score(steps_df[['x','y']], steps_df['outcome'])

from sklearn.ensemble import RandomForestRegressor

rfr = RandomForestRegressor(max_depth=8)
rfr.fit(steps_df[['x','y']], steps_df['outcome'])
plot_contour_fromdf(steps_df, fill_value=0, model = rfr, levels=30)
```

# 41 Exercise RBF

## 41.1 Package Loading

```
%matplotlib inline
import numpy as np
from numpy.matlib import eye
import scipy.linalg
from numpy import linalg as LA
from pyspot.design.spacefilling import spacefilling
from pyspot.fun.objectivefunctions import analytical
import matplotlib.pyplot as plt
```

## 41.2 1.2 Define a small number

```
eps = np.sqrt(np.spacing(1))
```

## 41.3 1.3 The Sampling Plan ( $\mathbf{X}$ )

- We will use 256 points.
- The first 10 points are shown below.

```
gen = spacefilling(2)
rng = np.random.RandomState(1)
lower = np.array([-1,-1])
upper = np.array([2,2])
X = gen.scipy_lhd(256, lower=lower, upper = upper)
X[1:10]
```

## 41.4 1.4 The Objective Function

- Here we use  $\sum_{i=1}^n (x_i - 1)^2$ .
- `f_map()` is a helper function that maps  $f$  to the entries (points) in the matrix  $X$ .

```
def f(x):
    return np.sum((x-1.0)**2)

def f_map(x):
    return np.array(list(map(f, x)))

y = f_map(X)
y[1:10]
```

- Alternatively, we can use pre-defined functions from the `pyspot` package:

```
# fun = analytical(sigma=0).fun_branin
# fun = analytical(sigma=0).fun_sphere

XX, YY = np.meshgrid(np.linspace(-1, 2, 128), np.linspace(-1, 2, 128))
zz = np.array([f_map(np.array([xi, yi])).reshape(-1,2) for xi, yi in zip(np.ravel(XX), np.ravel(YY))])

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
co = ax.pcolormesh(XX, YY, zz, vmin=-1, vmax=1, cmap='RdBu_r')

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
co = ax.contourf(XX, YY, zz, levels=np.linspace(0,2, 10))
```

## 42 2 The Gram Matrix

```
def build_Gram(X):
    """
    Construction of the Gram matrix.
    """
    n = X.shape[0]
    G = np.zeros((n, n))
    for i in range(n):
        for j in range(i, n):
            G[i, j] = np.linalg.norm(X[i] - X[j])
    G = G + G.T
    return G

G = build_Gram(X)
G
```

## 43 3 The Radial Basis Functions

```
def basis_linear(r):
    return r*r*r

def basis_gauss(r, sigma = 1e-1):
    return np.exp(-r**2/sigma)
```

- We select the Gaussian basis function for the following examples:

```
basis = basis_gauss
```

### 43.1 4 The $\Psi$ Matrix

```
def build_Phi(G, basis, eps=np.sqrt(np.spacing(1))):
    n = G.shape[0]
    Phi = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            Phi[i,j] = basis(G[i,j])
    Phi = Phi + np.multiply(np.mat(eye(n)), eps)
    return Phi

Phi = build_Phi(G, basis=basis)
Phi[0:3,0:3]
```

## 44 5 Inverting $\Psi$ via Cholesky Factorization

- There are two different implementations of the Cholesky factorization in Python:
  - `numpy`'s `linalg.cholesky()` and
  - `scipy`'s `linalg.cholesky()`
- We will use `numpy`'s version.

```
def get_rbf_weights(Phi, y):  
    """  
    Calculating the weights of the radial basis function surrogate.  
    Cholesky factorization used.  
    LU decomposition otherwise (not implemented yet).  
    """  
    # U = scipy.linalg.cholesky(Phi, lower=True)  
    U = np.linalg.cholesky(Phi)  
    U = U.T  
    # w = U\U'\ModelInfo.y  
    w = np.linalg.solve(U, np.linalg.solve(U.T, y))  
    return w  
  
w = get_rbf_weights(Phi, y)  
w[0:3]
```

# 45 6 Predictions

## 45.1 6.1 The Predictor

```
def pred_rbf(x, X, basis, w):
    n = X.shape[0]
    d = np.zeros((n))
    phi = np.zeros((n))
    for i in range(n):
        d[i] = np.linalg.norm(x - X[i])
    for i in range(n):
        phi[i] = basis(d[i])
    return w @ phi
```

## 45.2 6.2 Testing some Example Points

```
x = X[0]
x
```

## 45.3 6.1 The RBF Prediction $\hat{f}$

```
pred_rbf(x=x, X=X, basis=basis, w=w)
```

## 45.4 6.2 The Original (True) Value $f$

```
f_map(np.array(x).reshape(1,-1))
```

## 45.5 6.3 Visualizations

```
XX, YY = np.meshgrid(np.linspace(-1, 2, 128), np.linspace(-1, 2, 128))
zz = np.array([pred_rbf(x=np.array([xi, yi]), X=X, basis=basis, w=w) for xi, yi in zip(np.ran
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
co = ax.pcolormesh(XX, YY, zz, vmin=-1, vmax=1, cmap='RdBu_r')

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
co = ax.contourf(XX, YY, zz, levels=np.linspace(0,2, 5))
```

## 46 7 Note

- The original function  $f$  is cheaper than the surrogate  $\hat{f}$  in this example, because we have chosen a simple analytical function as the ground truth.
- This is not the case in real-world settings.

# 47 8 Cholesky Factorization

## 47.1 8.1 $A = U^T U$

- $U$  is an upper triangular matrix

```
def cholesky_U(A):
    N = A.shape[0]
    U = np.zeros((N,N))
    for k in range(0,N):
        # compute diagonal entry
        U[k,k] = A[k,k]
        for j in range(0,k):
            U[k,k] = U[k,k] - U[j,k]*U[j,k]
        U[k,k] = np.sqrt(U[k,k])
        # compute remaining column
        for i in range(k+1,N):
            U[k,i] = A[k,i]
            for j in range(0,k):
                U[k,i] = U[k,i] - U[j,i]*U[j,k]
            U[k,i] = U[k,i] / U[k,k]
    return U
```

## 47.2 8.2 $A = LL^T$

- $L$  is a lower triangular matrix

```
def cholesky_L(A):
    N = A.shape[0]
    L = np.zeros((N,N))
    for k in range(0,N):
        # compute diagonal entry
        L[k,k] = A[k,k]
        for j in range(0,k):
```

```

L[k,k] = L[k,k] - L[k,j]*L[k,j]
L[k,k] = np.sqrt(L[k,k])
# compute remaining column
for i in range(k+1,N):
    L[i,k] = A[i,k]
    for j in range(0,k):
        L[i,k] = L[i,k] - L[i,j]*L[k,j]
    L[i,k] = L[i,k] / L[k,k]
return L

```

### 47.3 8.3 Example

```

A = np.array([[4, 2, 4, 4], [2, 10, 5, 2], [4, 5, 9, 6], [4, 2, 6, 9]])
A

```

### 47.4 8.4 Check: Is $A$ positive definite?

```
assert(np.all(np.linalg.eigvals(A) > 0))
```

### 47.5 8.5.1 $A = U^T U$

- Perform Cholesky Factorization

```

U = cholesky_U(A)
U

```

- Test Result

```
U.T @ U
```

## 47.6 8.5.2 $A = LL^T$

```
L = cholesky_L(A)  
L
```

- Test Result

```
L @ L.T
```

# 48 9 Exercises

## 48.1 9.1 Gaussian Basis Function

### 48.1.1 9.1.1

- Plot the Gaussian Basis Function `basis_gauss` in the range from -2 to 2 using `matplotlib.pyplot`
  - Hint: Check the [matplotlib documentation](#) for examples.
  - Generate a plot with several `sigma` values, e.g., 0.1, 1.0, and 10.

### 48.1.2 9.1.2

- What is the meaning of the `sigma` parameter: Can you explain its influence / effect on the model quality?
  - Is the `sigma` value important?

## 48.2 9.2 Linear Basis Function

- Select the linear basis function?
- What errors occur?
- Do you have any ideas how to fix this error?

## **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.

# B 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”

## B.1 Code cells

The code cells are used to execute the code. They are following the logic of the chosen 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).

## B.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 Sheet](#)

## B.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.

### B.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. Choose the appropriate “Raw NBConvert Format” within the cell

Data Science is fun

## C Install packages

Because python is a heavily used programming language, there are many different packages that can make your life easier. Sadly, there are only a few standard packages that are already included in your python environment. If you have the need to install a new package in your environment, you can simply do that by executing 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 package you want to install

**Hint:** It is often useful to restart the kernel after installing a package, otherwise loading the package could lead to an error.

## D 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:

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.express 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

# E 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.

## F 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

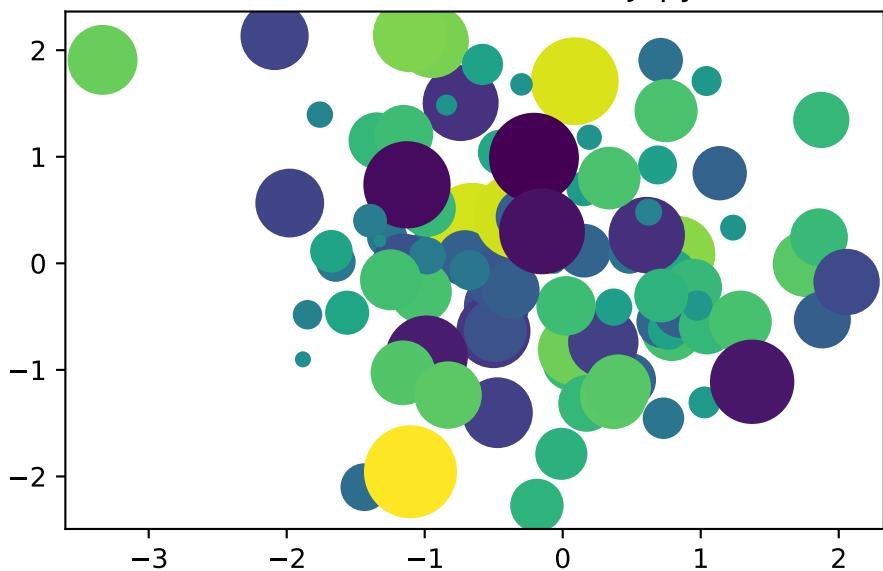
If you combine everything you can create beautiful graphics

```
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!



# G 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/>.

## G.1 Example: `spot`

```
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
```

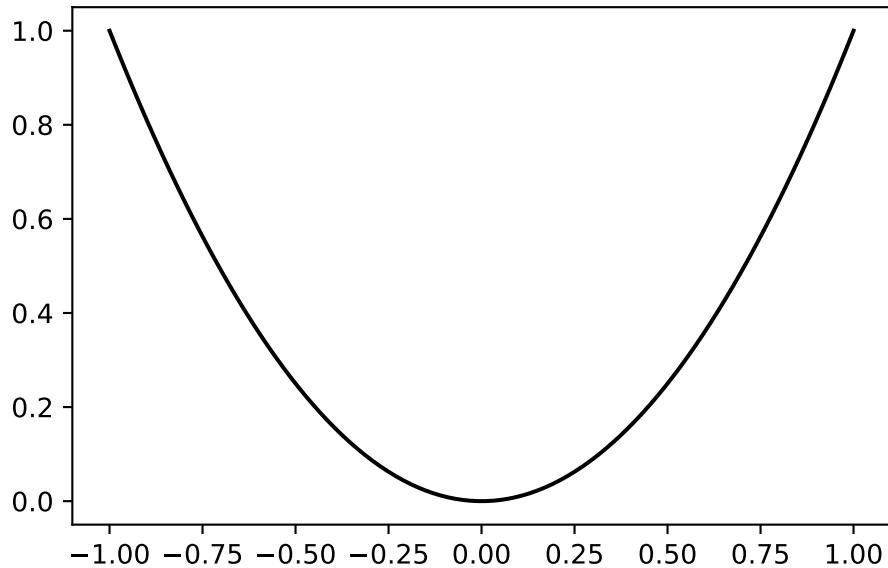
### G.1.1 The Objective Function

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()
```



```
spot_1 = spot.Spot(fun=fun,
                    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,
                    show_models=True,
                    fun_control = {},
                    design_control={"init_size": 5,
                                   "repeats": 1},
                    surrogate_control={"noise": False,
                                      "cod_type": "norm",
                                      "min_theta": -4,
                                      "max_theta": 3,
                                      "n_theta": 1,
                                      "model_optimizer": differential_evolution,
                                      "model_fun_evals": 1000},
```

```
})
```

`spot`'s `__init__` method sets the control parameters. There are two parameter groups:

1. external parameters can be specified by the user
2. internal parameters, which are handled by `spot`.

### G.1.2 External Parameters

external parameter	type	description	default	mandatory
<code>fun</code>	object	objective function		yes
<code>lower</code>	array	lower bound		yes
<code>upper</code>	array	upper bound		yes
<code>fun_evals</code>	int	number of function evaluations	15	no
<code>fun_evals</code>	int	number of function evaluations	15	no
<code>fun_control</code>	dict	noise etc.	{}	n
<code>max_time</code>	int	max run time budget	<code>inf</code>	no
<code>noise</code>	bool	if repeated evaluations of <code>fun</code> results in different values, then <code>noise</code> should be set to <code>True</code> .	<code>False</code>	no

---

external parameter	type	description	default	mandatory
<b>tolerance_x</b>	float	tolerance for new x solutions. Minimum distance of new solutions, generated by <code>suggest_new_X</code> , to already existing solutions. If zero (which is the default), every new solution is accepted.	0	no
<b>var_type</b>	list	list of type information, can be either "num" or "factor"	["num"]	no
<b>infill_criterion</b>	string	Can be "y", "s", "y" "ei" (negative expected improvement), or "all"		no
<b>n_points</b>	int	number of infill points	1	no
<b>seed</b>	int	initial seed. If <code>Spot.run()</code> is called twice, different results will be generated. To reproduce results, the <code>seed</code> can be used.	123	no

external parameter	type	description	default	mandatory
log_level	int	log level with the following settings: NOTSET (0), DEBUG (10: Detailed information, typically of interest only when diagnosing problems.), INFO (20: Confirmation that things are working as expected.), WARNING (30: An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.), ERROR (40: Due to a more serious problem, the software has not been able to perform some function.), and CRITICAL (50: A serious error, indicating that the program itself may be unable to continue running.)	50	no

external parameter	type	description	default	mandatory
<code>show_models</code>	bool	Plot model. Currently only 1-dim functions are supported	<code>False</code>	no
<code>design</code>	object	experimental design	<code>None</code>	no
<code>design_control</code>	dict	control parameters	see below	no
<code>surrogate</code>		surrogate model	<code>kriging</code>	no
<code>surrogate_control</code>		control parameters	see below	no
<code>optimizer</code>	object	optimizer	see below	no
<code>optimizer_control</code>		control parameters	see below	no

- Besides these single parameters, the following parameter dictionaries can be specified by the user:

- `fun_control`
- `design_control`
- `surrogate_control`
- `optimizer_control`

## G.2 The `fun_control` Dictionary

external parameter	type	description	default	mandatory
<code>sigma</code>	float	noise: standard deviation	0	yes
<code>seed</code>	int	seed for rng	124	yes

## G.3 The `design_control` Dictionary

external parameter	type	description	default	mandatory
<code>init_size</code>	int	initial sample size	10	yes

external parameter	type	description	default	mandatory
<b>repeats</b>	int	number of repeats of the initial samples	1	yes

## G.4 The surrogate\_control Dictionary

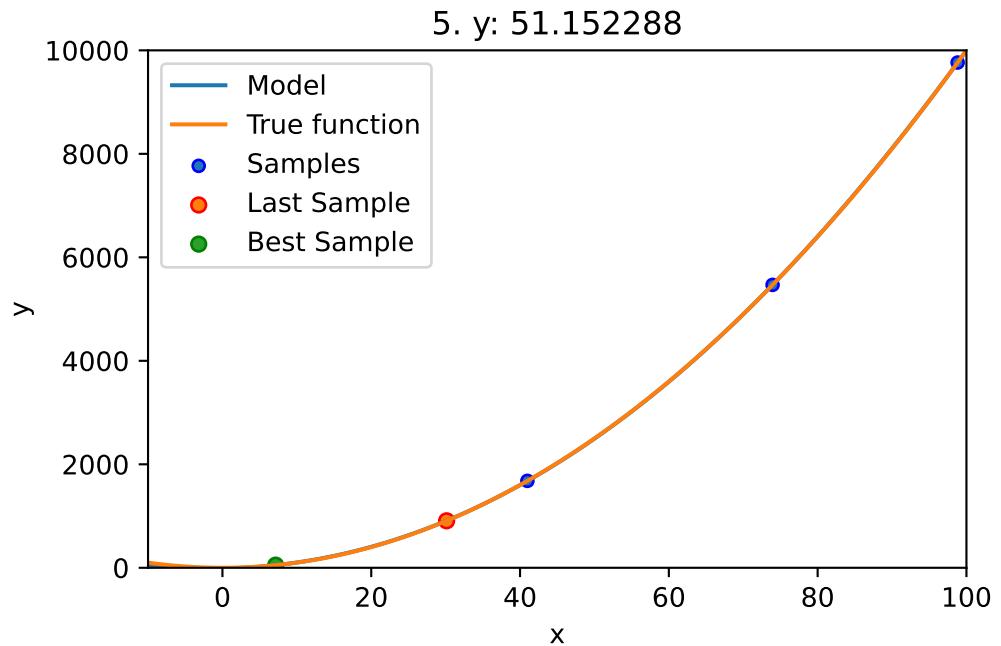
external parameter	type	description	default	mandatory
<b>noise</b>				
<b>model_optimizer</b>	object	optimizer	<b>differential_evolution</b>	
<b>model_fun_evals</b>				
<b>min_theta</b>			-3.	
<b>max_theta</b>			3.	
<b>n_theta</b>			1	
<b>n_p</b>			1	
<b>optim_p</b>			<b>False</b>	
<b>cod_type</b>			"norm"	
<b>var_type</b>				
<b>use_cod_y</b>	bool		<b>False</b>	

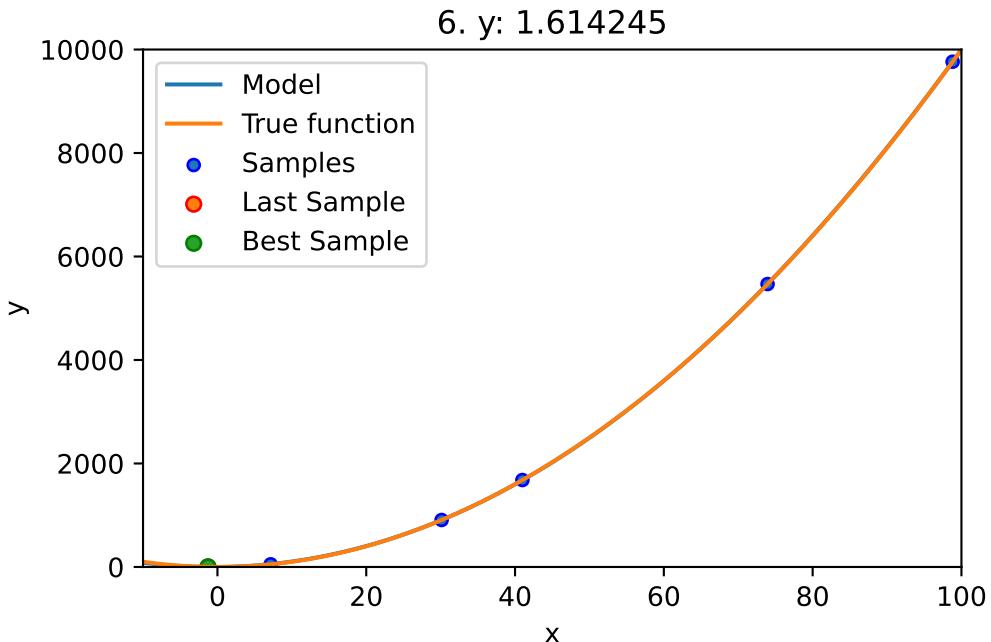
## G.5 The optimizer\_control Dictionary

external parameter	type	description	default	mandatory
<b>max_iter</b>	int	max number of iterations. Note: these are the cheap evaluations on the surrogate.	1000	no

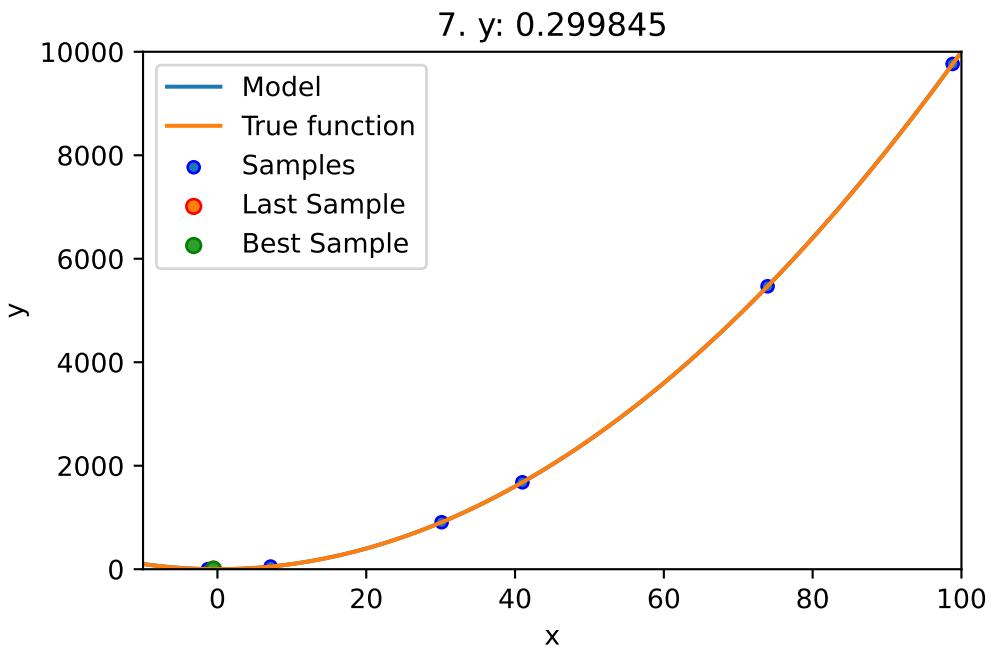
## G.6 Run

```
spot_1.run()
```





spotPython tuning: 1.6142446477388548 [#####-] 85.71%



```
spotPython tuning: 0.29984480579304645 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2d284bfa0>
```

## G.7 Print the Results

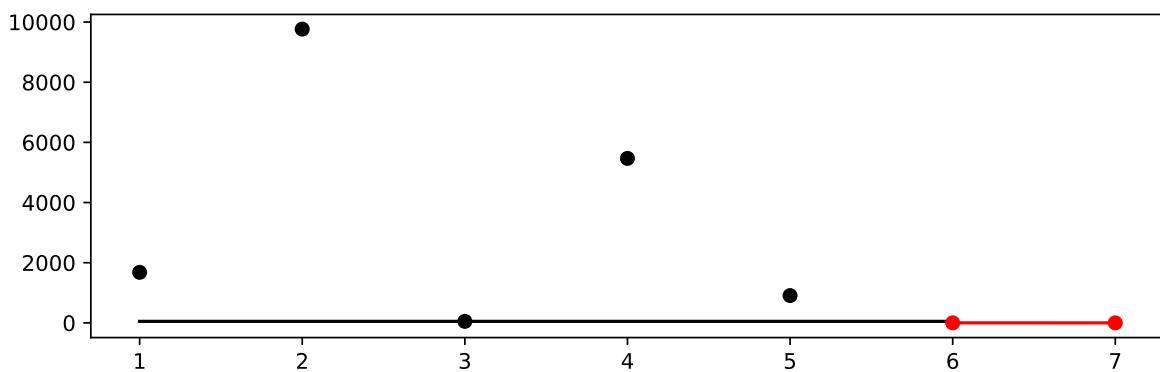
```
spot_1.print_results()
```

```
min y: 0.29984480579304645  
x0: -0.5475808668982568
```

```
[['x0', -0.5475808668982568]]
```

## G.8 Show the Progress

```
spot_1.plot_progress()
```

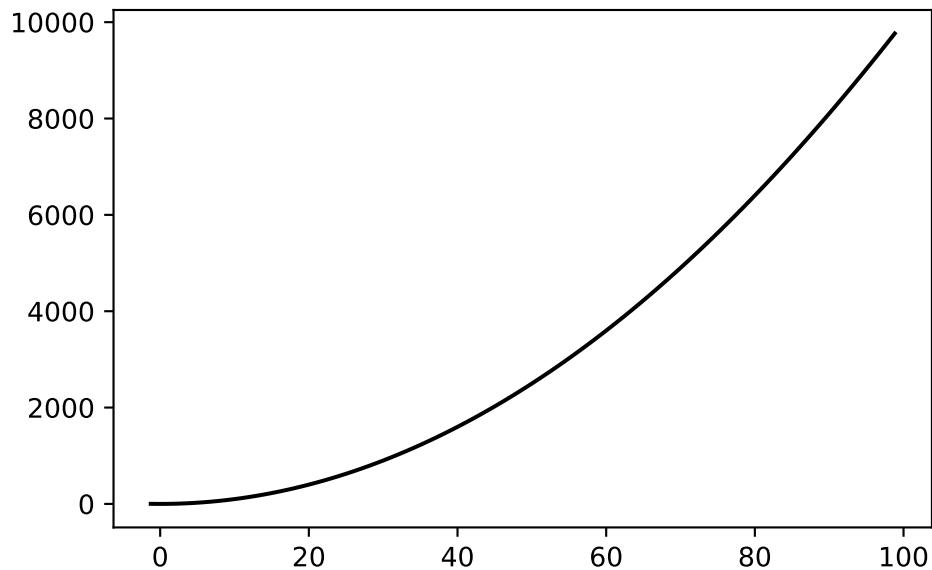


## G.9 Visualize the Surrogate

- The plot method of the `kriging` surrogate is used.
- Note: the plot uses the interval defined by the ranges of the natural variables.

```
spot_1.surrogate.plot()
```

<Figure size 2700x1800 with 0 Axes>



## G.10 Run With a Specific Start Design

```
spot_x0 = spot.Spot(fun=fun,
                     lower = np.array([-10, -10]),
                     upper = np.array([10, 10]),
                     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,
                     show_models=False,
                     fun_control = {},
                     design_control={"init_size": 5,
                                    "repeats": 1},
                     surrogate_control={"noise": False},
```

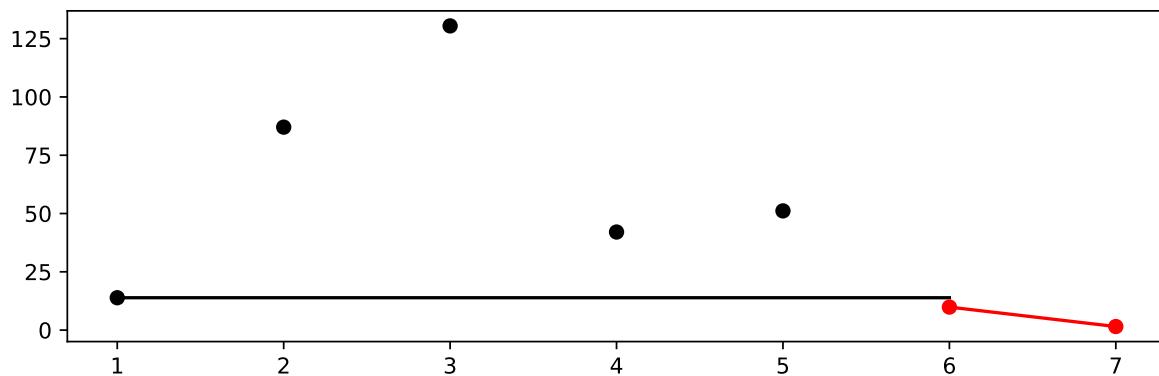
```

    "cod_type": "norm",
    "min_theta": -4,
    "max_theta": 3,
    "n_theta": 2,
    "model_optimizer": differential_evolution,
    "model_fun_evals": 1000,
)
spot_x0.run(X_start=np.array([0.5, -0.5]))
spot_x0.plot_progress()

```

spotPython tuning: 9.869670875300805 [#####-] 85.71%

spotPython tuning: 1.5261843012598162 [#####] 100.00% Done...



## G.11 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]

```

## G.12 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 ]]))
```

## G.13 Surrogates

### G.13.1 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  $f$ .

## G.14 Demo/Test: Objective Function Fails

SPOT expects `np.nan` values from failed objective function values. These are handled. Note: SPOT's counter considers only successful executions of the objective function.

```
import numpy as np
from spotPython.fun.objectivefunctions import analytical
```

```

from spotPython.spot import spot
import numpy as np
from math import inf
# number of initial points:
ni = 20
# number of points
n = 30

fun = analytical().fun_random_error
lower = np.array([-1])
upper = np.array([1])
design_control={"init_size": ni}

spot_1 = spot.Spot(fun=fun,
                    lower = lower,
                    upper= upper,
                    fun_evals = n,
                    show_progress=False,
                    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

```

```

[ 0.53176481 -0.9053821 -0.02203599 -0.21843718  0.78240941 -0.58120945
 -0.3923345          nan  0.31802454 -0.68898927 -0.75129705  0.97550354
  0.41757584  0.0786237   0.82585329  0.23700598 -0.49274073 -0.82319082
 -0.17991251  0.1481835 ]
[-1.]

```

[0.17335968]

[-0.58552368]

[-0.20126111]

[-0.60100809]

$[-0.97897336]$   
 $[-0.2748985]$

$[0.8359486]$

$[0.99035591]$   
 $[0.01641232]$

$[0.5629346]$

## G.15 PyTorch: Detailed Description of the Data Splitting

### G.15.1 Description of the "train\_hold\_out" Setting

The "train\_hold\_out" setting is used by default. It uses the loss function specified in `fun_control` and the metric specified in `fun_control`.

1. First, the method `HyperTorch().fun_torch` is called.
2. `fun_torch()`, which is implemented in the file `hypertorch.py`, calls `evaluate_hold_out()` as follows:

```
df_eval, _ = evaluate_hold_out(  
    model,  
    train_dataset=fun_control["train"],  
    shuffle=self.fun_control["shuffle"],  
    loss_function=self.fun_control["loss_function"],  
    metric=self.fun_control["metric_torch"],  
    device=self.fun_control["device"],  
    show_batch_interval=self.fun_control["show_batch_interval"],  
    path=self.fun_control["path"],  
    task=self.fun_control["task"],  
    writer=self.fun_control["writer"],  
    writerId=config_id,  
)
```

Note: Only the data set `fun_control["train"]` is used for training and validation. It is used in `evaluate_hold_out` as follows:

```
trainloader, valloader = create_train_val_data_loaders(  
    dataset=train_dataset, batch_size=batch_size_instance, shuffle=shuffle  
)
```

`create_train_val_data_loaders()` splits the `train_dataset` into `trainloader` and `valloader` using `torch.utils.data.random_split()` as follows:

```
def create_train_val_data_loaders(dataset, batch_size, shuffle, num_workers=0):  
    test_abs = int(len(dataset) * 0.6)  
    train_subset, val_subset = random_split(dataset, [test_abs, len(dataset) - test_abs])  
    trainloader = torch.utils.data.DataLoader(  
        train_subset, batch_size=int(batch_size), shuffle=shuffle, num_workers=num_workers  
)  
    valloader = torch.utils.data.DataLoader(
```

```

        val_subset, batch_size=int(batch_size), shuffle=shuffle, num_workers=num_workers
    )
    return trainloader, valloader

```

The optimizer is set up as follows:

```

optimizer_instance = net.optimizer
lr_mult_instance = net.lr_mult
sgd_momentum_instance = net.sgd_momentum
optimizer = optimizer_handler(
    optimizer_name=optimizer_instance,
    params=net.parameters(),
    lr_mult=lr_mult_instance,
    sgd_momentum=sgd_momentum_instance,
)

```

3. `evaluate_hold_out()` sets the `net` attributes such as `epochs`, `batch_size`, `optimizer`, and `patience`. For each epoch, the methods `train_one_epoch()` and `validate_one_epoch()` are called, the former for training and the latter for validation and early stopping. The validation loss from the last epoch (not the best validation loss) is returned from `evaluate_hold_out`.
4. The method `train_one_epoch()` is implemented as follows:

```

def train_one_epoch(
    net,
    trainloader,
    batch_size,
    loss_function,
    optimizer,
    device,
    show_batch_interval=10_000,
    task=None,
):
    running_loss = 0.0
    epoch_steps = 0
    for batch_nr, data in enumerate(trainloader, 0):
        input, target = data
        input, target = input.to(device), target.to(device)
        optimizer.zero_grad()
        output = net(input)
        if task == "regression":

```

```

        target = target.unsqueeze(1)
        if target.shape == output.shape:
            loss = loss_function(output, target)
        else:
            raise ValueError(f"Shapes of target and output do not match:
                            {target.shape} vs {output.shape}")
    elif task == "classification":
        loss = loss_function(output, target)
    else:
        raise ValueError(f"Unknown task: {task}")
    loss.backward()
    torch.nn.utils.clip_grad_norm_(net.parameters(), max_norm=1.0)
    optimizer.step()
    running_loss += loss.item()
    epoch_steps += 1
    if batch_nr % show_batch_interval == (show_batch_interval - 1):
        print(
            "Batch: %5d. Batch Size: %d. Training Loss (running): %.3f"
            % (batch_nr + 1, int(batch_size), running_loss / epoch_steps)
        )
        running_loss = 0.0
    return loss.item()

```

5. The method `validate_one_epoch()` is implemented as follows:

```

def validate_one_epoch(net, valloader, loss_function, metric, device, task):
    val_loss = 0.0
    val_steps = 0
    total = 0
    correct = 0
    metric.reset()
    for i, data in enumerate(valloader, 0):
        # get batches
        with torch.no_grad():
            input, target = data
            input, target = input.to(device), target.to(device)
            output = net(input)
            # print(f"target: {target}")
            # print(f"output: {output}")
            if task == "regression":
                target = target.unsqueeze(1)

```

```

        if target.shape == output.shape:
            loss = loss_function(output, target)
        else:
            raise ValueError(f"Shapes of target and output
                            do not match: {target.shape} vs {output.shape}")
        metric_value = metric.update(output, target)
    elif task == "classification":
        loss = loss_function(output, target)
        metric_value = metric.update(output, target)
        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()
    else:
        raise ValueError(f"Unknown task: {task}")
    val_loss += loss.cpu().numpy()
    val_steps += 1
loss = val_loss / val_steps
print(f"Loss on hold-out set: {loss}")
if task == "classification":
    accuracy = correct / total
    print(f"Accuracy on hold-out set: {accuracy}")
# metric on all batches using custom accumulation
metric_value = metric.compute()
metric_name = type(metric).__name__
print(f"{metric_name} value on hold-out data: {metric_value}")
return metric_value, loss

```

### G.15.1.1 Description of the "test\_hold\_out" Setting

It uses the loss function specified in `fun_control` and the metric specified in `fun_control`.

1. First, the method `HyperTorch().fun_torch` is called.
2. `fun_torc()` calls `spotPython.torch.traintest.evaluate_hold_out()` similar to the "`train_hold_out`" setting with one exception: It passes an additional `test` data set to `evaluate_hold_out()` as follows:

```
test_dataset=fun_control["test"]
```

`evaluate_hold_out()` calls `create_train_test_data_loaders` instead of `create_train_val_data_loaders`. The two data sets are used in `create_train_test_data_loaders` as follows:

```

def create_train_test_data_loaders(dataset, batch_size, shuffle, test_dataset,
    num_workers=0):
    trainloader = torch.utils.data.DataLoader(
        dataset, batch_size=int(batch_size), shuffle=shuffle,
        num_workers=num_workers
    )
    testloader = torch.utils.data.DataLoader(
        test_dataset, batch_size=int(batch_size), shuffle=shuffle,
        num_workers=num_workers
    )
    return trainloader, testloader

```

3. The following steps are identical to the "train\_hold\_out" setting. Only a different data loader is used for testing.

#### **G.15.1.2 Detailed Description of the "train\_cv" Setting**

It uses the loss function specified in `fun_control` and the metric specified in `fun_control`.

1. First, the method `HyperTorch().fun_torch` is called.
2. `fun_torch()` calls `spotPython.torch.traintest.evaluate_cv()` as follows (Note: Only the data set `fun_control["train"]` is used for CV.):

```

df_eval, _ = evaluate_cv(
    model,
    dataset=fun_control["train"],
    shuffle=self.fun_control["shuffle"],
    device=self.fun_control["device"],
    show_batch_interval=self.fun_control["show_batch_interval"],
    task=self.fun_control["task"],
    writer=self.fun_control["writer"],
    writerId=config_id,
)

```

3. In 'evaluate\_cv()', the following steps are performed: The optimizer is set up as follows:

```

optimizer_instance = net.optimizer
lr_instance = net.lr
sgd_momentum_instance = net.sgd_momentum
optimizer = optimizer_handler(optimizer_name=optimizer_instance,
    params=net.parameters(), lr_mult=lr_mult_instance)

```

`evaluate_cv()` sets the `net` attributes such as `epochs`, `batch_size`, `optimizer`, and `patience`. CV is implemented as follows:

```
def evaluate_cv(
    net,
    dataset,
    shuffle=False,
    loss_function=None,
    num_workers=0,
    device=None,
    show_batch_interval=10_000,
    metric=None,
    path=None,
    task=None,
    writer=None,
    writerId=None,
):
    lr_mult_instance = net.lr_mult
    epochs_instance = net.epochs
    batch_size_instance = net.batch_size
    k_folds_instance = net.k_folds
    optimizer_instance = net.optimizer
    patience_instance = net.patience
    sgd_momentum_instance = net.sgd_momentum
    removed_attributes, net = get_removed_attributes_and_base_net(net)
    metric_values = {}
    loss_values = {}
    try:
        device = getDevice(device=device)
        if torch.cuda.is_available():
            device = "cuda:0"
            if torch.cuda.device_count() > 1:
                print("We will use", torch.cuda.device_count(), "GPUs!")
                net = nn.DataParallel(net)
        net.to(device)
        optimizer = optimizer_handler(
            optimizer_name=optimizer_instance,
            params=net.parameters(),
            lr_mult=lr_mult_instance,
            sgd_momentum=sgd_momentum_instance,
        )
        kfold = KFold(n_splits=k_folds_instance, shuffle=shuffle)
```

```

for fold, (train_ids, val_ids) in enumerate(kfold.split(dataset)):
    print(f"Fold: {fold + 1}")
    train_subampler = torch.utils.data.SubsetRandomSampler(train_ids)
    val_subampler = torch.utils.data.SubsetRandomSampler(val_ids)
    trainloader = torch.utils.data.DataLoader(
        dataset, batch_size=batch_size_instance,
        sampler=train_subampler, num_workers=num_workers
    )
    valloader = torch.utils.data.DataLoader(
        dataset, batch_size=batch_size_instance,
        sampler=val_subampler, num_workers=num_workers
    )
    # each fold starts with new weights:
    reset_weights(net)
    # Early stopping parameters
    best_val_loss = float("inf")
    counter = 0
    for epoch in range(epochs_instance):
        print(f"Epoch: {epoch + 1}")
        # training loss from one epoch:
        training_loss = train_one_epoch(
            net=net,
            trainloader=trainloader,
            batch_size=batch_size_instance,
            loss_function=loss_function,
            optimizer=optimizer,
            device=device,
            show_batch_interval=show_batch_interval,
            task=task,
        )
        # Early stopping check. Calculate validation loss from one epoch:
        metric_values[fold], loss_values[fold] = validate_one_epoch(
            net, valloader=valloader, loss_function=loss_function,
            metric=metric, device=device, task=task
        )
        # Log the running loss averaged per batch
        metric_name = "Metric"
        if metric is None:
            metric_name = type(metric).__name__
            print(f"{metric_name} value on hold-out data:
                  {metric_values[fold]}")

```

```

        if writer is not None:
            writer.add_scalars(
                "evaluate_cv fold:" + str(fold + 1) +
                ". Train & Val Loss and Val Metric" + writerId,
                {"Train loss": training_loss, "Val loss":
                 loss_values[fold], metric_name: metric_values[fold]},
                epoch + 1,
            )
            writer.flush()
        if loss_values[fold] < best_val_loss:
            best_val_loss = loss_values[fold]
            counter = 0
            # save model:
            if path is not None:
                torch.save(net.state_dict(), path)
        else:
            counter += 1
            if counter >= patience_instance:
                print(f"Early stopping at epoch {epoch}")
                break
        df_eval = sum(loss_values.values()) / len(loss_values.values())
        df_metrics = sum(metric_values.values()) / len(metric_values.values())
        df_preds = np.nan
    except Exception as err:
        print(f"Error in Net_Core. Call to evaluate_cv() failed. {err=},
              {type(err)=}")
        df_eval = np.nan
        df_preds = np.nan
    add_attributes(net, removed_attributes)
    if writer is not None:
        metric_name = "Metric"
        if metric is None:
            metric_name = type(metric).__name__
        writer.add_scalars(
            "CV: Val Loss and Val Metric" + writerId,
            {"CV-loss": df_eval, metric_name: df_metrics},
            epoch + 1,
        )
        writer.flush()
    return df_eval, df_preds, df_metrics

```

4. The method `train_fold()` is implemented as shown above.

5. The method `validate_one_epoch()` is implemented as shown above. In contrast to the hold-out setting, it is called for each of the  $k$  folds. The results are stored in a dictionaries `metric_values` and `loss_values`. The results are averaged over the  $k$  folds and returned as `df_eval`.

#### G.15.1.3 Detailed Description of the "test\_cv" Setting

It uses the loss function specified in `fun_control` and the metric specified in `fun_control`.

1. First, the method `HyperTorch().fun_torch` is called.
2. `fun_torc()` calls `spotPython.torch.taintest.evaluate_cv()` as follows:

```
df_eval, _ = evaluate_cv(
    model,
    dataset=fun_control["test"],
    shuffle=self.fun_control["shuffle"],
    device=self.fun_control["device"],
    show_batch_interval=self.fun_control["show_batch_interval"],
    task=self.fun_control["task"],
    writer=self.fun_control["writer"],
    writerId=config_id,
)
```

Note: The data set `fun_control["test"]` is used for CV. The rest is the same as for the "train\_cv" setting.

#### G.15.1.4 Detailed Description of the Final Model Training and Evaluation

There are two methods that can be used for the final evaluation of a Pytorch model:

1. "train\_tuned" and
2. "test\_tuned".

`train_tuned()` is just a wrapper to `evaluate_hold_out` using the `train` data set. It is implemented as follows:

```
def train_tuned(
    net,
    train_dataset,
    shuffle,
    loss_function,
    metric,
```

```

        device=None,
        show_batch_interval=10_000,
        path=None,
        task=None,
        writer=None,
    ):
        evaluate_hold_out(
            net=net,
            train_dataset=train_dataset,
            shuffle=shuffle,
            test_dataset=None,
            loss_function=loss_function,
            metric=metric,
            device=device,
            show_batch_interval=show_batch_interval,
            path=path,
            task=task,
            writer=writer,
        )

```

The `test_tuned()` procedure is implemented as follows:

```

def test_tuned(net, shuffle, test_dataset=None, loss_function=None,
               metric=None, device=None, path=None, task=None):
    batch_size_instance = net.batch_size
    removed_attributes, net = get_removed_attributes_and_base_net(net)
    if path is not None:
        net.load_state_dict(torch.load(path))
        net.eval()
    try:
        device = getDevice(device=device)
        if torch.cuda.is_available():
            device = "cuda:0"
            if torch.cuda.device_count() > 1:
                print("We will use", torch.cuda.device_count(), "GPUs!")
                net = nn.DataParallel(net)
        net.to(device)
        valloader = torch.utils.data.DataLoader(
            test_dataset, batch_size=int(batch_size_instance),
            shuffle=shuffle,
            num_workers=0
        )

```

```
metric_value, loss = validate_one_epoch(
    net, valloader=valloader, loss_function=loss_function,
    metric=metric, device=device, task=task
)
df_eval = loss
df_metric = metric_value
df_preds = np.nan
except Exception as err:
    print(f"Error in Net_Core. Call to test_tuned() failed. {err=},
          {type(err)=}")
    df_eval = np.nan
    df_metric = np.nan
    df_preds = np.nan
add_attributes(net, removed_attributes)
print(f"Final evaluation: Validation loss: {df_eval}")
print(f"Final evaluation: Validation metric: {df_metric}")
print("-----")
return df_eval, df_preds, df_metric
```

# References

- 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](https://github.com/sequential-parameter-optimization/spotPython/blob/main/notebooks/14_spot_ray_hpt_torch_cifar10.ipynb).
- Bartz-Beielstein, Thomas, Jürgen Branke, Jörn Mehnen, and Olaf Mersmann. 2014. “Evolutionary Algorithms.” *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 4 (3): 178–95.
- Bartz-Beielstein, Thomas, Carola Doerr, Jakob Bossek, Sowmya Chandrasekaran, Tome Eftimov, Andreas Fischbach, Pascal Kerschke, et al. 2020. “Benchmarking in Optimization: Best Practice and Open Issues.” arXiv. <https://arxiv.org/abs/2007.03488>.
- Bartz-Beielstein, Thomas, Christian Lasarczyk, and Mike Preuss. 2005. “Sequential Parameter Optimization.” In *Proceedings 2005 Congress on Evolutionary Computation (CEC'05), Edinburgh, Scotland*, edited by B McKay et al., 773–80. Piscataway NJ: IEEE Press.
- Lewis, R M, V Torczon, and M W Trosset. 2000. “Direct search methods: Then and now.” *Journal of Computational and Applied Mathematics* 124 (1–2): 191–207.
- Li, Lisha, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization.” *arXiv e-Prints*, March, arXiv:1603.06560.
- Meignan, David, Sigrid Knust, Jean-Marc Frayet, Gilles Pesant, and Nicolas Gaud. 2015. “A Review and Taxonomy of Interactive Optimization Methods in Operations Research.” *ACM Transactions on Interactive Intelligent Systems*, September.
- Montiel, Jacob, Max Halford, Saulo Martiello Mastelini, Geoffrey Bolmier, Raphael Soury, Robin Vaysse, Adil Zouitine, et al. 2021. “River: Machine Learning for Streaming Data in Python.”
- PyTorch. 2023a. “Hyperparameter Tuning with Ray Tune.” [https://pytorch.org/tutorials/beginner/hyperparameter\\_tuning\\_tutorial.html](https://pytorch.org/tutorials/beginner/hyperparameter_tuning_tutorial.html).
- . 2023b. “Training a Classifier.” [https://pytorch.org/tutorials/beginner/blitz/cifar10\\_tutorial.html](https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html).