

# **Hyperparameter Tuning Cookbook**

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

Thomas Bartz-Beielstein

Jul 17, 2023

# Table of contents

<b>Preface: Optimization and Hyperparameter Tuning</b>	<b>13</b>
Book Structure . . . . .	14
Software Used in this Book . . . . .	16
<b>I Spot as an Optimizer</b>	<b>17</b>
<b>1 Introduction to spotPython</b>	<b>18</b>
1.1 Example: Spot and the Sphere Function . . . . .	18
1.1.1 The Objective Function: Sphere . . . . .	19
1.2 Spot Parameters: fun_evals, init_size and show_models . . . . .	21
1.3 Print the Results . . . . .	23
1.4 Show the Progress . . . . .	23
1.5 Visualizing the Optimization and Hyperparameter Tuning Process with TensorBoard . . . . .	23
<b>2 Multi-dimensional Functions</b>	<b>27</b>
2.1 Example: Spot and the 3-dim Sphere Function . . . . .	27
2.1.1 The Objective Function: 3-dim Sphere . . . . .	27
2.1.2 Results . . . . .	29
2.1.3 A Contour Plot . . . . .	29
2.1.4 TensorBoard . . . . .	31
2.2 Conclusion . . . . .	32
2.3 Exercises . . . . .	33
2.3.1 The Three Dimensional fun_cubed . . . . .	33
2.3.2 The Ten Dimensional fun_wing_wt . . . . .	33
2.3.3 The Three Dimensional fun_runge . . . . .	33
2.3.4 The Three Dimensional fun_linear . . . . .	34
<b>3 Isotropic and Anisotropic Kriging</b>	<b>35</b>
3.1 Example: Isotropic Spot Surrogate and the 2-dim Sphere Function . . . . .	35
3.1.1 The Objective Function: 2-dim Sphere . . . . .	35
3.1.2 Results . . . . .	36
3.2 Example With Anisotropic Kriging . . . . .	37
3.2.1 Taking a Look at the theta Values . . . . .	39

3.3	Exercises . . . . .	40
3.3.1	<code>fun_branin</code> . . . . .	40
3.3.2	<code>fun_sin_cos</code> . . . . .	41
3.3.3	<code>fun_runge</code> . . . . .	41
3.3.4	<code>fun_wingwt</code> . . . . .	41
<b>4</b>	<b>Using sklearn Surrogates in spotPython</b>	<b>42</b>
4.1	Example: Branin Function with <code>spotPython</code> 's Internal Kriging Surrogate . . . . .	42
4.1.1	The Objective Function Branin . . . . .	42
4.1.2	Running the surrogate model based optimizer <code>Spot</code> : . . . . .	43
4.1.3	<code>TensorBoard</code> . . . . .	44
4.1.4	Print the Results . . . . .	44
4.1.5	Show the Progress and the Surrogate . . . . .	46
4.2	Example: Using Surrogates From <code>scikit-learn</code> . . . . .	47
4.2.1	<code>GaussianProcessRegressor</code> as a Surrogate . . . . .	47
4.3	Example: One-dimensional Sphere Function With <code>spotPython</code> 's Kriging . . . . .	49
4.3.1	Results . . . . .	55
4.4	Example: Sklearn Model <code>GaussianProcess</code> . . . . .	56
4.5	Exercises . . . . .	63
4.5.1	<code>DecisionTreeRegressor</code> . . . . .	63
4.5.2	<code>RandomForestRegressor</code> . . . . .	64
4.5.3	<code>linear_model.LinearRegression</code> . . . . .	64
4.5.4	<code>linear_model.Ridge</code> . . . . .	64
4.6	Exercise 2 . . . . .	64
<b>5</b>	<b>Sequential Parameter Optimization: Using <code>scipy</code> Optimizers</b>	<b>65</b>
5.1	The Objective Function Branin . . . . .	65
5.2	The Optimizer . . . . .	66
5.2.1	<code>TensorBoard</code> . . . . .	68
5.3	Print the Results . . . . .	68
5.4	Show the Progress . . . . .	68
5.5	Exercises . . . . .	71
5.5.1	<code>dual_annealing</code> . . . . .	71
5.5.2	<code>direct</code> . . . . .	71
5.5.3	<code>shgo</code> . . . . .	71
5.5.4	<code>basinhopping</code> . . . . .	71
5.5.5	Performance Comparison . . . . .	71
<b>6</b>	<b>Sequential Parameter Optimization: Gaussian Process Models</b>	<b>73</b>
6.1	Gaussian Processes Regression: Basic Introductory <code>scikit-learn</code> Example . . . . .	73
6.1.1	Train and Test Data . . . . .	74
6.1.2	Building the Surrogate With <code>Sklearn</code> . . . . .	74
6.1.3	Plotting the <code>SklearnModel</code> . . . . .	74

6.1.4	The <code>spotPython</code> Version . . . . .	75
6.1.5	Visualizing the Differences Between the <code>spotPython</code> and the <code>sklearn</code> Model Fits . . . . .	76
6.2	Exercises . . . . .	77
6.2.1	<code>Schonlau</code> Example Function . . . . .	77
6.2.2	<code>Forrester</code> Example Function . . . . .	77
6.2.3	<code>fun_runge</code> Function (1-dim) . . . . .	78
6.2.4	<code>fun_cubed</code> (1-dim) . . . . .	79
6.2.5	The Effect of Noise . . . . .	79
<b>7</b>	<b>Expected Improvement</b>	<b>80</b>
7.1	Example: Spot and the 1-dim Sphere Function . . . . .	80
7.1.1	The Objective Function: 1-dim Sphere . . . . .	80
7.1.2	Results . . . . .	82
7.2	Same, but with EI as <code>infill_criterion</code> . . . . .	83
7.3	Non-isotropic Kriging . . . . .	85
7.4	Using <code>sklearn</code> Surrogates . . . . .	88
7.4.1	The spot Loop . . . . .	88
7.4.2	<code>spot</code> : The Initial Model . . . . .	89
7.4.3	<code>Init</code> : Build Initial Design . . . . .	91
7.4.4	Evaluate . . . . .	93
7.4.5	Build Surrogate . . . . .	93
7.4.6	A Simple Predictor . . . . .	93
7.5	Gaussian Processes regression: basic introductory example . . . . .	93
7.6	The Surrogate: Using <code>scikit-learn</code> models . . . . .	96
7.7	Additional Examples . . . . .	99
7.7.1	Optimize on Surrogate . . . . .	103
7.7.2	Evaluate on Real Objective . . . . .	103
7.7.3	Impute / Infill new Points . . . . .	103
7.8	Tests . . . . .	103
7.9	EI: The Famous Schonlau Example . . . . .	104
7.10	EI: The Forrester Example . . . . .	106
7.11	Noise . . . . .	109
7.12	Cubic Function . . . . .	112
7.13	Factors . . . . .	118
<b>8</b>	<b>Hyperparameter Tuning and Noise</b>	<b>120</b>
8.1	Example: Spot and the Noisy Sphere Function . . . . .	120
8.1.1	The Objective Function: Noisy Sphere . . . . .	120
8.2	Print the Results . . . . .	127
8.3	Noise and Surrogates: The Nugget Effect . . . . .	128
8.3.1	The Noisy Sphere . . . . .	128

8.4 Exercises . . . . .	131
8.4.1 Noisy <code>fun_cubed</code> . . . . .	131
8.4.2 <code>fun_runge</code> . . . . .	132
8.4.3 <code>fun_forrester</code> . . . . .	132
8.4.4 <code>fun_xsin</code> . . . . .	132
<b>9 Handling Noise: Optimal Computational Budget Allocation in Spot</b>	<b>133</b>
9.1 Example: Spot, OCBA, and the Noisy Sphere Function . . . . .	133
9.1.1 The Objective Function: Noisy Sphere . . . . .	133
9.2 Print the Results . . . . .	139
9.3 Noise and Surrogates: The Nugget Effect . . . . .	139
9.3.1 The Noisy Sphere . . . . .	139
9.4 Exercises . . . . .	142
9.4.1 Noisy <code>fun_cubed</code> . . . . .	142
9.4.2 <code>fun_runge</code> . . . . .	143
9.4.3 <code>fun_forrester</code> . . . . .	143
9.4.4 <code>fun_xsin</code> . . . . .	143
<b>II Hyperparameter Tuning</b>	<b>144</b>
<b>10 HPT: sklearn SVC on Moons Data</b>	<b>145</b>
10.1 Step 1: Setup . . . . .	145
10.2 Step 2: Initialization of the Empty <code>fun_control</code> Dictionary . . . . .	145
10.3 Step 3: SKlearn Load Data (Classification) . . . . .	146
10.4 Step 4: Specification of the Preprocessing Model . . . . .	148
10.5 Step 5: Select Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	148
10.6 Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	151
10.6.1 Modify hyperparameter of type numeric and integer (boolean) . . . . .	151
10.6.2 Modify hyperparameter of type factor . . . . .	152
10.6.3 Optimizers . . . . .	152
10.7 Step 7: Selection of the Objective (Loss) Function . . . . .	152
10.7.1 Predict Classes or Class Probabilities . . . . .	153
10.8 Step 8: Calling the SPOT Function . . . . .	153
10.8.1 Preparing the SPOT Call . . . . .	153
10.8.2 The Objective Function . . . . .	154
10.8.3 Run the Spot Optimizer . . . . .	154
10.8.4 Starting the Hyperparameter Tuning . . . . .	155
10.9 Step 9: Results . . . . .	156
10.9.1 Show variable importance . . . . .	158
10.9.2 Get Default Hyperparameters . . . . .	158
10.9.3 Get SPOT Results . . . . .	159

10.9.4	Plot: Compare Predictions . . . . .	160
10.9.5	Detailed Hyperparameter Plots . . . . .	162
10.9.6	Parallel Coordinates Plot . . . . .	163
10.9.7	Plot all Combinations of Hyperparameters . . . . .	163
<b>11</b>	<b>river Hyperparameter Tuning: Hoeffding Adaptive Tree Regressor with Friedman Drift Data</b>	<b>164</b>
11.1	Setup . . . . .	164
11.2	Initialization of the <code>fun_control</code> Dictionary . . . . .	165
11.3	Load Data: The Friedman Drift Data . . . . .	166
11.4	Specification of the Preprocessing Model . . . . .	167
11.5	SelectSelect Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	167
11.6	Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	168
11.7	Selection of the Objective Function . . . . .	169
11.8	Calling the SPOT Function . . . . .	171
11.8.1	Prepare the SPOT Parameters . . . . .	171
11.8.2	The Objective Function . . . . .	171
11.8.3	Run the Spot Optimizer . . . . .	171
11.8.4	TensorBoard . . . . .	173
11.8.5	Results . . . . .	175
11.9	The Larger Data Set . . . . .	176
11.10	Get Default Hyperparameters . . . . .	177
11.10.1	Show Predictions . . . . .	179
11.11	Get SPOT Results . . . . .	180
11.12	Visualize Regression Trees . . . . .	183
11.12.1	Spot Model . . . . .	183
11.13	Detailed Hyperparameter Plots . . . . .	184
11.14	Parallel Coordinates Plots . . . . .	187
11.15	Plot all Combinations of Hyperparameters . . . . .	187
<b>12</b>	<b>HPT: PyTorch With spotPython and Ray Tune on CIFAR10</b>	<b>188</b>
12.1	Step 1: Setup . . . . .	189
12.2	Step 2: Initialization of the <code>fun_control</code> Dictionary . . . . .	190
12.3	Step 3: PyTorch Data Loading . . . . .	191
12.4	Step 4: Specification of the Preprocessing Model . . . . .	191
12.5	Step 5: Select Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	192
12.5.1	The <code>Net_Core</code> class . . . . .	194
12.5.2	Comparison of the Approach Described in the PyTorch Tutorial With <code>spotPython</code> . . . . .	194
12.5.3	The Search Space: Hyperparameters . . . . .	195
12.5.4	Configuring the Search Space With Ray Tune . . . . .	195
12.5.5	Configuring the Search Space With <code>spotPython</code> . . . . .	196

12.6 Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	198
12.6.1 Optimizers . . . . .	199
12.7 Step 7: Selection of the Objective (Loss) Function . . . . .	201
12.7.1 Evaluation: Data Splitting . . . . .	201
12.7.2 Hold-out Data Split . . . . .	201
12.7.3 Cross-Validation . . . . .	202
12.7.4 Overview of the Evaluation Settings . . . . .	203
12.7.5 Evaluation: Loss Functions and Metrics . . . . .	204
12.8 Step 8: Calling the SPOT Function . . . . .	205
12.8.1 Preparing the SPOT Call . . . . .	205
12.8.2 The Objective Function <code>fun_torch</code> . . . . .	206
12.8.3 Using Default Hyperparameters or Results from Previous Runs . . . . .	206
12.8.4 Starting the Hyperparameter Tuning . . . . .	206
12.9 Step 9: Tensorboard . . . . .	213
12.9.1 Tensorboard: Start Tensorboard . . . . .	213
12.9.2 Saving the State of the Notebook . . . . .	213
12.10 Step 10: Results . . . . .	215
12.10.1 Get the Tuned Architecture (SPOT Results) . . . . .	217
12.10.2 Get Default Hyperparameters . . . . .	217
12.10.3 Evaluation of the Default Architecture . . . . .	218
12.10.4 Evaluation of the Tuned Architecture . . . . .	219
12.10.5 Detailed Hyperparameter Plots . . . . .	221
12.11 Summary and Outlook . . . . .	224
12.12 Appendix . . . . .	225
12.12.1 Sample Output From Ray Tune's Run . . . . .	225
<b>13 HPT: <code>sklearn RandomForestClassifier</code> VBDP Data</b>	<b>227</b>
13.1 Step 1: Setup . . . . .	227
13.2 Step 2: Initialization of the Empty <code>fun_control</code> Dictionary . . . . .	228
13.3 Step 3: PyTorch Data Loading . . . . .	228
13.3.1 Load Data: Classification VBDP . . . . .	228
13.3.2 Holdout Train and Test Data . . . . .	229
13.4 Step 4: Specification of the Preprocessing Model . . . . .	230
13.5 Step 5: Select Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	231
13.6 Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	232
13.6.1 Modify hyperparameter of type numeric and integer (boolean) . . . . .	232
13.6.2 Modify hyperparameter of type factor . . . . .	233
13.6.3 Optimizers . . . . .	233
13.6.4 Selection of the Objective: Metric and Loss Functions . . . . .	233
13.7 Step 7: Selection of the Objective (Loss) Function . . . . .	234
13.7.1 Metric Function . . . . .	234

13.7.2	Evaluation on Hold-out Data . . . . .	235
13.7.3	OOB Score . . . . .	235
13.8	Step 8: Calling the SPOT Function . . . . .	236
13.8.1	Preparing the SPOT Call . . . . .	236
13.8.2	The Objective Function . . . . .	237
13.8.3	Run the Spot Optimizer . . . . .	237
13.9	Step 9: Tensorboard . . . . .	240
13.10	Step 10: Results . . . . .	240
13.10.1	Show variable importance . . . . .	241
13.10.2	Get Default Hyperparameters . . . . .	241
13.10.3	Get SPOT Results . . . . .	242
13.10.4	Evaluate SPOT Results . . . . .	243
13.10.5	Handling Non-deterministic Results . . . . .	244
13.10.6	Evalution of the Default Hyperparameters . . . . .	244
13.10.7	Plot: Compare Predictions . . . . .	245
13.10.8	Cross-validated Evaluations . . . . .	247
13.10.9	Detailed Hyperparameter Plots . . . . .	248
13.10.10	Parallel Coordinates Plot . . . . .	252
13.10.11	Plot all Combinations of Hyperparameters . . . . .	252
<b>14</b>	<b>HPT: sklearn XGB Classifier VBDP Data</b>	<b>253</b>
14.1	Step 1: Setup . . . . .	253
14.2	Step 2: Initialization of the Empty <code>fun_control</code> Dictionary . . . . .	254
14.3	Step 3: PyTorch Data Loading . . . . .	254
14.3.1	1. Load Data: Classification VBDP . . . . .	254
14.3.2	Holdout Train and Test Data . . . . .	255
14.4	Step 4: Specification of the Preprocessing Model . . . . .	256
14.5	Step 5: Select Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	256
14.6	Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	258
14.6.1	Modify hyperparameter of type numeric and integer (boolean) . . . . .	258
14.6.2	Modify hyperparameter of type factor . . . . .	259
14.6.3	Optimizers . . . . .	259
14.7	Step 7: Selection of the Objective (Loss) Function . . . . .	259
14.7.1	Evaluation . . . . .	259
14.7.2	Selection of the Objective: Metric and Loss Functions . . . . .	259
14.7.3	Loss Function . . . . .	260
14.7.4	Metric Function . . . . .	260
14.7.5	Evaluation on Hold-out Data . . . . .	261
14.8	Step 8: Calling the SPOT Function . . . . .	262
14.8.1	Preparing the SPOT Call . . . . .	262
14.8.2	The Objective Function . . . . .	262
14.8.3	Run the Spot Optimizer . . . . .	263

14.9 Step 9: Tensorboard . . . . .	265
14.10 Step 10: Results . . . . .	265
14.10.1 Show variable importance . . . . .	266
14.10.2 Get Default Hyperparameters . . . . .	266
14.10.3 Get SPOT Results . . . . .	267
14.10.4 Evaluate SPOT Results . . . . .	268
14.10.5 Handling Non-deterministic Results . . . . .	269
14.10.6 Evaluation of the Default Hyperparameters . . . . .	269
14.10.7 Plot: Compare Predictions . . . . .	270
14.10.8 Cross-validated Evaluations . . . . .	272
14.10.9 Detailed Hyperparameter Plots . . . . .	273
14.10.10 Parallel Coordinates Plot . . . . .	276
14.10.11 Plot all Combinations of Hyperparameters . . . . .	276
<b>15 HPT: sklearn SVC VBDP Data</b>	<b>277</b>
15.1 Step 1: Setup . . . . .	277
15.2 Step 2: Initialization of the Empty <code>fun_control</code> Dictionary . . . . .	278
15.3 Step 3: PyTorch Data Loading . . . . .	278
15.3.1 1. Load Data: Classification VBDP . . . . .	278
15.3.2 Holdout Train and Test Data . . . . .	279
15.4 Step 4: Specification of the Preprocessing Model . . . . .	280
15.5 Step 5: Select Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	280
15.6 Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	282
15.6.1 Modify hyperparameter of type numeric and integer (boolean) . . . . .	282
15.6.2 Modify hyperparameter of type factor . . . . .	282
15.6.3 Optimizers . . . . .	283
15.6.4 Selection of the Objective: Metric and Loss Functions . . . . .	283
15.7 Step 7: Selection of the Objective (Loss) Function . . . . .	283
15.7.1 Metric Function . . . . .	283
15.7.2 Evaluation on Hold-out Data . . . . .	284
15.8 Step 8: Calling the SPOT Function . . . . .	285
15.8.1 Preparing the SPOT Call . . . . .	285
15.8.2 The Objective Function . . . . .	286
15.8.3 Run the Spot Optimizer . . . . .	286
15.9 Step 9: Tensorboard . . . . .	289
15.10 Step 10: Results . . . . .	289
15.10.1 Show variable importance . . . . .	290
15.10.2 Get Default Hyperparameters . . . . .	291
15.10.3 Get SPOT Results . . . . .	292
15.10.4 Evaluate SPOT Results . . . . .	293
15.10.5 Handling Non-deterministic Results . . . . .	294
15.10.6 Evaluation of the Default Hyperparameters . . . . .	294

15.10.7 Plot: Compare Predictions . . . . .	295
15.10.8 Cross-validated Evaluations . . . . .	296
15.10.9 Detailed Hyperparameter Plots . . . . .	297
15.10.10 Parallel Coordinates Plot . . . . .	299
15.10.11 Plot all Combinations of Hyperparameters . . . . .	299
<b>16 HPT: sklearn KNN Classifier VBDP Data</b>	<b>300</b>
16.1 Step 1: Setup . . . . .	300
16.2 Step 2: Initialization of the Empty <code>fun_control</code> Dictionary . . . . .	301
16.2.1 Load Data: Classification VBDP . . . . .	301
16.2.2 Holdout Train and Test Data . . . . .	302
16.3 Step 4: Specification of the Preprocessing Model . . . . .	303
16.4 Step 5: Select Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	303
16.5 Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	305
16.5.1 Modify hyperparameter of type numeric and integer (boolean) . . . . .	305
16.5.2 Modify hyperparameter of type factor . . . . .	305
16.5.3 Optimizers . . . . .	306
16.5.4 Selection of the Objective: Metric and Loss Functions . . . . .	306
16.6 Step 7: Selection of the Objective (Loss) Function . . . . .	306
16.6.1 Metric Function . . . . .	306
16.6.2 Evaluation on Hold-out Data . . . . .	307
16.7 Step 8: Calling the SPOT Function . . . . .	308
16.7.1 Preparing the SPOT Call . . . . .	308
16.7.2 The Objective Function . . . . .	309
16.7.3 Run the Spot Optimizer . . . . .	309
16.8 Step 9: Tensorboard . . . . .	312
16.9 Step 10: Results . . . . .	313
16.9.1 Show variable importance . . . . .	313
16.9.2 Get Default Hyperparameters . . . . .	314
16.9.3 Get SPOT Results . . . . .	315
16.9.4 Evaluate SPOT Results . . . . .	315
16.9.5 Handling Non-deterministic Results . . . . .	316
16.9.6 Evalution of the Default Hyperparameters . . . . .	317
16.9.7 Plot: Compare Predictions . . . . .	317
16.9.8 Cross-validated Evaluations . . . . .	319
16.9.9 Detailed Hyperparameter Plots . . . . .	320
16.9.10 Parallel Coordinates Plot . . . . .	321
16.9.11 Plot all Combinations of Hyperparameters . . . . .	321
<b>17 HPT PyTorch Lightning: VBDP</b>	<b>322</b>
17.1 Step 1: Setup . . . . .	323
17.2 Step 2: Initialization of the <code>fun_control</code> Dictionary . . . . .	323

17.3 Step 3: PyTorch Data Loading . . . . .	324
17.3.1 Lightning Dataset and DataModule . . . . .	324
17.4 Step 4: Preprocessing . . . . .	324
17.5 Step 5: Select the NN Model ( <code>algorithm</code> ) and <code>core_model_hyper_dict</code> . . . . .	325
17.6 Step 6: Modify <code>hyper_dict</code> Hyperparameters for the Selected Algorithm aka <code>core_model</code> . . . . .	325
17.7 Step 7: Data Splitting, the Objective (Loss) Function and the Metric . . . . .	327
17.7.1 Evaluation . . . . .	327
17.7.2 Loss Functions and Metrics . . . . .	327
17.7.3 Metric . . . . .	327
17.8 Step 8: Calling the SPOT Function . . . . .	327
17.8.1 Preparing the SPOT Call . . . . .	327
17.8.2 The Objective Function <code>fun</code> . . . . .	328
17.8.3 Starting the Hyperparameter Tuning . . . . .	328
17.9 Step 9: Tensorboard . . . . .	333
17.10 Step 10: Results . . . . .	333
17.10.1 Get the Tuned Architecture . . . . .	335
17.10.2 Cross Validation With Lightning . . . . .	336
17.10.3 Detailed Hyperparameter Plots . . . . .	340
17.10.4 Parallel Coordinates Plot . . . . .	340
17.10.5 Plot all Combinations of Hyperparameters . . . . .	341
17.10.6 Visualizing the Activation Distribution . . . . .	341
17.11 Submission . . . . .	343
17.12 Appendix . . . . .	345
17.12.1 Differences to the spotPython Approaches for <code>torch</code> , <code>sklearn</code> and <code>river</code> . . . . .	345
17.12.2 Taking a Look at the Data . . . . .	346
17.12.3 The MAPK Metric . . . . .	347
<b>Appendices</b>	<b>348</b>
<b>A Documentation of the Sequential Parameter Optimization</b>	<b>348</b>
A.1 Example: <code>spot</code> . . . . .	348
A.1.1 The Objective Function . . . . .	348
A.1.2 External Parameters . . . . .	350
A.2 The <code>fun_control</code> Dictionary . . . . .	353
A.3 The <code>design_control</code> Dictionary . . . . .	353
A.4 The <code>surrogate_control</code> Dictionary . . . . .	354
A.5 The <code>optimizer_control</code> Dictionary . . . . .	354
A.6 Run . . . . .	355
A.7 Print the Results . . . . .	357
A.8 Show the Progress . . . . .	357
A.9 Visualize the Surrogate . . . . .	357

A.10 Init: Build Initial Design . . . . .	358
A.11 Replicability . . . . .	359
A.12 Surrogates . . . . .	360
A.12.1 A Simple Predictor . . . . .	360
A.13 Demo/Test: Objective Function Fails . . . . .	360
A.14 PyTorch: Detailed Description of the Data Splitting . . . . .	363
A.14.1 Description of the " <code>train_hold_out</code> " Setting . . . . .	363
<b>References</b>	<b>374</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 12 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 12.1 describes the setup of the tuners. Section 12.3 describes the data loading. Section 12.5 describes the model to be tuned. The search space is introduced in Section 12.5.3. Optimizers are presented in Section 12.6.1. How to split the data in train, validation, and test sets is described in Section 12.7.1. The selection of the loss function and metrics is described in Section 12.7.5. Section 12.8.1 describes the preparation of the `spotPython` call. The objective function is described in Section 12.8.2. How to use results from previous runs and default hyperparameter configurations is described in Section 12.8.3. Starting the tuner is shown in Section 12.8.4. TensorBoard can be used to visualize the results as shown in Section 12.9. Results are discussed and explained in Section 12.10. Section 12.11 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 13 describes the hyperparameter tuning of a `random forest classifier` from `scikit-learn` with `spotPython`. Chapter 14 describes the hyperparameter tuning of an `XGBoost classifier` from `scikit-learn` with `spotPython`. Chapter 15 describes the hyperparameter tuning of a `support vector classifier` from `scikit-learn` with `spotPython`. Chapter 16 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 17. 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`.

# **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

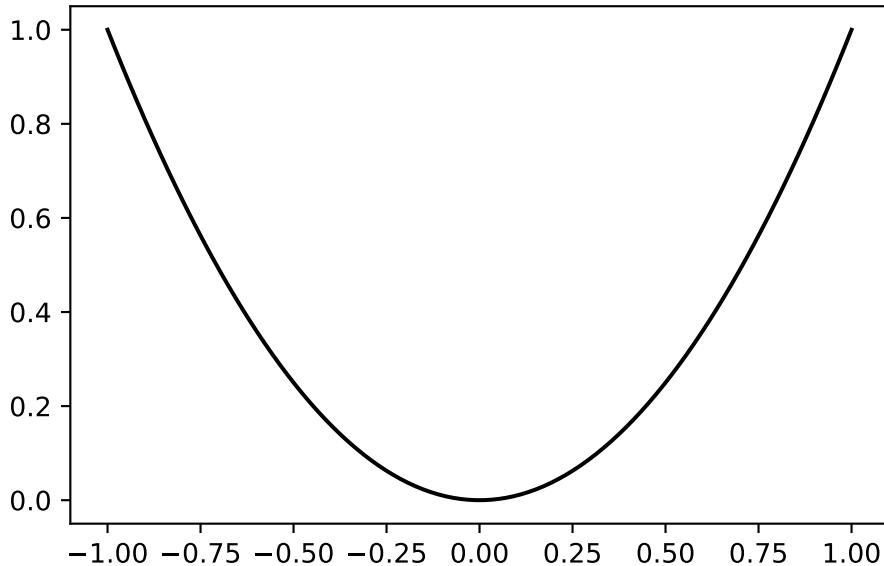
```

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: 7.263311682641849e-09 [#####---] 73.33%
spotPython tuning: 7.263311682641849e-09 [#####----] 80.00%
spotPython tuning: 7.263311682641849e-09 [#####----] 86.67%
spotPython tuning: 7.263311682641849e-09 [#####----] 93.33%
spotPython tuning: 3.696886711914087e-10 [#####----] 100.00% Done...

```

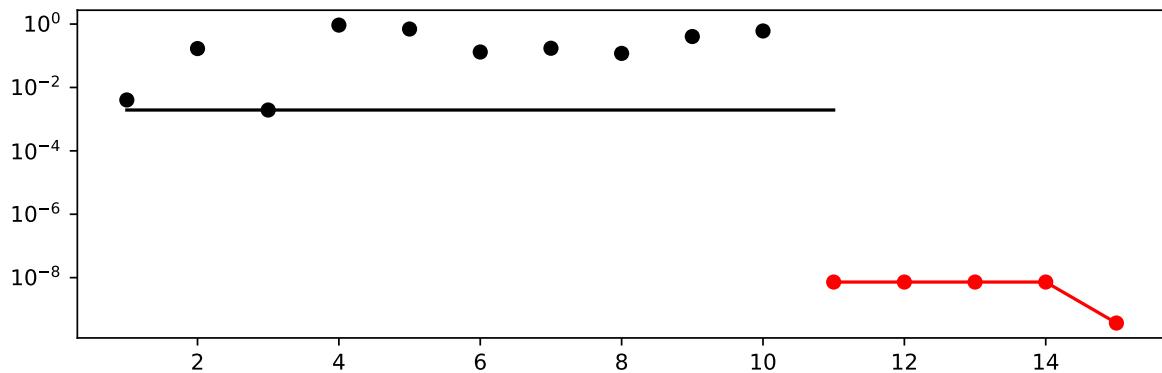
<spotPython.spot.spot.Spot at 0x168361ff0>

```
spot_0.print_results()
```

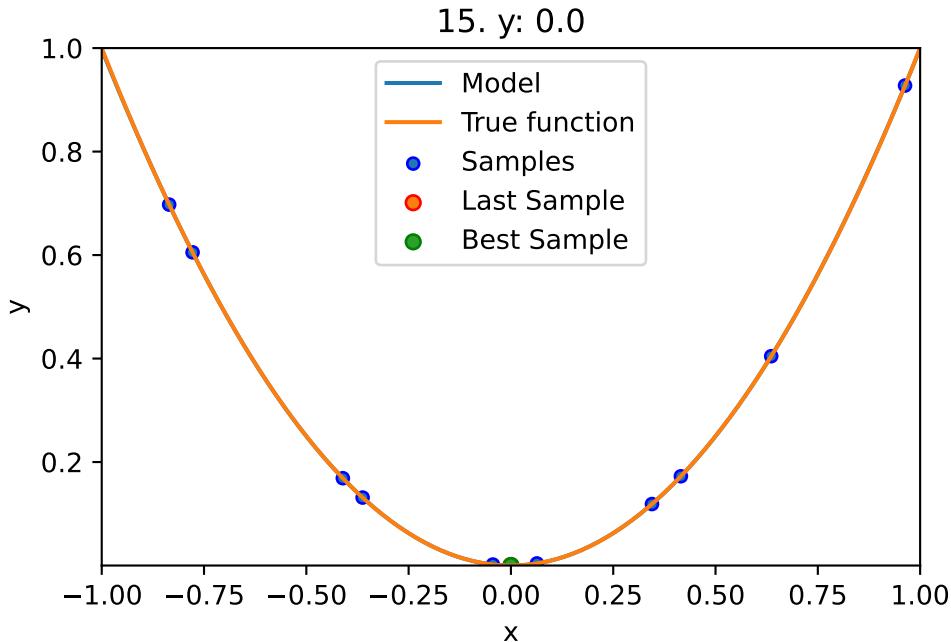
```
min y: 3.696886711914087e-10
x0: 1.922728975158508e-05
```

```
[['x0', 1.922728975158508e-05]]
```

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



```
spot_0.plot_model()
```



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

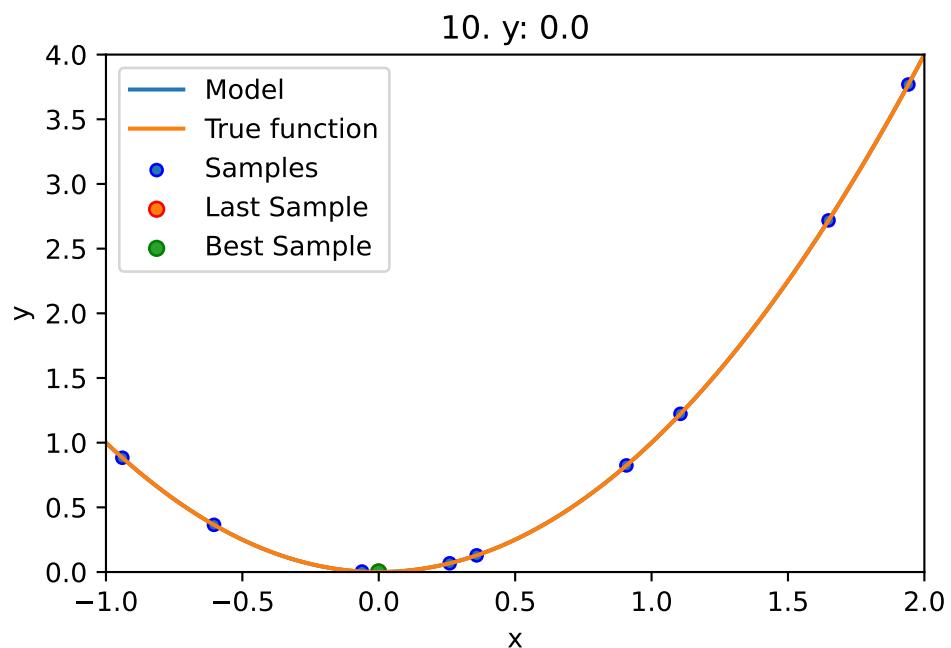
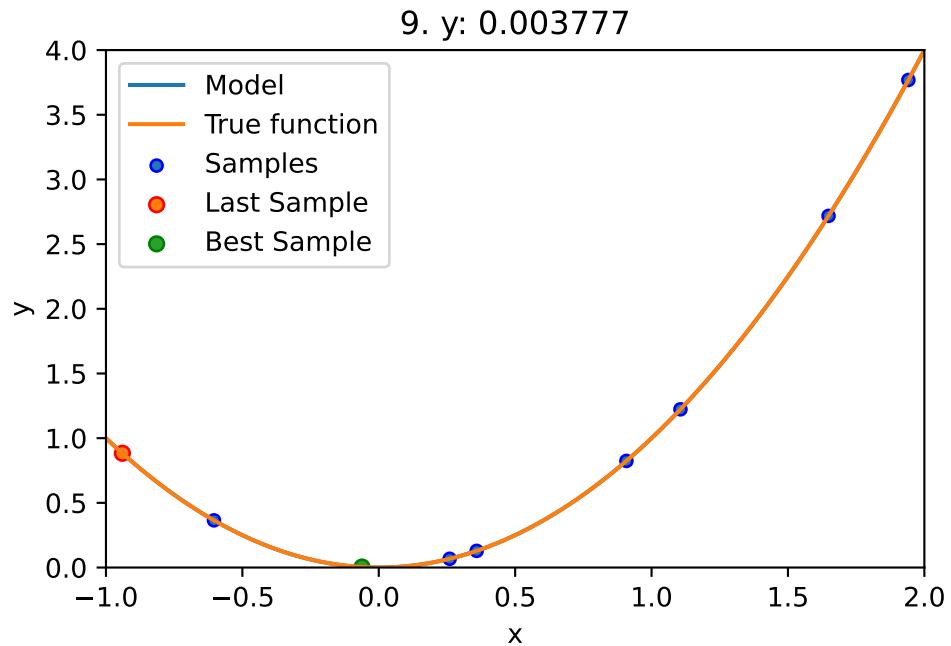
We will modify three parameters:

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: 3.6779240309761575e-07 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x168f98910>
```

## 1.3 Print the Results

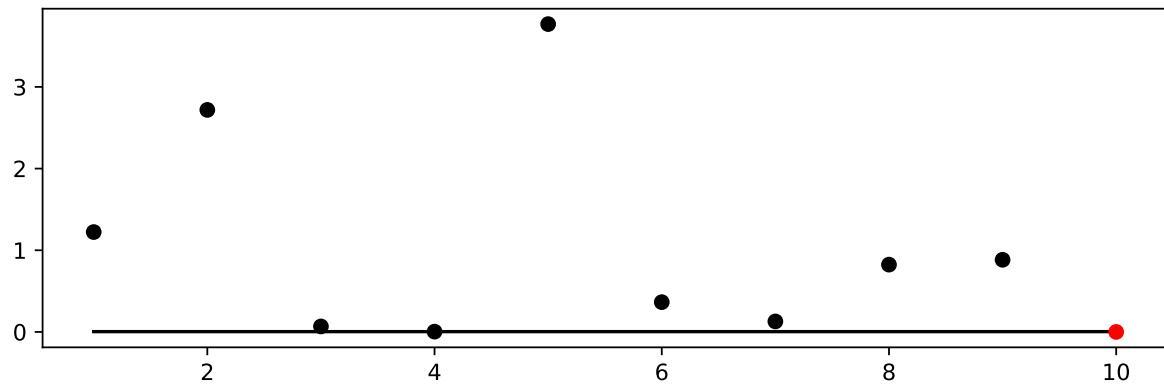
```
spot_1.print_results()
```

```
min y: 3.6779240309761575e-07
x0: -0.0006064589047063418
```

```
[['x0', -0.0006064589047063418]]
```

## 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\_bartz09\_2023-07-17\_18-00-57

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: 2.7705924100183687e-05 [#####----] 60.00%

spotPython tuning: 7.364661789374228e-07 [#####---] 70.00%

spotPython tuning: 7.364661789374228e-07 [#####----] 80.00%

spotPython tuning: 3.5490065465299805e-07 [#####----] 90.00%

spotPython tuning: 7.234315072455918e-09 [#####----] 100.00% Done...

```
<spotPython.spot.spot.Spot at 0x2c439be50>
```

Now we can start TensorBoard in the background. The TensorBoard process will read the TensorBoard files and visualize the hyperparameter tuning process. From the terminal, we can start TensorBoard with the following command:

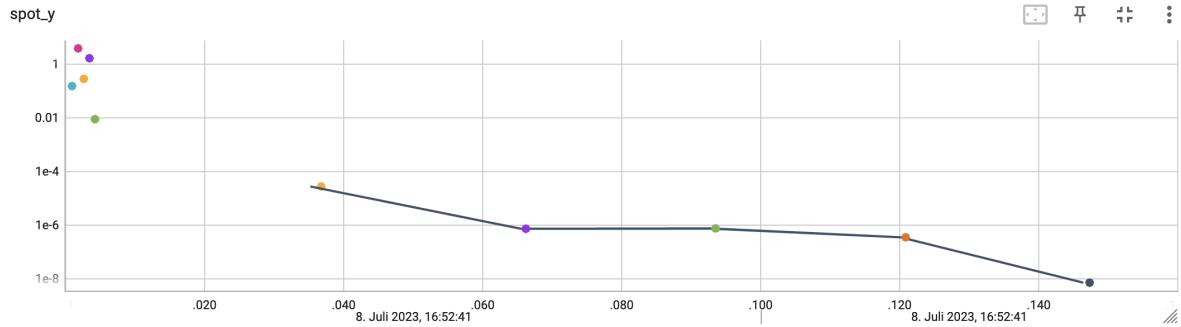
```
tensorboard --logdir=".runs"
```

`logdir` is the directory where the TensorBoard files are stored. In our case, the TensorBoard files are stored in the directory `./runs`.

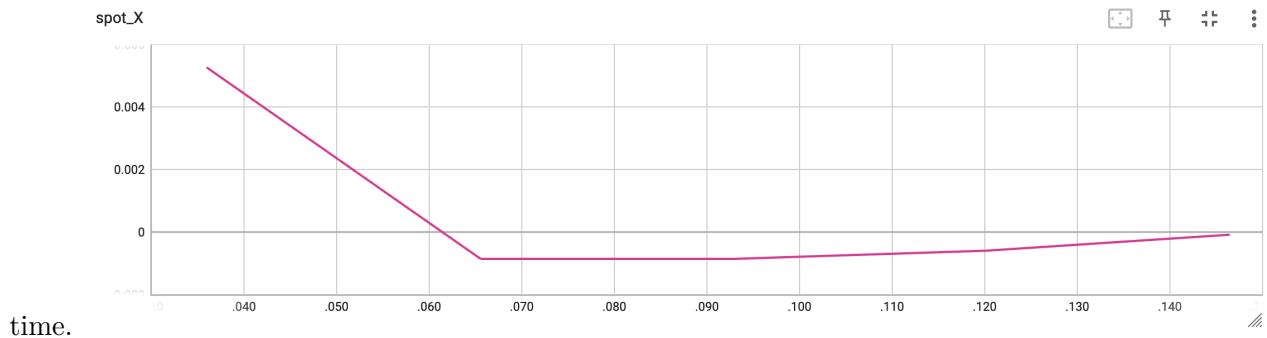
TensorBoard will start a web server on port 6006. We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The first TensorBoard visualization shows the objective function values plotted against the wall time. The wall time is the time that has passed since the start of the hyperparameter tuning process. The five initial design points are shown in the upper left region of the plot. The line visualizes the optimization process.



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



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.

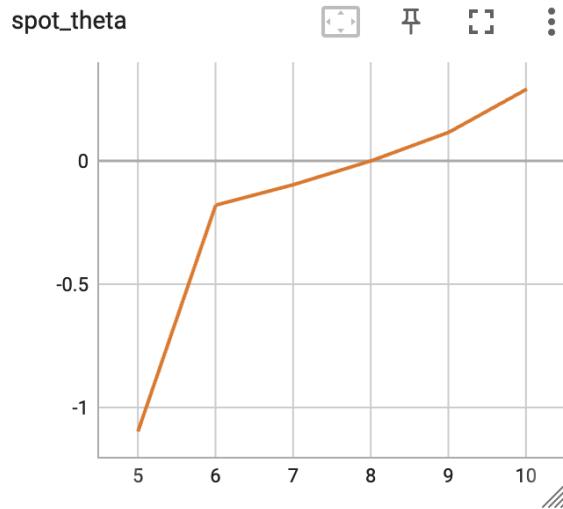


Figure 1.1: TensorBoard visualization of the spotPython process.

## 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\_bartz09\_2023-07-17\_18-02-37

```
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: 0.03443344056467332 [#####---] 73.33%

spotPython tuning: 0.03134865993507926 [#####---] 80.00%

spotPython tuning: 0.0009629342967936851 [#####---] 86.67%

spotPython tuning: 8.541951463966474e-05 [#####---] 93.33%

spotPython tuning: 6.285135731399678e-05 [#####] 100.00% Done...

<spotPython.spot.spot.Spot at 0x28319da50>

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: 6.285135731399678e-05
```

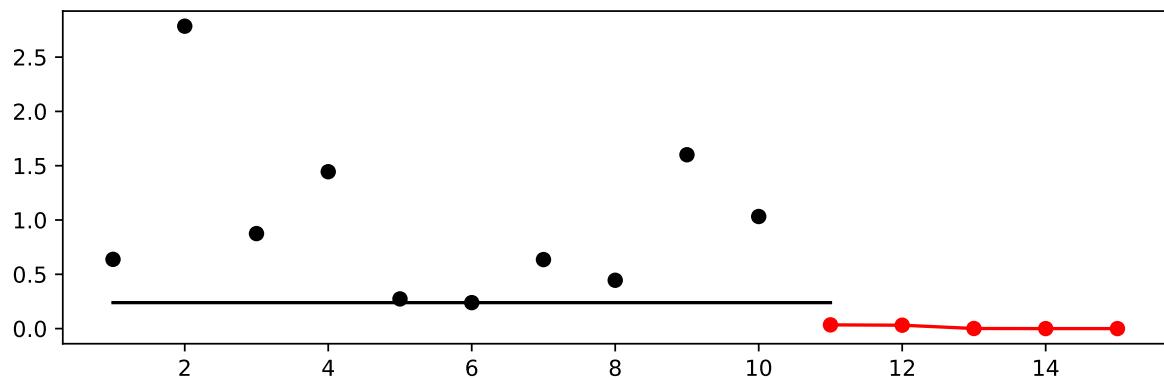
```
Pressure: 0.005236109709736696
```

```
Temp: 0.0019572552655686714
```

```
Lambda: 0.005621713639718905
```

```
[['Pressure', 0.005236109709736696],  
 ['Temp', 0.0019572552655686714],  
 ['Lambda', 0.005621713639718905]]
```

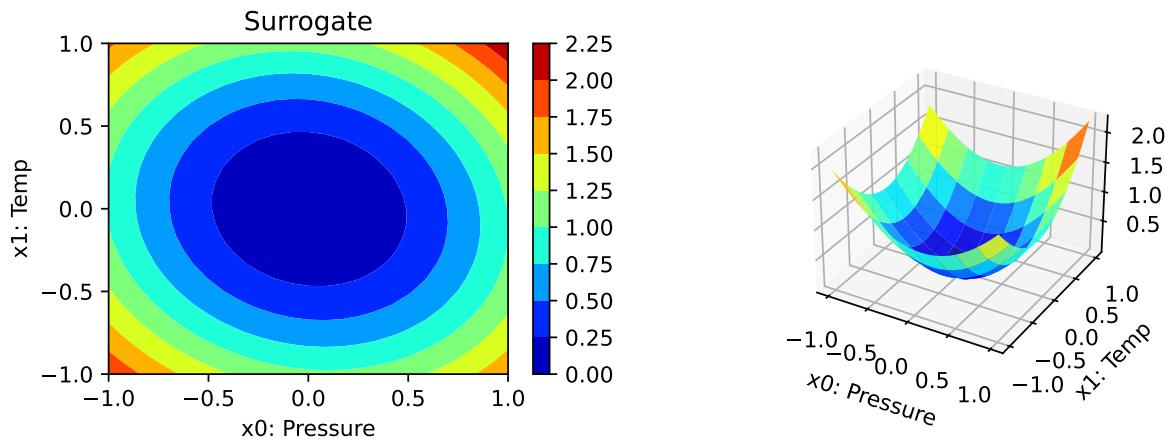
```
spot_3.plot_progress()
```



### 2.1.3 A Contour Plot

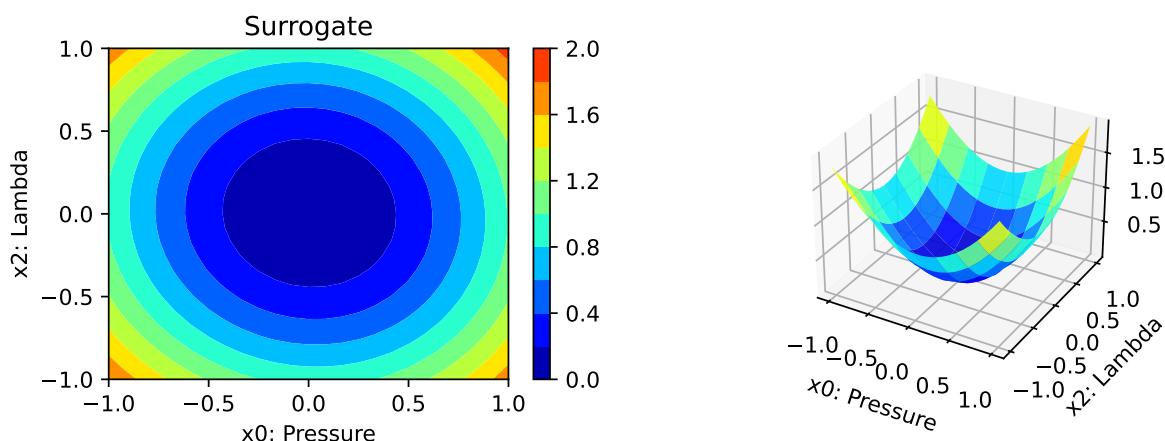
- We can select two dimensions, say  $i = 0$  and  $j = 1$ , and generate a contour plot as follows.
  - Note: We have specified identical `min_z` and `max_z` values to generate comparable plots!

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



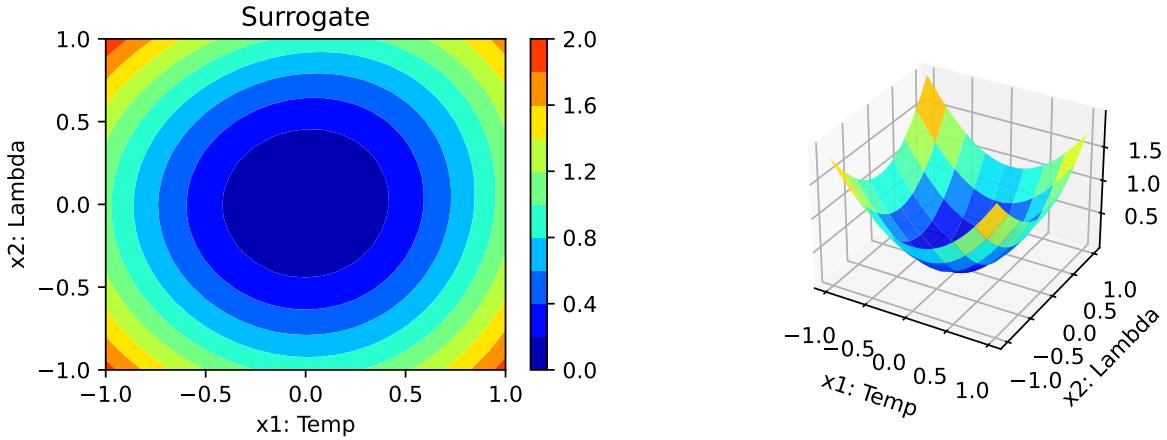
- In a similar manner, we can plot dimension  $i = 0$  and  $j = 2$ :

```
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:

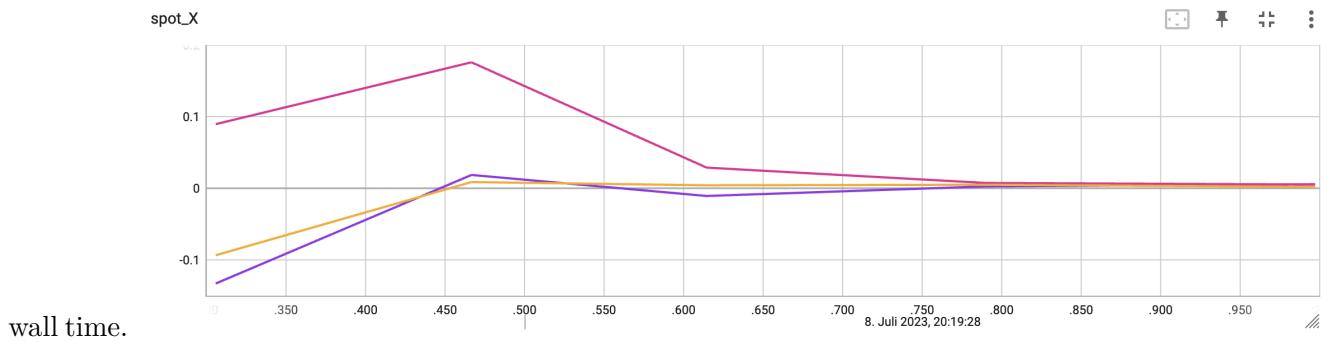
```
spot_3.print_importance()
```

```
Pressure: 99.35185545837122
Temp: 99.99999999999999
Lambda: 94.31627052007231
```

```
[['Pressure', 99.35185545837122],
 ['Temp', 99.99999999999999],
 ['Lambda', 94.31627052007231]]
```

### 2.1.4 TensorBoard

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



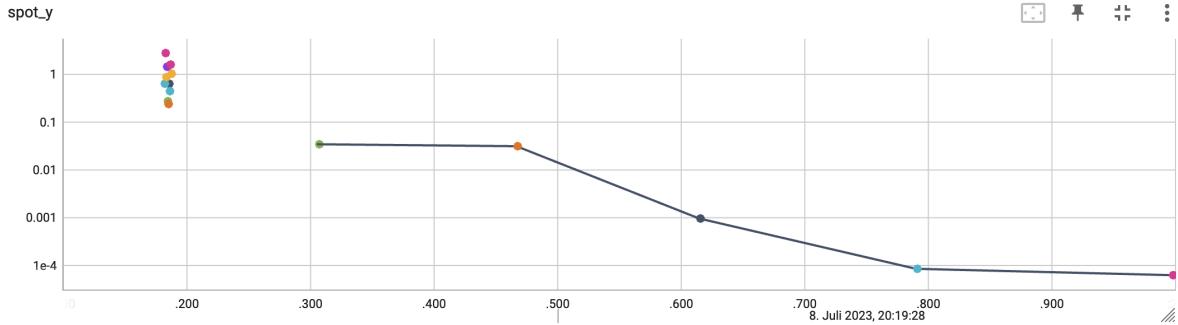


Figure 2.1: TensorBoard visualization of the spotPython process. Objective function values plotted against wall time.

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.

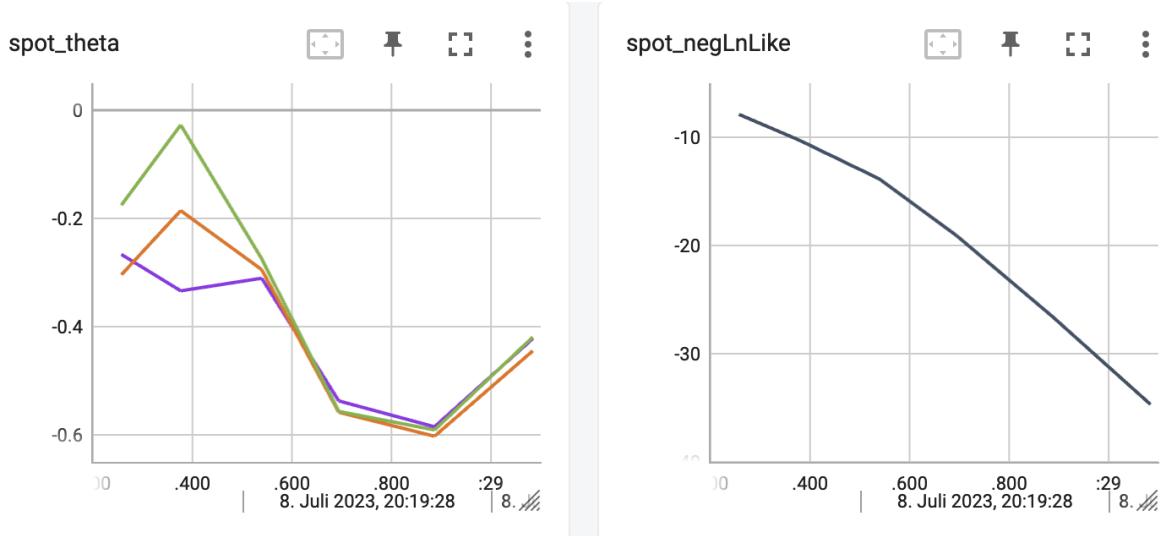


Figure 2.2: TensorBoard visualization of the spotPython surrogate model.

## 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: 2.093282610941807e-05 [#####---] 73.33%
spotPython tuning: 2.093282610941807e-05 [#####----] 80.00%
spotPython tuning: 2.093282610941807e-05 [#####----] 86.67%
spotPython tuning: 2.093282610941807e-05 [#####----] 93.33%
spotPython tuning: 2.093282610941807e-05 [#####----] 100.00% Done...
<spotPython.spot.spot.Spot at 0x14eb299c0>
```

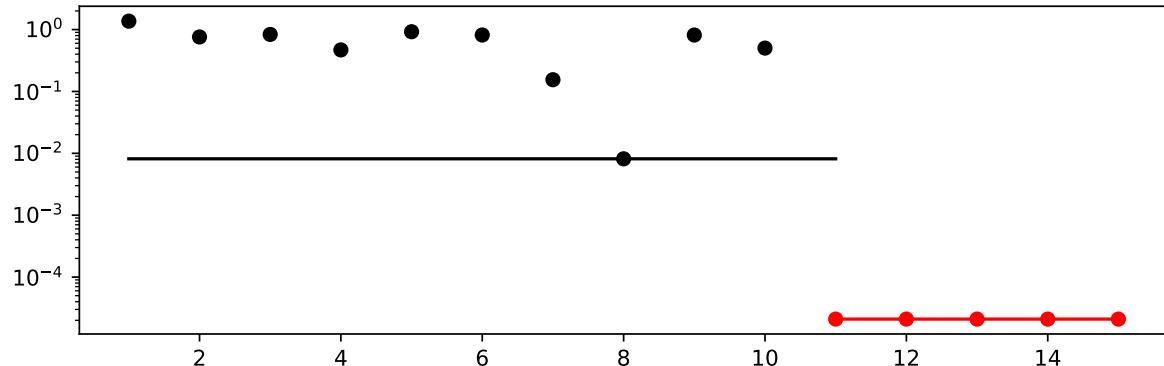
### 3.1.2 Results

```
spot_2.print_results()

min y: 2.093282610941807e-05
x0: 0.0016055267473267492
x1: 0.00428428640184529

[['x0', 0.0016055267473267492], ['x1', 0.00428428640184529]]
```

```
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\_bartz09\_2023-07-17\_08-48-59

```
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: 1.991152317760403e-05 [#####---] 73.33%

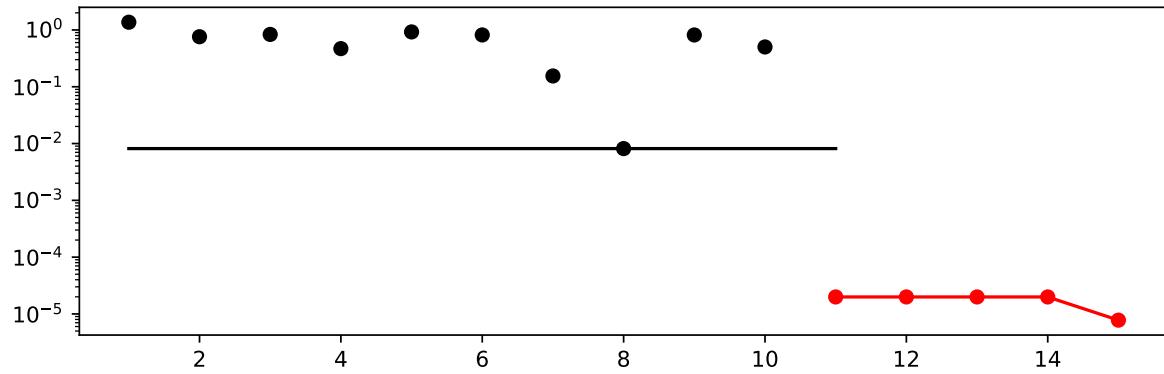
spotPython tuning: 1.991152317760403e-05 [#####---] 80.00%

```
spotPython tuning: 1.991152317760403e-05 [#####-] 86.67%
spotPython tuning: 1.991152317760403e-05 [#####-] 93.33%
spotPython tuning: 7.77061191821505e-06 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2a749fa60>
```

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

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

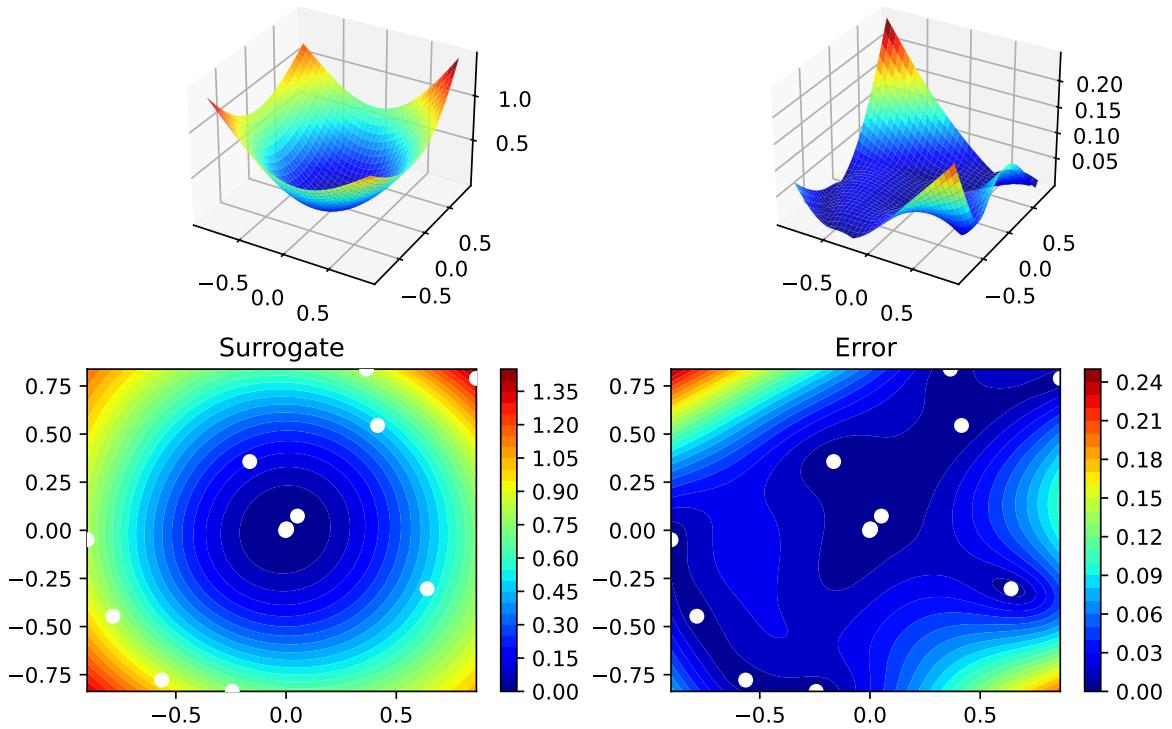


```
spot_2_anisotropic.print_results()
```

```
min y: 7.77061191821505e-06
x0: -0.0024488252797500764
x1: -0.0013318658594137815

[['x0', -0.0024488252797500764], ['x1', -0.0013318658594137815]]
```

```
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([0.19447342, 0.30813872])
```

- 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([0.26287447])
```

### 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.

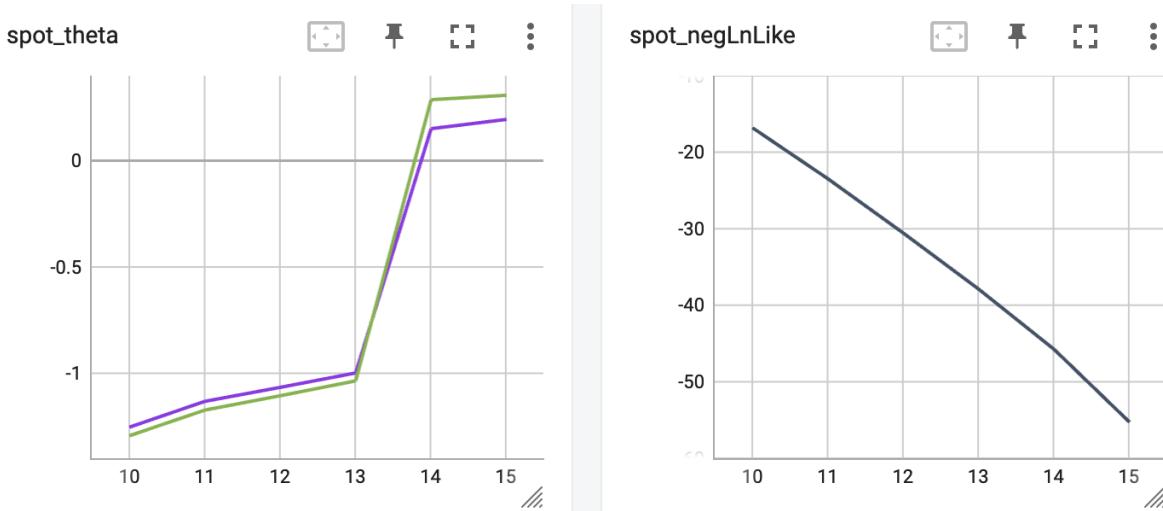


Figure 3.1: TensorBoard visualization of the `spotPython` surrogate model.

## 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\_bartz09\_2023-07-17\_08-49-25

#### 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: 3.4474628349075243 [#####----] 55.00%

spotPython tuning: 3.4474628349075243 [#####----] 60.00%

spotPython tuning: 3.039485786016101 [#####----] 65.00%

```
spotPython tuning: 3.039485786016101 [#####---] 70.00%
spotPython tuning: 1.1632959357427755 [#####---] 75.00%
spotPython tuning: 0.6123887750698636 [#####---] 80.00%
spotPython tuning: 0.4575920097730535 [#####---] 85.00%
spotPython tuning: 0.3982295132785083 [#####---] 90.00%
spotPython tuning: 0.3982295132785083 [#####---] 95.00%
spotPython tuning: 0.3982295132785083 [#####---] 100.00% Done...
<spotPython.spot.spot.Spot at 0x13ab7cca0>
```

#### 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: 0.3982295132785083
x0: 3.135528626303215
x1: 2.2926027772585886

[['x0', 3.135528626303215], ['x1', 2.2926027772585886]]
```

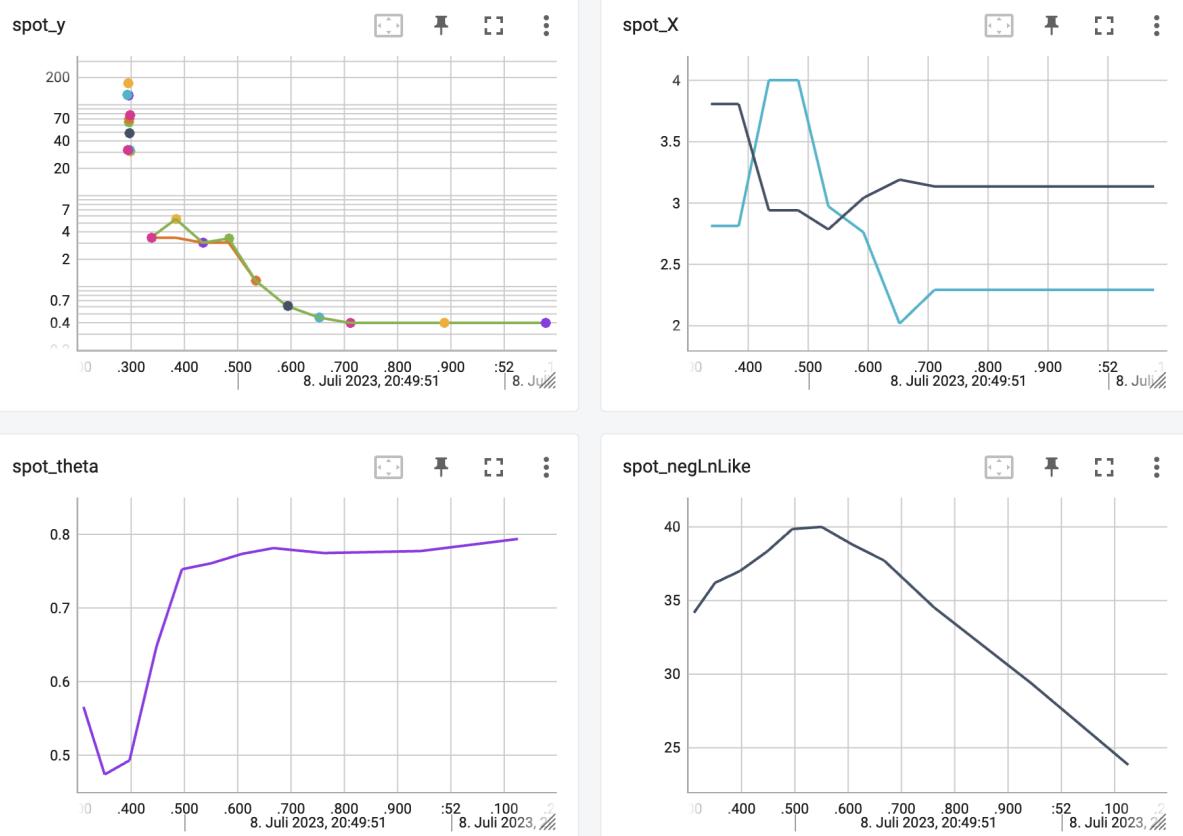
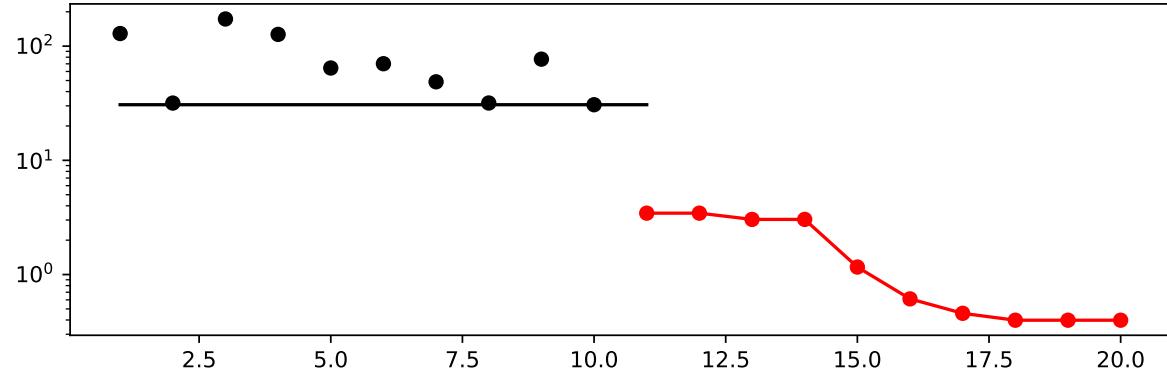


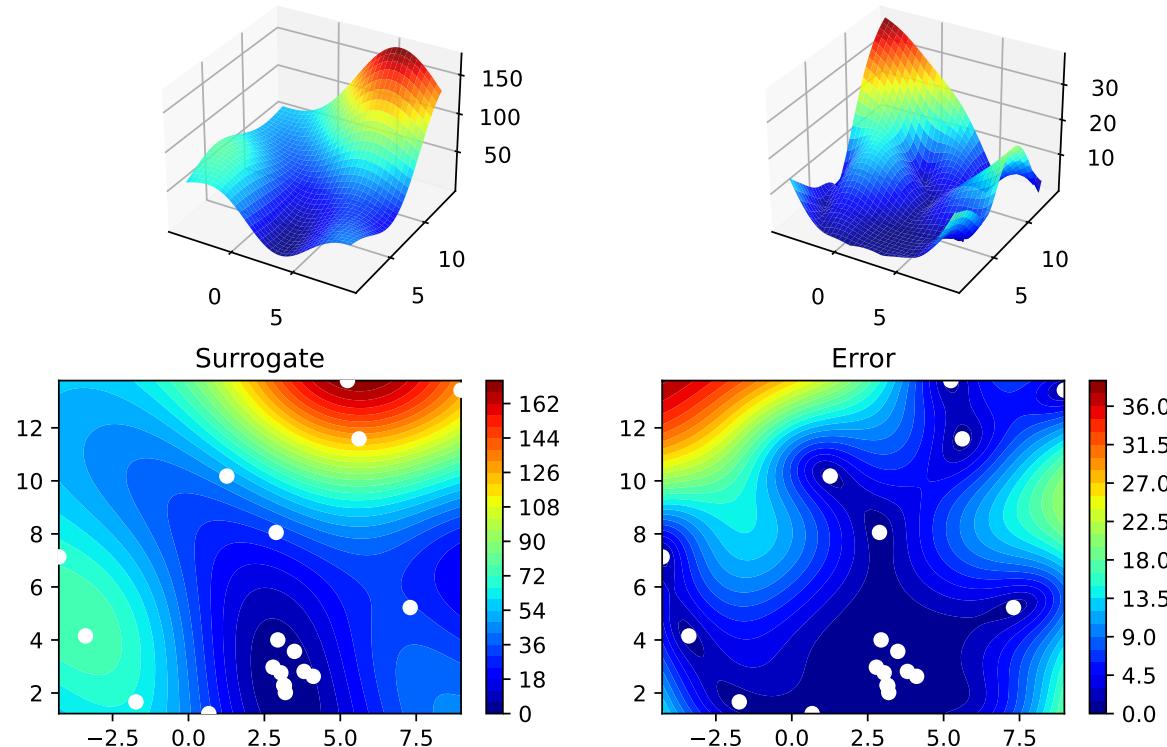
Figure 4.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

#### 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: 18.86511402323416 [#####----] 55.00%
```

```
spotPython tuning: 4.0669082302178285 [#####----] 60.00%
```

```
spotPython tuning: 3.4618162795514635 [#####----] 65.00%
```

```
spotPython tuning: 3.4618162795514635 [#####---] 70.00%
```

```
spotPython tuning: 1.3283163482563598 [#####---] 75.00%
```

```
spotPython tuning: 0.9542592376072765 [#####--] 80.00%
```

```
spotPython tuning: 0.9289433893626615 [#####--] 85.00%
```

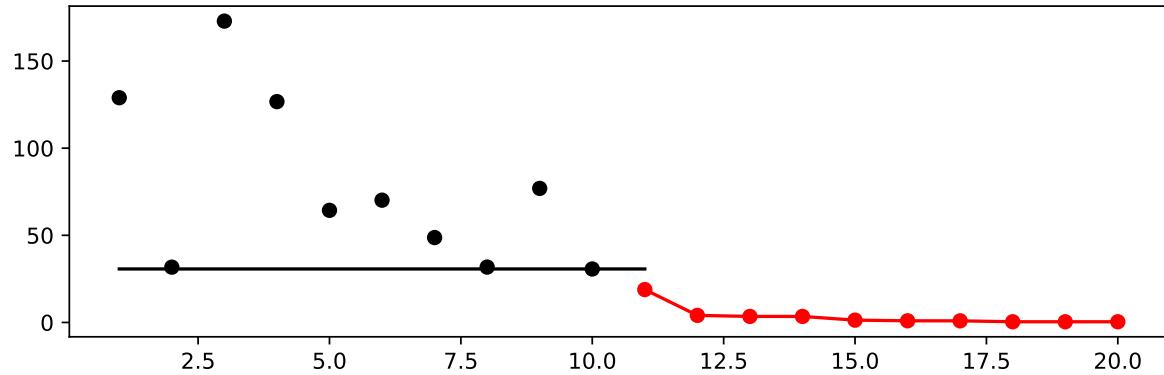
```
spotPython tuning: 0.3981201359931852 [#####--] 90.00%
```

```
spotPython tuning: 0.39799355388506363 [#####] 95.00%
```

```
spotPython tuning: 0.39799355388506363 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2bc4f6020>
```

```
spot_2_GP.plot_progress()
```



```
spot_2_GP.print_results()
```

```
min y: 0.39799355388506363
x0: 3.1460470114516994
x1: 2.2748359190479013
```

```
[['x0', 3.1460470114516994], ['x1', 2.2748359190479013]]
```

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

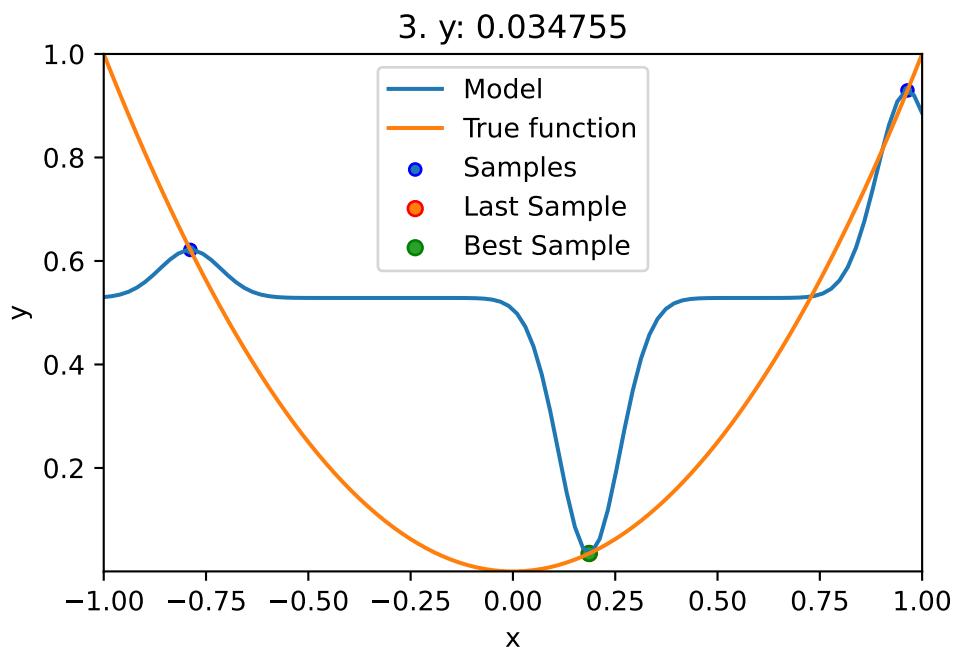
- In this example, we will use an one-dimensional function, which allows us to visualize the optimization process.
  - `show_models= True` is added to the argument list.

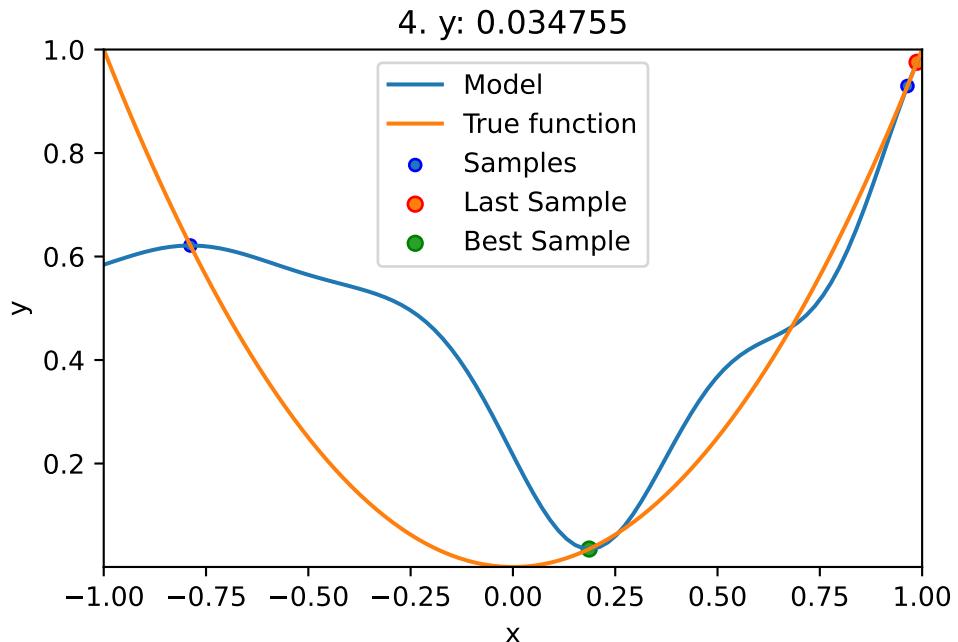
```

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

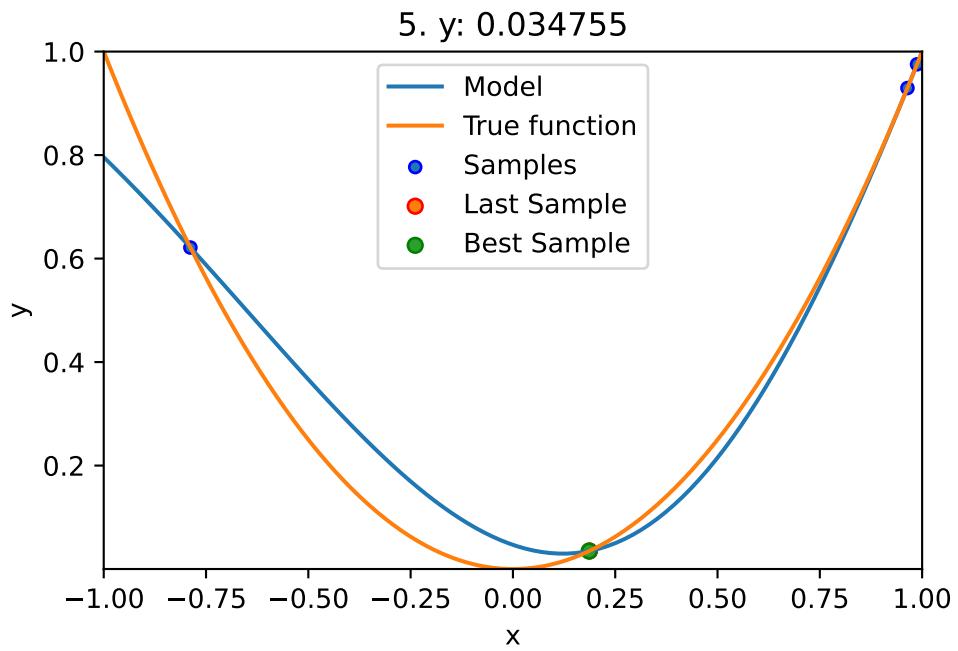
spot_1 = spot.Spot(fun=fun,
                    lower = lower,
                    upper = upper,
                    fun_evals = 10,
                    max_time = inf,
                    seed=123,
                    show_models= True,
                    tolerance_x = np.sqrt(np.spacing(1)),
   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

```

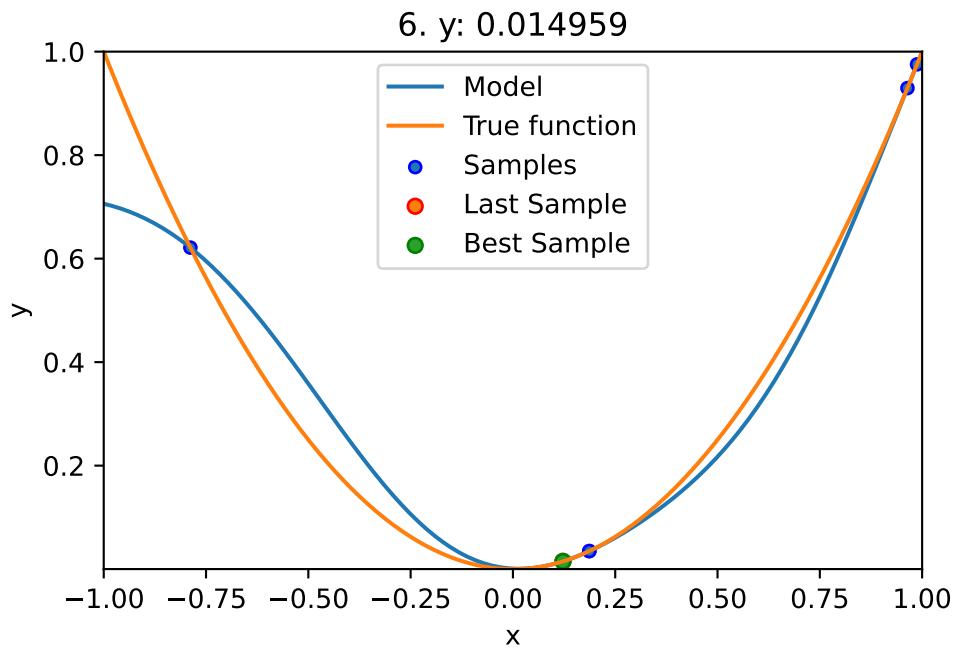




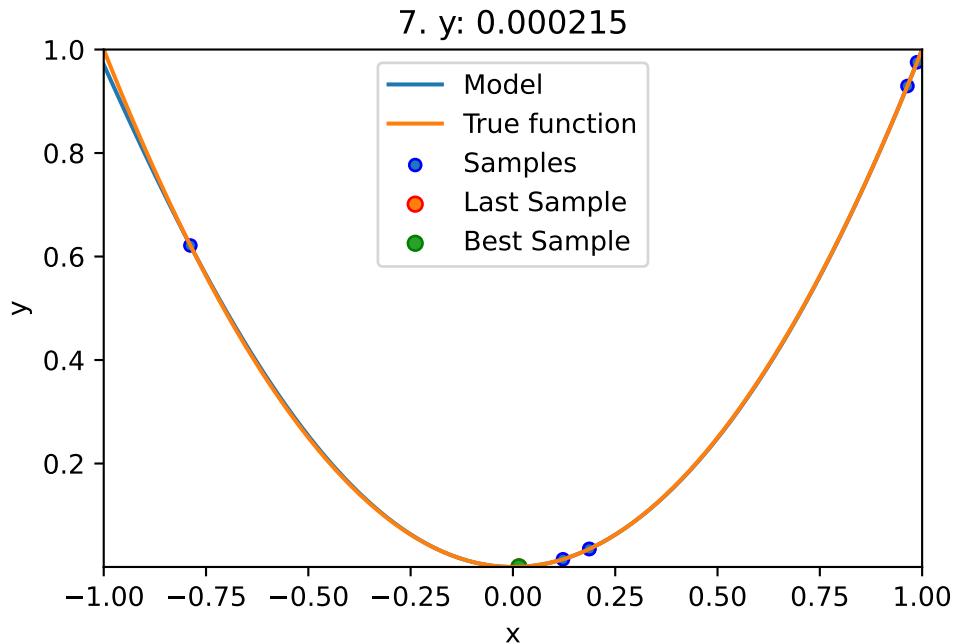
spotPython tuning: 0.03475493366922229 [#####-----] 40.00%



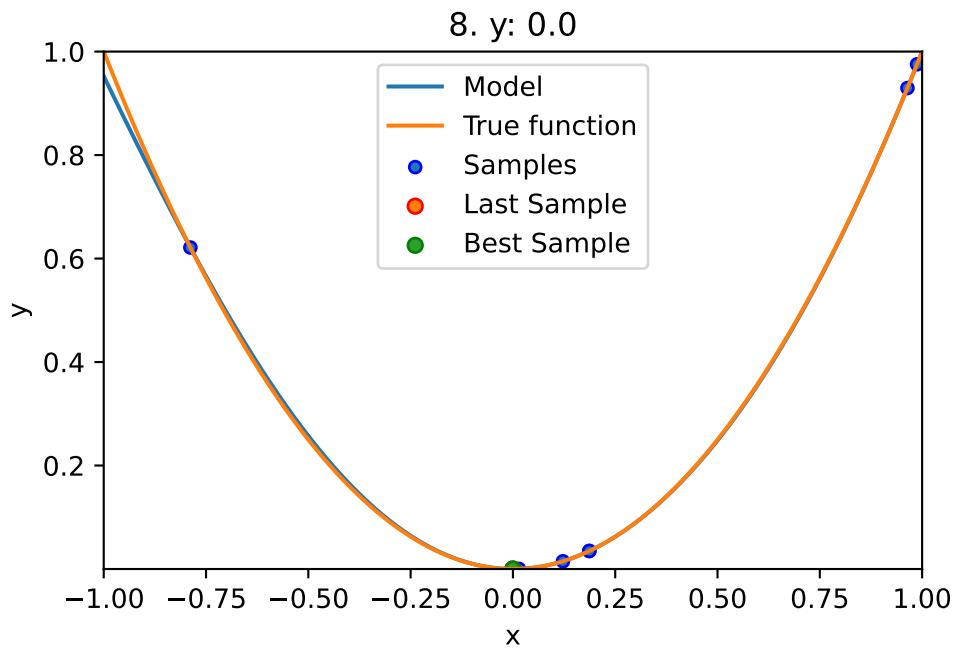
```
spotPython tuning: 0.03475493366922229 [#####----] 50.00%
```



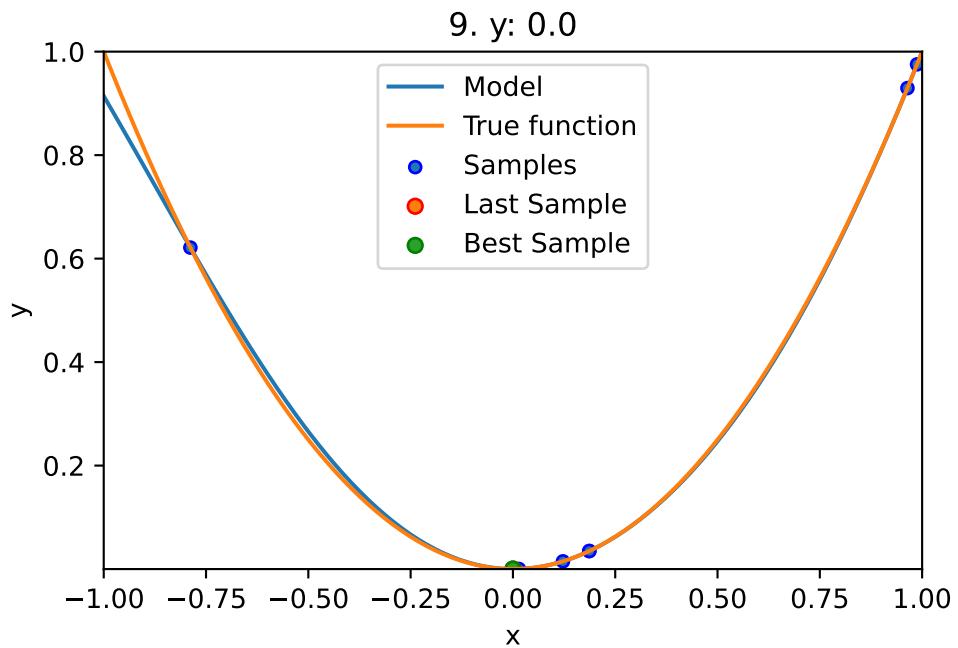
```
spotPython tuning: 0.014958671130600643 [#####----] 60.00%
```



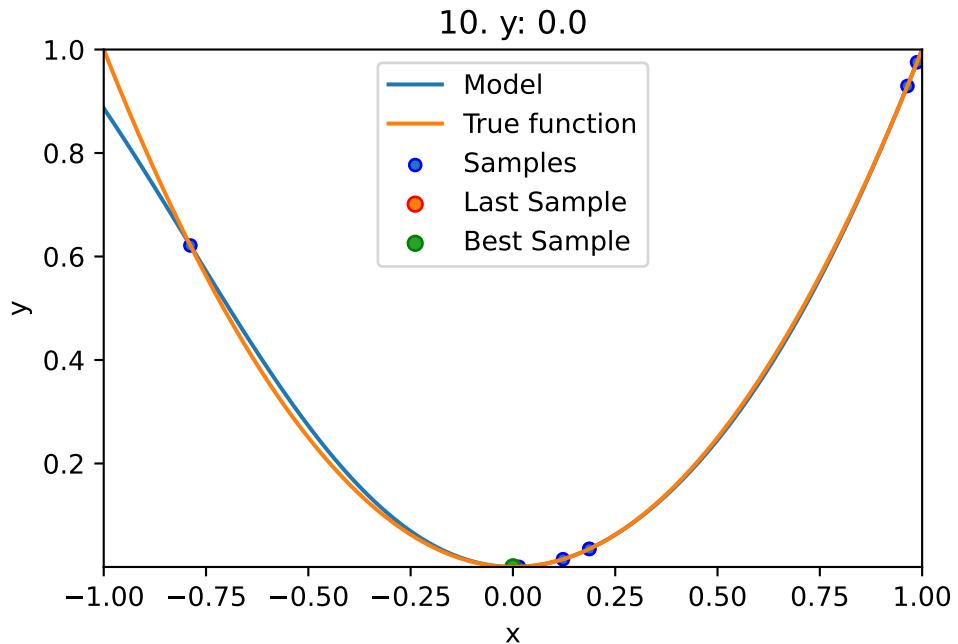
```
spotPython tuning: 0.0002154633036537174 [#####---] 70.00%
```



```
spotPython tuning: 4.41925228274096e-08 [#####--] 80.00%
```



```
spotPython tuning: 4.41925228274096e-08 [#####--] 90.00%
```



```
spotPython tuning: 4.41925228274096e-08 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2bd636b90>
```

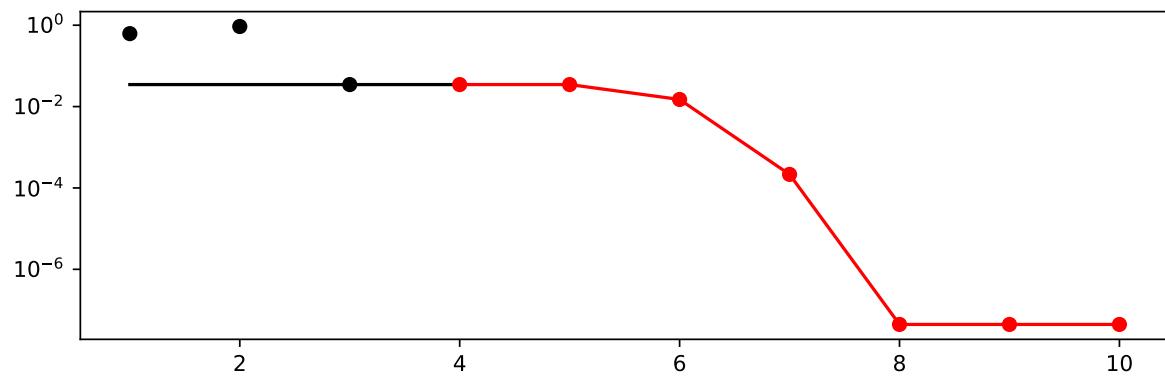
#### 4.3.1 Results

```
spot_1.print_results()
```

```
min y: 4.41925228274096e-08
x0: -0.00021022017702259125
```

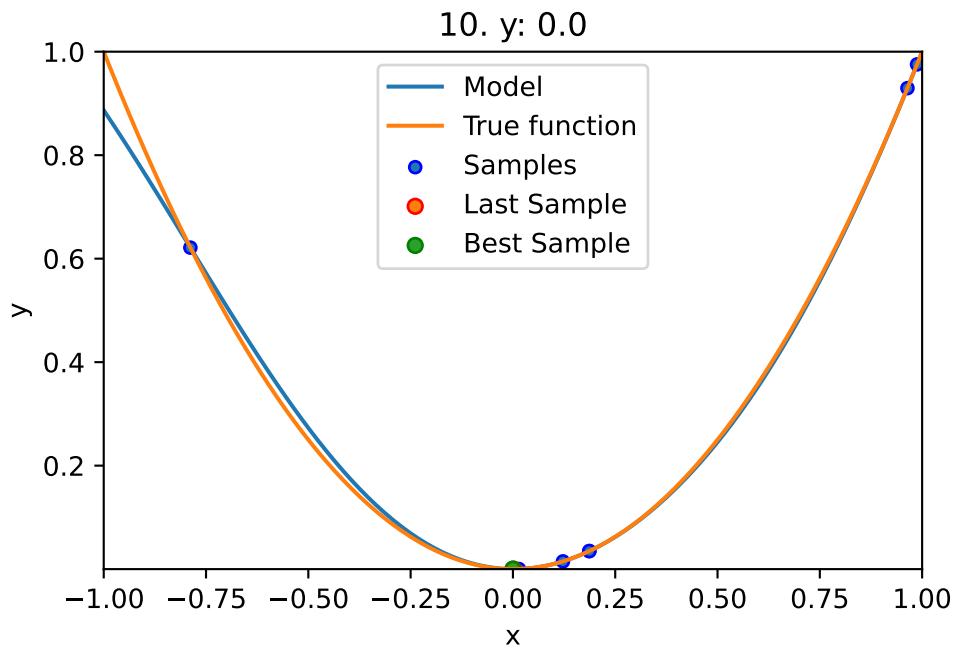
```
[['x0', -0.00021022017702259125]]
```

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



- The method `plot_model` plots the final surrogate:

```
spot_1.plot_model()
```



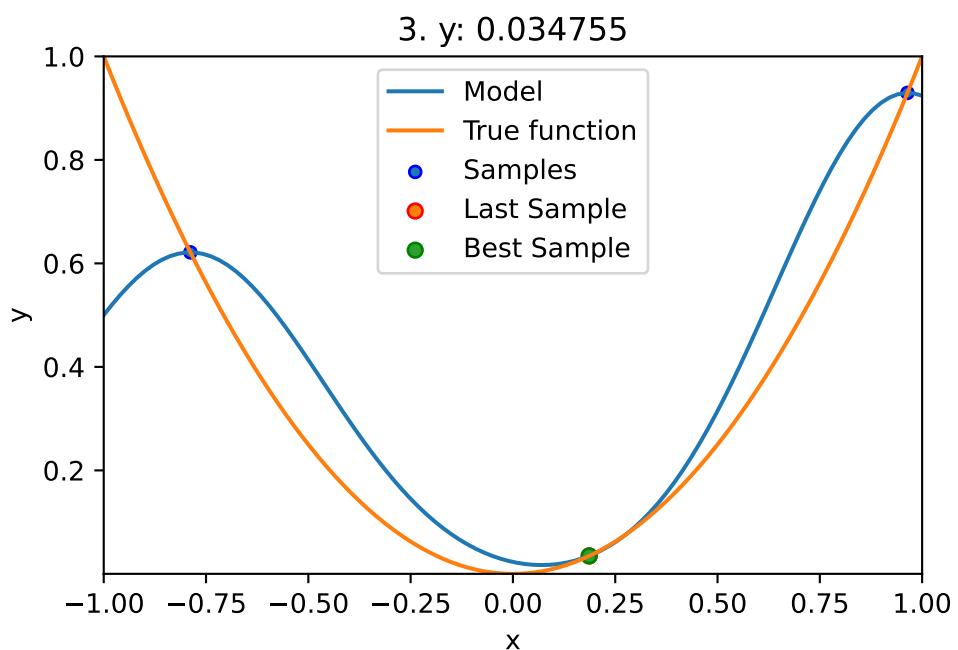
#### 4.4 Example: Sklearn Model GaussianProcess

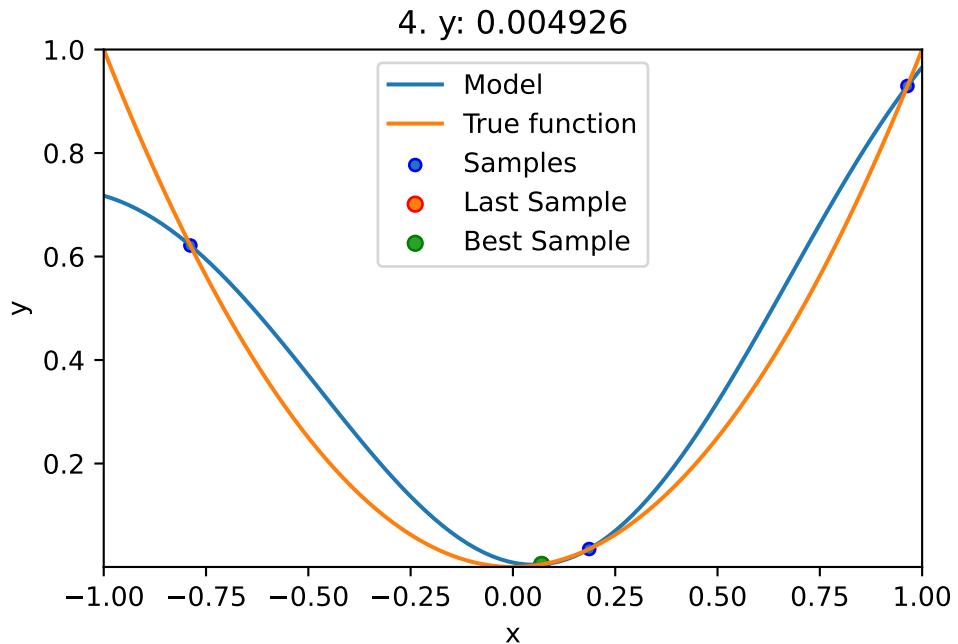
- This example visualizes the search process on the `GaussianProcessRegression` surrogate from `sklearn`.
- Therefore `surrogate = S_GP` is added to the argument list.

```

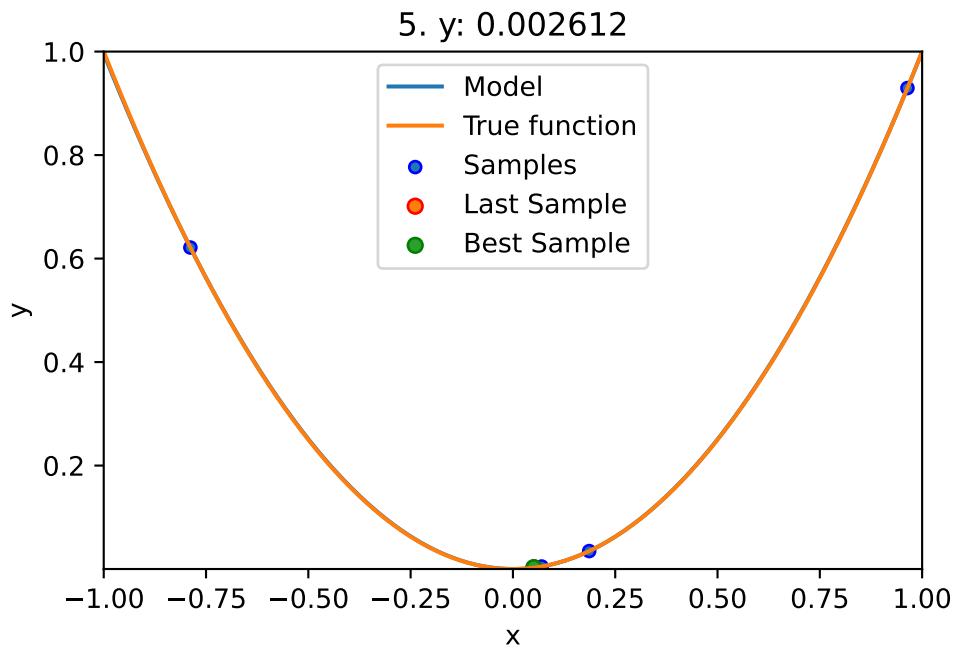
fun = analytical(seed=123).fun_sphere
spot_1_GP = spot.Spot(fun=fun,
                      lower = lower,
                      upper = upper,
                      fun_evals = 10,
                      max_time = inf,
                      seed=123,
                      show_models= True,
        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

```

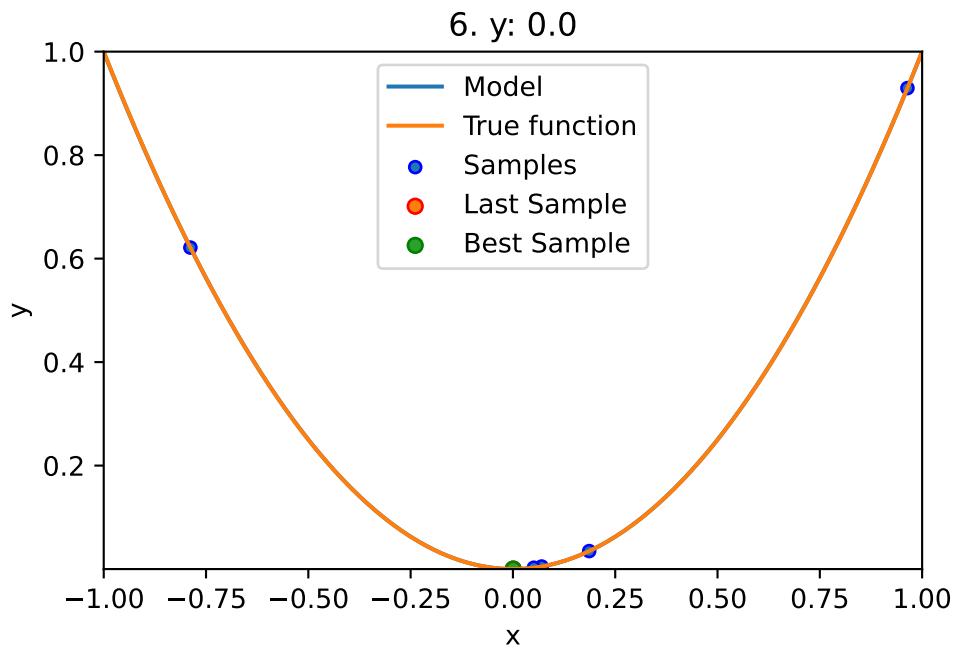




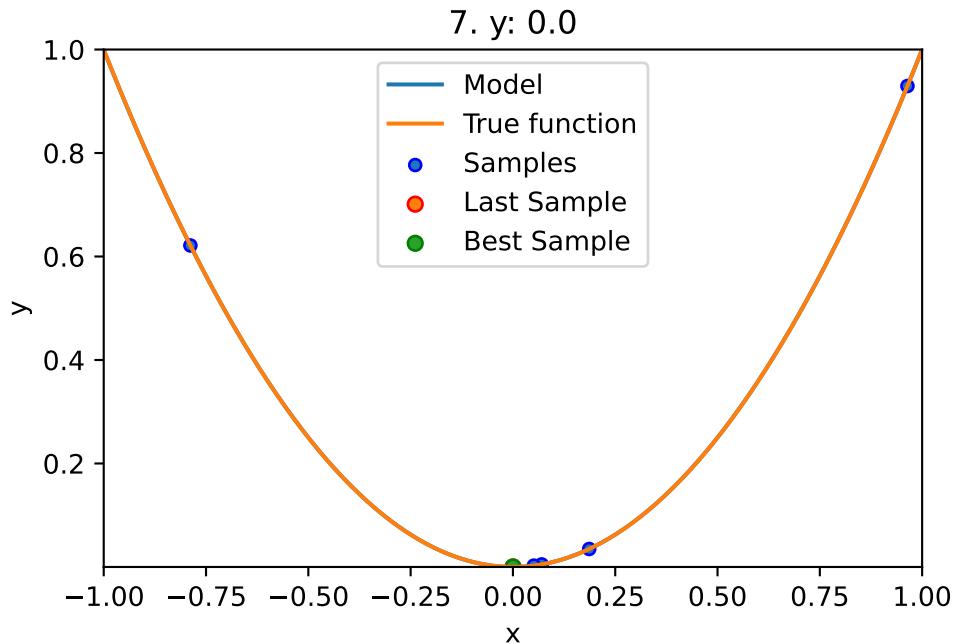
```
spotPython tuning: 0.004925761656816393 [#####-----] 40.00%
```



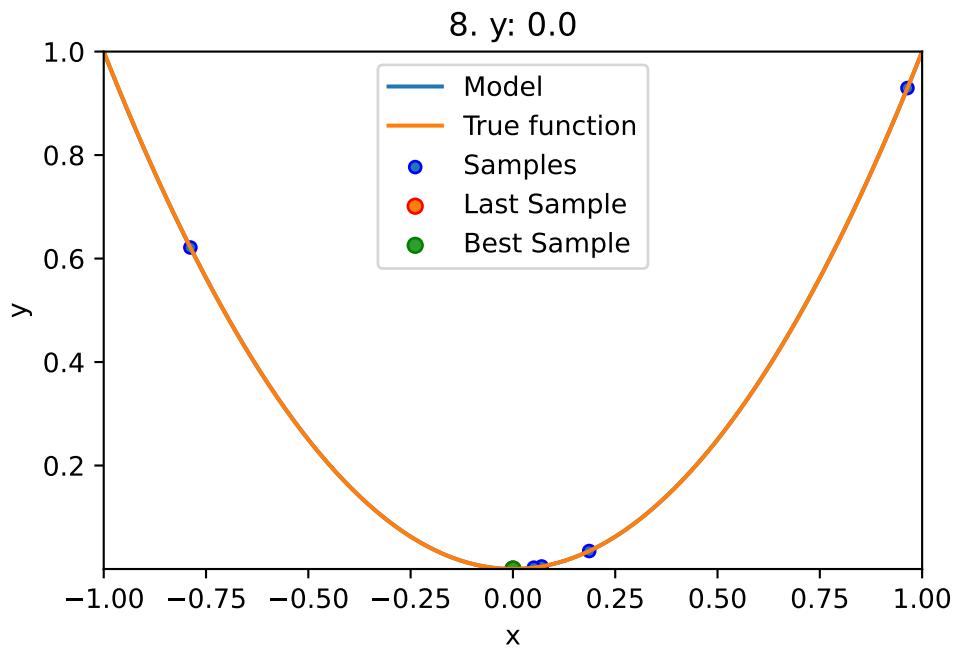
```
spotPython tuning: 0.0026120758453649505 [#####----] 50.00%
```



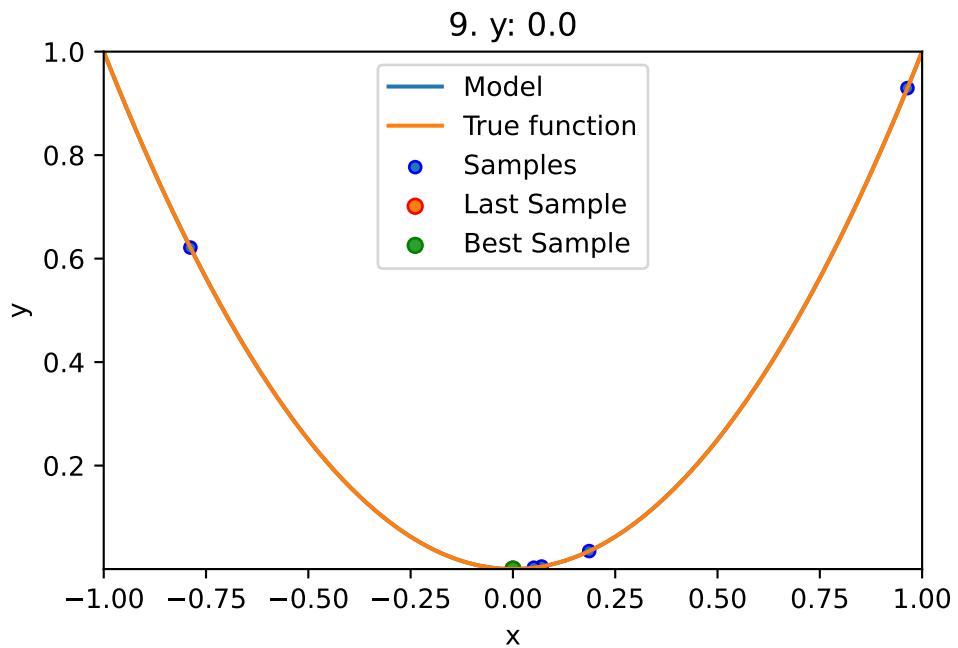
```
spotPython tuning: 4.492968068412204e-07 [#####----] 60.00%
```



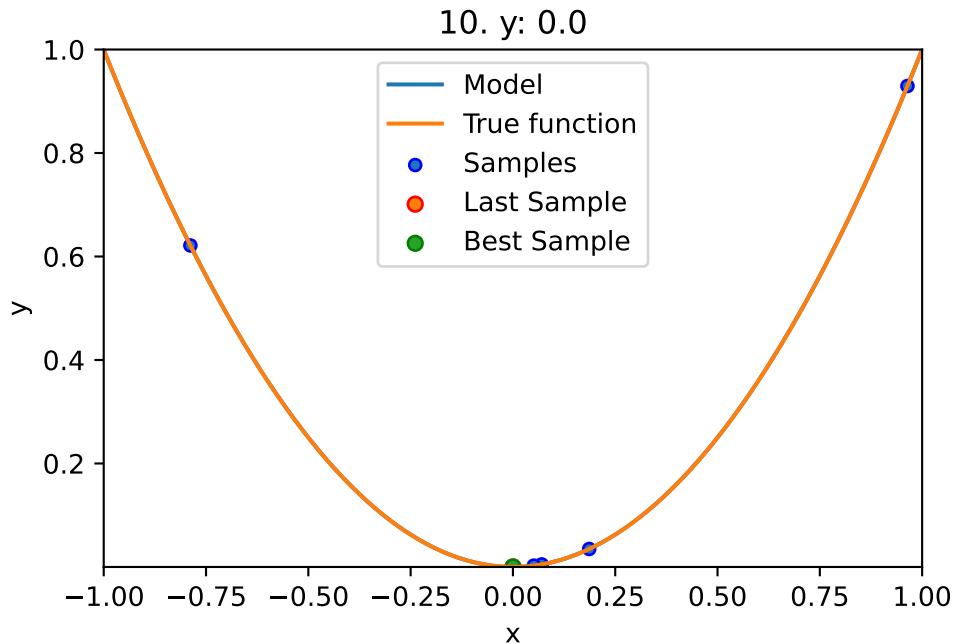
```
spotPython tuning: 5.520019085369139e-08 [#####---] 70.00%
```



```
spotPython tuning: 1.8830522883506717e-08 [#####--] 80.00%
```



```
spotPython tuning: 1.2165253306918689e-08 [#####--] 90.00%
```



```
spotPython tuning: 1.0471089618292772e-08 [#####] 100.00% Done...
```

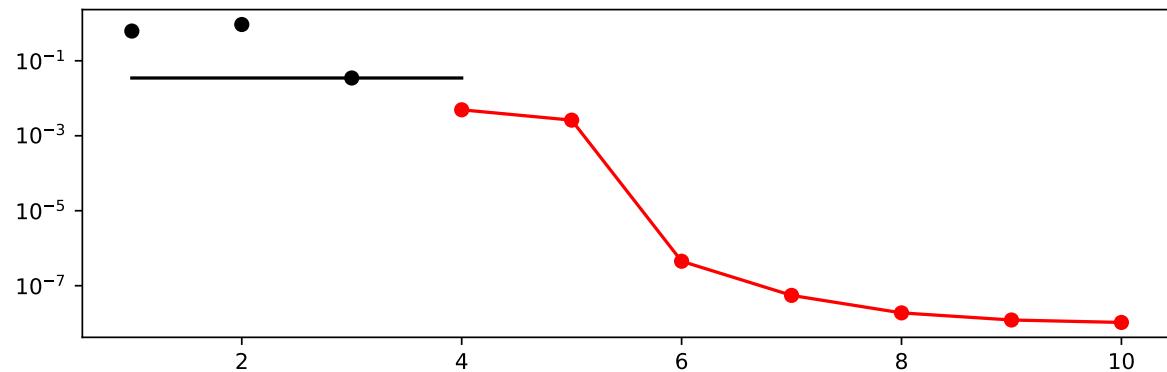
```
<spotPython.spot.spot.Spot at 0x2bdbe5960>
```

```
spot_1_GP.print_results()
```

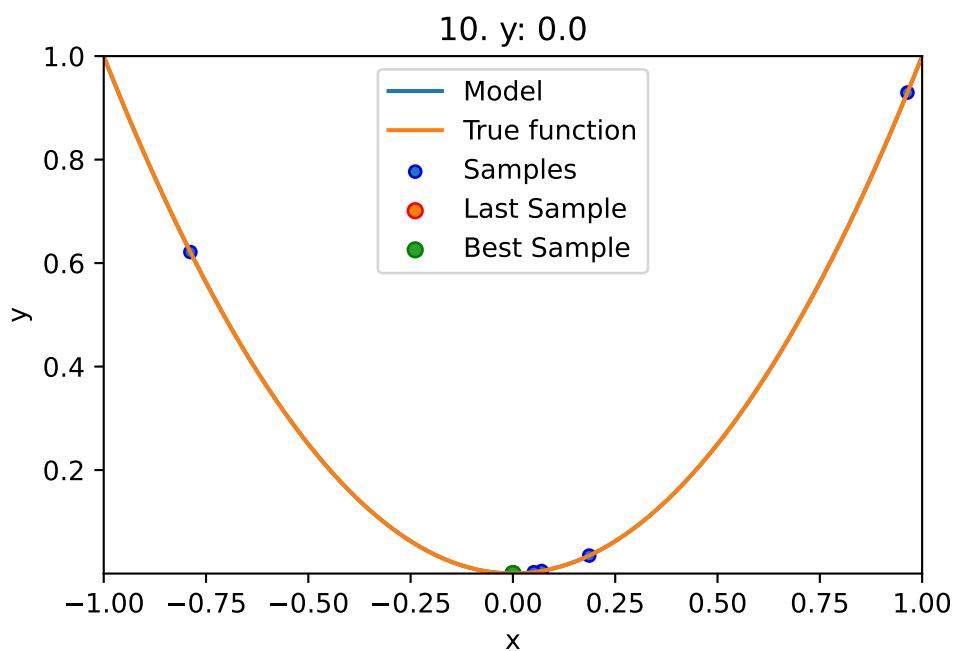
```
min y: 1.0471089618292772e-08
x0: 0.00010232834220436082
```

```
[['x0', 0.00010232834220436082]]
```

```
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:

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

- 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 Evaluation 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#glo-bal-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\_\_bartz09\_2023-07-17\_08-49-54

```

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: 2.5179080007735086 [#####----] 65.00%

spotPython tuning: 1.0168713401682457 [#####---] 70.00%

spotPython tuning: 0.4160575412800043 [#####--] 75.00%

```
spotPython tuning: 0.40966080781404557 [#####--] 80.00%
spotPython tuning: 0.40966080781404557 [#####--] 85.00%
spotPython tuning: 0.39989087044857285 [#####--] 90.00%
spotPython tuning: 0.3996741243343038 [#####--] 95.00%
spotPython tuning: 0.39951958110619046 [#####--] 100.00% Done...
<spotPython.spot.spot.Spot at 0x2bcddf910>
```

### 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: 0.39951958110619046
x0: -3.1570201165683587
x1: 12.289980569430284

[['x0', -3.1570201165683587], ['x1', 12.289980569430284]]
```

## 5.4 Show the Progress

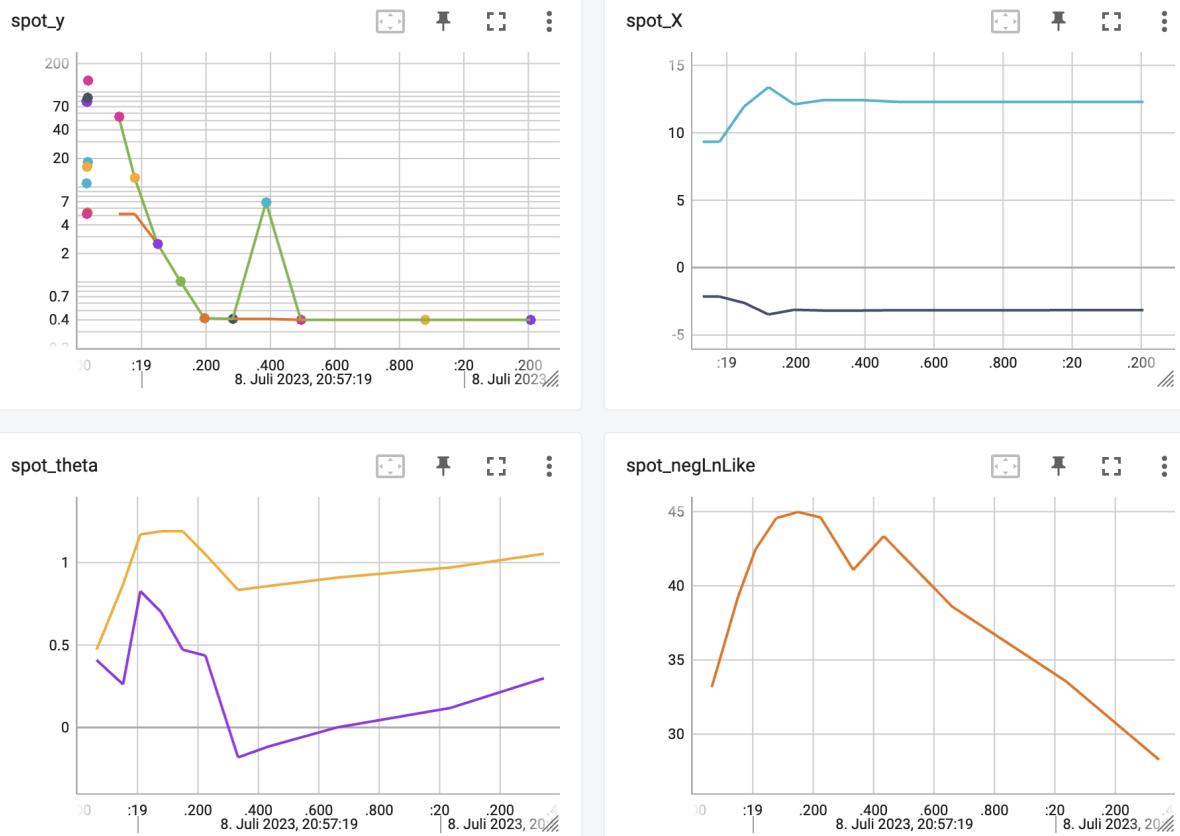
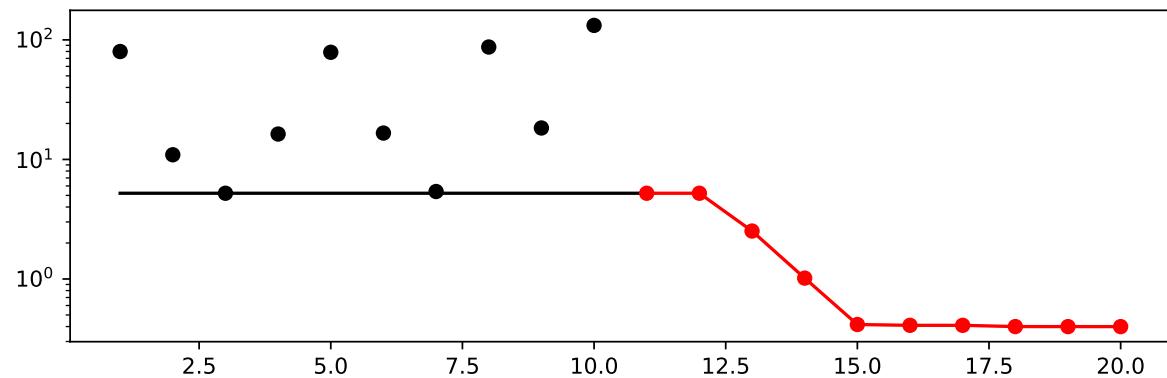
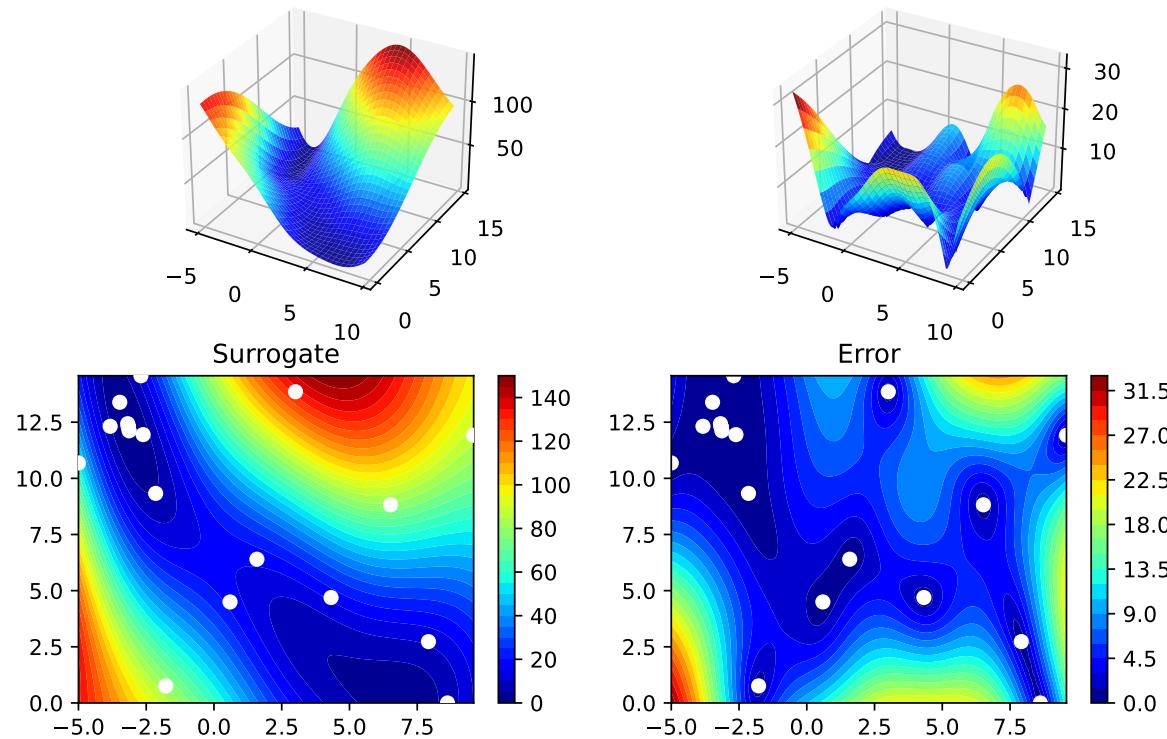


Figure 5.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

```
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/plot\\_gp.html](https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gp.html)
- After fitting our model, we see that the hyperparameters of the kernel have been optimized.
- Now, we will use our kernel to compute the mean prediction of the full dataset and plot the 95% confidence interval.

### 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

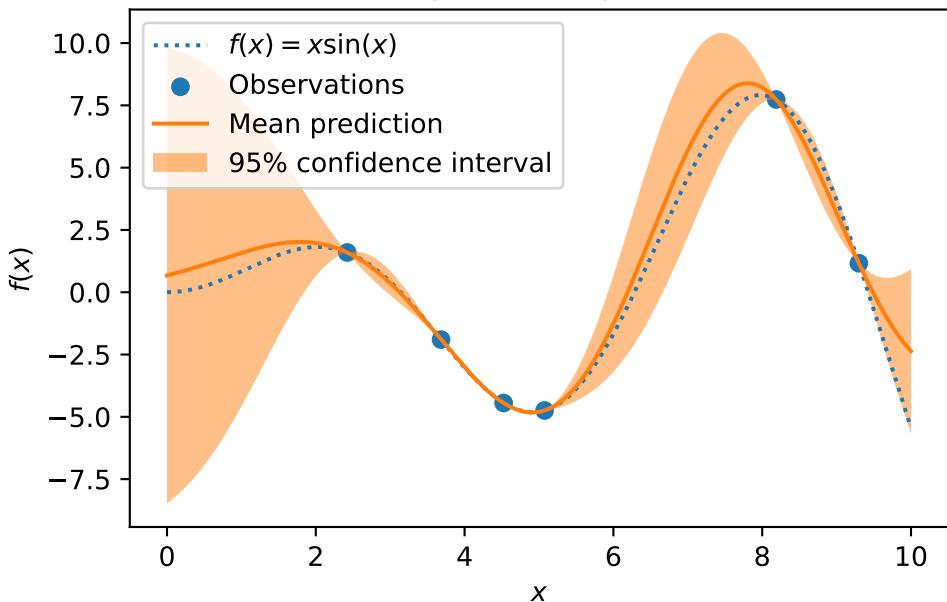
- The model building with `sklearn` consists of three steps:
  1. Instantiating the model, then
  2. fitting the model (using `fit`), and
  3. making predictions (using `predict`)

```
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")

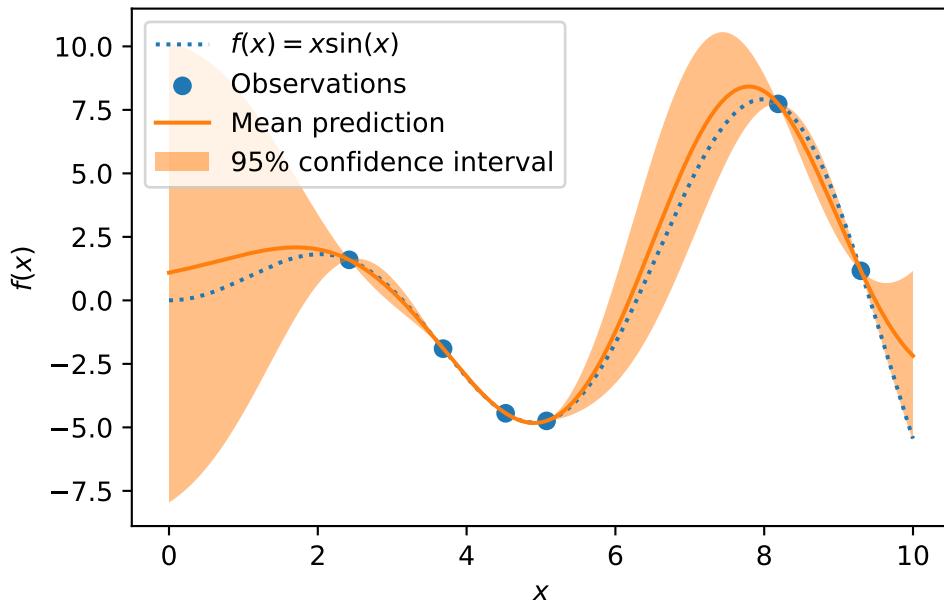
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, S_mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
    S_mean_prediction - 1.96 * S_std_prediction,
    S_mean_prediction + 1.96 * S_std_prediction,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

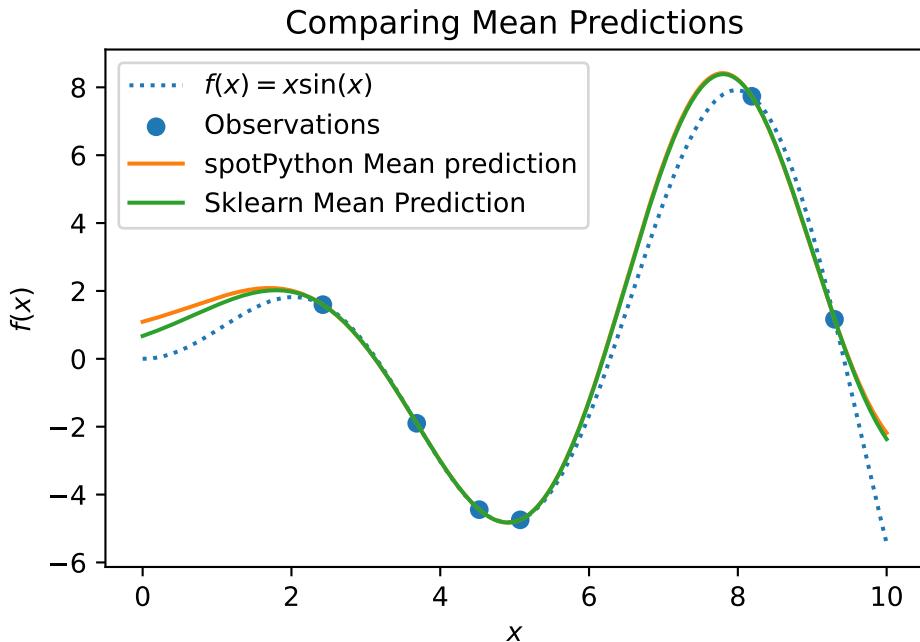


### 6.1.5 Visualizing the Differences Between the spotPython and the sklearn Model Fits

```

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:

$$f(x) = (6x-2)^2 \sin(12x-4) \text{ for } x \in [0,1].$$

- 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\_bartz09\_2023-07-17\_08-50-45

```

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: 7.263311682641849e-09 [#####----] 44.00%

spotPython tuning: 7.263311682641849e-09 [#####----] 48.00%

spotPython tuning: 7.263311682641849e-09 [#####----] 52.00%

spotPython tuning: 7.263311682641849e-09 [#####----] 56.00%

spotPython tuning: 3.696886711914087e-10 [#####----] 60.00%

spotPython tuning: 3.696886711914087e-10 [#####----] 64.00%

spotPython tuning: 3.696886711914087e-10 [#####----] 68.00%

```

spotPython tuning: 3.696886711914087e-10 [#####---] 72.00%
spotPython tuning: 3.696886711914087e-10 [#####---] 76.00%
spotPython tuning: 3.696886711914087e-10 [#####---] 80.00%
spotPython tuning: 3.696886711914087e-10 [#####---] 84.00%
spotPython tuning: 3.696886711914087e-10 [#####---] 88.00%
spotPython tuning: 1.3792745942664307e-11 [#####---] 92.00%
spotPython tuning: 1.3792745942664307e-11 [#####---] 96.00%
spotPython tuning: 1.3792745942664307e-11 [#####---] 100.00% Done...
<spotPython.spot.spot.Spot at 0x103347430>

```

### 7.1.2 Results

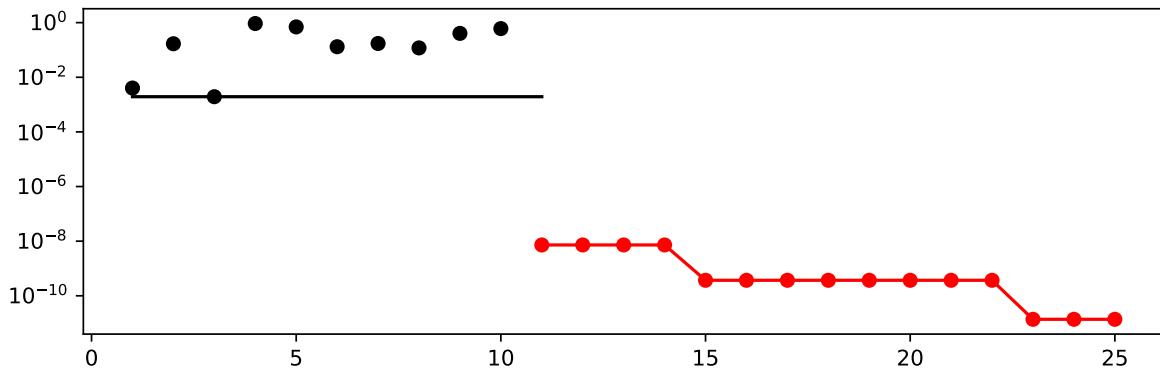
```

spot_1.print_results()

min y: 1.3792745942664307e-11
x0: 3.7138586325632142e-06

[['x0', 3.7138586325632142e-06]]
```

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



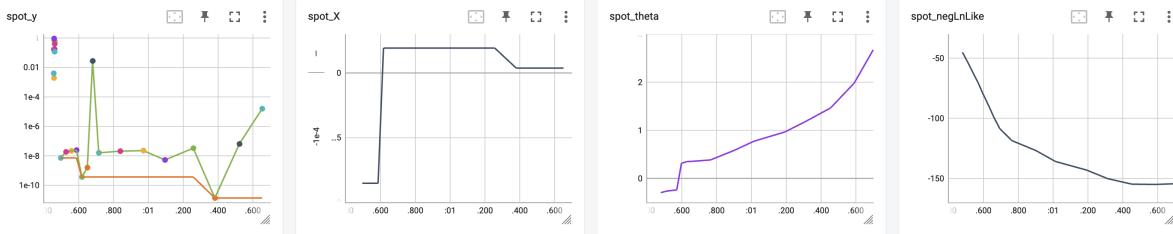


Figure 7.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

## 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\_bartz09\_2023-07-17\_08-50-47

```
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: 1.1630341306771934e-08 [#####-----] 44.00%

spotPython tuning: 1.1630341306771934e-08 [#####-----] 48.00%

spotPython tuning: 1.1630341306771934e-08 [#####-----] 52.00%

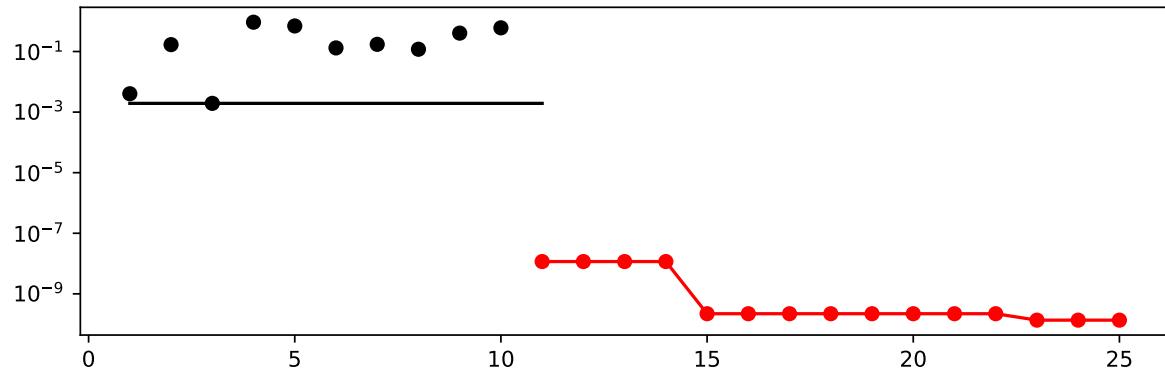
```

spotPython tuning: 1.1630341306771934e-08 [#####----] 56.00%
spotPython tuning: 2.207887258868953e-10 [#####----] 60.00%
spotPython tuning: 2.207887258868953e-10 [#####----] 64.00%
spotPython tuning: 2.207887258868953e-10 [#####----] 68.00%
spotPython tuning: 2.207887258868953e-10 [#####----] 72.00%
spotPython tuning: 2.207887258868953e-10 [#####----] 76.00%
spotPython tuning: 2.207887258868953e-10 [#####----] 80.00%
spotPython tuning: 2.207887258868953e-10 [#####----] 84.00%
spotPython tuning: 2.207887258868953e-10 [#####----] 88.00%
spotPython tuning: 1.3536080613078865e-10 [#####----] 92.00%
spotPython tuning: 1.3536080613078865e-10 [#####----] 96.00%
spotPython tuning: 1.3536080613078865e-10 [#####----] 100.00% Done...

```

<spotPython.spot.spot at 0x16e9e4c40>

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



```

spot_1_ei.print_results()

min y: 1.3536080613078865e-10
x0: 1.1634466301931888e-05

[['x0', 1.1634466301931888e-05]]

```

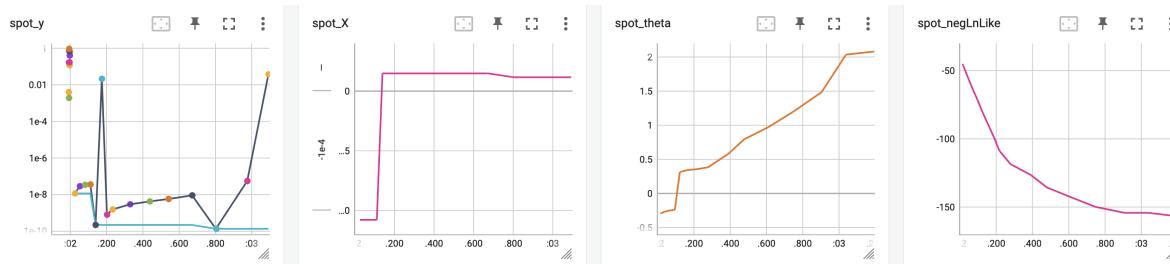


Figure 7.2: TensorBoard visualization of the spotPython optimization process and the surrogate model. Expected improvement, isotropic Kriging.

### 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\_bartz09\_2023-07-17\_08-50-49

```

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: 1.754686753274553e-05 [#####-----] 44.00%
```

```
spotPython tuning: 1.754686753274553e-05 [#####-----] 48.00%
```

```
spotPython tuning: 1.754686753274553e-05 [#####-----] 52.00%
```

```
spotPython tuning: 1.0120806700557811e-05 [#####-----] 56.00%
```

```
spotPython tuning: 1.0120806700557811e-05 [#####-----] 60.00%
```

```
spotPython tuning: 1.8779971830281702e-07 [#####-----] 64.00%
```

```
spotPython tuning: 1.8779971830281702e-07 [#####-----] 68.00%
```

```
spotPython tuning: 1.8779971830281702e-07 [#####-----] 72.00%
```

```
spotPython tuning: 1.8779971830281702e-07 [#####-----] 76.00%
```

```
spotPython tuning: 1.8779971830281702e-07 [#####-----] 80.00%
```

```
spotPython tuning: 1.8779971830281702e-07 [#####-----] 84.00%
```

```
spotPython tuning: 1.8779971830281702e-07 [#####-----] 88.00%
```

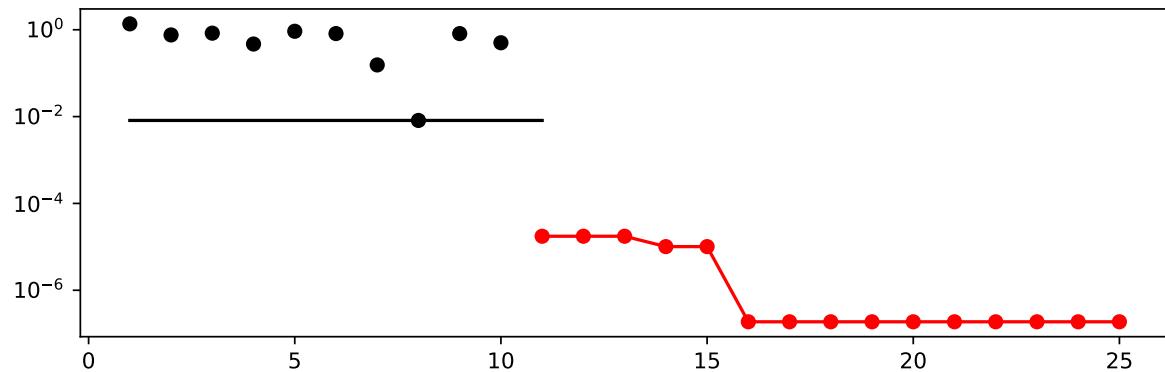
```
spotPython tuning: 1.8779971830281702e-07 [#####-----] 92.00%
```

```
spotPython tuning: 1.8779971830281702e-07 [#####] 96.00%
```

```
spotPython tuning: 1.8779971830281702e-07 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x16ebefee0>
```

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



```
spot_2_ei_noniso.print_results()
```

```
min y: 1.8779971830281702e-07
x0: -0.0002783721390529846
x1: 0.0003321274913371111
```

```
[['x0', -0.0002783721390529846], ['x1', 0.0003321274913371111]]
```

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

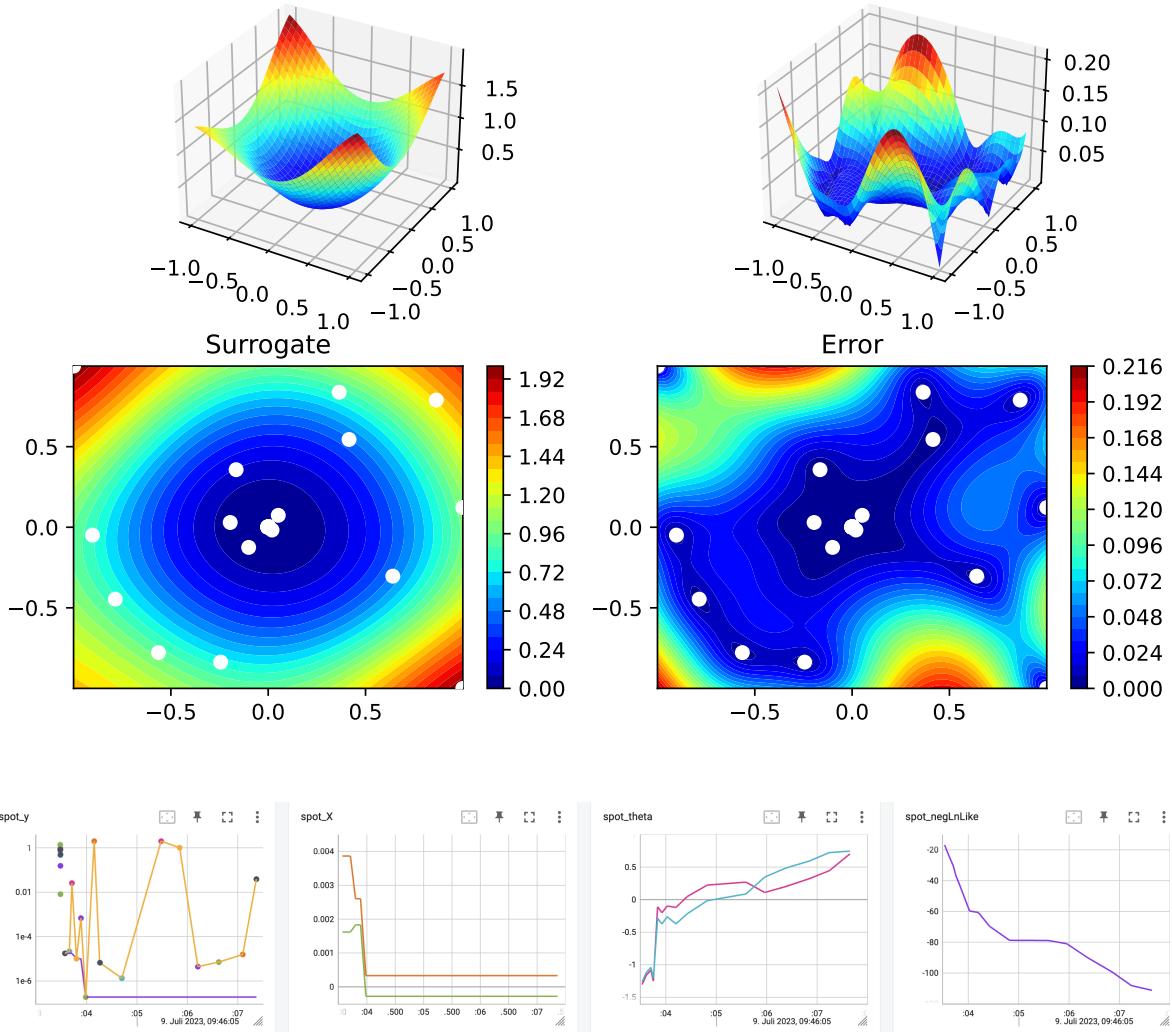


Figure 7.3: TensorBoard visualization of the spotPython optimization process and the surrogate model. Expected improvement, isotropic Kriging.

## 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:

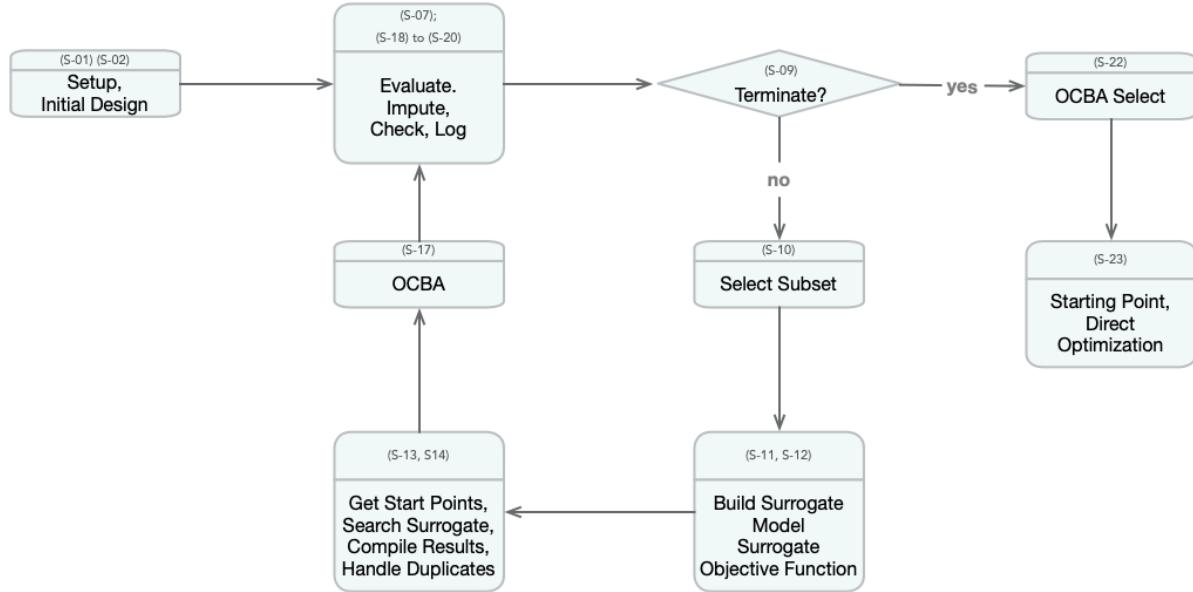


Figure 7.4: Visual representation of the model based search with SPOT. Taken from: Bartz-Beielstein, T., and Zaefferer, M. Hyperparameter tuning approaches. In Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide, E. Bartz, T. Bartz-Beielstein, M. Zaefferer, and O. Mersmann, 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 [bart21].

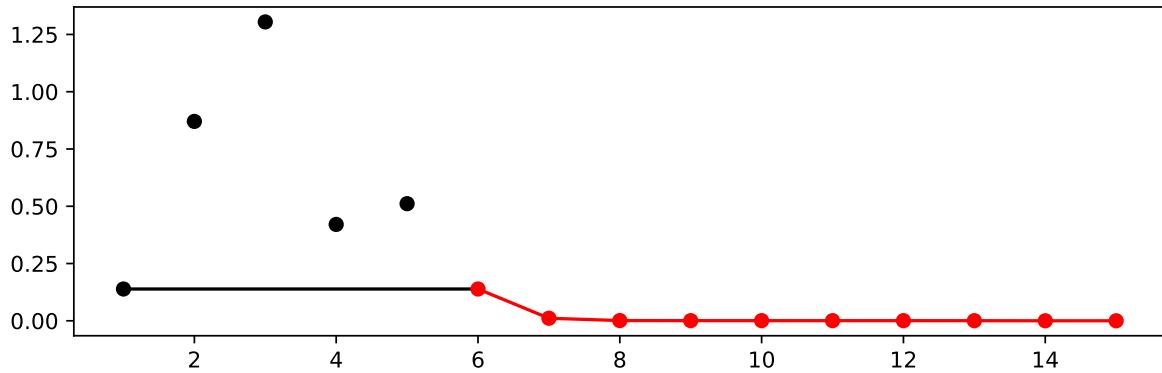
```

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: 0.0111581443080968 [#####-----] 46.67%
spotPython tuning: 0.0010079970679825743 [#####-----] 53.33%
spotPython tuning: 0.000631621365403864 [#####----] 60.00%
spotPython tuning: 0.0005883893741686826 [#####----] 66.67%
spotPython tuning: 0.00058412889636168 [#####---] 73.33%
spotPython tuning: 0.0005539414734082665 [#####---] 80.00%
spotPython tuning: 0.0004401288692983916 [#####--] 86.67%
spotPython tuning: 5.8179647898944394e-05 [#####--] 93.33%
spotPython tuning: 1.7928640814182596e-05 [#####-] 100.00% Done...
<spotPython.spot.spot at 0x16eb6e230>
```

```
spot_ei.plot_progress()
```



```

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

(1.3792745942664307e-11, 1.7928640814182596e-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)

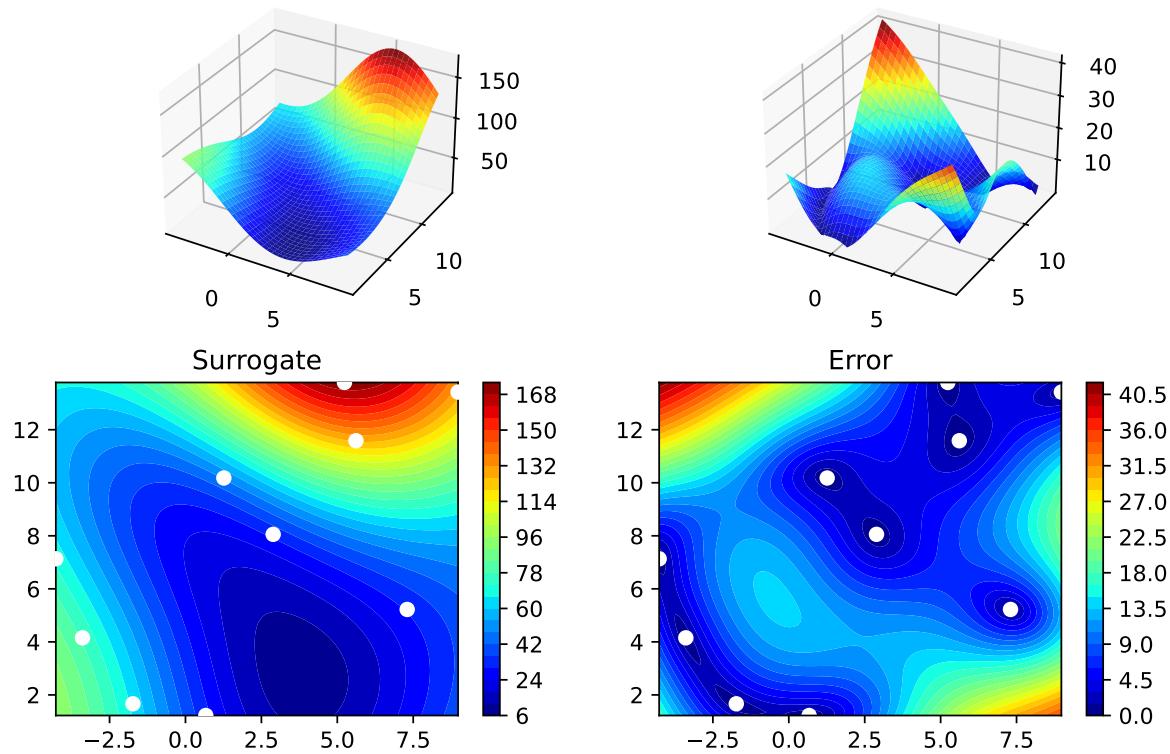
```

[[ 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]

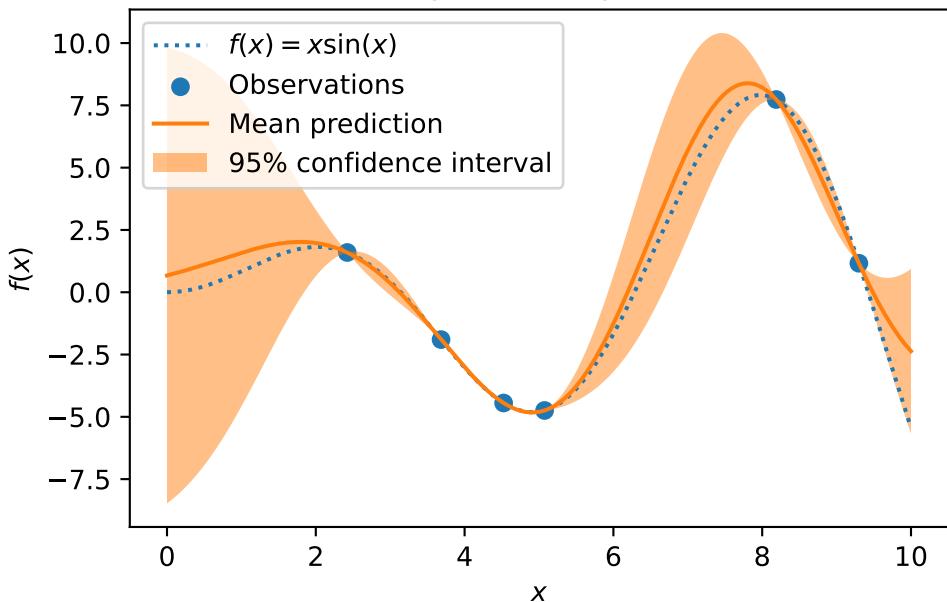
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
gaussian_process = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
gaussian_process.fit(X_train, y_train)
gaussian_process.kernel_

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

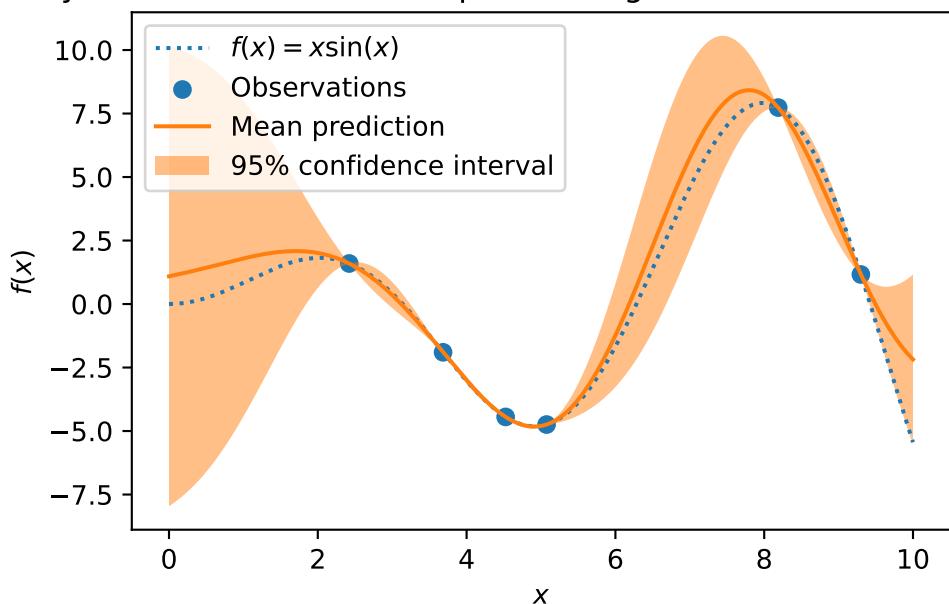
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
plt.fill_between(
```

```

        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()
```

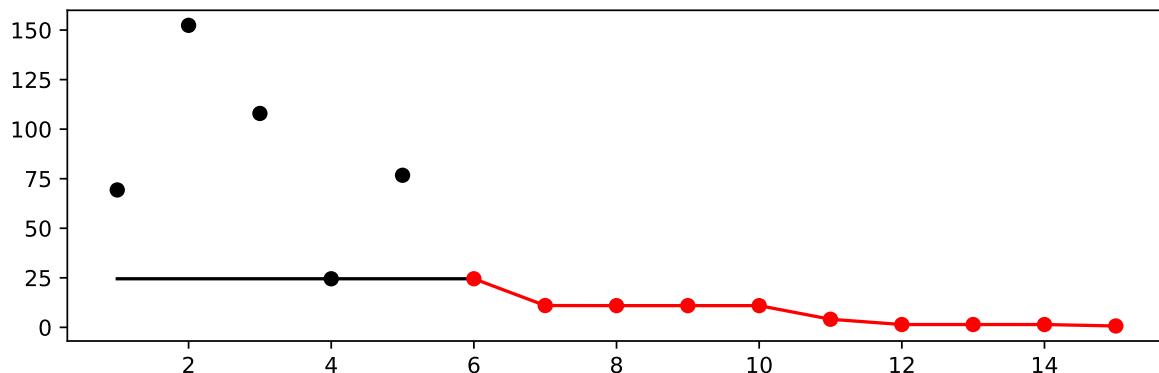
```
spotPython tuning: 24.51465459019188 [#####-----] 40.00%
spotPython tuning: 11.003073503598229 [#####-----] 46.67%
spotPython tuning: 10.960665185123245 [#####-----] 53.33%
spotPython tuning: 10.960665185123245 [#####----] 60.00%
spotPython tuning: 10.960665185123245 [#####---] 66.67%
spotPython tuning: 4.089511646427124 [#####---] 73.33%
spotPython tuning: 1.4230307255030858 [#####--] 80.00%
spotPython tuning: 1.4230307255030858 [#####-] 86.67%
spotPython tuning: 1.4230307255030858 [#####-] 93.33%
spotPython tuning: 0.6949448160267053 [#####-] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2a7d11570>
```

```
spot_GP.y
```

```
array([ 69.32459936, 152.38491454, 107.92560483, 24.51465459,
       76.73500031, 86.30425659, 11.0030735 , 10.96066519,
      16.06666933, 24.08428925, 4.08951165, 1.42303073,
      1.4736037 , 16.03577039, 0.69494482])
```

```
spot_GP.plot_progress()
```



```
spot_GP.print_results()
```

```
min y: 0.6949448160267053
```

```
x0: 3.3575232000433637
```

```
x1: 2.3847893450472464
```

```
[['x0', 3.3575232000433637], ['x1', 2.3847893450472464]]
```

## 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: 2.0398360048852566e-05 [#####----] 44.00%

spotPython tuning: 2.0398360048852566e-05 [#####----] 48.00%

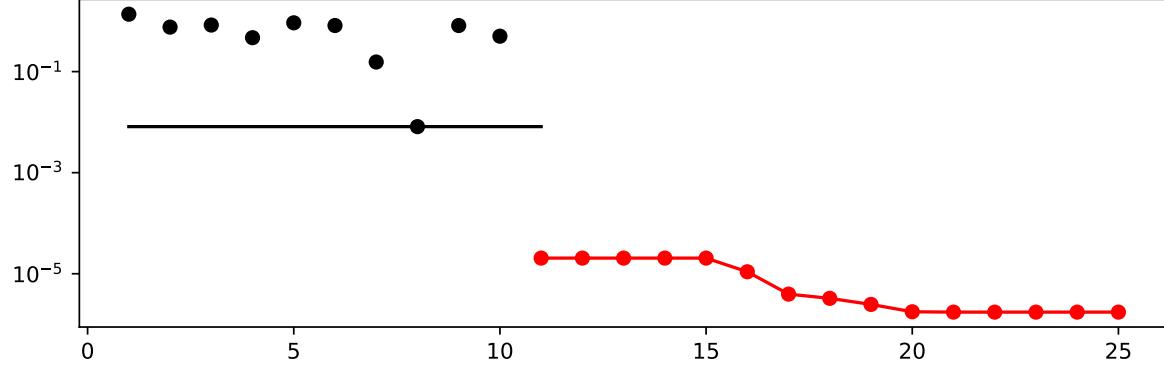
spotPython tuning: 2.0398360048852566e-05 [#####----] 52.00%

spotPython tuning: 2.0398360048852566e-05 [#####----] 56.00%

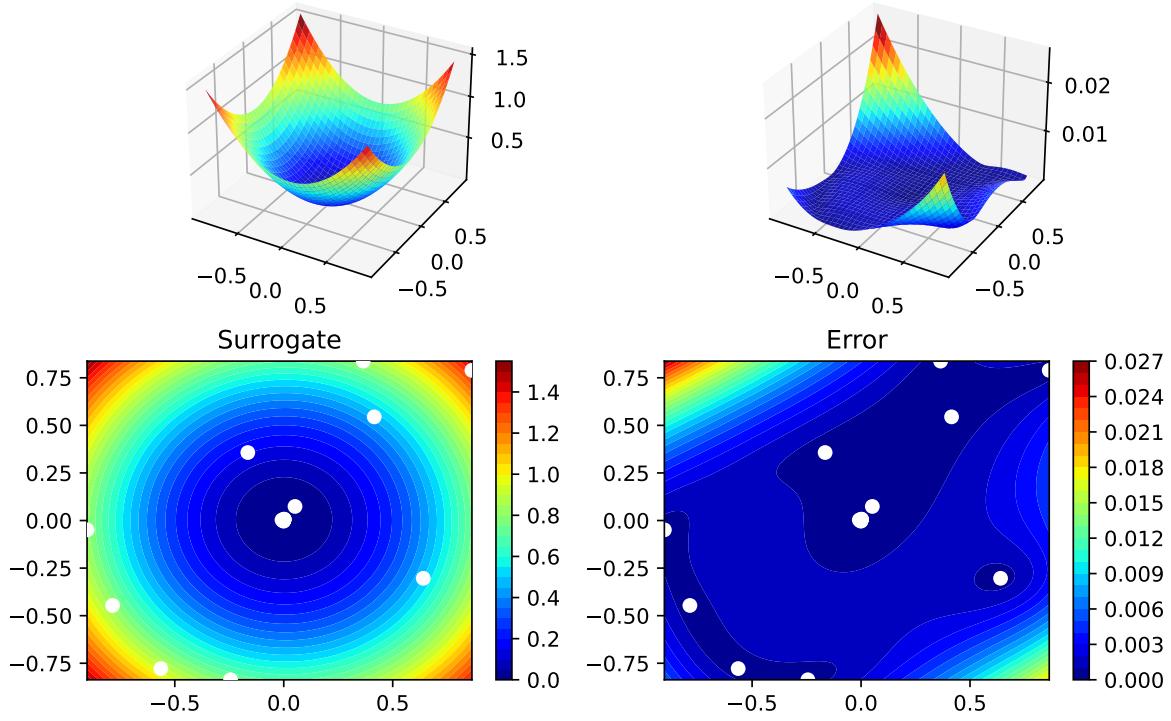
```
spotPython tuning: 2.0398360048852566e-05 [#####----] 60.00%
spotPython tuning: 1.0937897482978201e-05 [#####----] 64.00%
spotPython tuning: 3.950539536972047e-06 [#####---] 68.00%
spotPython tuning: 3.2602730419203698e-06 [#####---] 72.00%
spotPython tuning: 2.4704028732017656e-06 [#####--] 76.00%
spotPython tuning: 1.7687713431606244e-06 [#####--] 80.00%
spotPython tuning: 1.7395335905335862e-06 [#####--] 84.00%
spotPython tuning: 1.7395335905335862e-06 [#####--] 88.00%
spotPython tuning: 1.7395335905335862e-06 [#####--] 92.00%
spotPython tuning: 1.7395335905335862e-06 [#####--] 96.00%
spotPython tuning: 1.7395335905335862e-06 [#####--] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2c037a7d0>
```

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



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



```
spot_S_K.print_results()
```

```
min y: 1.7395335905335862e-06
x0: -0.0013044072412622557
x1: 0.0001950777780173277
```

```
[['x0', -0.0013044072412622557], ['x1', 0.0001950777780173277]]
```

### 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   ]
[[0.00160553  0.00428429]
 [0.00160553  0.00428429]]

```

## 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")

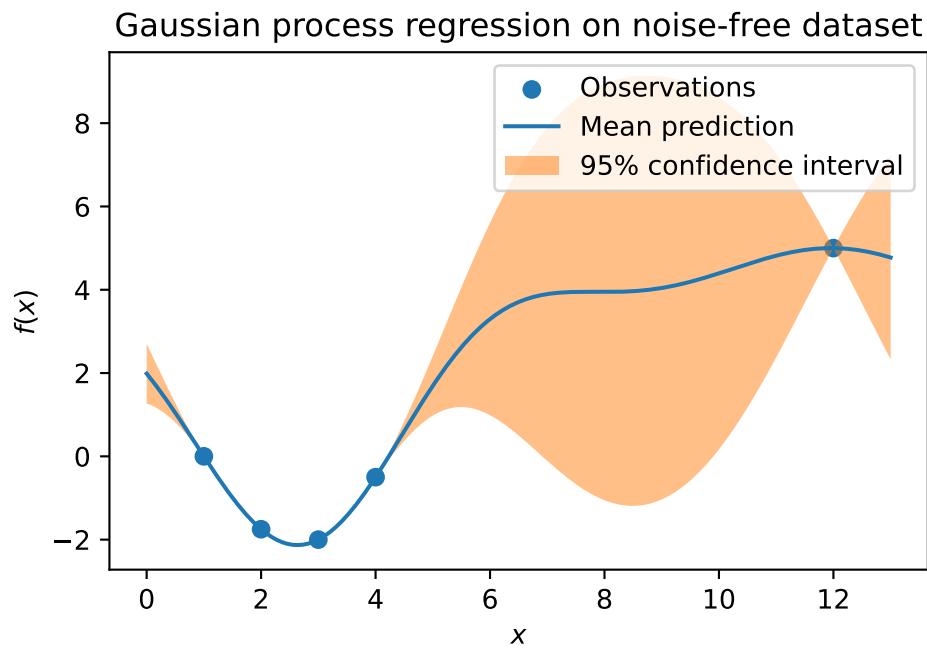
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
if True:
    plt.fill_between(
        X.ravel(),
        mean_prediction - 2 * std_prediction,
        mean_prediction + 2 * std_prediction,
        alpha=0.5,
        label=r"95% confidence interval",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

```

```

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

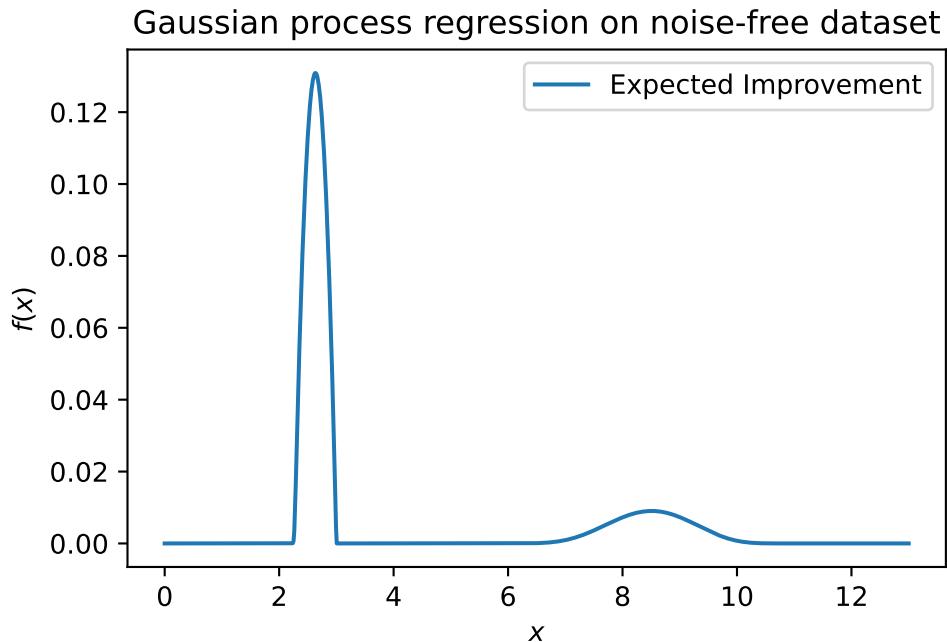
```



```

# 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([1.09276]),
 '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)

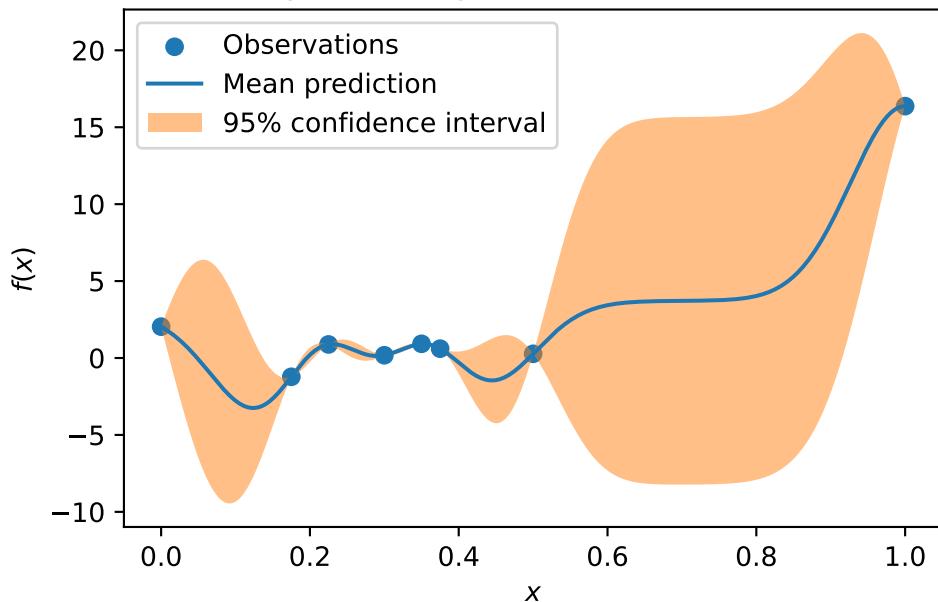
S = Kriging(name='kriging', seed=123, log_level=50, n_theta=1, noise=False, cod_type="nor"
S.fit(X_train, y_train)

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

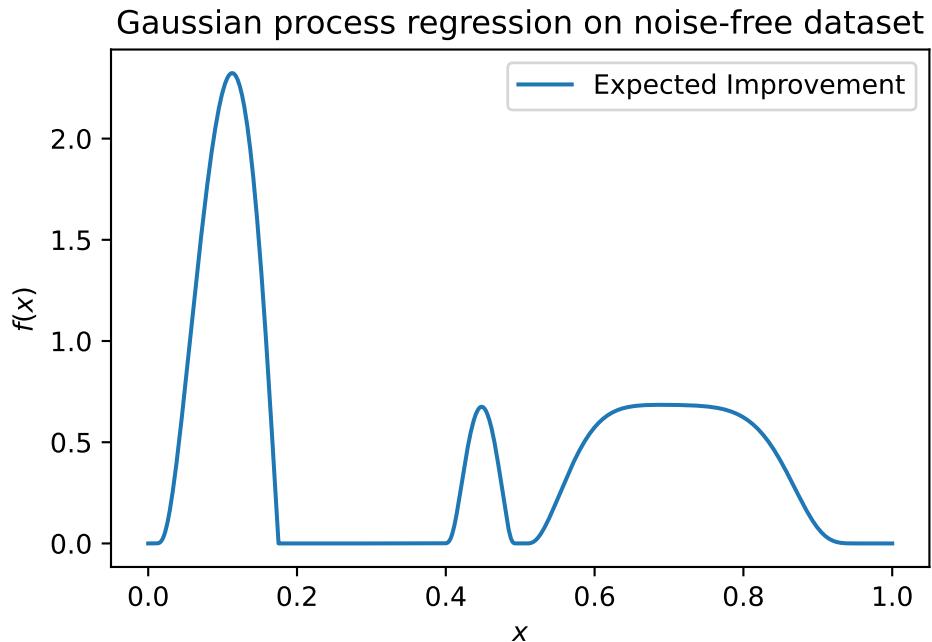
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
if True:
    plt.fill_between(
        X.ravel(),
        mean_prediction - 2 * std_prediction,
        mean_prediction + 2 * std_prediction,
        alpha=0.5,
        label=r"95% confidence interval",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

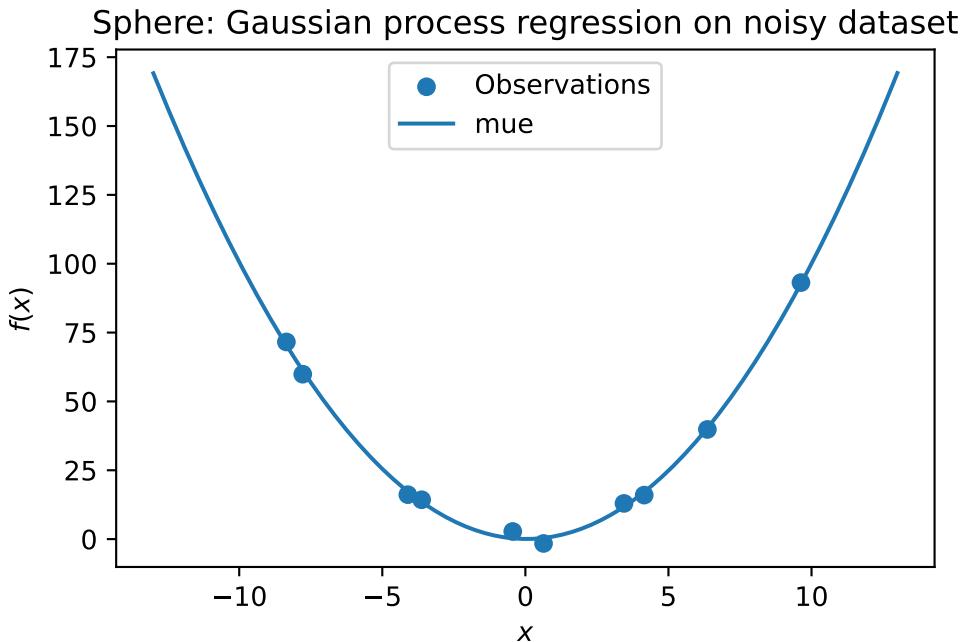
S = Kriging(name='kriging',
             seed=123,
             log_level=50,
             n_theta=1,
             noise=False)
S.fit(X_train, y_train)

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="mle")
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([25.26601605]),
  'theta': array([-1.98024488]),
  '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")

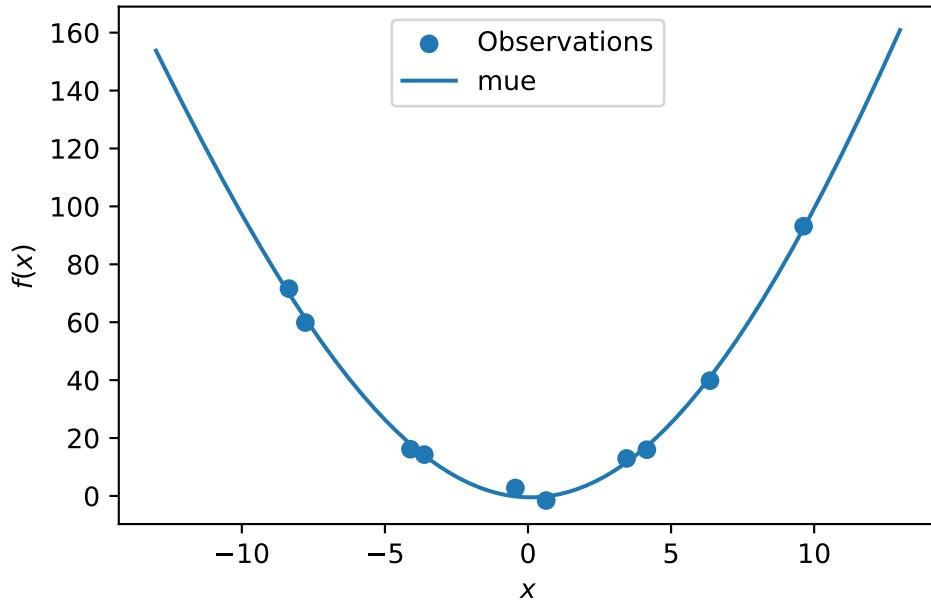
# plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
# plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mle")
```

```

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([5.20850907e-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")

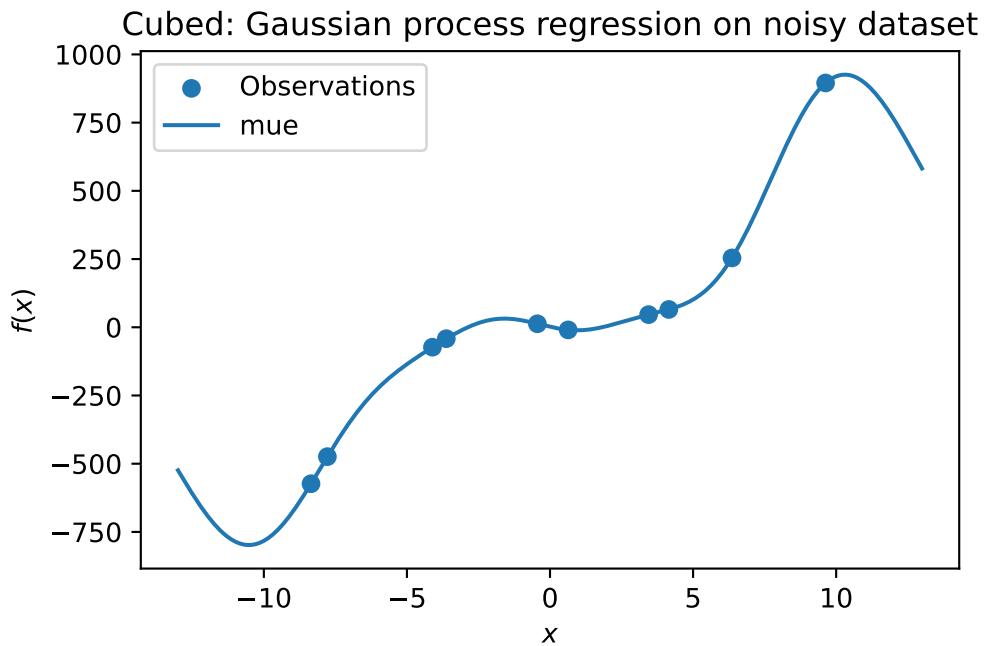
```

```

[[ 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]
```

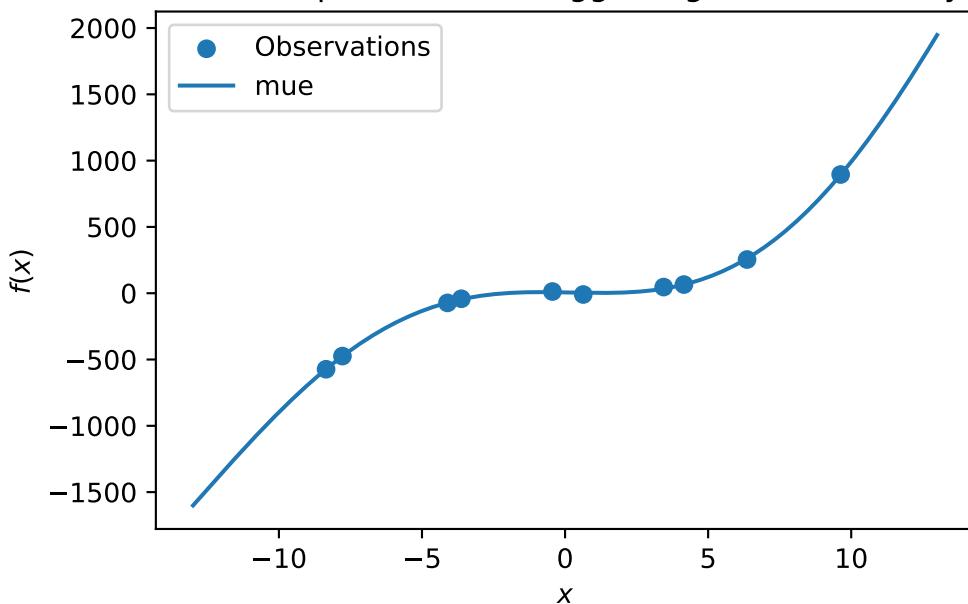


```
S = Kriging(name='kriging', seed=123, log_level=0, n_theta=1, noise=True)
S.fit(X_train, y_train)

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="mle")
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)

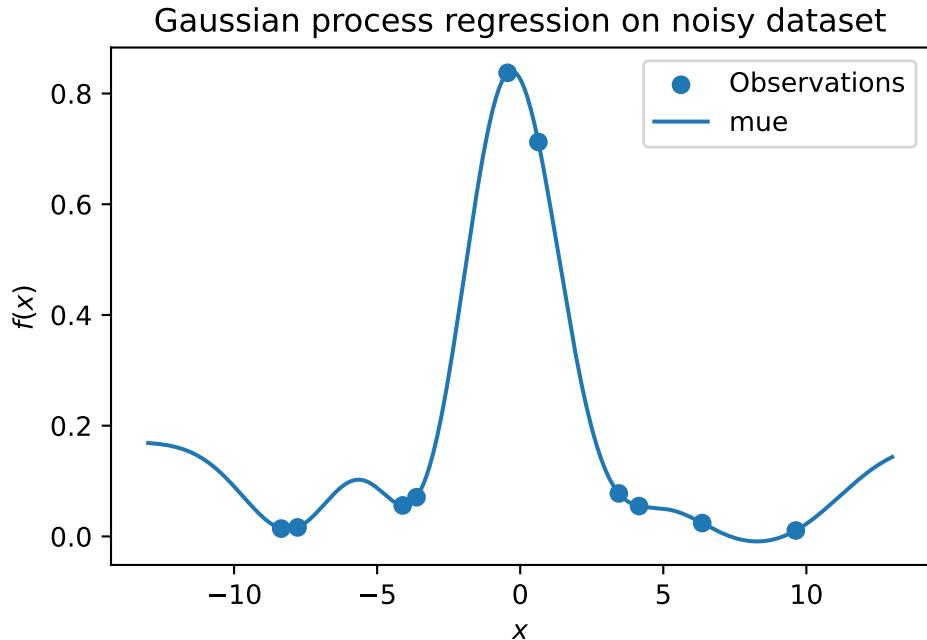
X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

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]
```

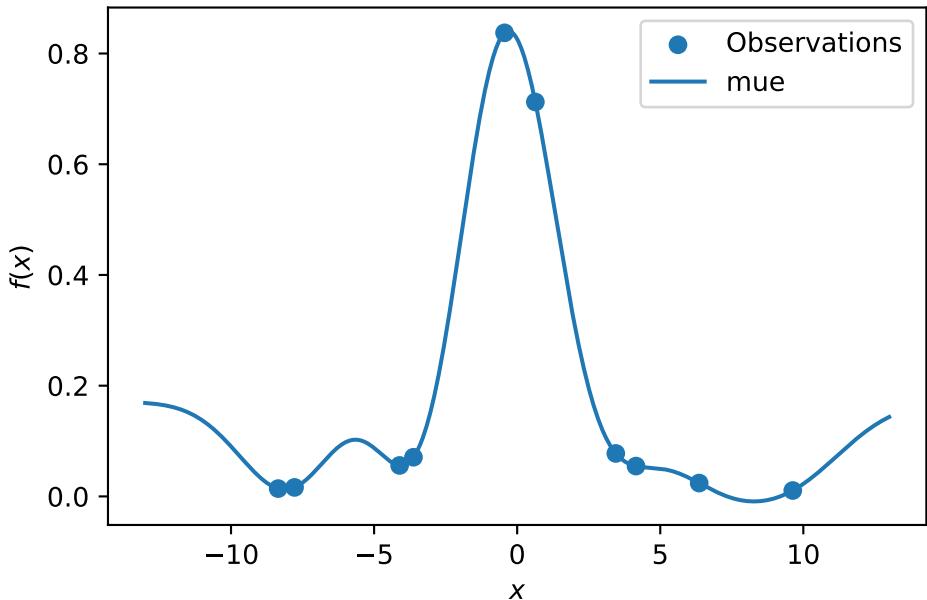


```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=True)
S.fit(X_train, y_train)

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=["n"])
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

```

-40.48225931963543

```
# 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\_bartz09\_2023-07-17\_08-51-27

### 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:

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

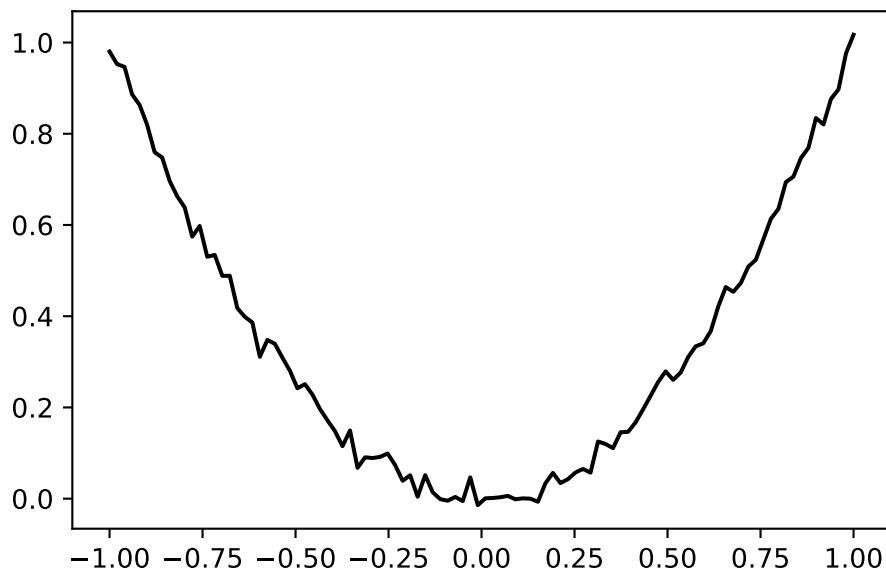
- Since `sigma` is set to 0.1, noise is added to the function:

```
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()
```

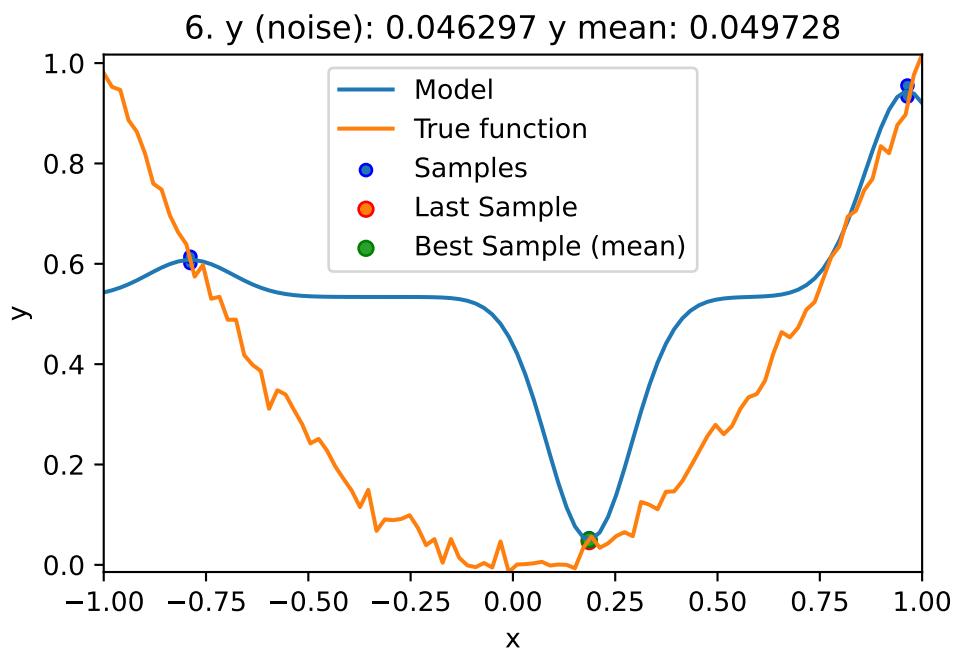


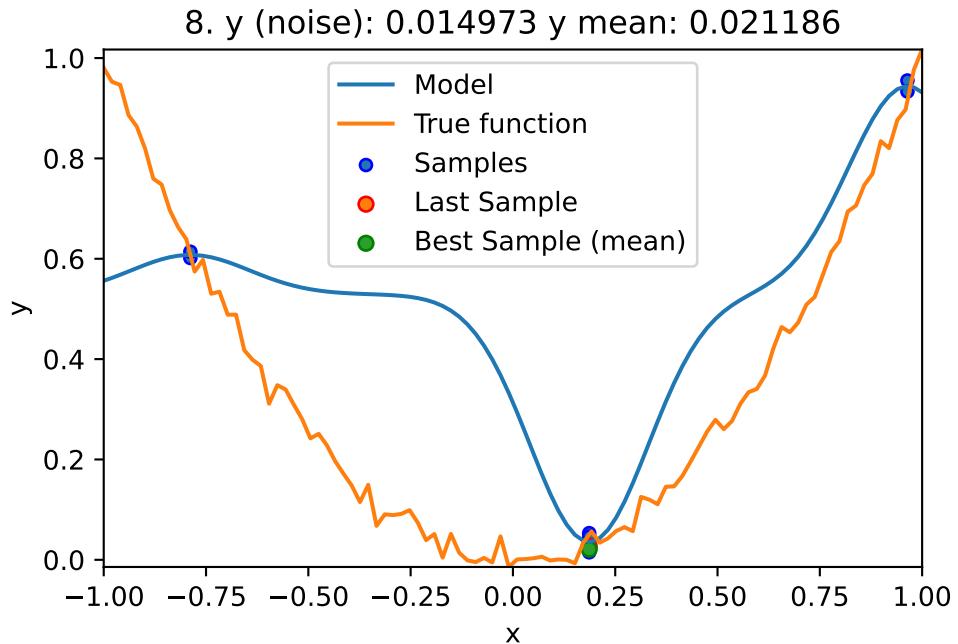
Spot is adopted as follows to cope with noisy functions:

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)

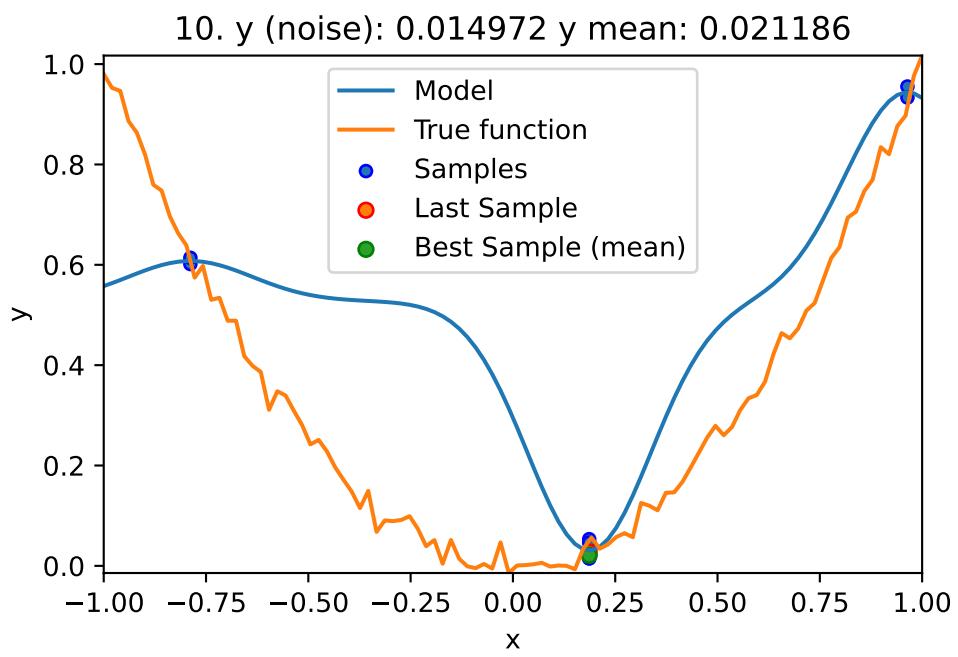
```
spot_1_noisy = spot.Spot(fun=fun,  
                         lower = np.array([-1]),  
                         upper = np.array([1]),  
                         fun_evals = 20,  
                         fun_repeats = 2,
```

```
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()
```

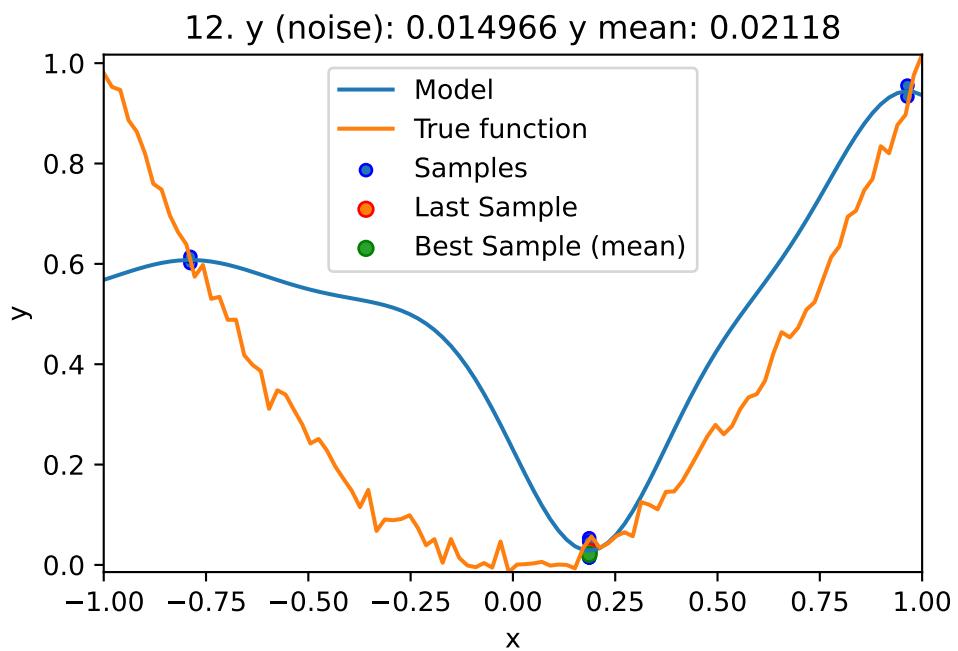




spotPython tuning: 0.01497250376669991 [#####-----] 40.00%

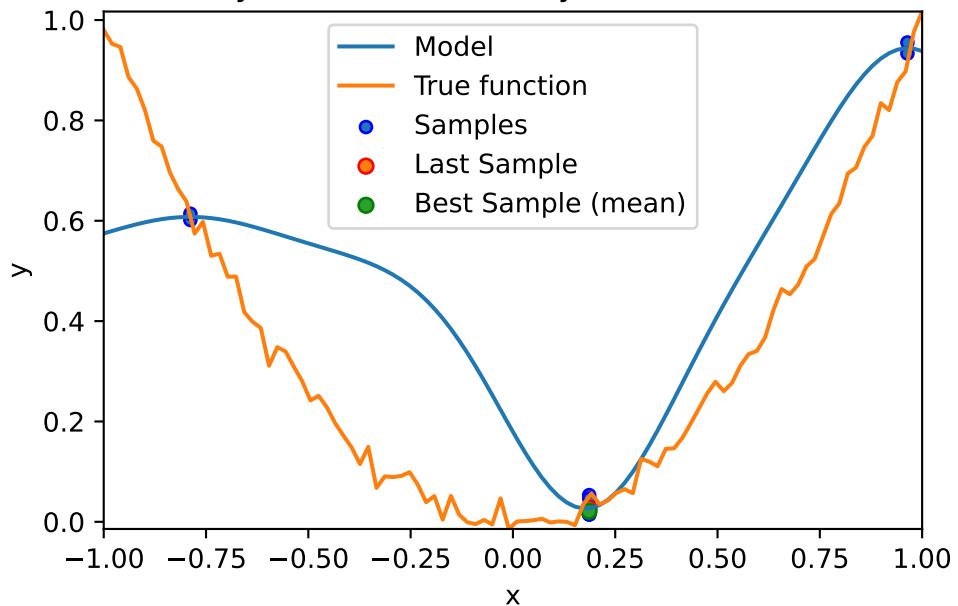


spotPython tuning: 0.01497226931667417 [#####----] 50.00%



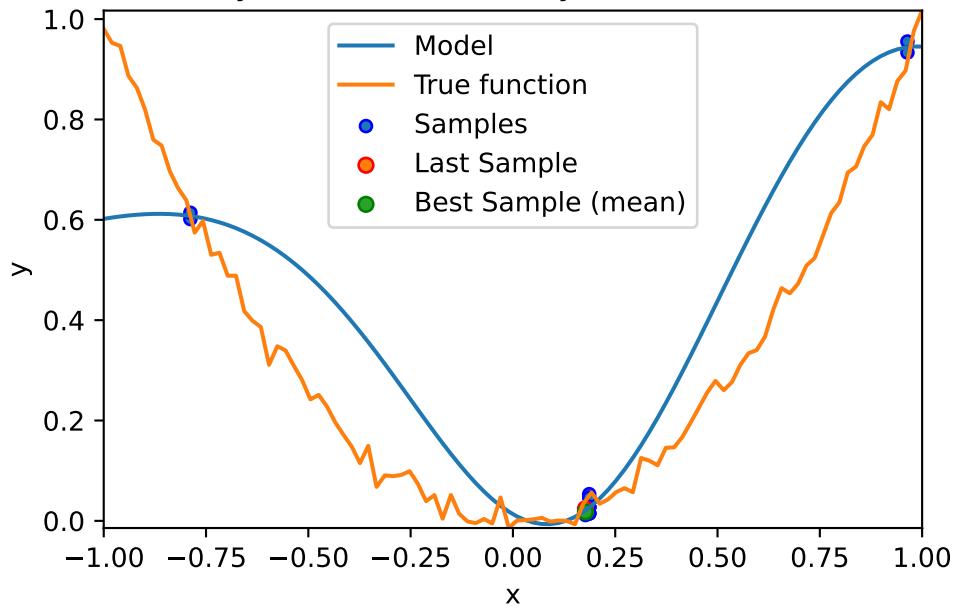
spotPython tuning: 0.01496618769080537 [#####----] 60.00%

14. y (noise): 0.014808 y mean: 0.021021

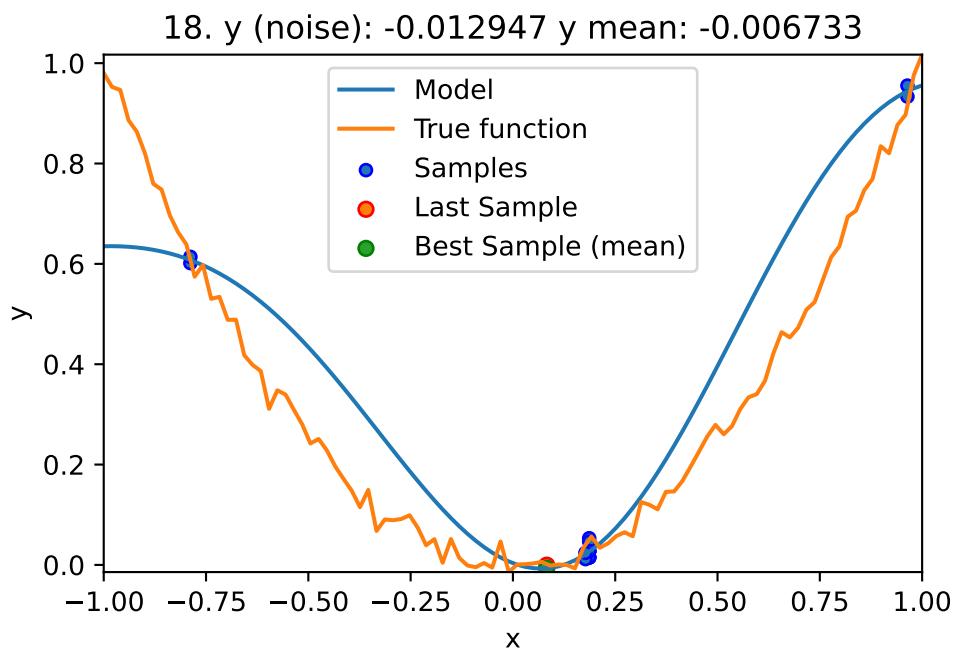


spotPython tuning: 0.014808104491512888 [#####---] 70.00%

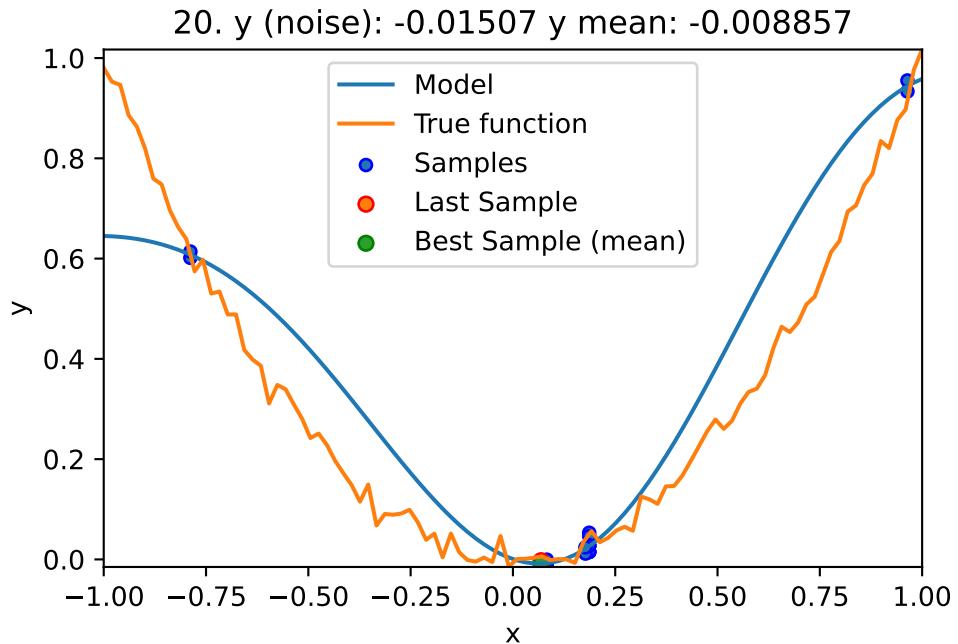
16. y (noise): 0.011631 y mean: 0.017845



```
spotPython tuning: 0.011631261600357518 [#####--] 80.00%
```



```
spotPython tuning: -0.012946672238374722 [#####--] 90.00%
```



```
spotPython tuning: -0.015070457665271902 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x16d2e6fe0>
```

## 8.2 Print the Results

```
spot_1_noisy.print_results()
```

```
min y: -0.015070457665271902
x0: 0.06864378589271657
min mean y: -0.008857110676472227
x0: 0.06864378589271657
```

```
[['x0', 0.06864378589271657], ['x0', 0.06864378589271657]]
```

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

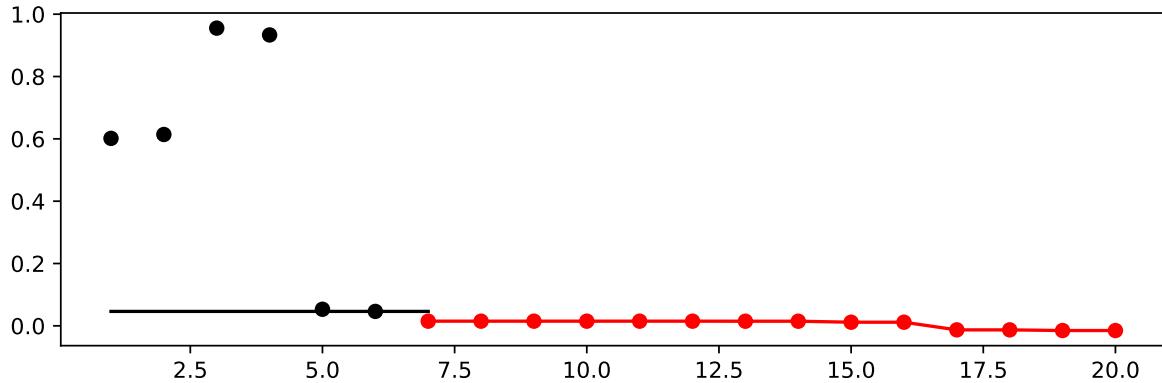


Figure 8.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

## 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:

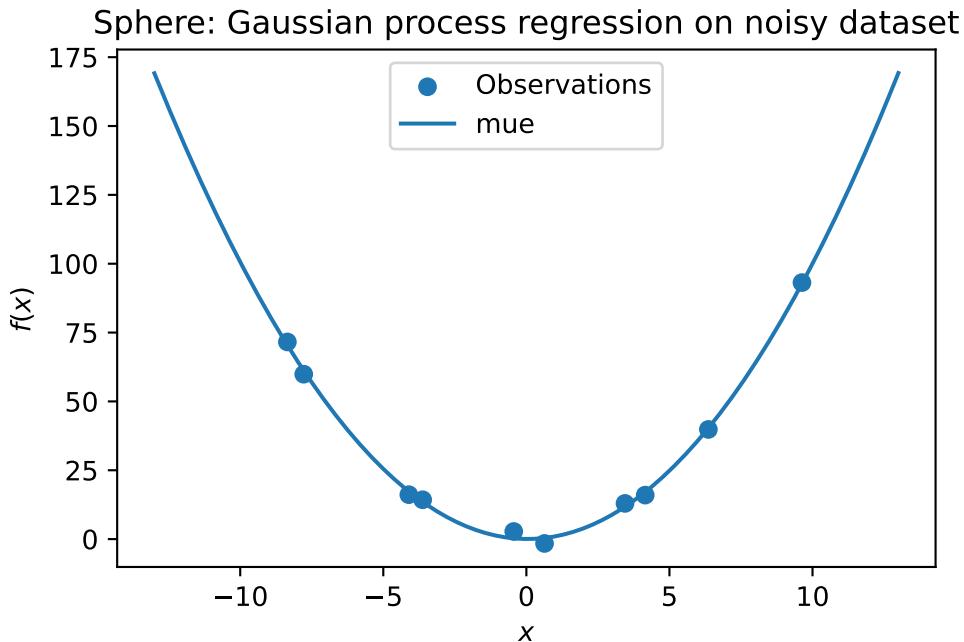
```

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")

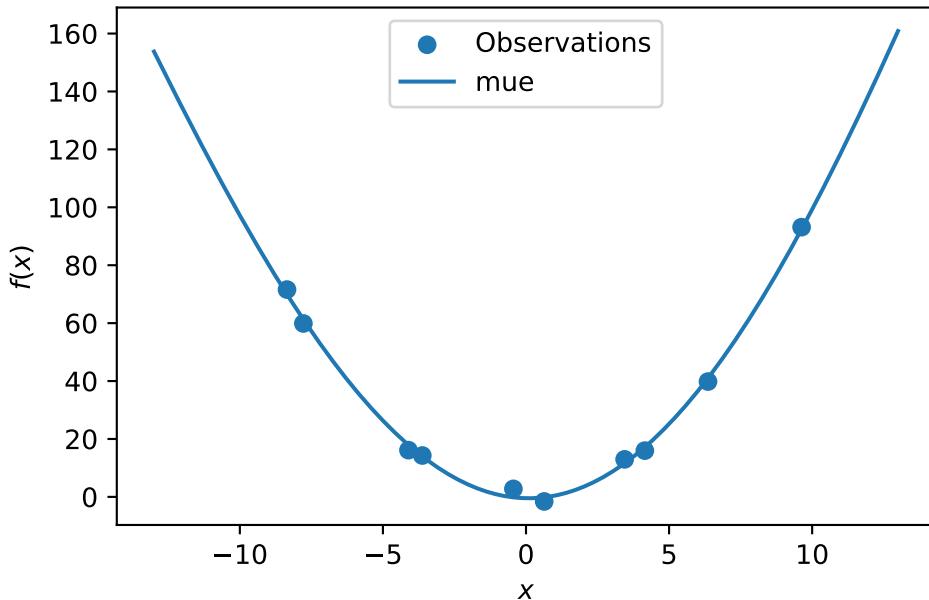
```



- 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
```

```
5.2085090734655785e-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\_bartz09\_2023-07-17\_08-51-43

### 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)

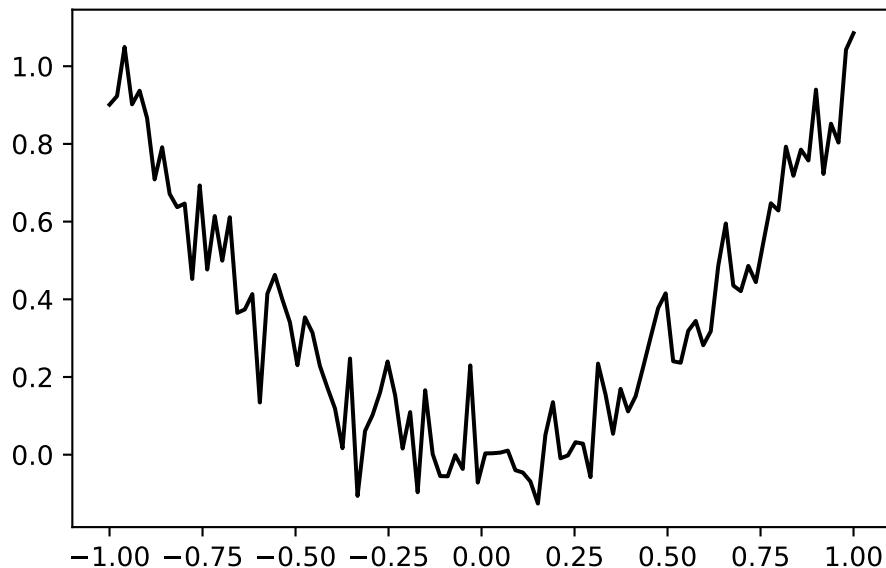
```

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]),

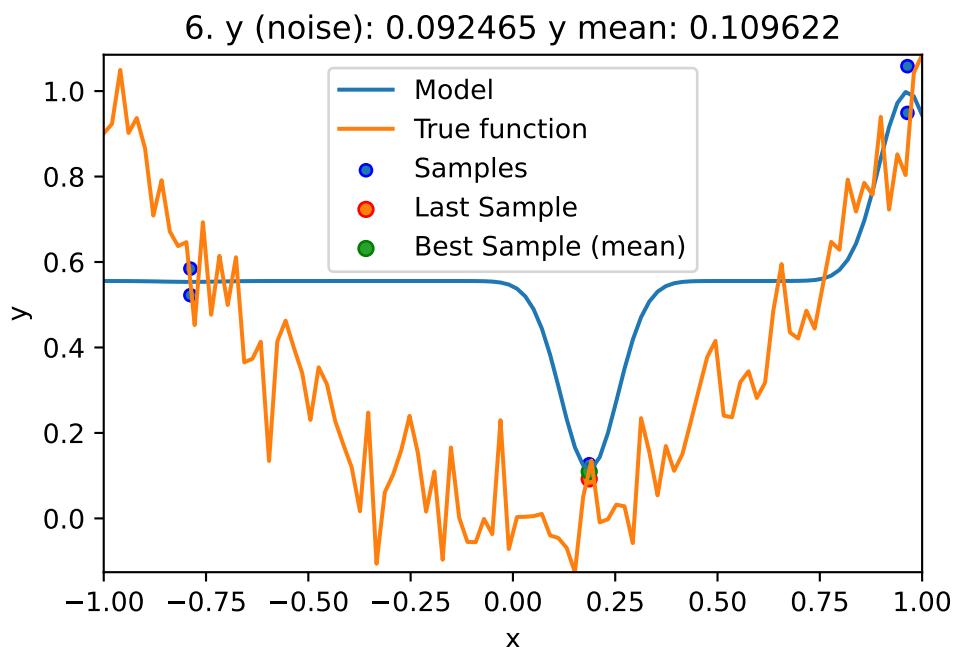
```

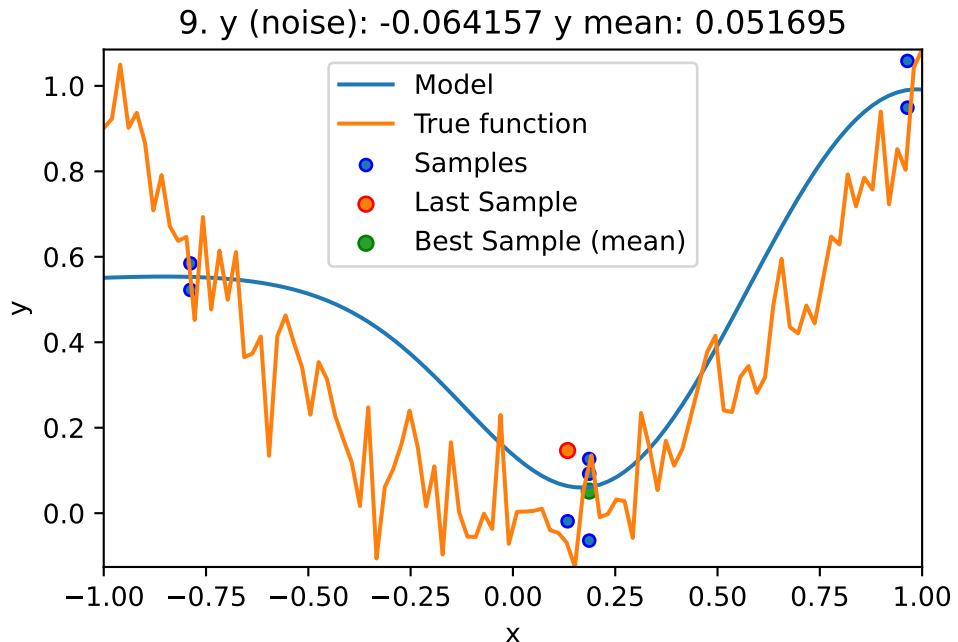
```

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})

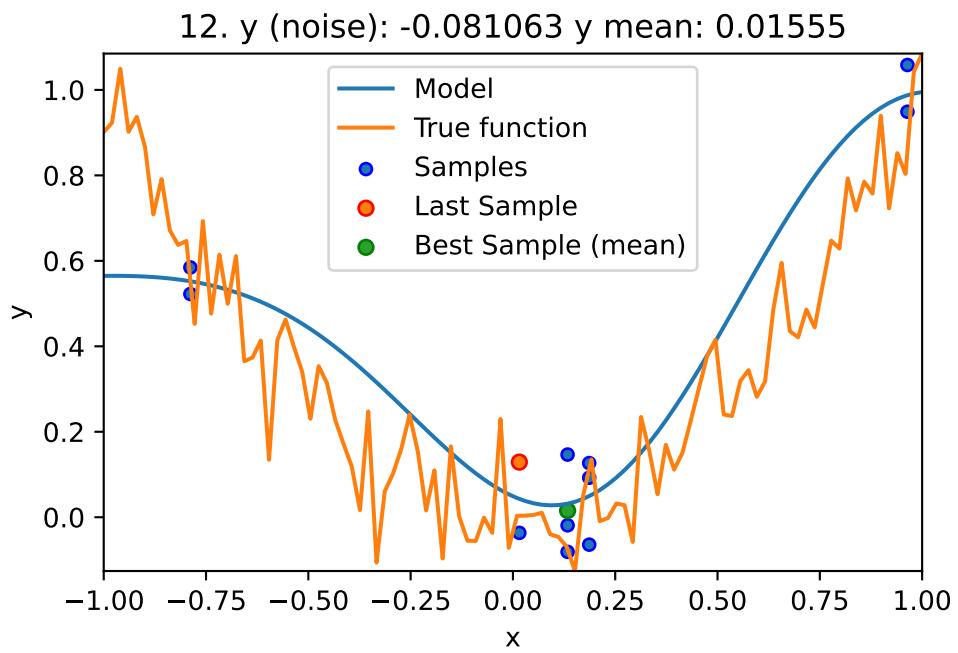
```

```
spot_1_noisy.run()
```

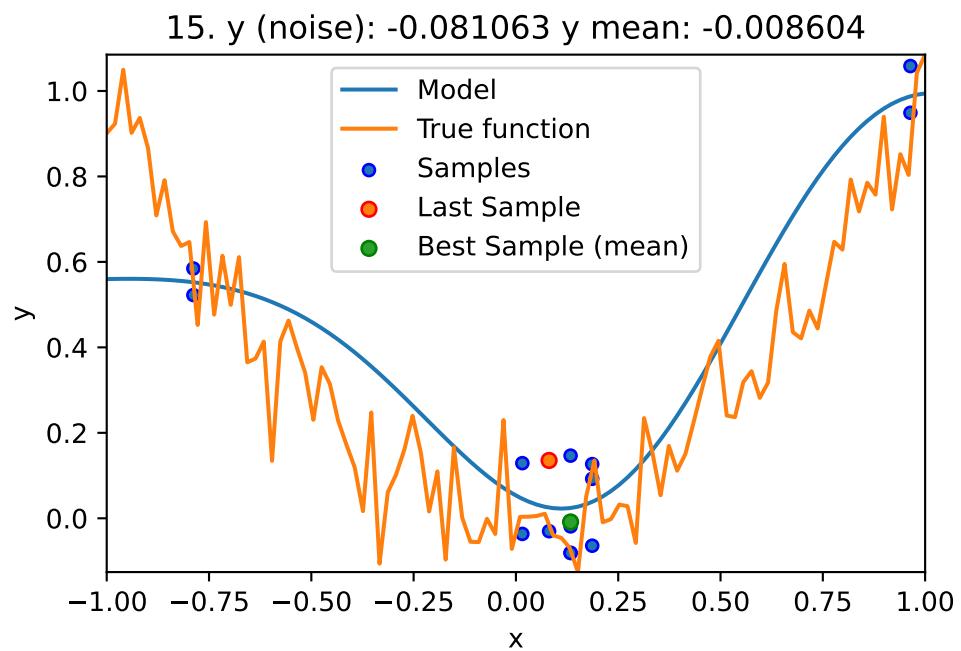




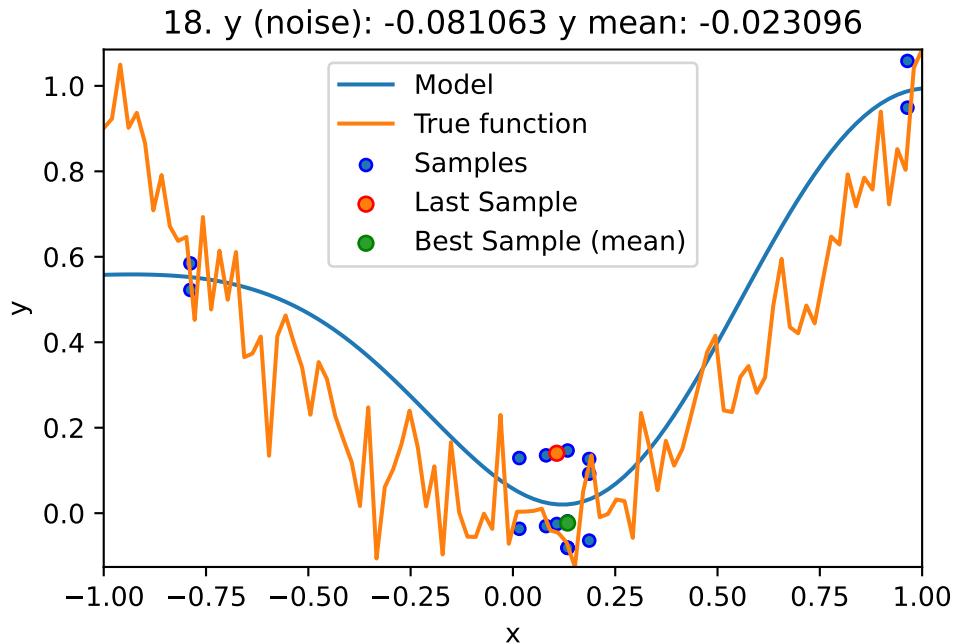
spotPython tuning: -0.0641572013655628 [#####-----] 45.00%



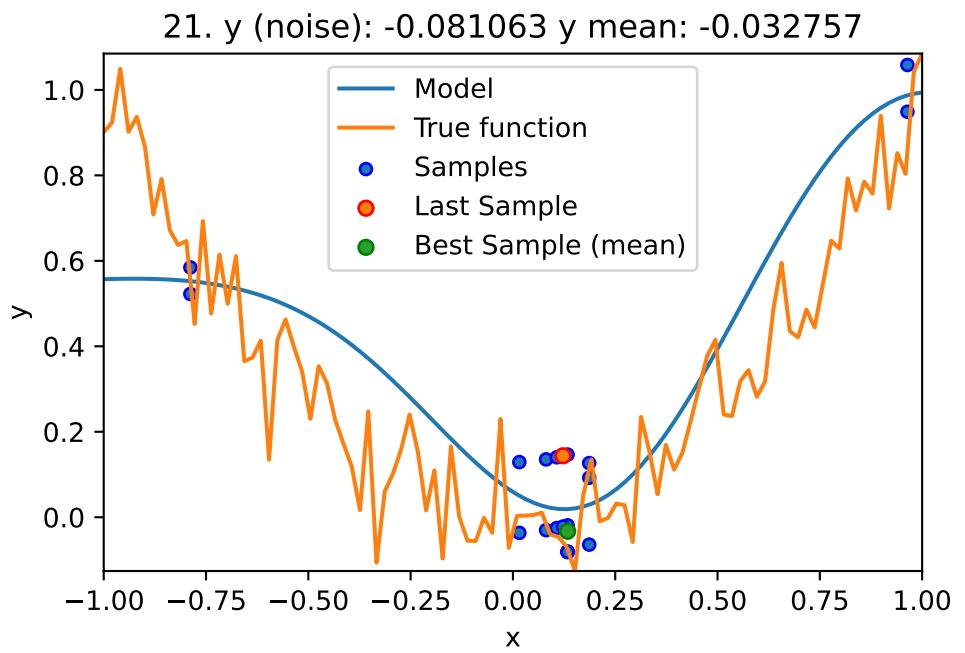
```
spotPython tuning: -0.08106318979661208 [#####----] 60.00%
```



```
spotPython tuning: -0.08106318979661208 [#####----] 75.00%
```



spotPython tuning: -0.08106318979661208 [#####-] 90.00%



```
spotPython tuning: -0.08106318979661208 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x17d973f70>
```

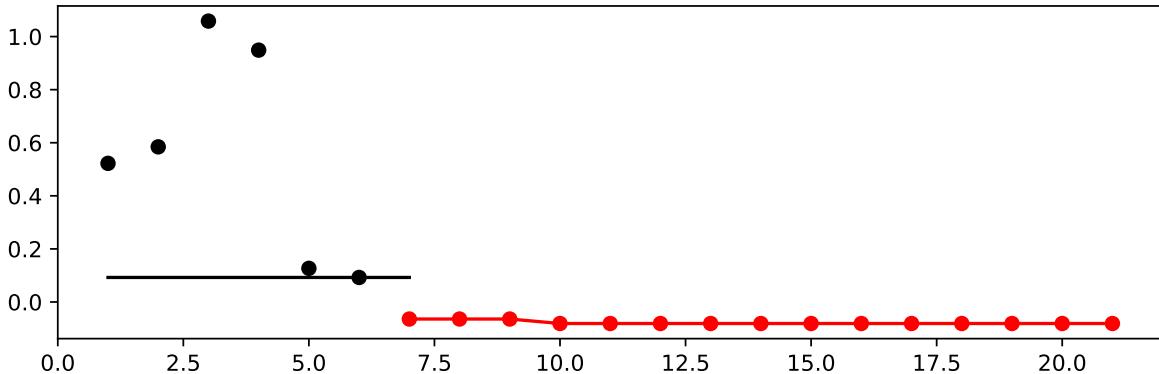
## 9.2 Print the Results

```
spot_1_noisy.print_results()
```

```
min y: -0.08106318979661208
x0: 0.1335999447536301
min mean y: -0.03275683462132762
x0: 0.1335999447536301
```

```
[['x0', 0.1335999447536301], ['x0', 0.1335999447536301]]
```

```
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:

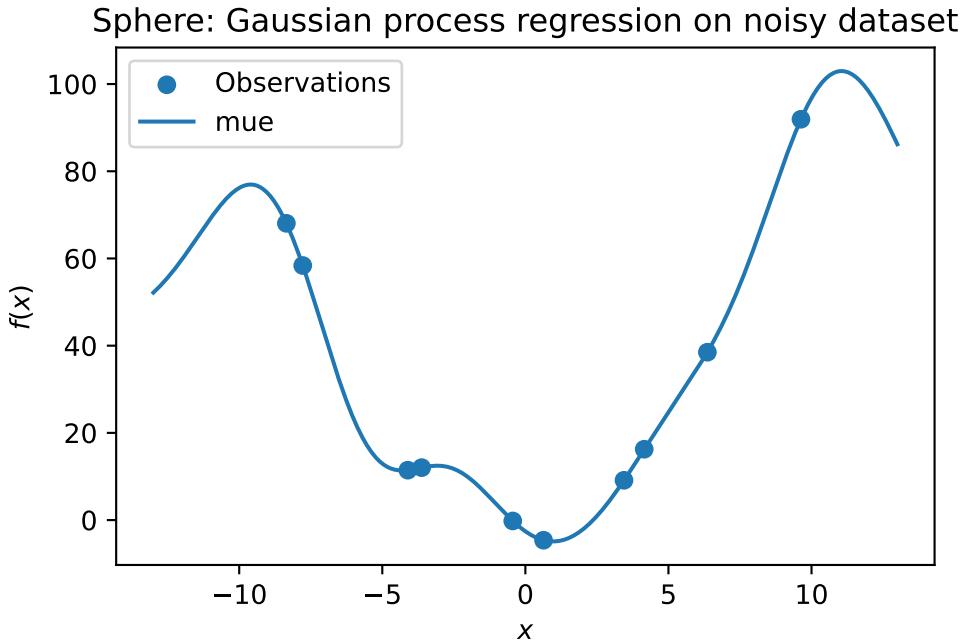
```

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")

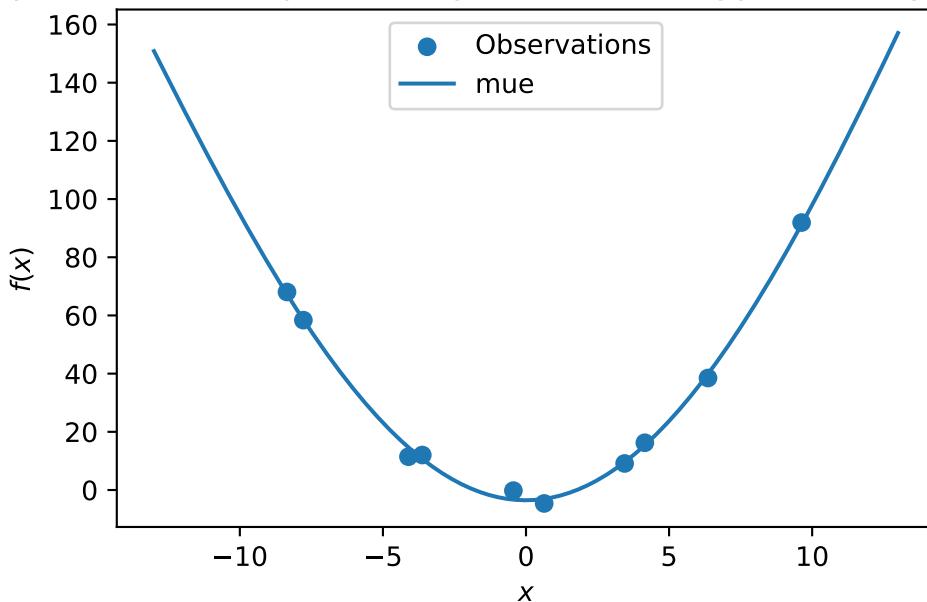
```



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
```

```
9.088150066416743e-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

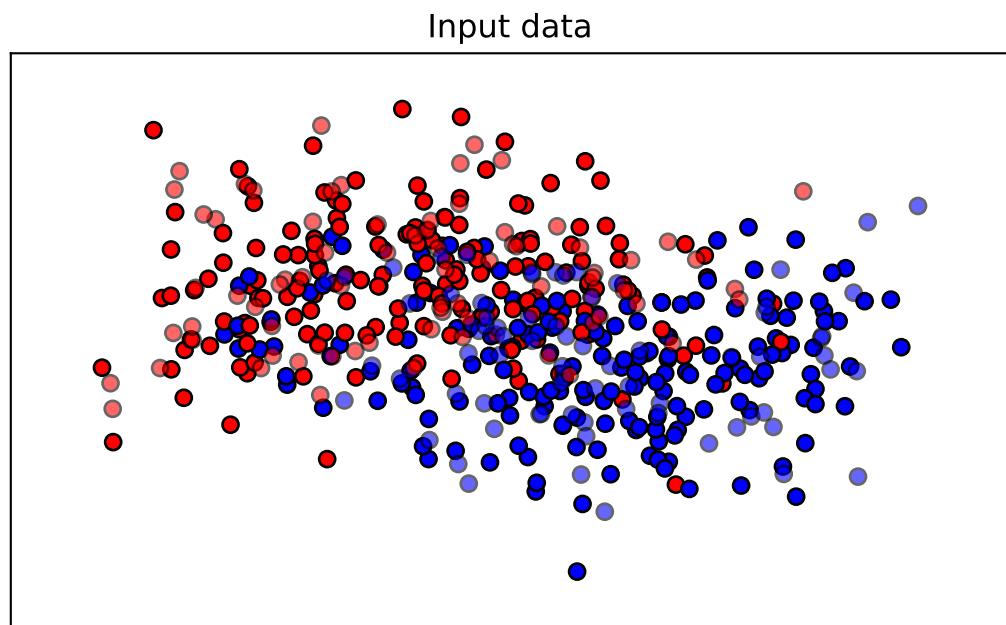
x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

```

```

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}]}
```

## i 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 12.6.

### 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 12.6.1.

## 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 [-----] 1.53%
```

```
spotPython tuning: 5.734217584632275 [-----] 3.47%
```

```
spotPython tuning: 5.734217584632275 [#-----] 5.57%
```

```
spotPython tuning: 5.734217584632275 [#-----] 7.52%
```

```
spotPython tuning: 5.734217584632275 [#-----] 9.29%
spotPython tuning: 5.734217584632275 [#-----] 11.16%
spotPython tuning: 5.734217584632275 [#-----] 13.14%
spotPython tuning: 5.734217584632275 [##-----] 21.26%
spotPython tuning: 5.734217584632275 [###-----] 29.05%
spotPython tuning: 5.734217584632275 [####-----] 38.06%
spotPython tuning: 5.734217584632275 [#####-----] 46.45%
spotPython tuning: 5.734217584632275 [#####-----] 56.03%
spotPython tuning: 5.734217584632275 [#####-----] 65.53%
spotPython tuning: 5.734217584632275 [#####-----] 73.32%
spotPython tuning: 5.734217584632275 [#####-----] 85.10%
spotPython tuning: 5.734217584632275 [#####-----] 93.14%
spotPython tuning: 5.734217584632275 [#####-----] 99.32%
spotPython tuning: 5.734217584632275 [#####-----] 100.00% Done...
<spotPython.spot.spot.Spot at 0x297b03f70>
```

## 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")
```

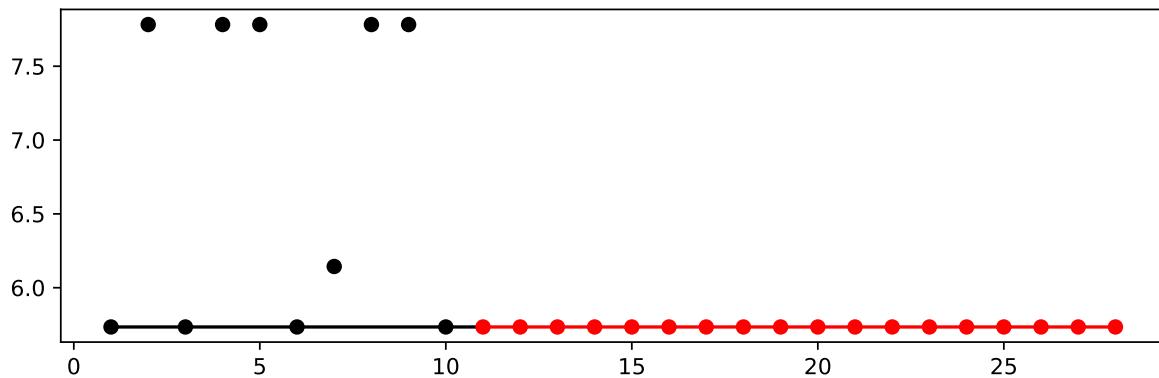


Figure 10.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

- 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+"_importance.png")
```

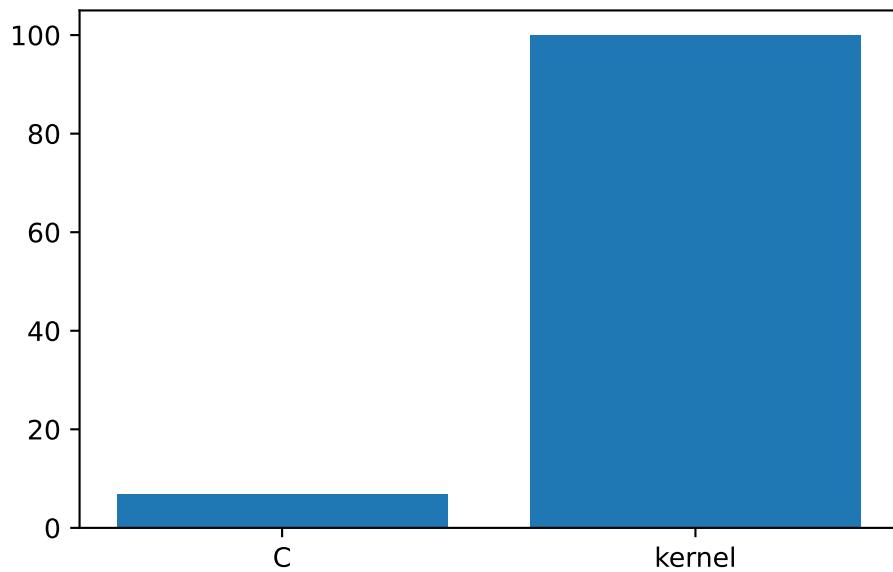


Figure 10.2: Variable importance plot, threshold 0.025.

### 10.9.2 Get Default Hyperparameters

```
from spotPython.hyperparameters.values import get_default_values, transform_hyper_parameter_values
values_default = get_default_values(fun_control)
values_default = transform_hyper_parameter_values(fun_control=fun_control, hyper_parameter_values_default=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_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': 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))])

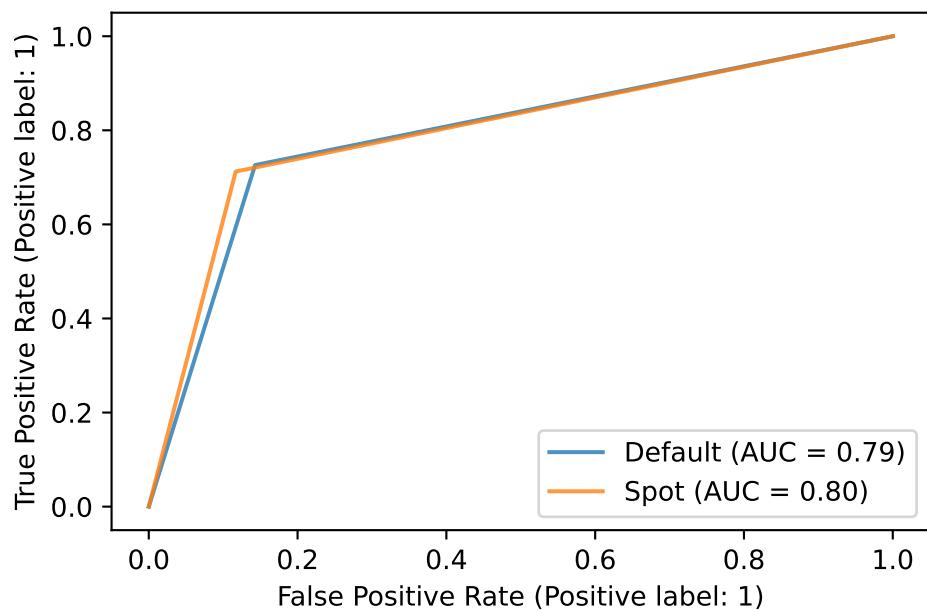
```

#### 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"])

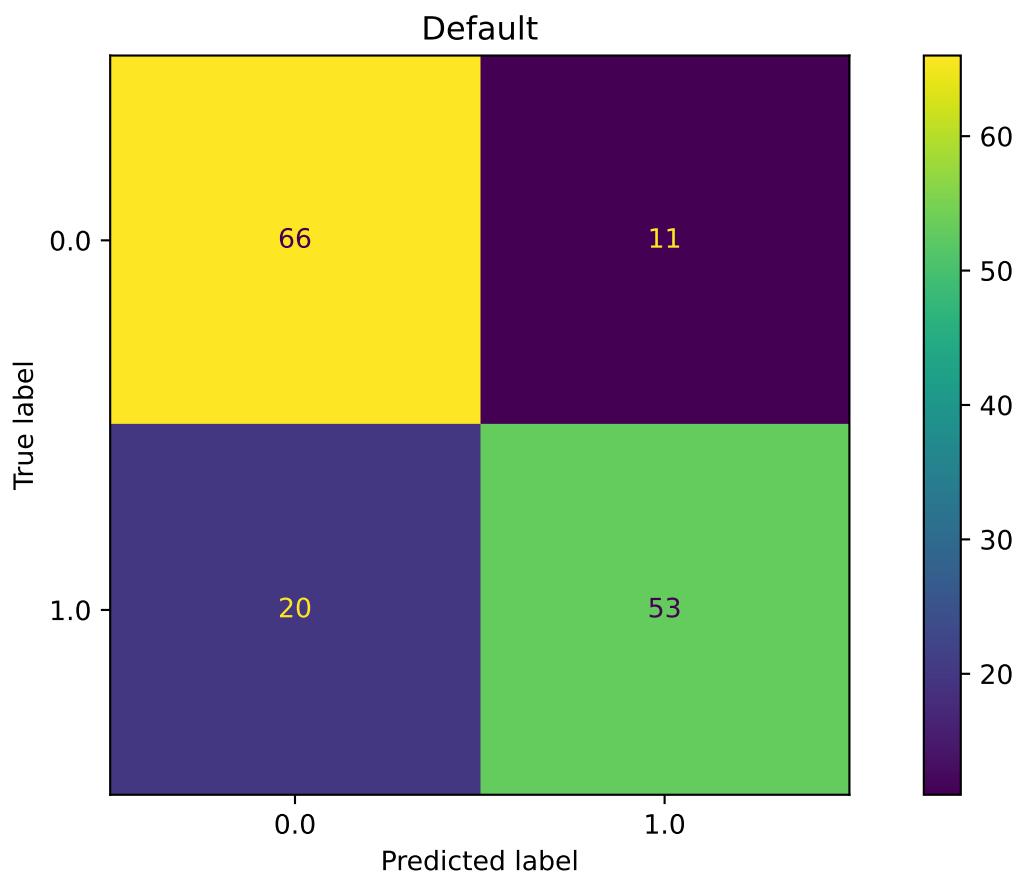
```



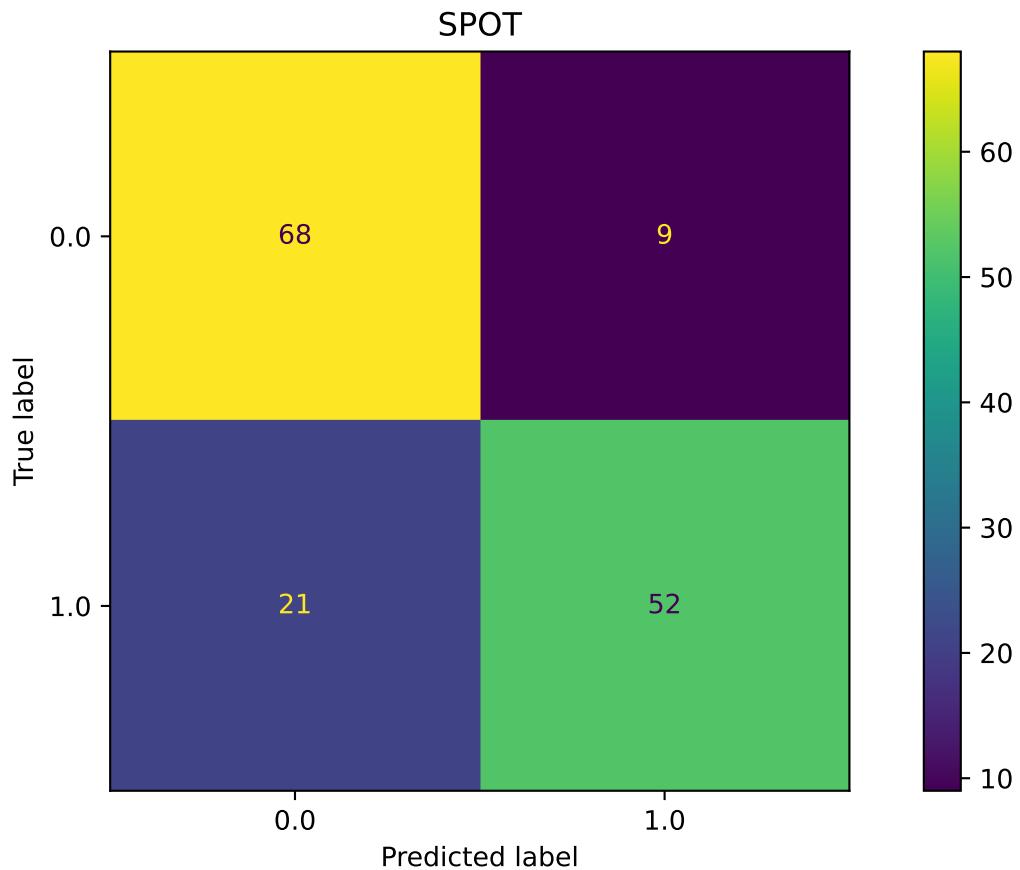
```

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")
```



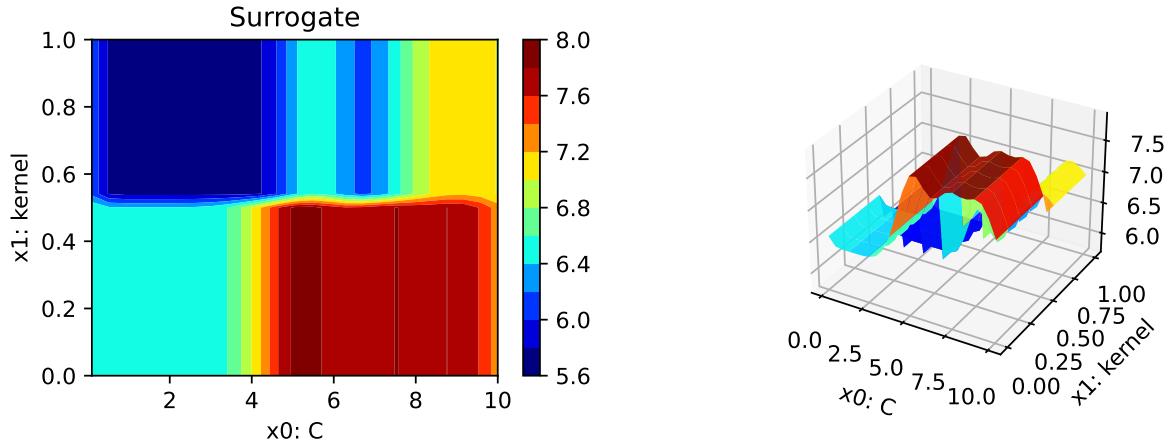
```
min(spot_tuner.y), max(spot_tuner.y)
```

```
(5.734217584632275, 7.782152436286657)
```

### 10.9.5 Detailed Hyperparameter Plots

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

```
C: 6.78742297418671
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="10-river"
```

```

K = .1

import os
from spotPython.utils.file import get_experiment_name
experiment_name = get_experiment_name(prefix=PREFIX)
print(experiment_name)

```

10-river\_bartz09\_2023-07-17\_08-56-52

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

experiment_name = get_experiment_name(prefix=PREFIX)
fun_control = fun_control_init(
    spot_tensorboard_path=get_spot_tensorboard_path(experiment_name),
    TENSORBOARD_CLEAN=True)

```

### 💡 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 11.8.4 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_prune`. 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_prune", [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_preprune	factor	0	0	0	None

## 11.7 Selection of the Objective 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 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

**i** 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: 2.1954027176053987 [##-----] 18.25%

```
spotPython tuning: 2.1954027176053987 [###-----] 34.32%
spotPython tuning: 2.1954027176053987 [#####-----] 48.89%
spotPython tuning: 2.1558528518089006 [#####----] 63.67%
spotPython tuning: 2.1189652804422368 [#####---] 75.61%
spotPython tuning: 2.1189652804422368 [#####--] 84.84%
spotPython tuning: 2.1189652804422368 [#####] 100.00% Done...
<spotPython.spot.spot.Spot at 0x2a258b730>
```

#### 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.

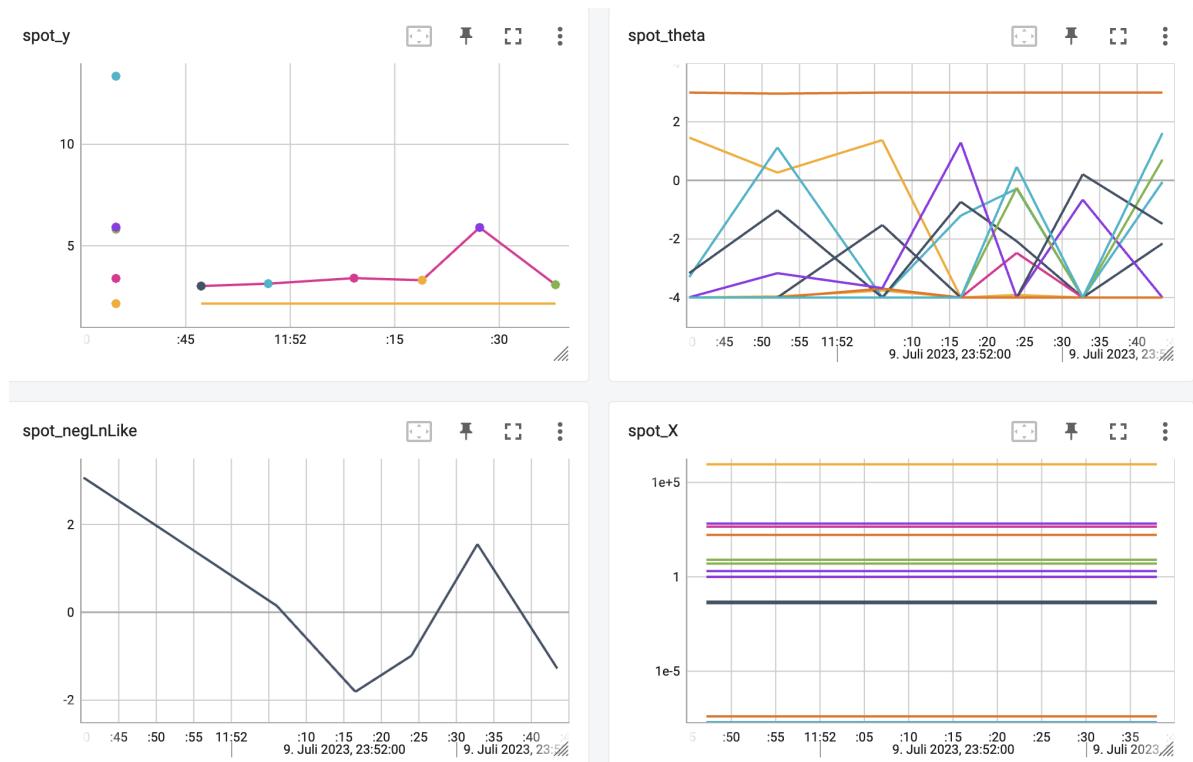


Figure 11.1: TensorBoard visualization of the spotPython optimization process and the surrogate model.

### 11.8.5 Results

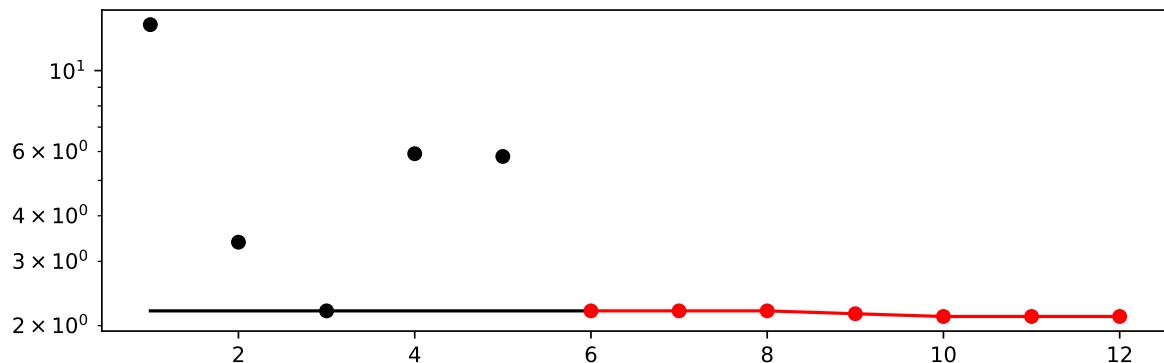
After the hyperparameter tuning run is finished, the results can be saved and reloaded with the following commands:

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

```
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.

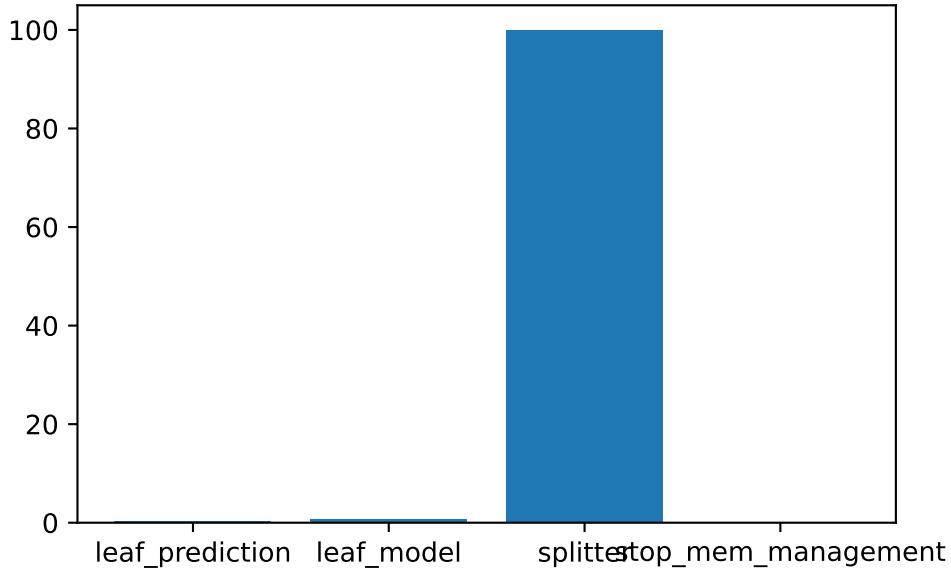
```
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

name	type	default	lower	upper	type
grace_period	int	200	10.0	1000.0	50
max_depth	int	20	2.0	20.0	10
delta	float	1e-07	1e-10	1e-06	1e-06
tau	float	0.05	0.01	0.1	0.01
leaf_prediction	factor	mean	0.0	2.0	0.0

leaf_model	factor	LinearRegression	0.0	2.0
model_selector_decay	float	0.95	0.9	0.99
splitter	factor	EBSTSplitter	0.0	2.0
min_samples_split	int	5	2.0	10.0
bootstrap_sampling	factor	0	0.0	1.0
drift_window_threshold	int	300	100.0	500.0
switch_significance	float	0.05	0.01	0.1
binary_split	factor	0	0.0	1.0
max_size	float	500.0	100.0	1000.0
memory_estimate_period	int	1000000	1000000.0	1000000.0
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.

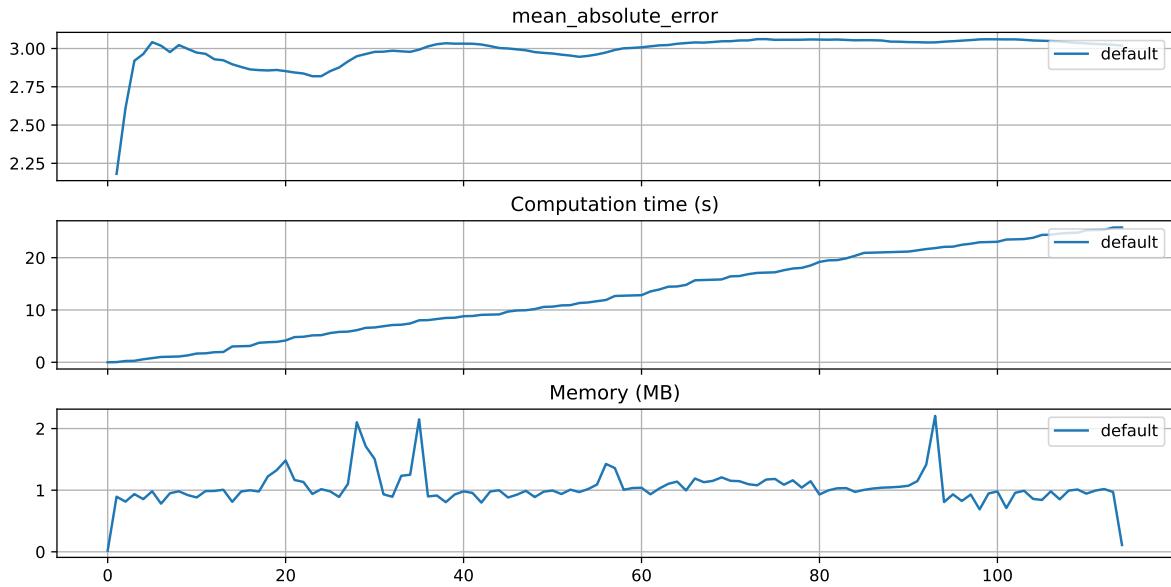
The model with the default hyperparameters can be trained and evaluated with the following commands:

```
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., `scaoe` (metric), runtime, and memory consumption, can be visualized with the following commands:

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

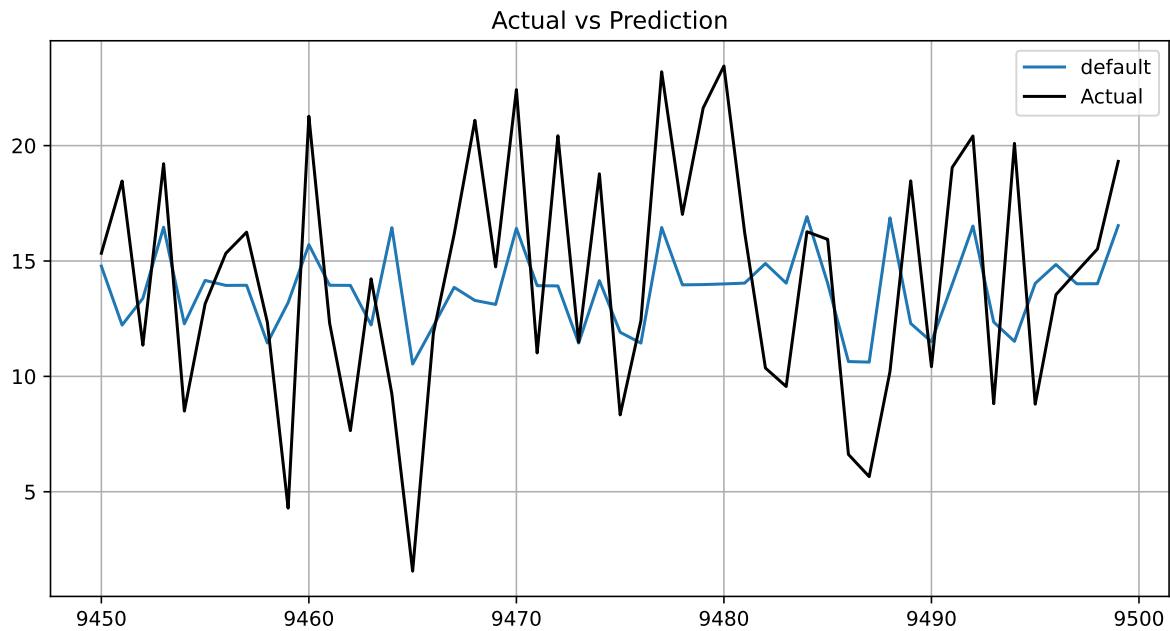


### 11.10.1 Show Predictions

- 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_co
```



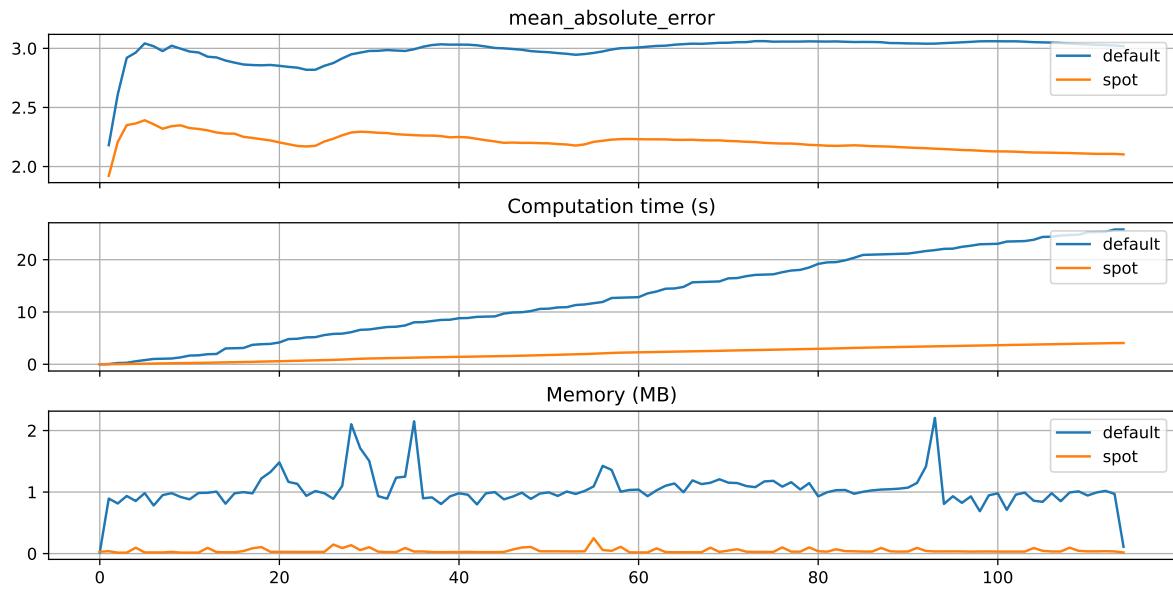
## 11.11 Get SPOT Results

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

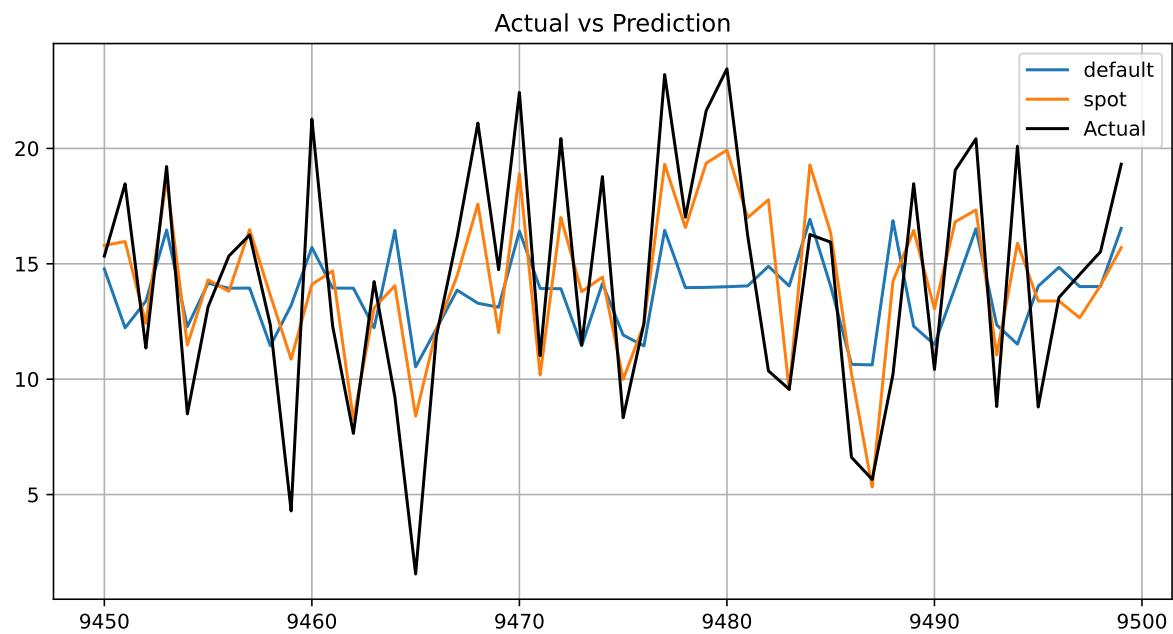
```
from spotPython.hyperparameters.values import get_one_core_model_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
model_spot = get_one_core_model_from_X(X, fun_control)

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
```



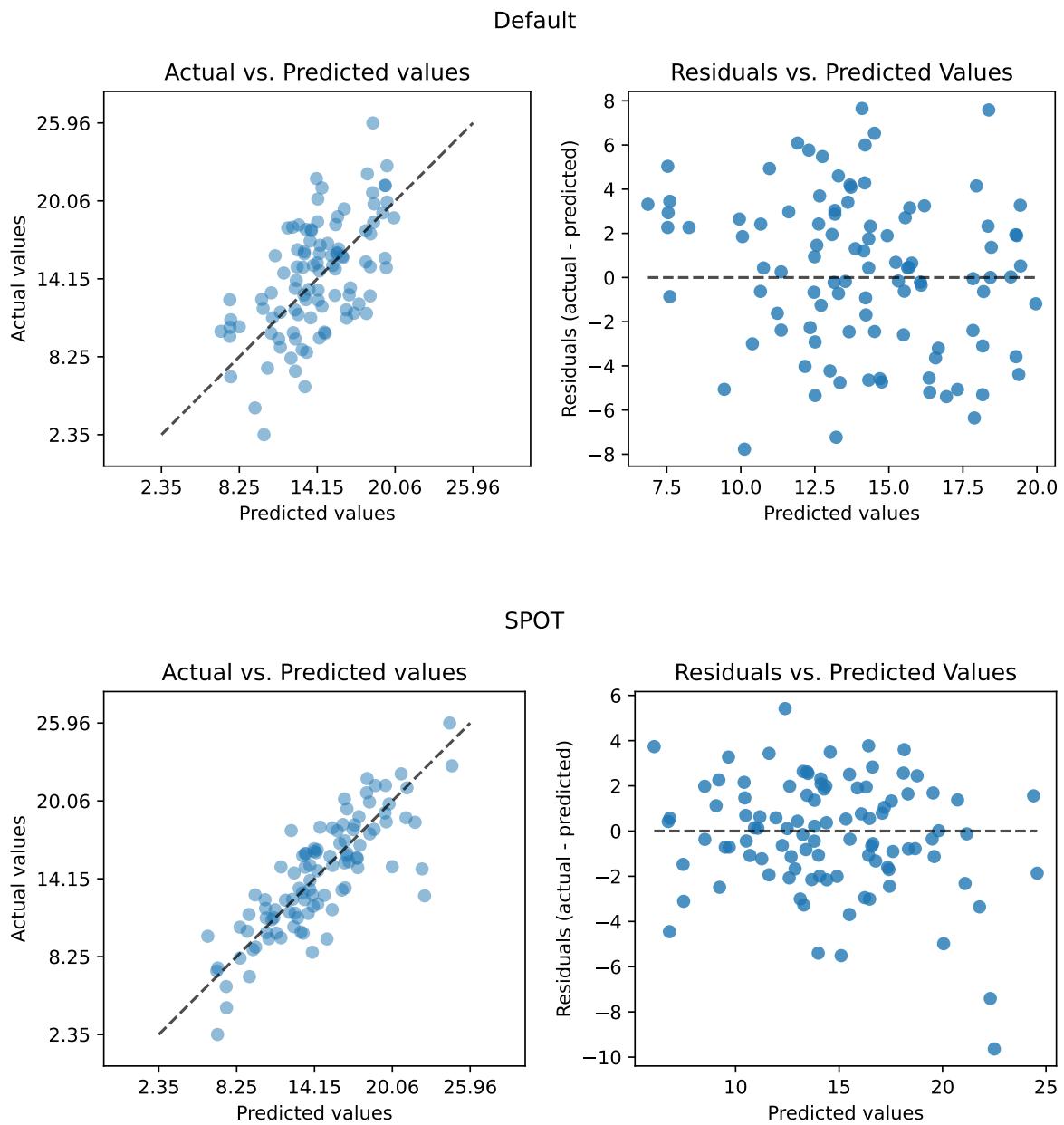
```
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b], df_true_spot[a:b]], targ
```



```

from spotPython.plot.validation import plot_actual_vs_predicted
plot_actual_vs_predicted(y_test=df_true_default["y"], y_pred=df_true_default["Prediction"])
plot_actual_vs_predicted(y_test=df_true_spot["y"], y_pred=df_true_spot["Prediction"], title="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': 21,
 'n_branches': 10,
 'n_leaves': 11,
 'n_active_leaves': -3919,
 'n_inactive_leaves': 3956,
 'height': 5,
 'total_observed_weight': 39002.0,
 'n_alternate_trees': 21,
 'n_pruned_alternate_trees': 5,
 'n_switch_alternate_trees': 1}

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

```

Parameter	Default	Spot
n_nodes	35	21
n_branches	17	10
n_leaves	18	11
n_active_leaves	96	-3919
n_inactive_leaves	0	3956
height	6	5
total_observed_weight	39002	39002
n_alternate_trees	21	21
n_pruned_alternate_trees	6	5
n_switch_alternate_trees	2	1

## 11.13 Detailed Hyperparameter Plots

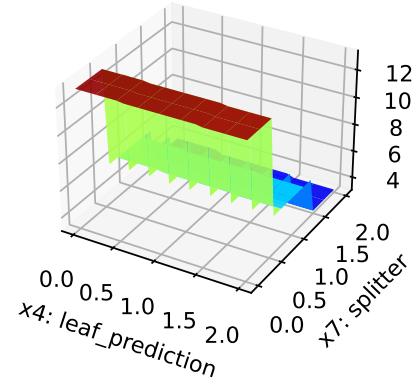
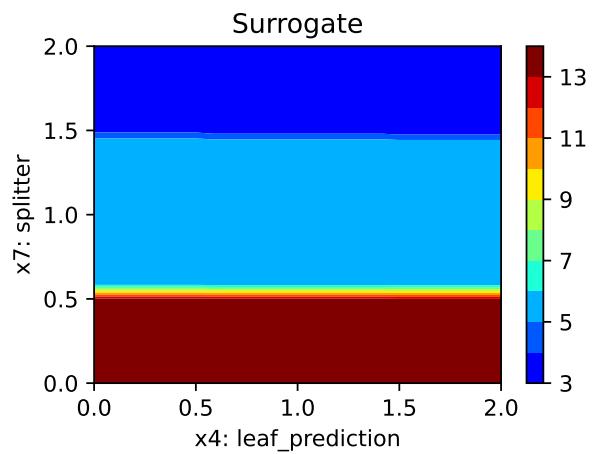
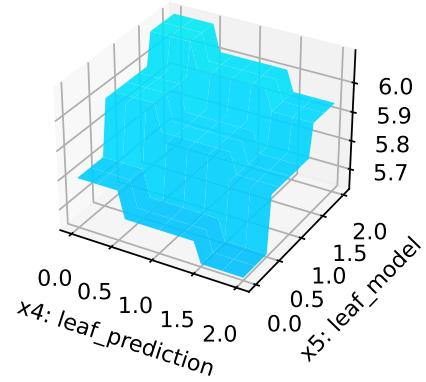
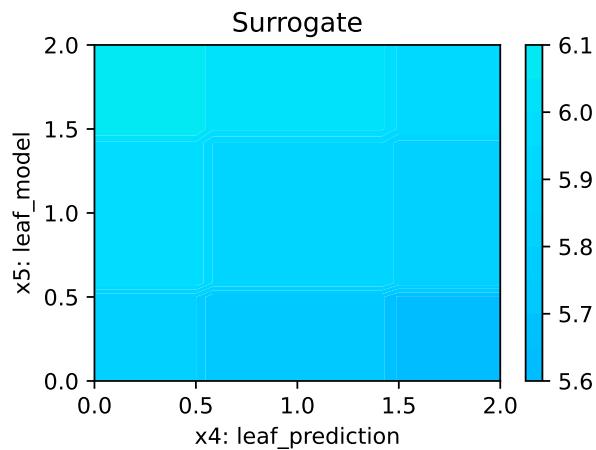
```

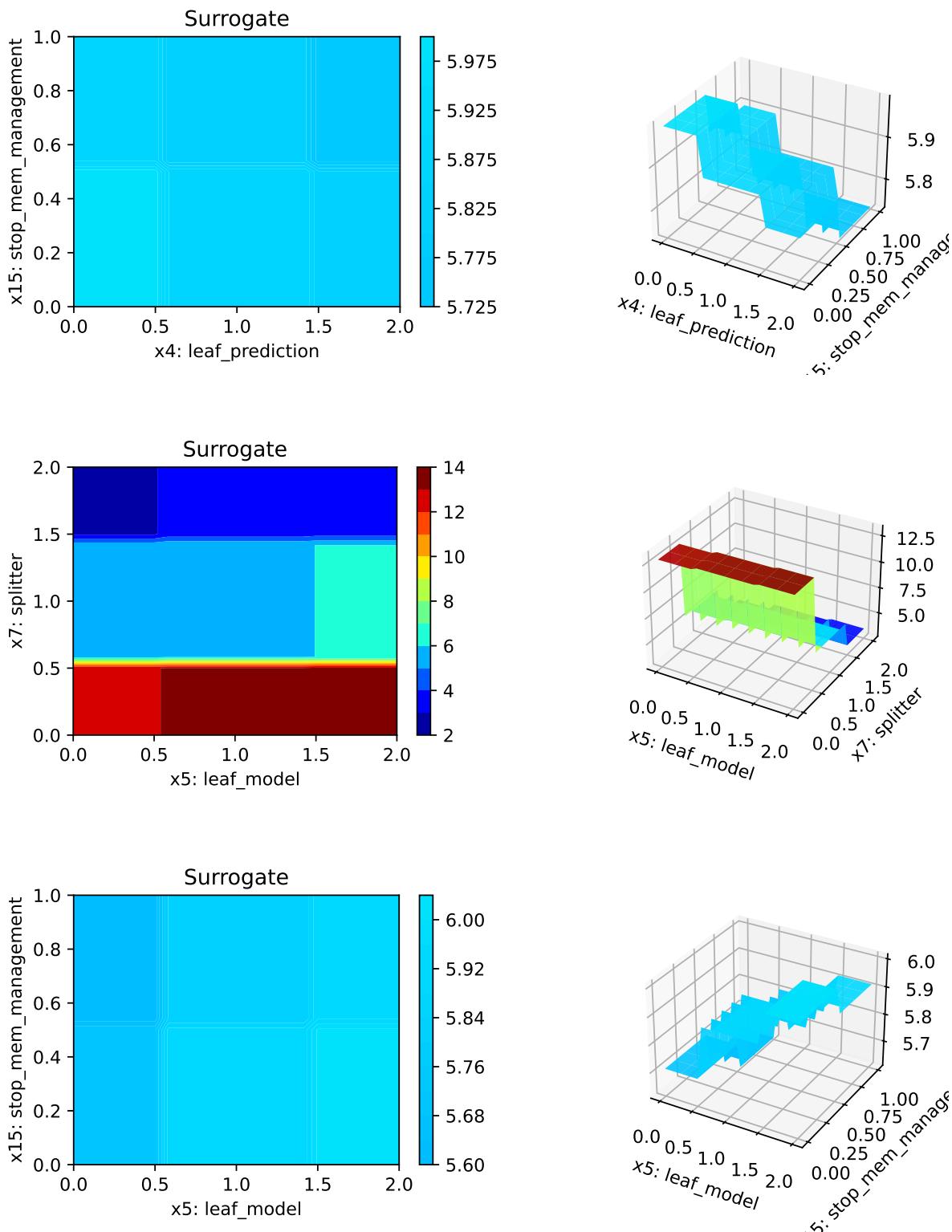
filename = "./figures/" + experiment_name
spot_tuner.plot_important_hyperparameter_contour(filename=filename)

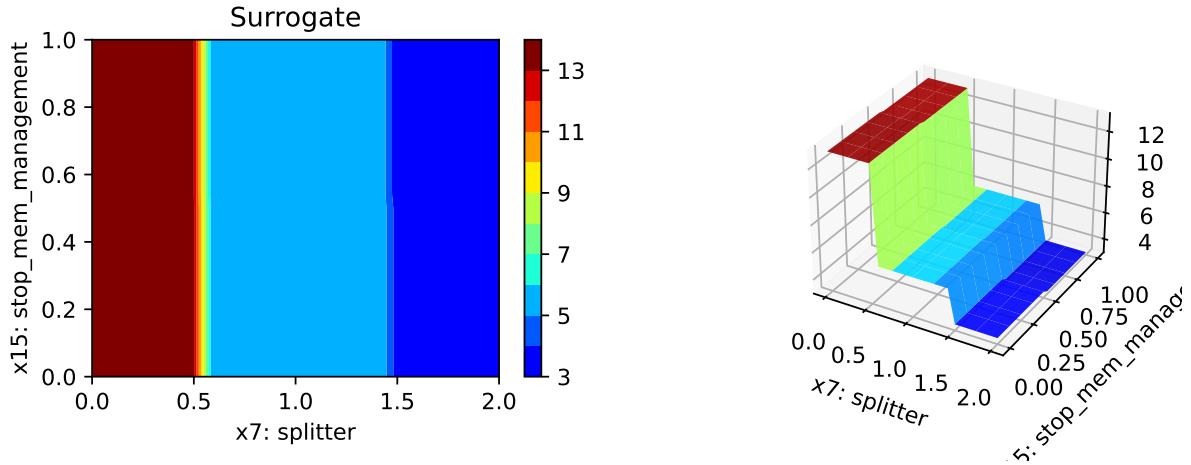
leaf_prediction: 0.25541913168507413

```

```
leaf_model: 0.771487672697361  
splitter: 100.0  
stop_mem_management: 0.14201240670317347
```







## 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 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.

**i** 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 17

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

1. Loading the data (training and test datasets), see Section 12.3.
2. Specification of the preprocessing model, see Section 12.4. 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 12.5. 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 12.6.0.1. 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 12.7.5.
6. Calling SPOT with the corresponding parameters, see Section 12.8.4. The results are stored in a dictionary and are available for further analysis.
7. Presentation, visualization and interpretation of the results, see Section 12.10.

`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>.

## 12.1 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")

```

## 12.2 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,)

```

## 12.3 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

## 12.4 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})
```

## 12.5 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 12.5.0.1 first, before the extended implementation with `spotPython` is shown in Section 12.5.0.2.

### 12.5.0.1 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)
```

### 12.5.0.2 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
```

### 12.5.1 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
```

### 12.5.2 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)
```

### 12.5.3 The Search Space: Hyperparameters

In Section 12.5.4, 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.

### 12.5.4 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.

## 12.5.5 Configuring the Search Space With `spotPython`

### 12.5.5.1 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}]}
```

## 12.6 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.

### 12.6.0.1 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.

### 12.6.0.2 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])
```

### 12.6.0.3 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 12.6.1, 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"])
```

## 12.6.1 Optimizers

Table 12.1 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 12.1: 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])
```

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

### 12.7.1 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.

### 12.7.2 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).

### 12.7.3 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.

## 12.7.4 Overview of the Evaluation Settings

### 12.7.4.1 Settings for the Hyperparameter Tuning

An overview of the training evaluations is shown in Table 12.2. "train\_cv" and "test\_cv" use `sklearn.model_selection.KFold()` internally. More details on the data splitting are provided in Section A.14 (in the Appendix).

Table 12.2: 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.

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

#### 12.7.4.2.1 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.

#### 12.7.4.2.2 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 A.14.1.4 describes the final evaluation of the tuned architecture.

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

## 12.7.5 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})

```

## 12.8 Step 8: Calling the SPOT Function

### 12.8.1 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  $2^4 = 16$ .

### 12.8.2 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
```

### 12.8.3 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)
```

### 12.8.4 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 12.9.

```
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: 0.3889499902725220 | Loss: 1.6403590366363525 | Acc: 0.3889500000000000.
Epoch: 2 | MulticlassAccuracy: 0.4578999876976013 | Loss: 1.4816969134330749 | Acc: 0.4579000000000000.
Epoch: 3 | MulticlassAccuracy: 0.4945999979972839 | Loss: 1.3767625138282775 | Acc: 0.4946000000000000.
Epoch: 4 | MulticlassAccuracy: 0.5118499994277954 | Loss: 1.3446329971313478 | Acc: 0.5118500000000000.
Epoch: 5 | MulticlassAccuracy: 0.5447499752044678 | Loss: 1.2767737101554870 | Acc: 0.5447500000000000.
Epoch: 6 | MulticlassAccuracy: 0.5664499998092651 | Loss: 1.2234437763214112 | Acc: 0.5664500000000000.
Epoch: 7 |

```

```
MulticlassAccuracy: 0.5648499727249146 | Loss: 1.2325385323524476 | Acc: 0.5648500000000000
Epoch: 8 |

MulticlassAccuracy: 0.5896499752998352 | Loss: 1.1611093239784240 | Acc: 0.5896500000000000
Epoch: 9 |

MulticlassAccuracy: 0.6015999913215637 | Loss: 1.1370150957107543 | Acc: 0.6016000000000000
Epoch: 10 |

MulticlassAccuracy: 0.6074000000953674 | Loss: 1.1378371593475343 | Acc: 0.6074000000000001
Epoch: 11 |

MulticlassAccuracy: 0.6036999821662903 | Loss: 1.1592556796073914 | Acc: 0.6037000000000000
Epoch: 12 |

MulticlassAccuracy: 0.5997499823570251 | Loss: 1.1987680685997009 | Acc: 0.5997500000000000
Early stopping at epoch 11
Returned to Spot: Validation loss: 1.1987680685997009

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

MulticlassAccuracy: 0.3920499980449677 | Loss: 1.6102165319681168 | Acc: 0.3920500000000000
Epoch: 2 |

MulticlassAccuracy: 0.4390000104904175 | Loss: 1.5077767979741097 | Acc: 0.4390000000000000
Epoch: 3 |

MulticlassAccuracy: 0.4700999855995178 | Loss: 1.4581756867766380 | Acc: 0.4701000000000000
Epoch: 4 |

MulticlassAccuracy: 0.4981499910354614 | Loss: 1.3969129746913911 | Acc: 0.4981500000000000
Epoch: 5 |

MulticlassAccuracy: 0.5059000253677368 | Loss: 1.3693460956692696 | Acc: 0.5059000000000000
Epoch: 6 |

MulticlassAccuracy: 0.5133500099182129 | Loss: 1.3540988440275192 | Acc: 0.5133500000000000
Epoch: 7 |
```

```
MulticlassAccuracy: 0.5081499814987183 | Loss: 1.3817692994177342 | Acc: 0.5081500000000000
Epoch: 8 |

MulticlassAccuracy: 0.5159500241279602 | Loss: 1.3653468480706215 | Acc: 0.5159500000000000
Returned to Spot: Validation loss: 1.3653468480706215

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

MulticlassAccuracy: 0.0958499982953072 | Loss: 2.3086834851264952 | Acc: 0.0958500000000000
Epoch: 2 |

MulticlassAccuracy: 0.0987000018358231 | Loss: 2.3107500833988190 | Acc: 0.0987000000000000
Epoch: 3 |

MulticlassAccuracy: 0.0958499982953072 | Loss: 2.3054559610605239 | Acc: 0.0958500000000000
Epoch: 4 |

MulticlassAccuracy: 0.1013000011444092 | Loss: 2.3091404678583145 | Acc: 0.1013000000000000
Epoch: 5 |

MulticlassAccuracy: 0.0958499982953072 | Loss: 2.3109533527135850 | Acc: 0.0958500000000000
Epoch: 6 |

MulticlassAccuracy: 0.0987000018358231 | Loss: 2.3080133529186249 | Acc: 0.0987000000000000
Early stopping at epoch 5
Returned to Spot: Validation loss: 2.308013352918625

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

MulticlassAccuracy: 0.3910000026226044 | Loss: 1.6194829273104667 | Acc: 0.3910000000000000
Epoch: 2 |

MulticlassAccuracy: 0.4532499909400940 | Loss: 1.5181912495672703 | Acc: 0.4532500000000000
Epoch: 3 |

MulticlassAccuracy: 0.5023999810218811 | Loss: 1.3594324642419815 | Acc: 0.5024000000000000
Epoch: 4 |
```

MulticlassAccuracy: 0.5066999793052673 | Loss: 1.3639220094040037 | Acc: 0.5067000000000000  
Epoch: 5 |

MulticlassAccuracy: 0.5313000082969666 | Loss: 1.3084210138827563 | Acc: 0.5313000000000000  
Epoch: 6 |

MulticlassAccuracy: 0.5376499891281128 | Loss: 1.3020537653062492 | Acc: 0.5376500000000000  
Epoch: 7 |

MulticlassAccuracy: 0.5404999852180481 | Loss: 1.2979997927054763 | Acc: 0.5405000000000000  
Epoch: 8 |

MulticlassAccuracy: 0.5505999922752380 | Loss: 1.2794678398683668 | Acc: 0.5506000000000000  
Returned to Spot: Validation loss: 1.2794678398683668

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

MulticlassAccuracy: 0.4688499867916107 | Loss: 1.4396714681148528 | Acc: 0.4688500000000000  
Epoch: 2 |

MulticlassAccuracy: 0.4978500008583069 | Loss: 1.3743870592117309 | Acc: 0.4978500000000000  
Epoch: 3 |

MulticlassAccuracy: 0.5149000287055969 | Loss: 1.3301207626819611 | Acc: 0.5149000000000000  
Epoch: 4 |

MulticlassAccuracy: 0.5352500081062317 | Loss: 1.2803554334163665 | Acc: 0.5352500000000000  
Epoch: 5 |

MulticlassAccuracy: 0.5407999753952026 | Loss: 1.2673199267387389 | Acc: 0.5407999999999999  
Epoch: 6 |

MulticlassAccuracy: 0.5474500060081482 | Loss: 1.2426155496835709 | Acc: 0.5474500000000000  
Epoch: 7 |

MulticlassAccuracy: 0.5532000064849854 | Loss: 1.2252585200309754 | Acc: 0.5532000000000000  
Epoch: 8 |

MulticlassAccuracy: 0.5598499774932861 | Loss: 1.2217366221427917 | Acc: 0.5598500000000000  
Epoch: 9 |

MulticlassAccuracy: 0.5702000260353088 | Loss: 1.2027698907375335 | Acc: 0.5702000000000000  
Epoch: 10 |

MulticlassAccuracy: 0.5695499777793884 | Loss: 1.1946598905563355 | Acc: 0.5695500000000000  
Epoch: 11 |

MulticlassAccuracy: 0.5720999836921692 | Loss: 1.1931119963169099 | Acc: 0.5721000000000001  
Epoch: 12 |

MulticlassAccuracy: 0.5777500271797180 | Loss: 1.1757407437086105 | Acc: 0.5777500000000000  
Epoch: 13 |

MulticlassAccuracy: 0.5833500027656555 | Loss: 1.1655059050798415 | Acc: 0.5833500000000000  
Epoch: 14 |

MulticlassAccuracy: 0.5854499936103821 | Loss: 1.1665637883186339 | Acc: 0.5854500000000000  
Epoch: 15 |

MulticlassAccuracy: 0.5885499715805054 | Loss: 1.1581050729990006 | Acc: 0.5885500000000000  
Epoch: 16 |

MulticlassAccuracy: 0.5877500176429749 | Loss: 1.1598053013563157 | Acc: 0.5877500000000000  
Returned to Spot: Validation loss: 1.1598053013563157

config: {'l1': 64, 'l2': 256, 'lr\_mult': 1.0, 'batch\_size': 16, 'epochs': 16, 'k\_folds': 0,  
Epoch: 1 |

MulticlassAccuracy: 0.4435999989509583 | Loss: 1.5161994444847107 | Acc: 0.4436000000000000  
Epoch: 2 |

MulticlassAccuracy: 0.4676499962806702 | Loss: 1.4507200250148773 | Acc: 0.4676500000000000  
Epoch: 3 |

MulticlassAccuracy: 0.4885500073432922 | Loss: 1.4064176963806152 | Acc: 0.4885500000000000  
Epoch: 4 |

```
MulticlassAccuracy: 0.4984500110149384 | Loss: 1.3765785826206207 | Acc: 0.4984500000000000
Epoch: 5 |

MulticlassAccuracy: 0.5091999769210815 | Loss: 1.3492139563083649 | Acc: 0.5092000000000000
Epoch: 6 |

MulticlassAccuracy: 0.5235000252723694 | Loss: 1.3260424315452575 | Acc: 0.5235000000000000
Epoch: 7 |

MulticlassAccuracy: 0.5347999930381775 | Loss: 1.2992566047668457 | Acc: 0.5348000000000001
Epoch: 8 |

MulticlassAccuracy: 0.5384500026702881 | Loss: 1.2924042490005494 | Acc: 0.5384500000000000
Epoch: 9 |

MulticlassAccuracy: 0.5433999896049500 | Loss: 1.2770100817918777 | Acc: 0.5434000000000000
Epoch: 10 |

MulticlassAccuracy: 0.5457999706268311 | Loss: 1.2646812784671784 | Acc: 0.5458000000000000
Epoch: 11 |

MulticlassAccuracy: 0.5486000180244446 | Loss: 1.2627830792903900 | Acc: 0.5486000000000000
Epoch: 12 |

MulticlassAccuracy: 0.5608000159263611 | Loss: 1.2396654787063599 | Acc: 0.5608000000000000
Epoch: 13 |

MulticlassAccuracy: 0.5554000139236450 | Loss: 1.2407209475994110 | Acc: 0.5554000000000000
Epoch: 14 |

MulticlassAccuracy: 0.5677000284194946 | Loss: 1.2263578844547272 | Acc: 0.5677000000000000
Epoch: 15 |

MulticlassAccuracy: 0.5665000081062317 | Loss: 1.2272662802696228 | Acc: 0.5665000000000000
Epoch: 16 |

MulticlassAccuracy: 0.5688999891281128 | Loss: 1.2138021411895752 | Acc: 0.5689000000000000
Returned to Spot: Validation loss: 1.2138021411895752
spotPython tuning: 1.1598053013563157 [#####] 100.00% Done...

<spotPython.spot.spot at 0x1553f78e0>
```

## 12.9 Step 9: Tensorboard

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

### 12.9.1 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.

### 12.9.2 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:
    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)
```

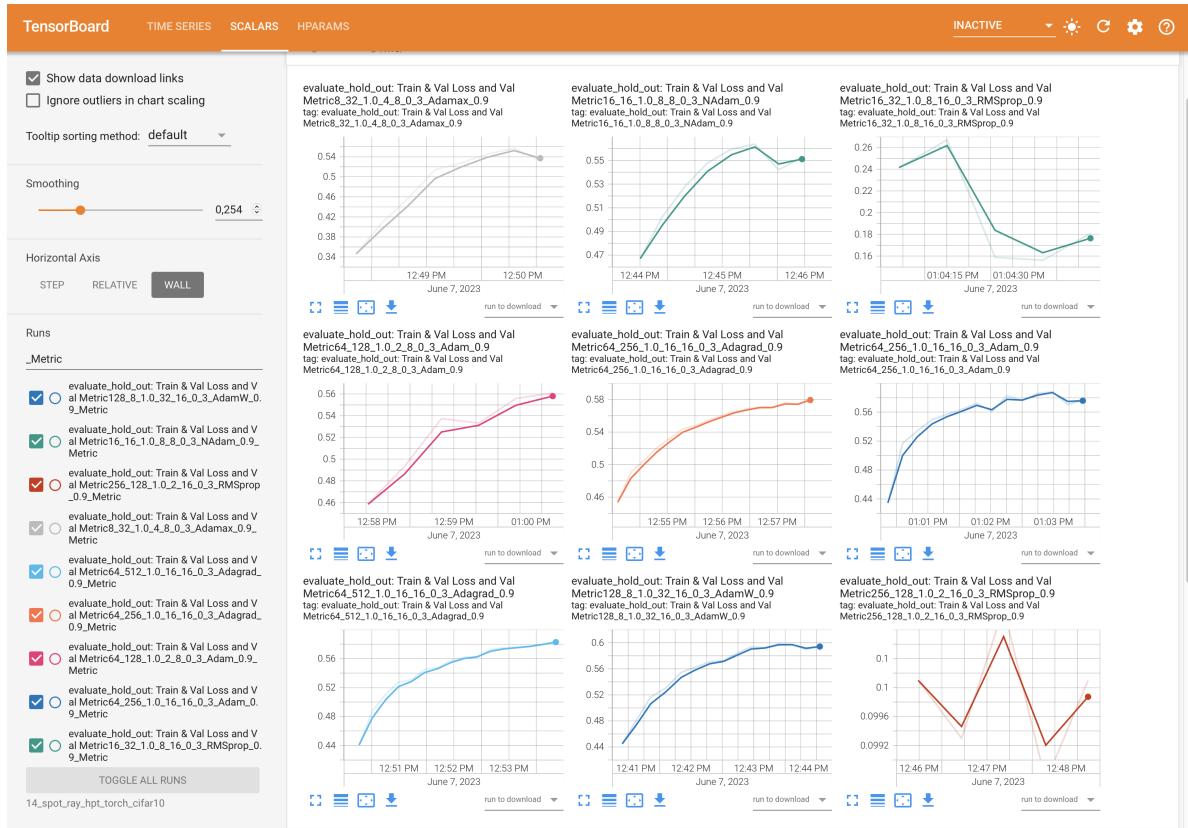


Figure 12.1: Tensorboard



Figure 12.2: Tensorboard

## 12.10 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")
```

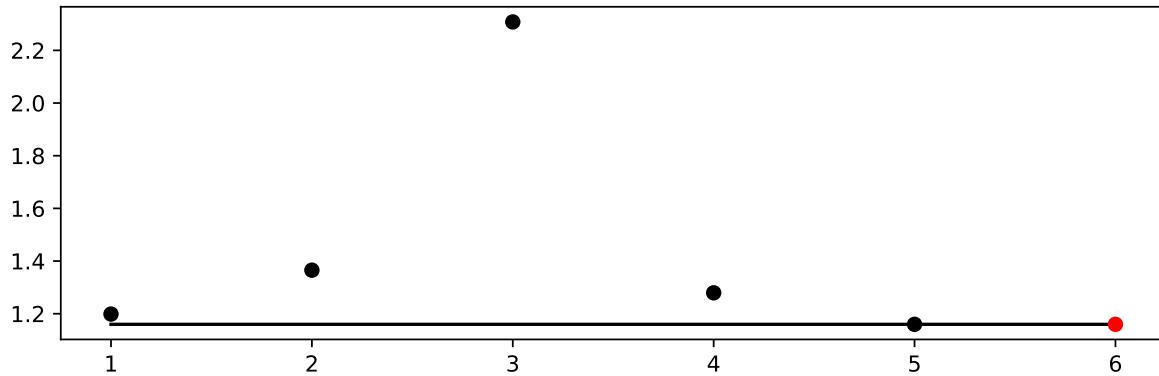


Figure 12.3: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

`?@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	6.0	transform_power_2_int
l2	int	5	2.0	9.0	9.0	transform_power_2_int
lr_mult	float	1.0	1.0	1.0	1.0	None
batch_size	int	4	1.0	5.0	4.0	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	1.0	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")
```

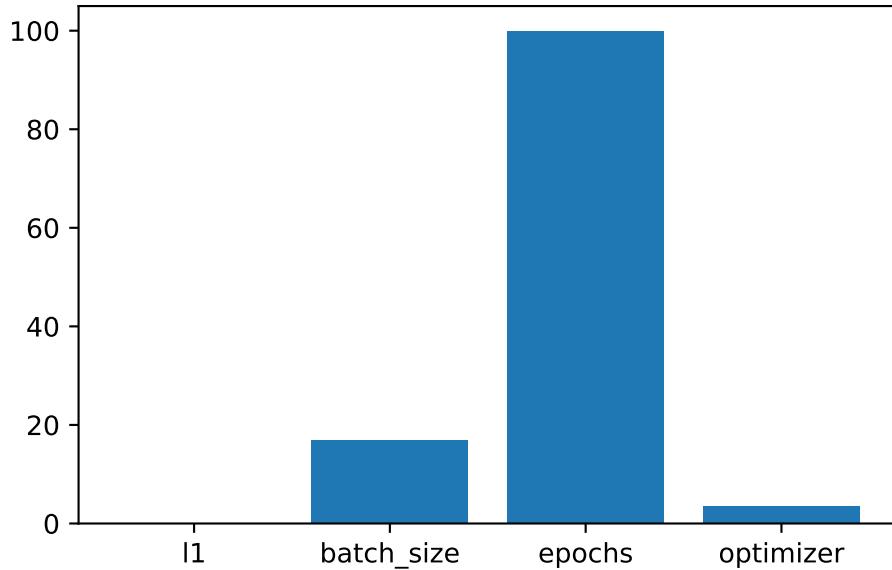


Figure 12.4: Variable importance plot, threshold 0.025.

### 12.10.1 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=64, bias=True)
    (fc2): Linear(in_features=64, out_features=512, bias=True)
    (fc3): Linear(in_features=512, out_features=10, bias=True)
)
```

### 12.10.2 Get Default Hyperparameters

In a similar manner as in Section 12.10.1, 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)
)
```

### 12.10.3 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: 0.1013000011444092 | Loss: 2.2993141119003297 | Acc: 0.1013000000000000.

Epoch: 2 |

MulticlassAccuracy: 0.1157499998807907 | Loss: 2.2862341335296632 | Acc: 0.1157500000000000.

Epoch: 3 |

MulticlassAccuracy: 0.1534000039100647 | Loss: 2.2558263620376588 | Acc: 0.1534000000000000.

Epoch: 4 |

MulticlassAccuracy: 0.2099500000476837 | Loss: 2.2096788969039918 | Acc: 0.2099500000000000.

Epoch: 5 |

```

MulticlassAccuracy: 0.2171999961137772 | Loss: 2.1583650140762329 | Acc: 0.2172000000000000.
Epoch: 6 |

MulticlassAccuracy: 0.2302500009536743 | Loss: 2.1003214435577391 | Acc: 0.2302500000000000.
Epoch: 7 |

MulticlassAccuracy: 0.2409500032663345 | Loss: 2.0469134126663207 | Acc: 0.2409500000000000.
Epoch: 8 |

MulticlassAccuracy: 0.2525500059127808 | Loss: 2.0065110932350159 | Acc: 0.2525500000000000.
Returned to Spot: Validation loss: 2.006511093235016

MulticlassAccuracy: 0.2576000094413757 | Loss: 2.0048375873565676 | Acc: 0.2576000000000000.
Final evaluation: Validation loss: 2.0048375873565676
Final evaluation: Validation metric: 0.25760000944137573
-----
(2.0048375873565676, nan, tensor(0.2576, device='mps:0'))

```

#### 12.10.4 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: 0.4553500115871429 | Loss: 1.4807784632682801 | Acc: 0.4553500000000000.  
Epoch: 2 |

MulticlassAccuracy: 0.4986500144004822 | Loss: 1.3824706964015960 | Acc: 0.4986500000000000.  
Epoch: 3 |

MulticlassAccuracy: 0.5169000029563904 | Loss: 1.3429181780815125 | Acc: 0.5169000000000000.  
Epoch: 4 |

MulticlassAccuracy: 0.5296999812126160 | Loss: 1.3132050466537475 | Acc: 0.5296999999999999.  
Epoch: 5 |

MulticlassAccuracy: 0.5366500020027161 | Loss: 1.2941528817415238 | Acc: 0.5366500000000000.  
Epoch: 6 |

MulticlassAccuracy: 0.5393499732017517 | Loss: 1.2878079622745513 | Acc: 0.5393500000000000.  
Epoch: 7 |

MulticlassAccuracy: 0.5490499734878540 | Loss: 1.2646987820148468 | Acc: 0.5490500000000000.  
Epoch: 8 |

MulticlassAccuracy: 0.5544499754905701 | Loss: 1.2544260616302489 | Acc: 0.5544500000000000.  
Epoch: 9 |

MulticlassAccuracy: 0.5620999932289124 | Loss: 1.2338094377756119 | Acc: 0.5621000000000000.  
Epoch: 10 |

MulticlassAccuracy: 0.5637500286102295 | Loss: 1.2312240300893784 | Acc: 0.5637500000000000.  
Epoch: 11 |

MulticlassAccuracy: 0.5688999891281128 | Loss: 1.2254522174358369 | Acc: 0.5689000000000000.  
Epoch: 12 |

```

MulticlassAccuracy: 0.5709999799728394 | Loss: 1.2168787500381471 | Acc: 0.5710000000000000
Epoch: 13 |

MulticlassAccuracy: 0.5732499957084656 | Loss: 1.2131404494524003 | Acc: 0.5732500000000000
Epoch: 14 |

MulticlassAccuracy: 0.5752500295639038 | Loss: 1.2019421347618102 | Acc: 0.5752500000000000
Epoch: 15 |

MulticlassAccuracy: 0.5807499885559082 | Loss: 1.1982519413948058 | Acc: 0.5807500000000000
Epoch: 16 |

MulticlassAccuracy: 0.5807999968528748 | Loss: 1.1949795161724091 | Acc: 0.5808000000000000
Returned to Spot: Validation loss: 1.194979516172409

MulticlassAccuracy: 0.5852000117301941 | Loss: 1.2008806512832642 | Acc: 0.5852000000000001
Final evaluation: Validation loss: 1.2008806512832642
Final evaluation: Validation metric: 0.5852000117301941
-----
(1.2008806512832642, nan, tensor(0.5852, device='mps:0'))

```

### 12.10.5 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)

11: 0.10134443931754378
batch_size: 16.862145330943314
epochs: 100.0
optimizer: 3.487626907795692

```

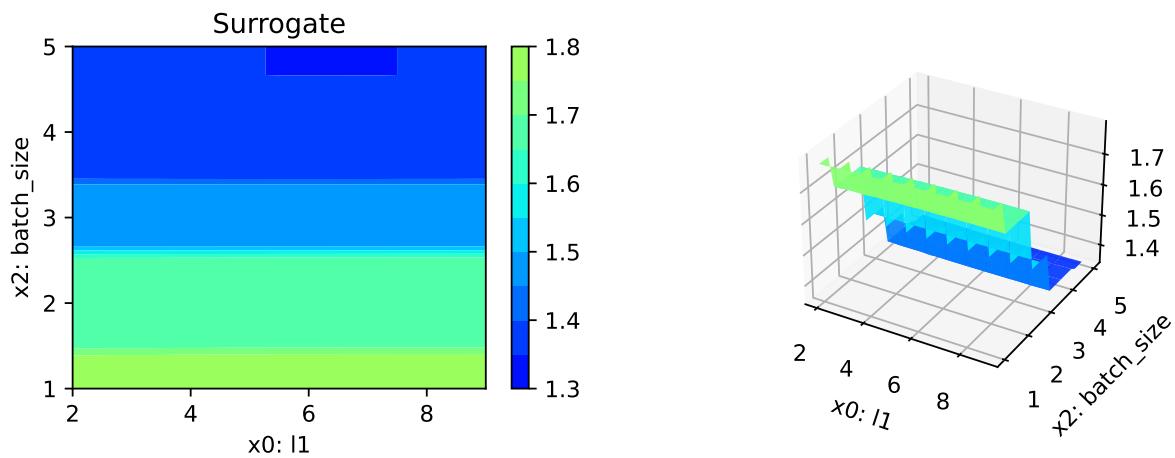
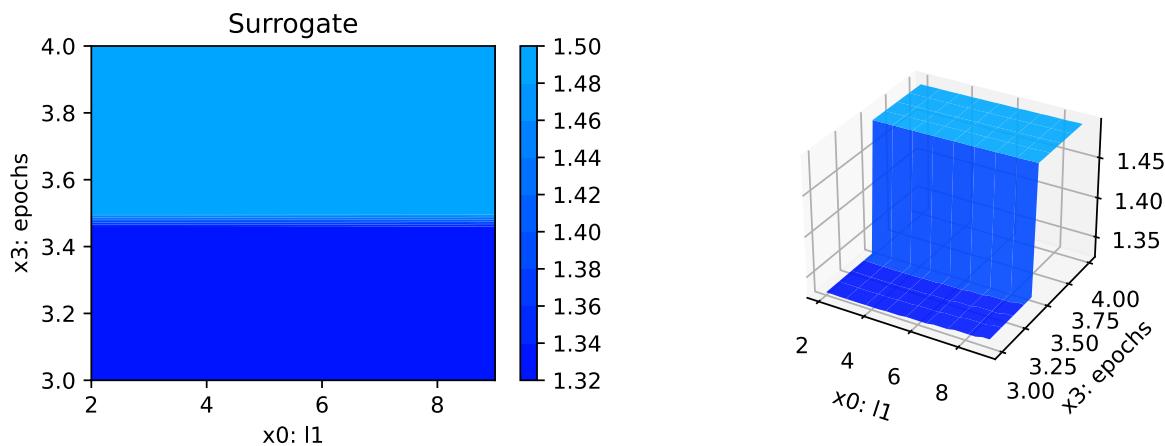
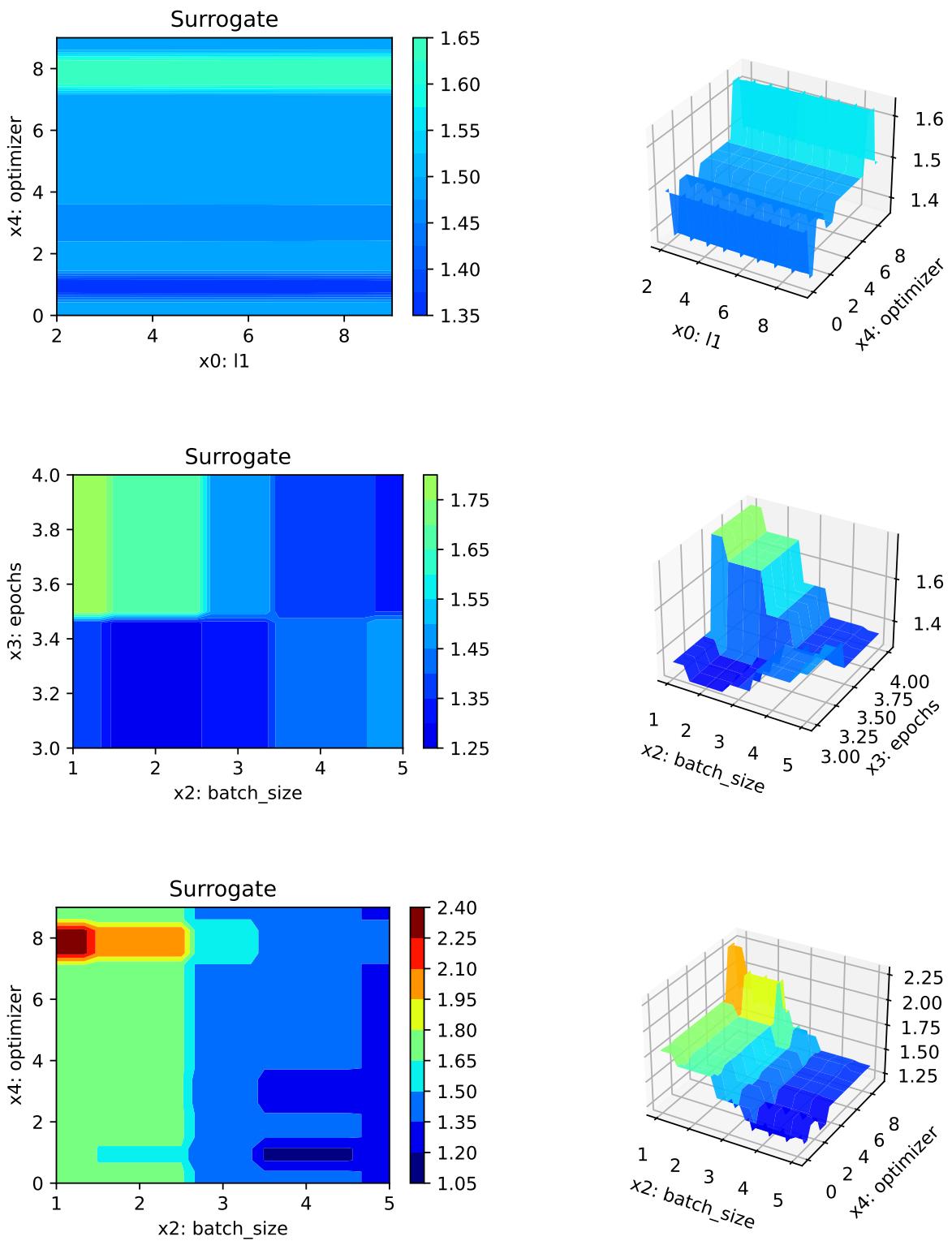
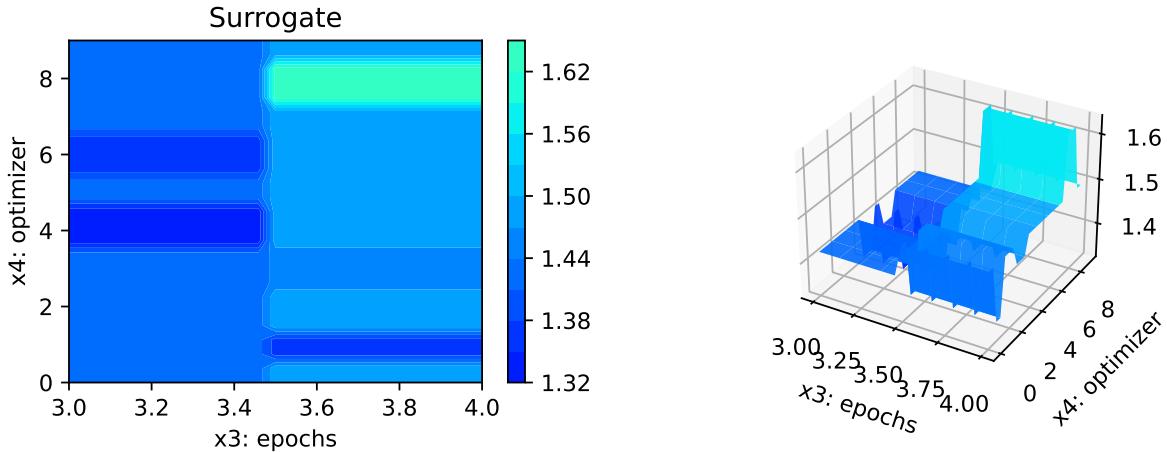


Figure 12.5: Contour plots.







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

## 12.11 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.

## 12.12 Appendix

### 12.12.1 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

Best trial config: {'11': 8, '12': 16, 'lr': 0.00276249, 'batch_size': 16, 'data_dir': '...'
Best trial final validation loss: 1.181501

Best trial final validation accuracy: 0.5836  
Best trial test set accuracy: 0.5806

# 13 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.

## 13.1 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")
```

## 13.2 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))
```

## 13.3 Step 3: PyTorch Data Loading

### 13.3.1 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`.

### 13.3.2 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})
```

## 13.4 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(),
# )
```

### 13.5 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

```

## 13.6 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model`

### 13.6.1 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])

```

### 13.6.2 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 12.6.

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 13.7.3.

```
modify_hyper_parameter_bounds(fun_control, "bootstrap", bounds=[0, 1])
modify_hyper_parameter_bounds(fun_control, "oob_score", bounds=[0, 0])
```

### 13.6.3 Optimizers

Optimizers are described in Section 12.6.1.

### 13.6.4 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.

## 13.7 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".

### 13.7.1 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.

#### 13.7.1.1 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}.
```

#### 13.7.1.2 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},
})
```

### 13.7.2 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",
})
```

### 13.7.3 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])

```

### 13.7.3.1 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,
# })

```

## 13.8 Step 8: Calling the SPOT Function

### 13.8.1 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

### 13.8.2 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
```

### 13.8.3 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 [-----] 1.76%

spotPython tuning: -0.8544973544973545 [-----] 3.80%

spotPython tuning: -0.8641975308641975 [#-----] 6.92%

spotPython tuning: -0.8641975308641975 [#-----] 9.12%

spotPython tuning: -0.8641975308641975 [#-----] 10.69%

spotPython tuning: -0.8641975308641975 [#-----] 12.42%

spotPython tuning: -0.8659611992945327 [#-----] 14.82%

spotPython tuning: -0.8686067019400352 [##-----] 17.34%

```
spotPython tuning: -0.8712522045855379 [##-----] 19.92%
spotPython tuning: -0.8712522045855379 [##-----] 22.59%
spotPython tuning: -0.8712522045855379 [###-----] 25.38%
spotPython tuning: -0.8712522045855379 [###-----] 27.49%
spotPython tuning: -0.8712522045855379 [###-----] 33.54%
spotPython tuning: -0.8712522045855379 [####-----] 40.15%
spotPython tuning: -0.879188712522046 [#####-----] 46.73%
spotPython tuning: -0.879188712522046 [#####-----] 52.89%
spotPython tuning: -0.879188712522046 [#####-----] 58.32%
spotPython tuning: -0.879188712522046 [#####-----] 63.55%
spotPython tuning: -0.879188712522046 [#####---] 69.01%
spotPython tuning: -0.879188712522046 [#####--] 76.03%
spotPython tuning: -0.879188712522046 [#####--] 82.55%
spotPython tuning: -0.879188712522046 [#####-] 88.23%
spotPython tuning: -0.879188712522046 [#####-] 94.82%
spotPython tuning: -0.879188712522046 [#####-] 100.00% Done...
<spotPython.spot.spot at 0x17fd5ace0>
```

## 13.9 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard as described in Section 12.9, see also the description in the documentation: [Tensorboard](#).

## 13.10 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")
```

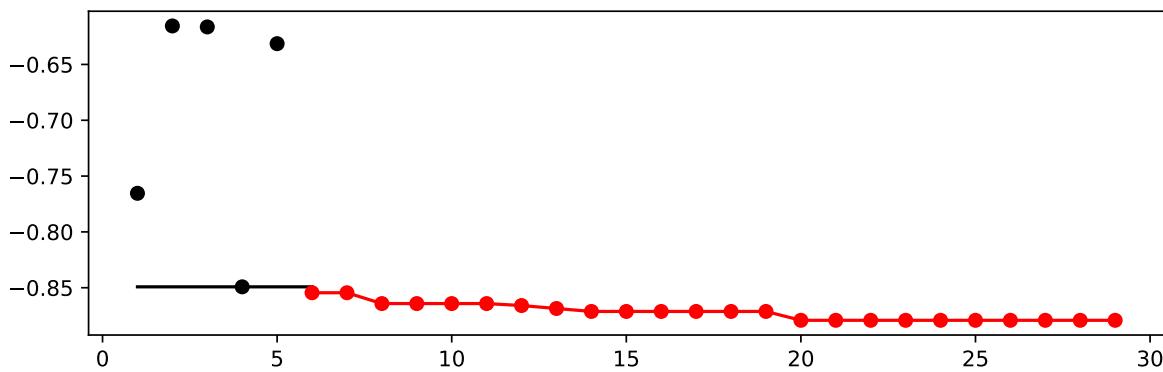


Figure 13.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

- 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	10.0
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	2.0
min_samples_leaf	int	1	1.0	25.0	1.0

min_weight_fraction_leaf	float	0.0	0.0	0.01	0.003002545876925399	
max_features	factor	sqrt	0.0	1.0	0.0	
max_leaf_nodes	int	10	7.0	12.0	10.0	
min_impurity_decrease	float	0.0	0.0	0.01	0.005762348549695934	
bootstrap	factor	1	1.0	1.0	1.0	
oob_score	factor	0	1.0	1.0	1.0	

### 13.10.1 Show variable importance

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



Figure 13.2: Variable importance plot, threshold 0.025.

### 13.10.2 Get Default Hyperparameters

```
from spotPython.hyperparameters.values import get_default_values, transform_hyper_parameter_values
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))])

```

### 13.10.3 Get SPOT Results

```

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

[[1.0000000e+01 1.0000000e+00 4.0000000e+00 2.0000000e+00
 1.0000000e+00 3.00254588e-03 0.0000000e+00 1.0000000e+01
 5.76234855e-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': 1024,
 'criterion': 'entropy',
 'max_depth': 16,
 'min_samples_split': 2,

```

```

'min_samples_leaf': 1,
'min_weight_fraction_leaf': 0.003002545876925399,
'max_features': 'sqrt',
'max_leaf_nodes': 1024,
'min_impurity_decrease': 0.005762348549695934,
'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=1024,
                      min_impurity_decrease=0.005762348549695934,
                      min_weight_fraction_leaf=0.003002545876925399,
                      n_estimators=1024, oob_score=1)

```

### 13.10.4 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

```

0.8465608465608465

```

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

```

### 13.10.5 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: 0.8524691358024691
std_res: 0.006887909906588555
min_res: 0.8386243386243385
max_res: 0.8703703703703703
median_res: 0.8544973544973544

```

### 13.10.6 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)
```

0.8597883597883599

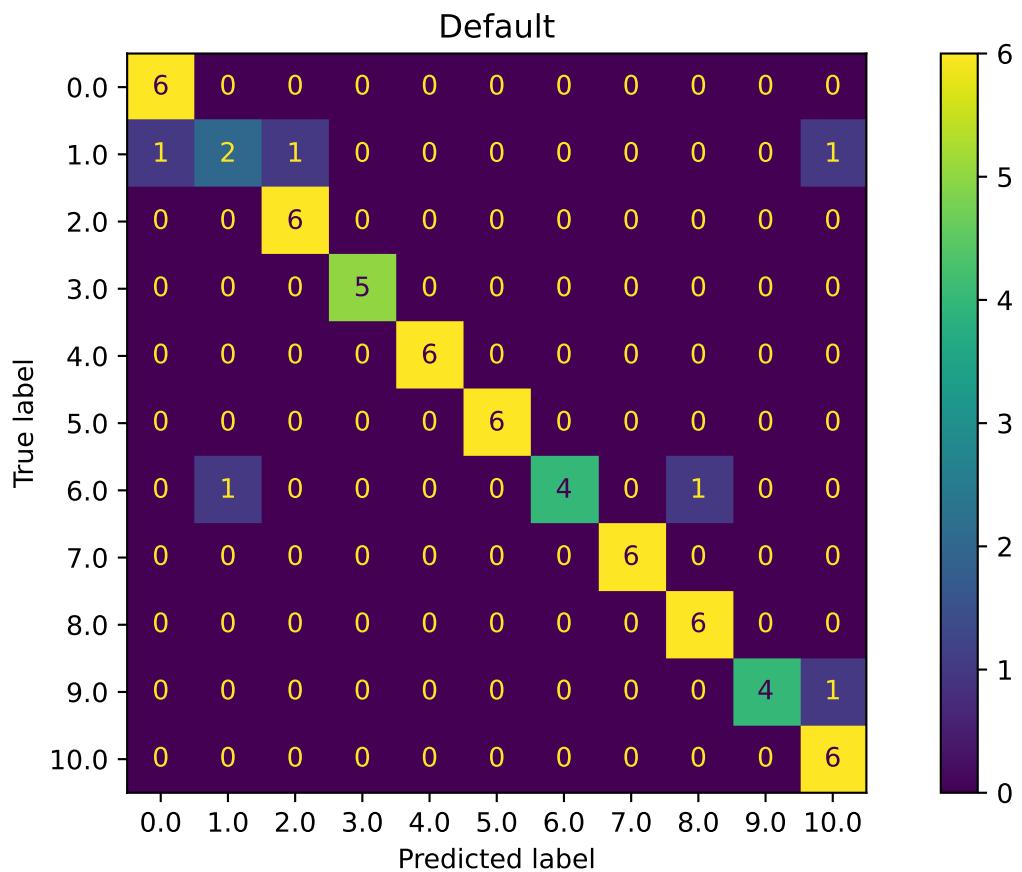
Since one single evaluation is not meaningful, we perform, similar to the evaluation of the SPOT results,  $n = 30$  runs of the default setting and calculate the mean and standard deviation of the performance metric.

```
_ = repeated_eval(30, model_default)

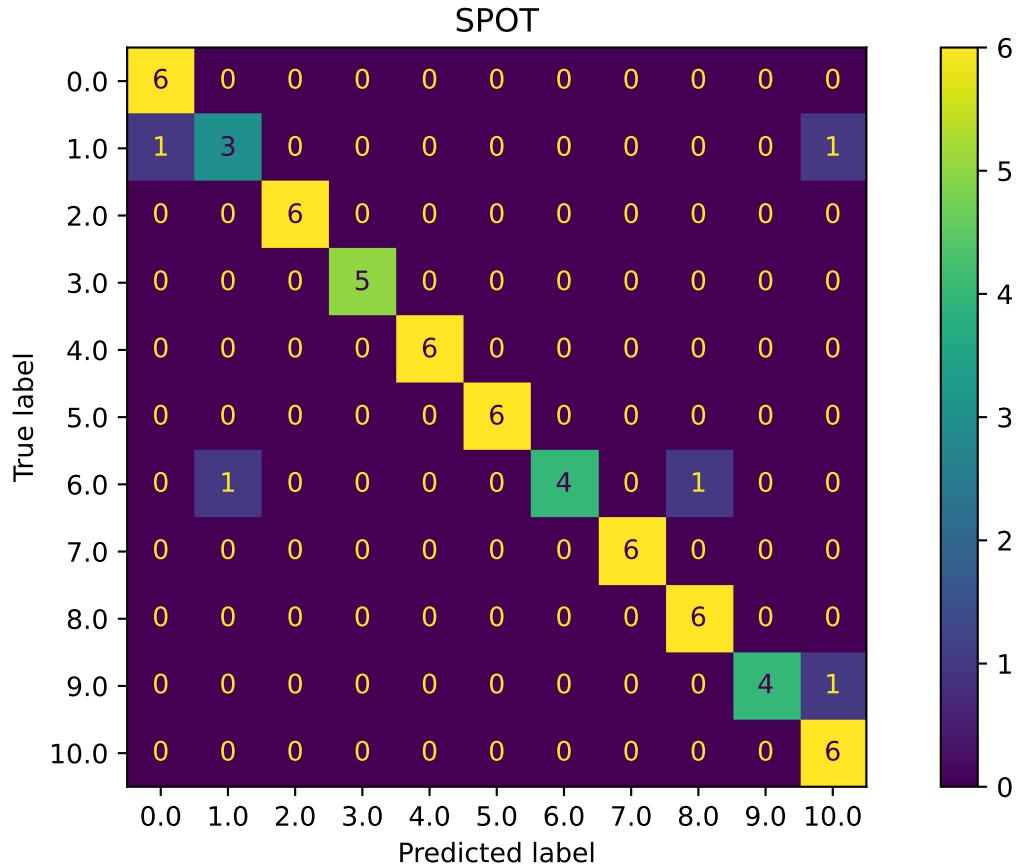
mean_res: 0.8509700176366846
std_res: 0.011538220270755644
min_res: 0.828042328042328
max_res: 0.8703703703703
median_res: 0.8505291005291006
```

### 13.10.7 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)
```

```
(-0.879188712522046, -0.6155202821869489)
```

### 13.10.8 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)
```

```
(0.8746588693957115, 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:

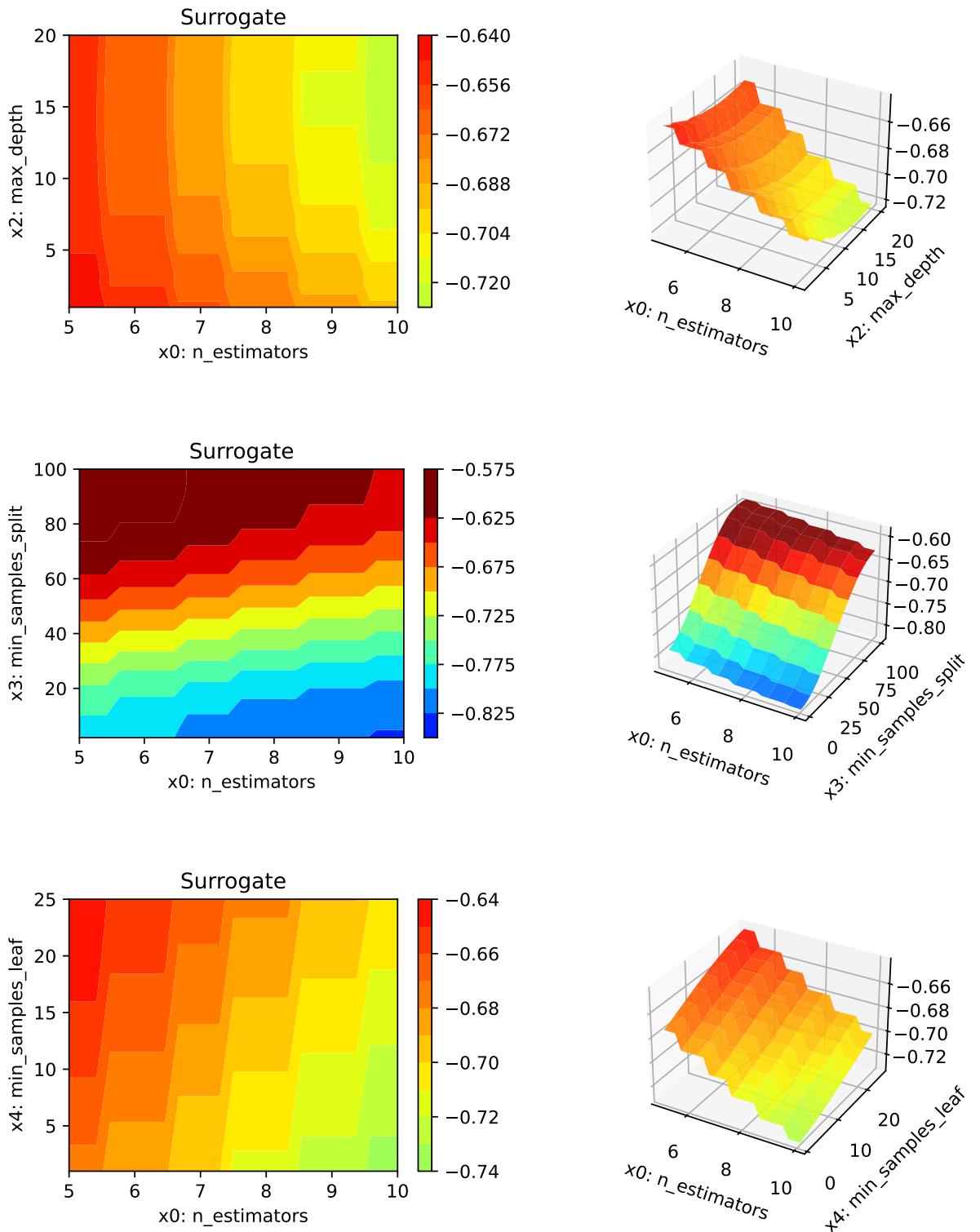
```
fun_control.update({
    "eval": "data_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

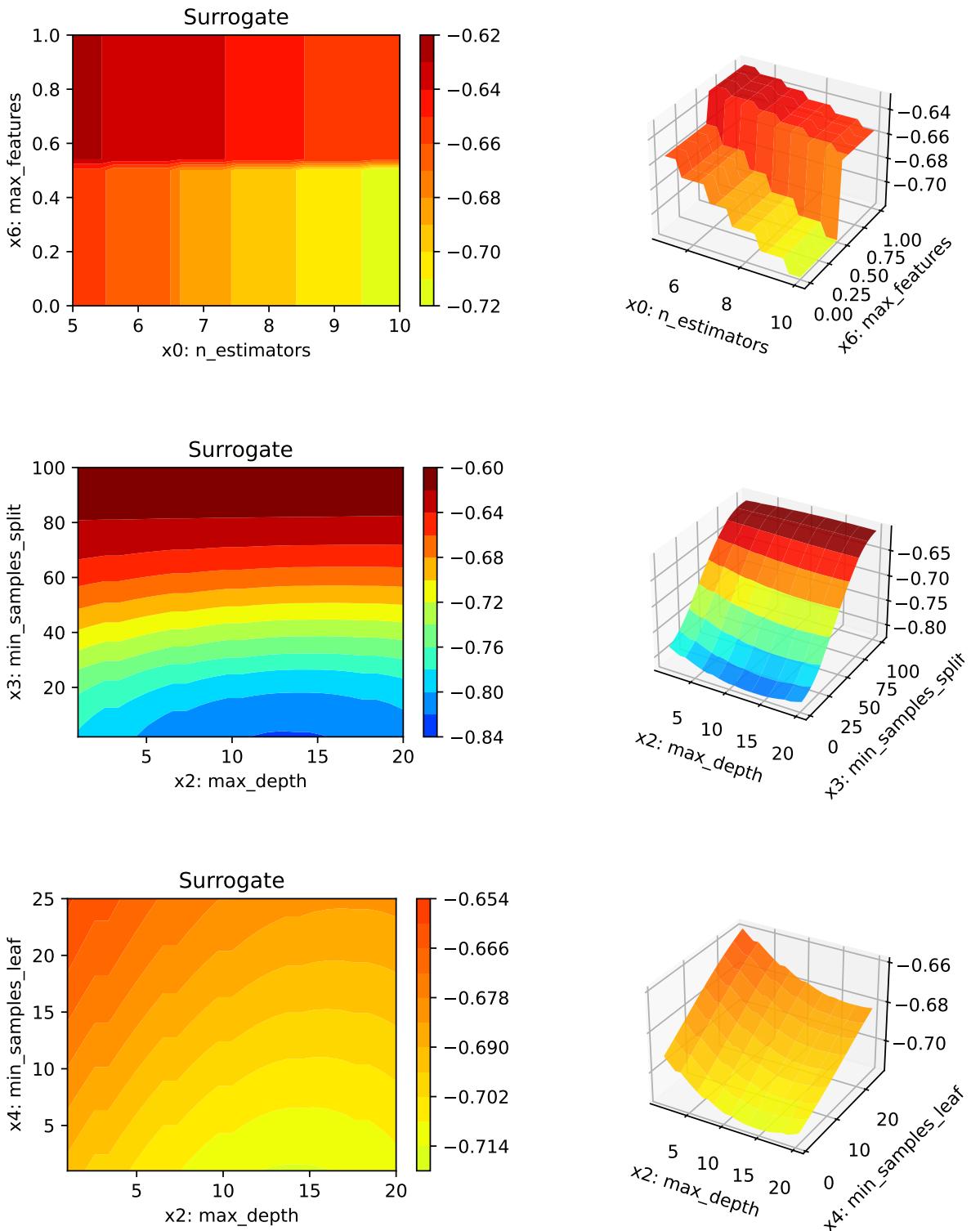
(0.879974358974359, None)

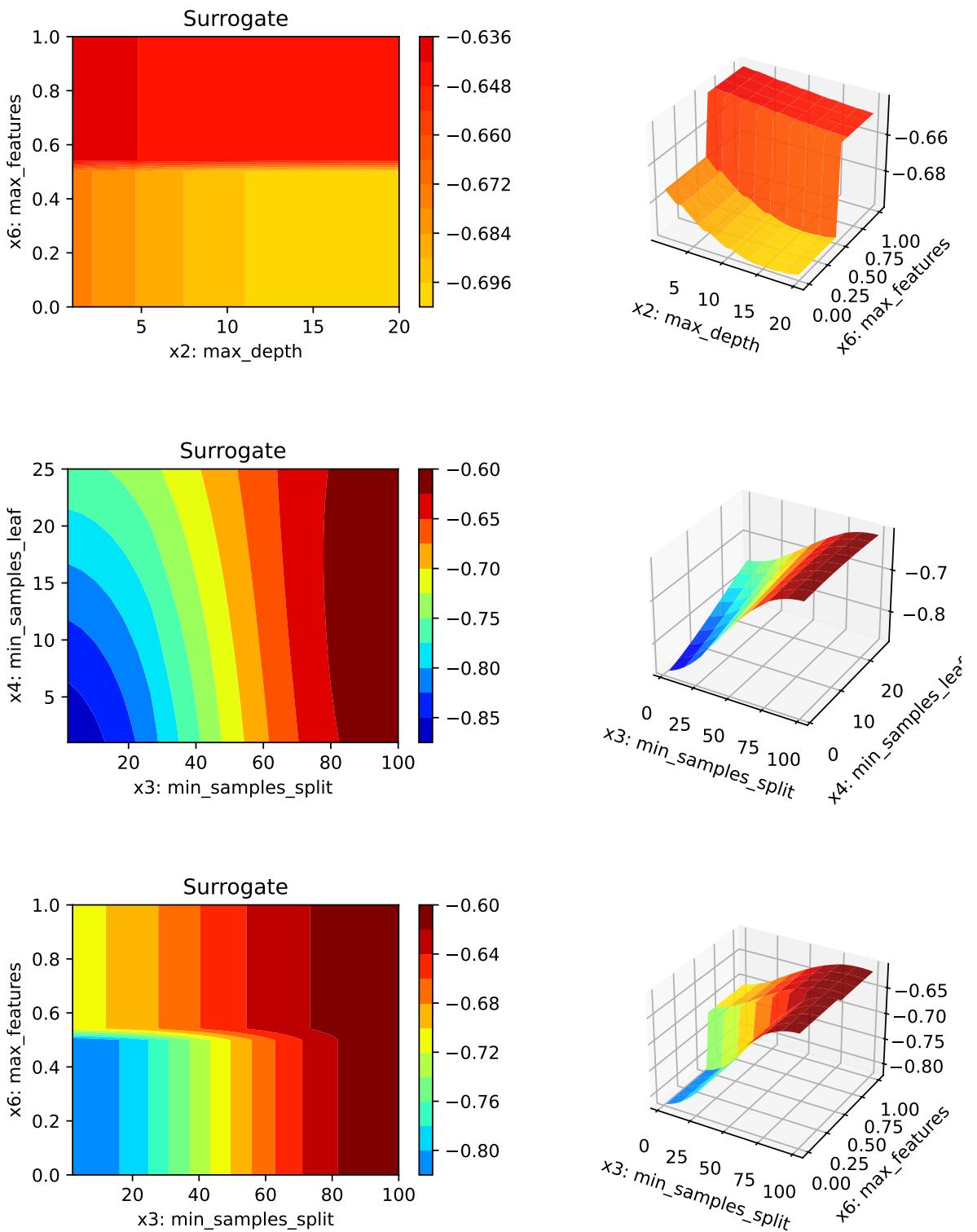
### 13.10.9 Detailed Hyperparameter Plots

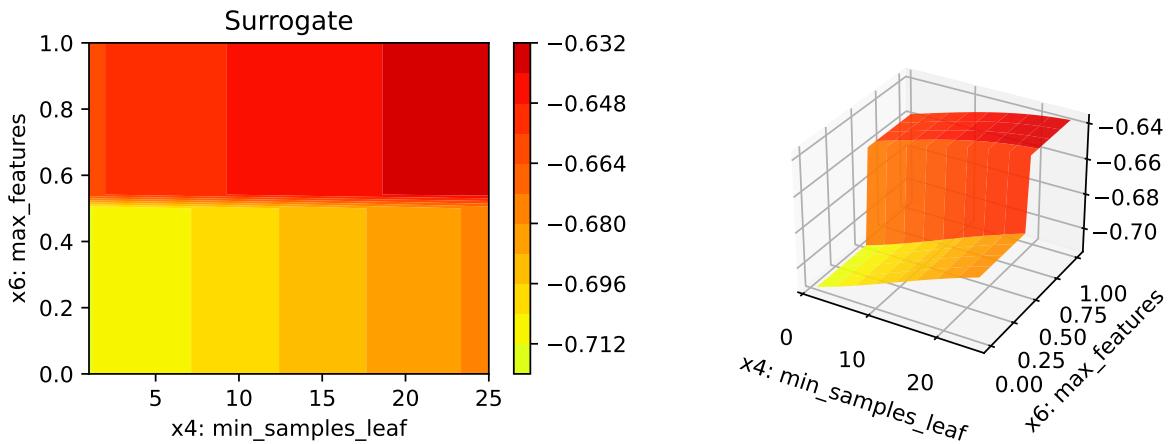
```
filename = "./figures/" + experiment_name
spot_tuner.plot_important_hyperparameter_contour(filename)

n_estimators: 0.13008809627543882
max_depth: 0.2405297237337157
min_samples_split: 1.346605405278484
min_samples_leaf: 0.19308393515083044
max_features: 100.0
```









### 13.10.10 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

### 13.10.11 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)
```

# 14 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.

## 14.1 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")
```

## 14.2 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))
```

## 14.3 Step 3: PyTorch Data Loading

### 14.3.1 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`.

#### 14.3.2 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})
```

## 14.4 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(),
# )
```

## 14.5 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
```

## 14.6 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model`

### 14.6.1 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])
```

### 14.6.2 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 12.6.

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"])
```

### 14.6.3 Optimizers

Optimizers are described in Section 12.6.1.

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

### 14.7.1 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).

### 14.7.2 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.

### 14.7.3 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".

### 14.7.4 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.

#### 14.7.4.1 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}.
```

#### 14.7.4.2 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},
})
```

### 14.7.5 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",
})
```

#### 14.7.5.1 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,
# })
```

## 14.8 Step 8: Calling the SPOT Function

### 14.8.1 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

### 14.8.2 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
```

### 14.8.3 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 [-----] 5.21%
spotPython tuning: -0.84375 [-----] 9.72%
spotPython tuning: -0.84375 [-----] 12.01%
spotPython tuning: -0.84375 [##-----] 15.54%
spotPython tuning: -0.84375 [###-----] 26.95%
spotPython tuning: -0.84375 [###-----] 32.56%
spotPython tuning: -0.84375 [####-----] 36.91%
spotPython tuning: -0.84375 [####-----] 39.93%
spotPython tuning: -0.84375 [#####-----] 45.14%
spotPython tuning: -0.84375 [#####-----] 47.63%
spotPython tuning: -0.8680555555555557 [#####-----] 57.62%
spotPython tuning: -0.8680555555555557 [#####-----] 63.53%
spotPython tuning: -0.8680555555555557 [#####----] 79.92%
spotPython tuning: -0.8680555555555557 [#####----] 91.80%
spotPython tuning: -0.8680555555555557 [#####----] 100.00% Done...
<spotPython.spot.spot.Spot at 0x2c01e7eb0>
```

## 14.9 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard as described in Section 12.9, see also the description in the documentation: [Tensorboard](#).

## 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")
```

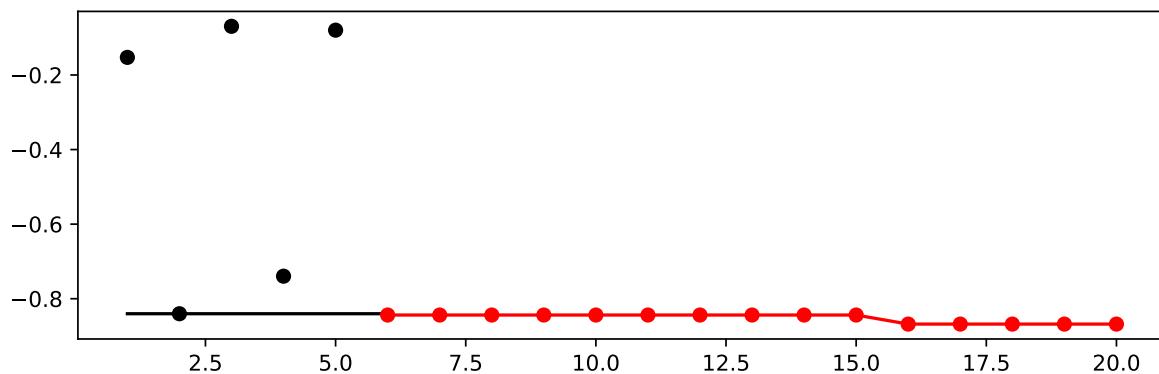


Figure 14.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

- Print the results

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

name	type	default	lower	upper	tuned	transformer
loss	factor	log_loss	0.0	0.0	0.0	None
learning_rate	float	-1.0	-5.0	0.0	-0.3668375393724054	transformer
max_iter	int	7	3.0	10.0	8.0	transformer
max_leaf_nodes	int	5	1.0	12.0	6.0	transformer
max_depth	int	2	1.0	20.0	17.0	transformer

min_samples_leaf   int   4	2.0   10.0	2.0   transform
l2_regularization   float   0.0	0.0   10.0	10.0   None
max_bins   int   255	127.0   255.0	140.0   None
early_stopping   factor   1	0.0   1.0	1.0   None
n_iter_no_change   int   10	5.0   20.0	8.0   None
tol   float   0.0001	1e-05   0.001	0.001   None

#### 14.10.1 Show variable importance

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

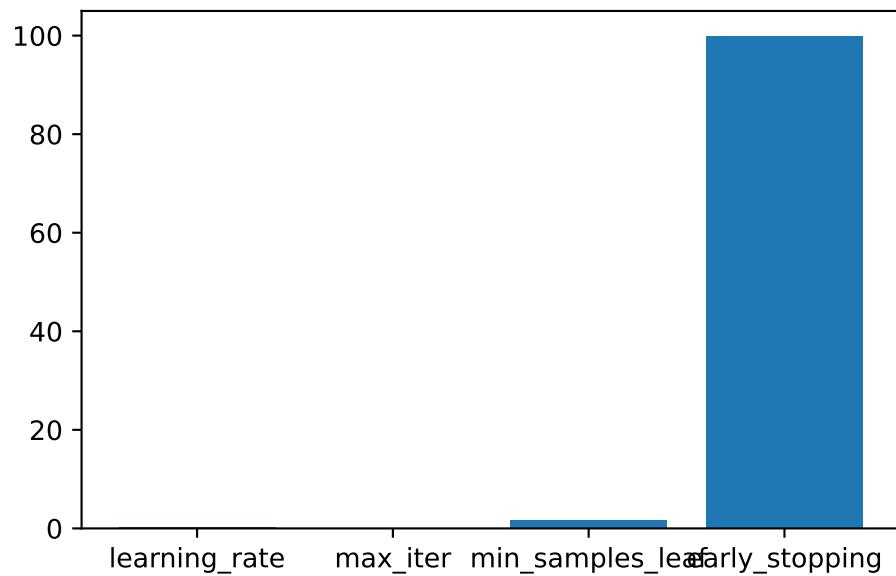


Figure 14.2: Variable importance plot, threshold 0.025.

#### 14.10.2 Get Default Hyperparameters

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

#### 14.10.3 Get SPOT Results

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

```
[[ 0.0000000e+00 -3.66837539e-01  8.0000000e+00  6.0000000e+00
  1.7000000e+01  2.0000000e+00  1.0000000e+01  1.4000000e+02
  1.0000000e+00  8.0000000e+00  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': 0.429697137559405,
 'max_iter': 256,
```

```

'max_leaf_nodes': 64,
'max_depth': 131072,
'min_samples_leaf': 4,
'l2_regularization': 10.0,
'max_bins': 140,
'early_stopping': 1,
'n_iter_no_change': 8,
'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, l2_regularization=10.0,
                               learning_rate=0.429697137559405, max_bins=140,
                               max_depth=131072, max_iter=256,
                               max_leaf_nodes=64, min_samples_leaf=4,
                               n_iter_no_change=8, tol=0.001)

```

#### 14.10.4 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

```

0.7910052910052912

```

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

```

#### 14.10.5 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: 0.7858906525573192
std_res: 0.01388041789767267
min_res: 0.7566137566137567
max_res: 0.8201058201058202
median_res: 0.7830687830687831

```

#### 14.10.6 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)
```

0.7592592592592592

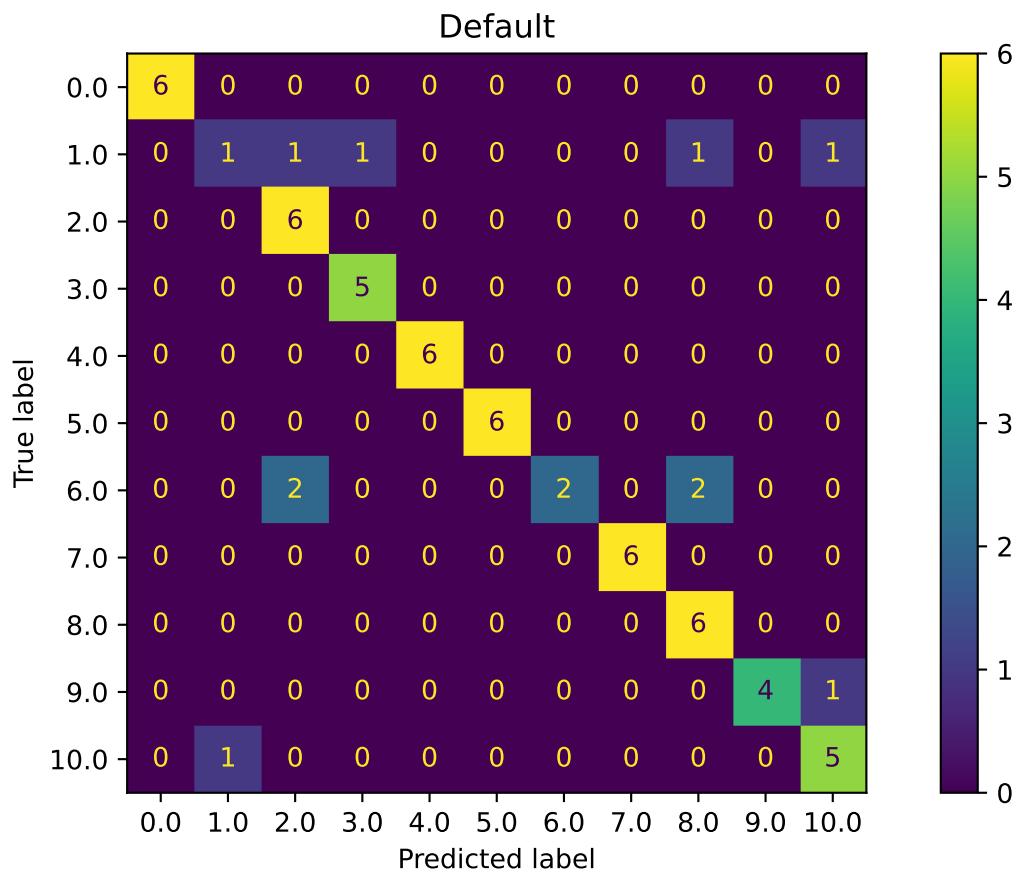
Since one single evaluation is not meaningful, we perform, similar to the evaluation of the SPOT results,  $n = 30$  runs of the default setting and calculate the mean and standard deviation of the performance metric.

```
_ = repeated_eval(30, model_default)
```

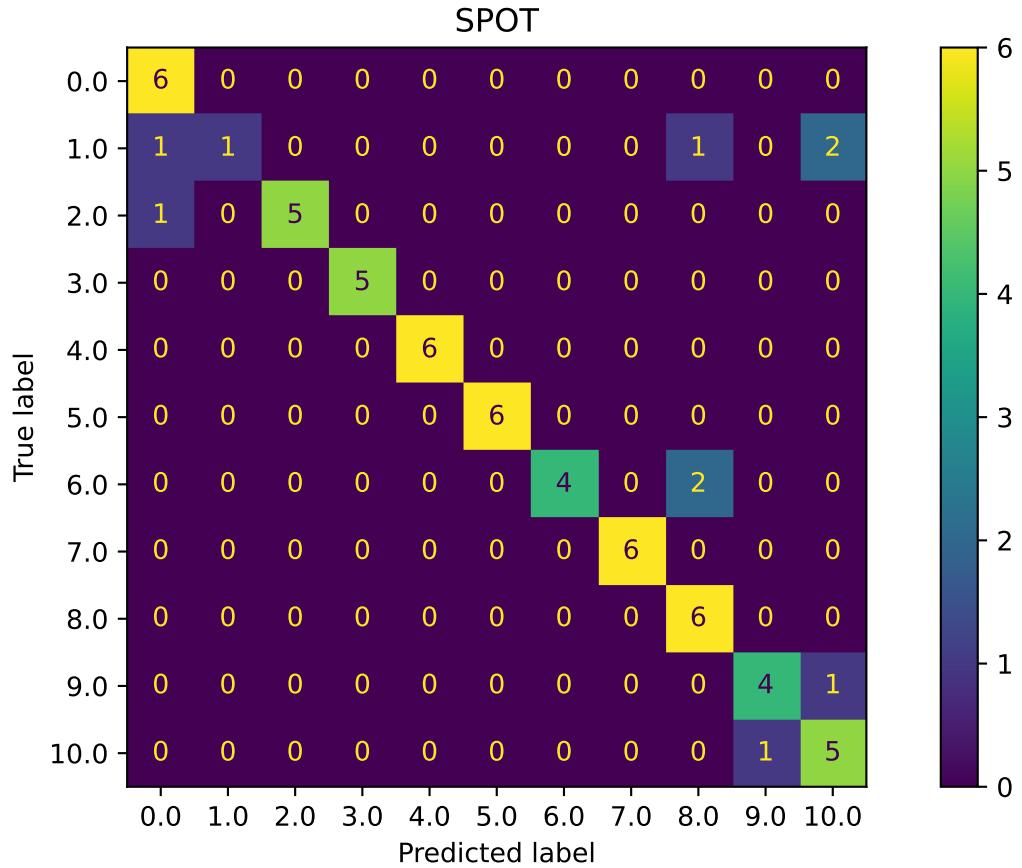
```
mean_res: 0.7952380952380952
std_res: 0.013824280735284227
min_res: 0.7671957671957672
max_res: 0.8253968253968255
median_res: 0.7962962962962963
```

#### 14.10.7 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)
```

```
(-0.8680555555555557, -0.0694444444444443)
```

#### 14.10.8 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)
```

```
(0.8021442495126706, 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 greater than n\_estimators=10')

(nan, None)

- This is the evaluation that will be used in the comparison:

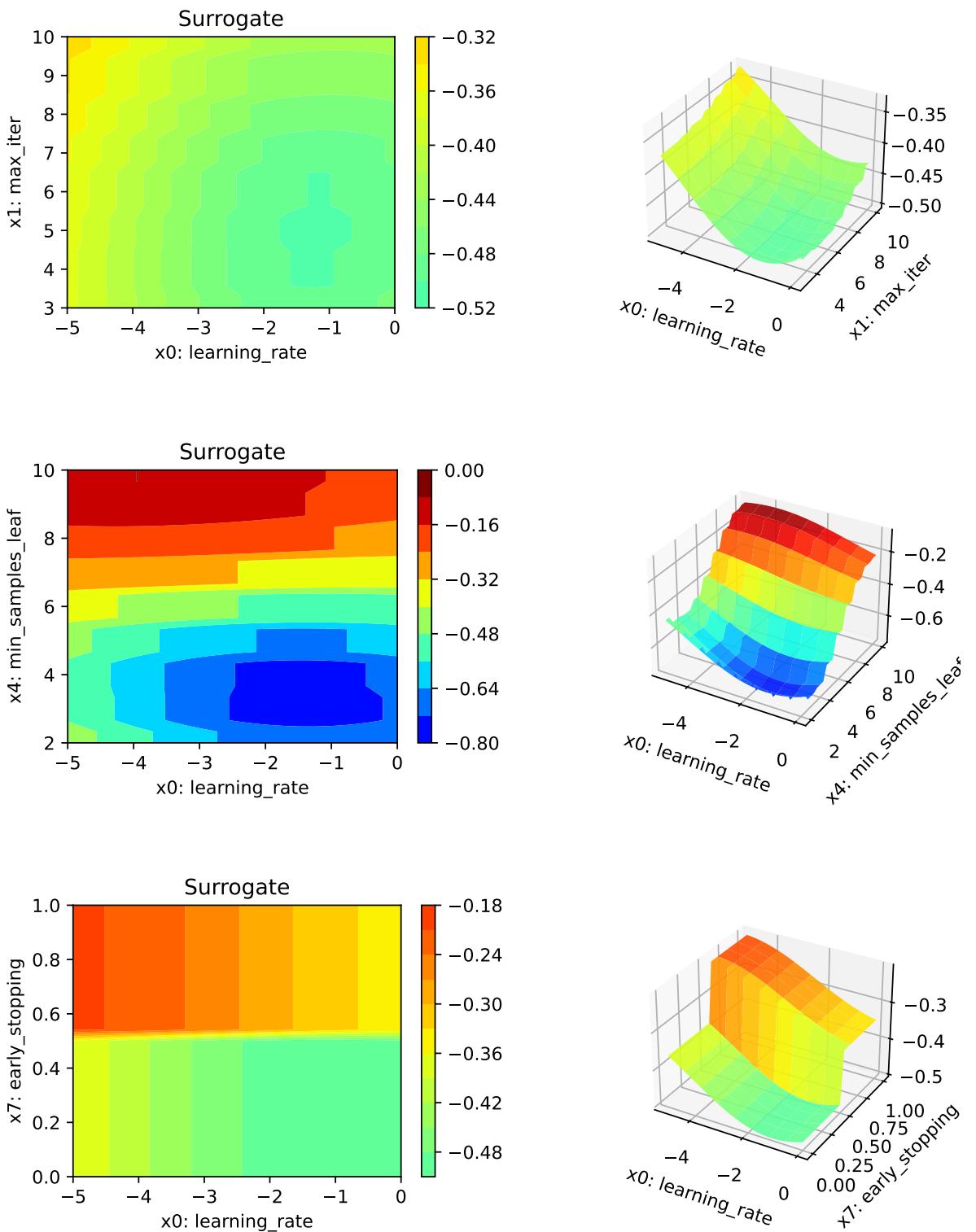
```
fun_control.update({
    "eval": "data_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

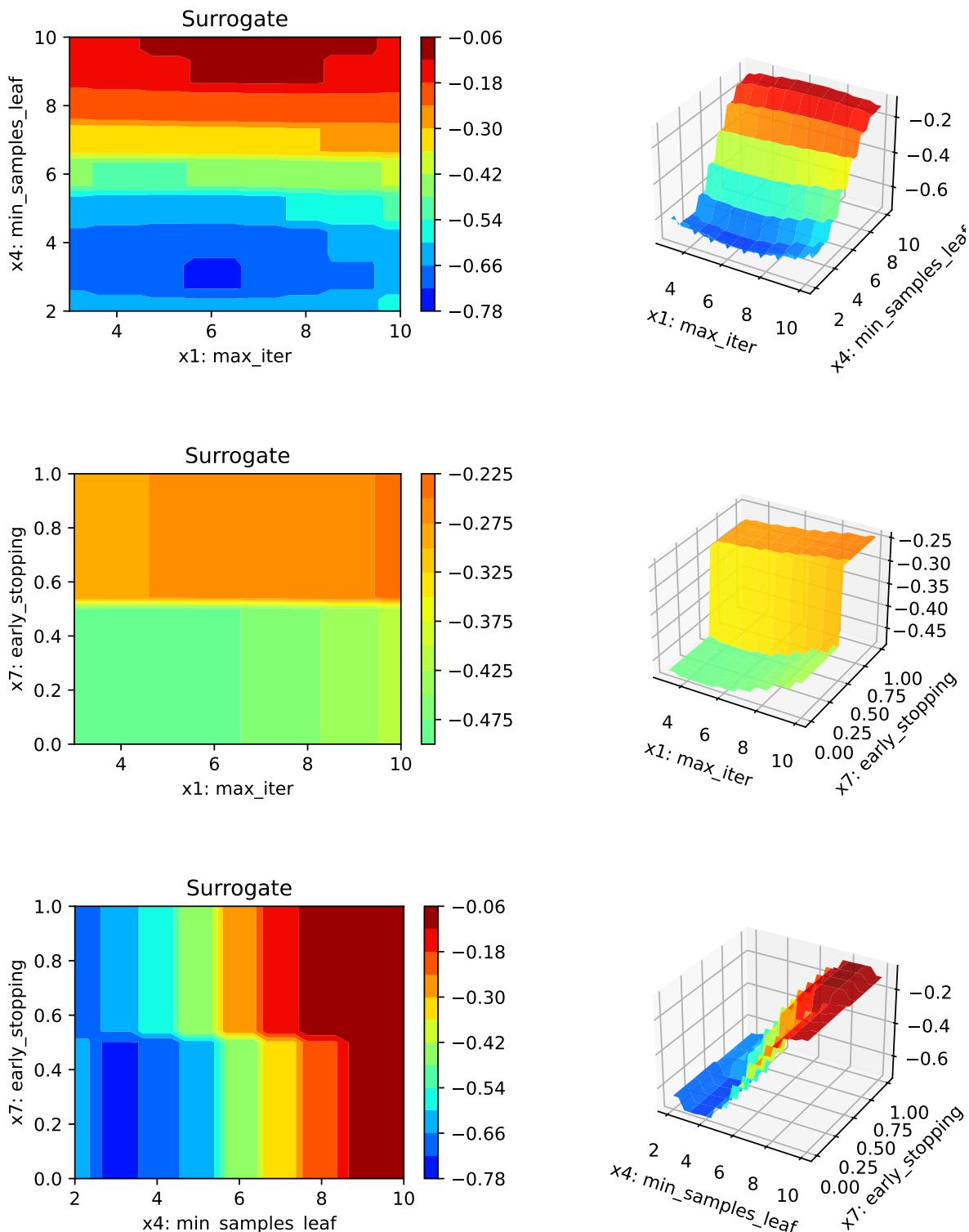
(0.8348974358974359, None)

#### 14.10.9 Detailed Hyperparameter Plots

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

learning_rate: 0.3567677848399139
max_iter: 0.13022107198445454
min_samples_leaf: 1.6076317023925468
early_stopping: 100.0
```





#### 14.10.10 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

#### 14.10.11 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)
```

# 15 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.

## 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 = "18"

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 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`.

### 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
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.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
```

## 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, "probability", bounds=[1, 1])
```

### 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 12.6.

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"])
```

### 15.6.3 Optimizers

Optimizers are described in Section 12.6.1.

### 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",
```

```
})
```

### 15.7.2.1 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,
# })
```

## 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
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	

### 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([[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 [-----] 0.69%
```

```
spotPython tuning: -0.875 [-----] 1.39%
```

```
spotPython tuning: -0.875 [-----] 1.95%
```

```
spotPython tuning: -0.875 [-----] 2.44%
```

```
spotPython tuning: -0.875 [-----] 2.92%
```

```
spotPython tuning: -0.875 [-----] 3.37%
```

```
spotPython tuning: -0.875 [-----] 4.03%
```

```
spotPython tuning: -0.875 [-----] 4.71%
```

```
spotPython tuning: -0.875 [#-----] 5.37%
```

```
spotPython tuning: -0.875 [#-----] 6.59%
```

```
spotPython tuning: -0.875 [#-----] 7.91%
spotPython tuning: -0.875 [#-----] 8.57%
spotPython tuning: -0.875 [#-----] 11.02%
spotPython tuning: -0.875 [#-----] 13.56%
spotPython tuning: -0.875 [##-----] 15.81%
spotPython tuning: -0.875 [##-----] 17.87%
spotPython tuning: -0.875 [##-----] 19.96%
spotPython tuning: -0.875 [##-----] 21.91%
spotPython tuning: -0.875 [##-----] 24.33%
spotPython tuning: -0.875 [###-----] 28.28%
spotPython tuning: -0.875 [###-----] 31.75%
spotPython tuning: -0.875 [####-----] 35.60%
spotPython tuning: -0.875 [####-----] 38.92%
spotPython tuning: -0.875 [####-----] 43.60%
spotPython tuning: -0.875 [#####-----] 47.09%
spotPython tuning: -0.8854166666666666 [#####-----] 50.64%
spotPython tuning: -0.8854166666666666 [#####-----] 54.78%
spotPython tuning: -0.8854166666666666 [#####-----] 60.03%
spotPython tuning: -0.8854166666666666 [#####---] 65.57%
```

```
spotPython tuning: -0.8854166666666666 [#####---] 70.76%
spotPython tuning: -0.8854166666666666 [#####---] 76.44%
spotPython tuning: -0.8854166666666666 [#####---] 82.52%
spotPython tuning: -0.8854166666666666 [#####---] 88.58%
spotPython tuning: -0.8854166666666666 [#####---] 95.42%
spotPython tuning: -0.8854166666666666 [#####---] 100.00% Done...
<spotPython.spot.spot.Spot at 0x17ff57c40>
```

## 15.9 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard as described in Section 12.9, 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")
```

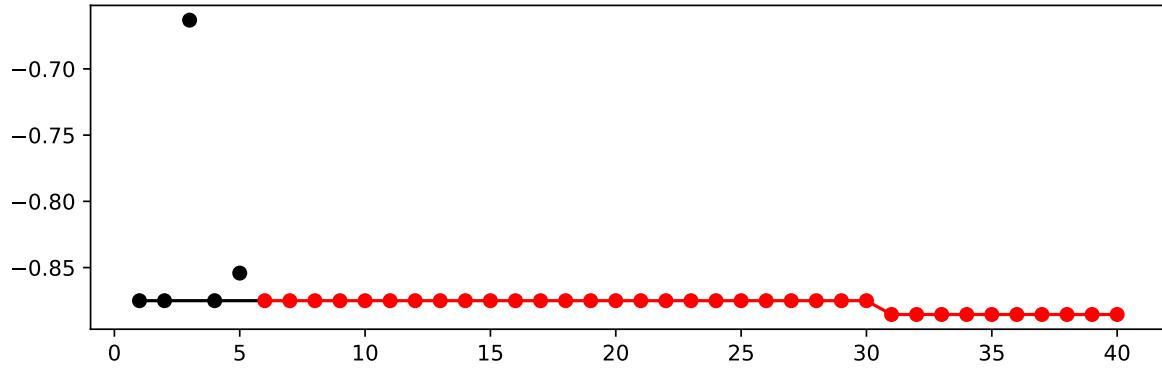


Figure 15.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

- Print the results

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

name	type	default	lower	upper	tuned	transform
C	float	1.0	0.1	10.0	4.809957939164208	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	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	1.0	1.0	1.0	None
tol	float	0.001	0.0001	0.01	0.003969298209225212	None
cache_size	float	200.0	100.0	400.0	145.9365804877652	None
break_ties	factor	0	0.0	1.0	0.0	None

### 15.10.1 Show variable importance

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

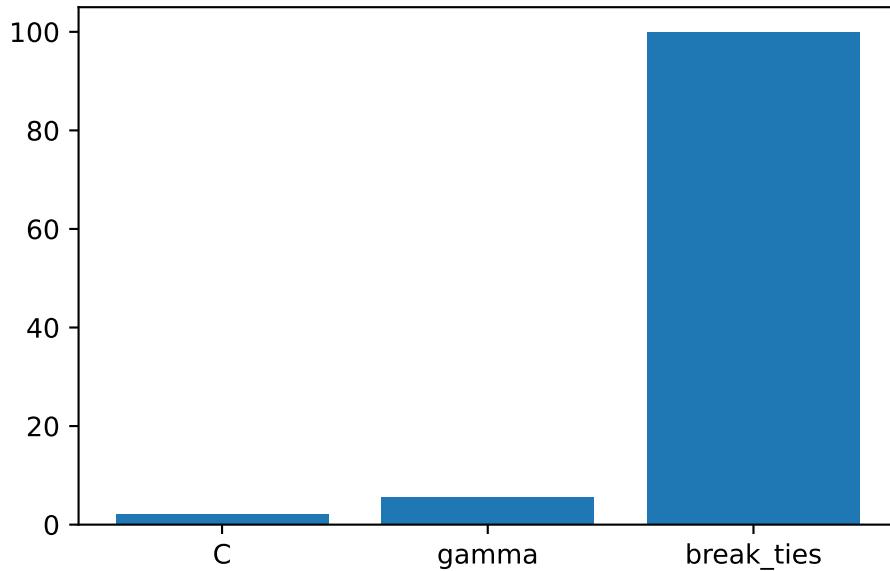


Figure 15.2: Variable importance plot, threshold 0.025.

### 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)

{'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))])
```

 Note

- Default value for “probability” is False, but we need it to be True for the metric “mapk\_score”.

```
values_default.update({"probability": 1})
```

### 15.10.3 Get SPOT Results

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

```
[[4.80995794e+00 0.0000000e+00 3.0000000e+00 0.0000000e+00
  0.0000000e+00 0.0000000e+00 1.0000000e+00 3.96929821e-03
  1.45936580e+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': 4.809957939164208,
 'kernel': 'rbf',
 'degree': 3,
 'gamma': 'scale',
 'coef0': 0.0,
 'shrinking': 0,
 'probability': 1,
 'tol': 0.003969298209225212,
 'cache_size': 145.9365804877652,
 '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=4.809957939164208, break_ties=0, cache_size=145.9365804877652,
probability=1, shrinking=0, tol=0.003969298209225212)
```

#### 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
```

```
0.8571428571428571
```

```
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: 0.862522045855379
std_res: 0.003580941887874279
min_res: 0.8571428571428571
max_res: 0.8650793650793651
median_res: 0.8650793650793651
```

### 15.10.6 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: 0.8545855379188712
std_res: 0.0041258157196788605
min_res: 0.8492063492063492
max_res: 0.8650793650793651
median_res: 0.8571428571428571

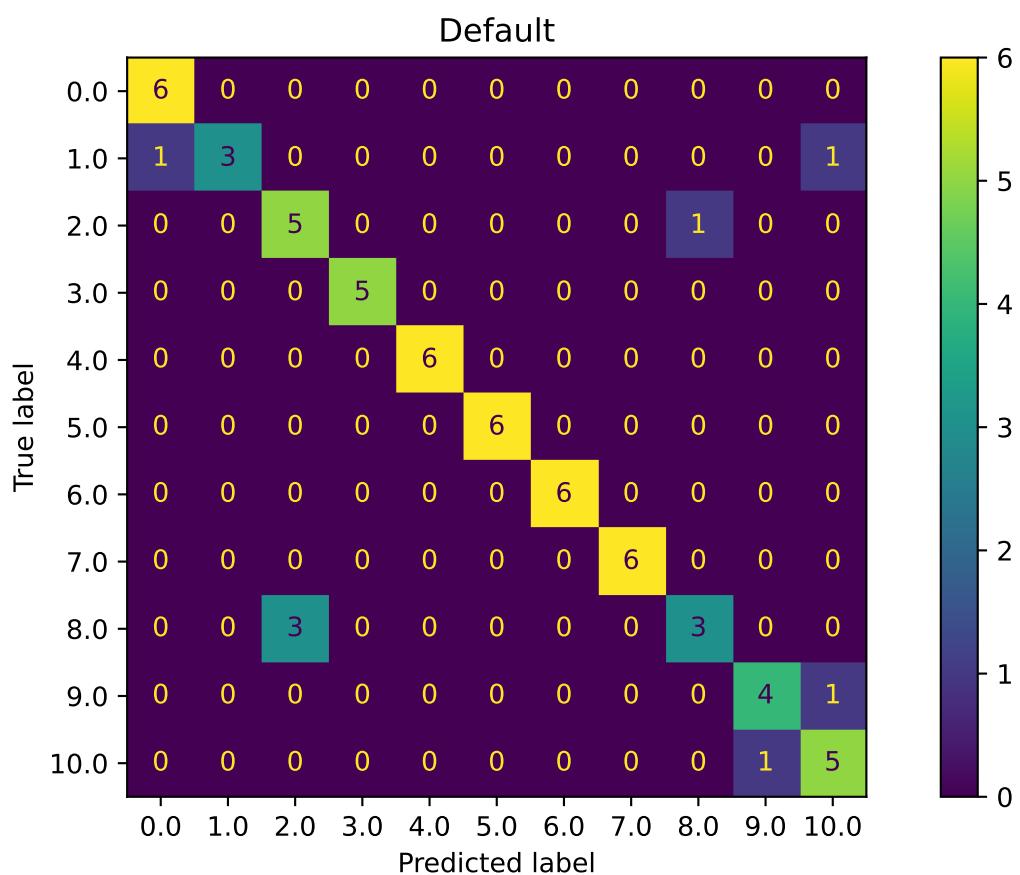
```

### 15.10.7 Plot: Compare Predictions

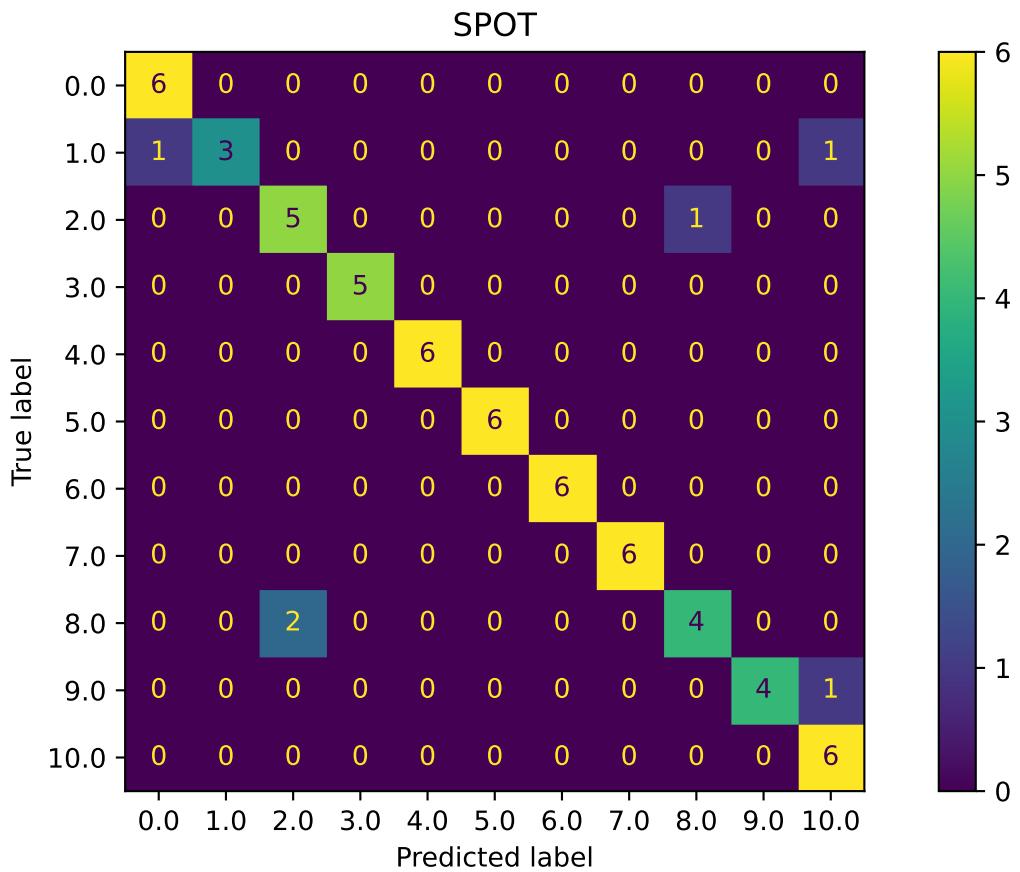
```

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)
```

```
(-0.8854166666666666, -0.04166666666666664)
```

### 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)
```

```
(0.8671539961013645, 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,
})
```

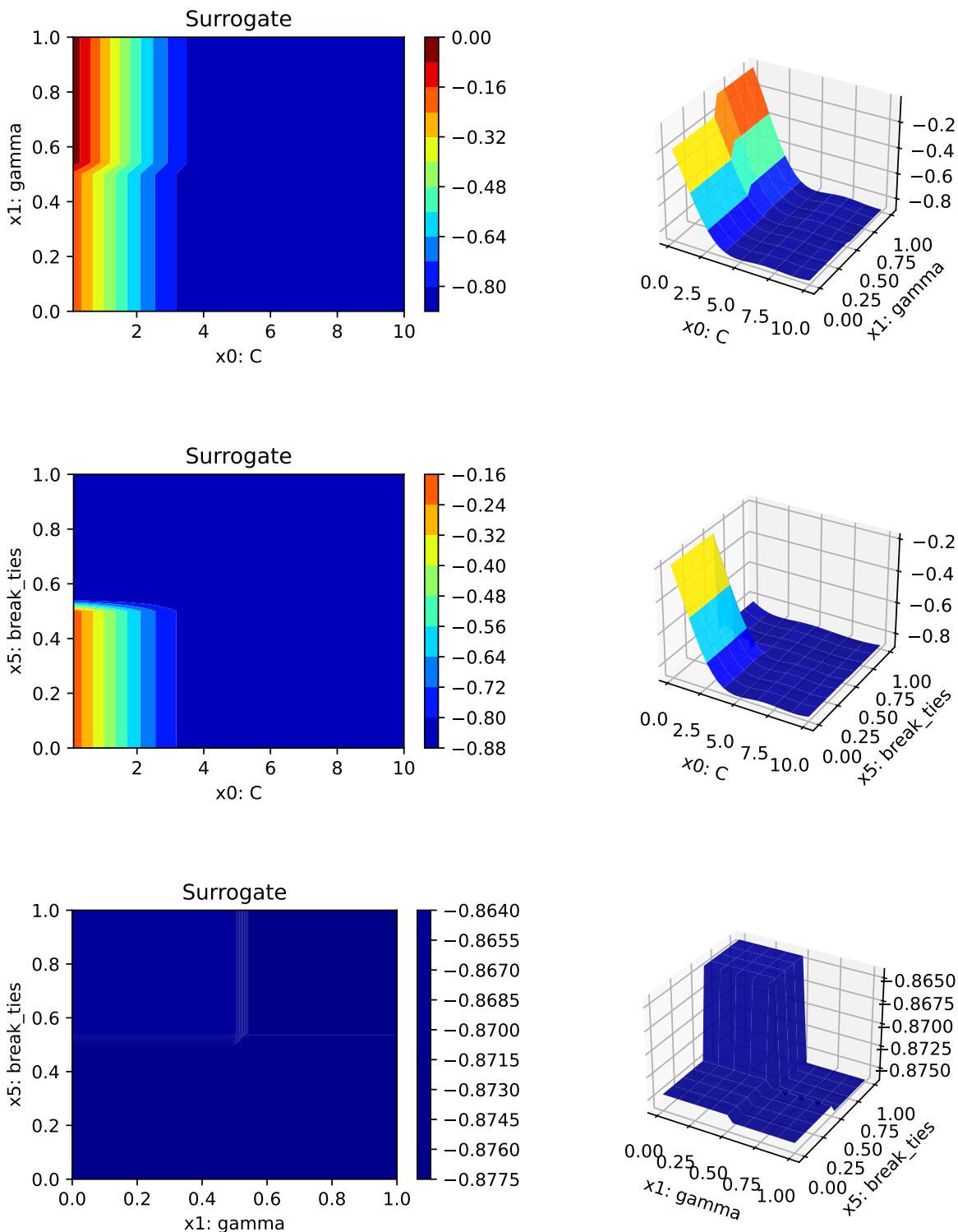
```
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

```
(0.882, None)
```

### 15.10.9 Detailed Hyperparameter Plots

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

```
C: 2.085932206795447
gamma: 5.522956421657414
break_ties: 100.000000000000001
```



### 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 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.

## 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 = "19"

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))
```

### 16.2.1 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	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`.

### 16.2.2 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})

```

## 16.3 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(),
# )

```

## 16.4 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
```

## 16.5 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model`

### 16.5.1 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])
```

### 16.5.2 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 12.6.

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"])
```

### 16.5.3 Optimizers

Optimizers are described in Section 12.6.1.

### 16.5.4 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.

## 16.6 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`".

### 16.6.1 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.

### 16.6.1.1 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}.
```

### 16.6.1.2 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},  
})
```

### 16.6.2 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",
```

```
})
```

### 16.6.2.1 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,
# })
```

## 16.7 Step 8: Calling the SPOT Function

### 16.7.1 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

### 16.7.2 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
```

### 16.7.3 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 [-----] 0.91%

spotPython tuning: -0.71875 [-----] 1.85%

spotPython tuning: -0.7326388888888888 [-----] 2.89%

spotPython tuning: -0.7326388888888888 [-----] 3.30%

spotPython tuning: -0.7326388888888888 [-----] 3.71%

spotPython tuning: -0.7326388888888888 [-----] 4.15%

spotPython tuning: -0.7326388888888888 [-----] 4.58%

spotPython tuning: -0.7326388888888888 [#-----] 5.04%

spotPython tuning: -0.7326388888888888 [#-----] 5.44%

spotPython tuning: -0.7326388888888888 [#-----] 5.91%

spotPython tuning: -0.7326388888888888 [#-----] 6.33%

spotPython tuning: -0.7326388888888888 [#-----] 7.87%

spotPython tuning: -0.7326388888888888 [#-----] 9.47%

spotPython tuning: -0.7326388888888888 [#-----] 11.02%

spotPython tuning: -0.7326388888888888 [#-----] 12.33%

spotPython tuning: -0.7326388888888888 [#-----] 13.90%

spotPython tuning: -0.7326388888888888 [##-----] 15.76%

spotPython tuning: -0.7326388888888888 [##-----] 17.19%

spotPython tuning: -0.7326388888888888 [##-----] 19.91%

spotPython tuning: -0.7465277777777777 [##-----] 21.44%

spotPython tuning: -0.7465277777777777 [##-----] 23.02%

spotPython tuning: -0.7465277777777777 [##-----] 24.40%

spotPython tuning: -0.7465277777777777 [###-----] 25.99%

spotPython tuning: -0.7465277777777777 [###-----] 27.57%

spotPython tuning: -0.7465277777777777 [###-----] 29.11%

spotPython tuning: -0.7465277777777777 [###-----] 30.49%

spotPython tuning: -0.7465277777777777 [###-----] 32.31%

spotPython tuning: -0.7465277777777777 [###-----] 34.69%

spotPython tuning: -0.7465277777777777 [####-----] 36.58%

spotPython tuning: -0.7465277777777777 [####-----] 38.43%

spotPython tuning: -0.7465277777777777 [####-----] 40.64%

spotPython tuning: -0.7465277777777777 [####-----] 43.54%

```
spotPython tuning: -0.7465277777777777 [#####----] 47.35%
spotPython tuning: -0.7465277777777777 [#####----] 50.21%
spotPython tuning: -0.7465277777777777 [#####----] 54.47%
spotPython tuning: -0.7465277777777777 [#####----] 57.77%
spotPython tuning: -0.7465277777777777 [#####----] 61.95%
spotPython tuning: -0.7465277777777777 [#####---] 65.40%
spotPython tuning: -0.7465277777777777 [#####---] 68.64%
spotPython tuning: -0.7465277777777777 [#####---] 71.76%
spotPython tuning: -0.7465277777777777 [#####---] 74.88%
spotPython tuning: -0.7465277777777777 [#####---] 78.26%
spotPython tuning: -0.7465277777777777 [#####---] 82.36%
spotPython tuning: -0.7465277777777777 [#####---] 85.23%
spotPython tuning: -0.7465277777777777 [#####---] 88.90%
spotPython tuning: -0.7465277777777777 [#####---] 93.99%
spotPython tuning: -0.7465277777777777 [#####---] 100.00% Done...
<spotPython.spot.spot.Spot at 0x28868a230>
```

## 16.8 Step 9: Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard as described in Section 12.9, see also the description in the documentation: [Tensorboard](#).

## 16.9 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")
```

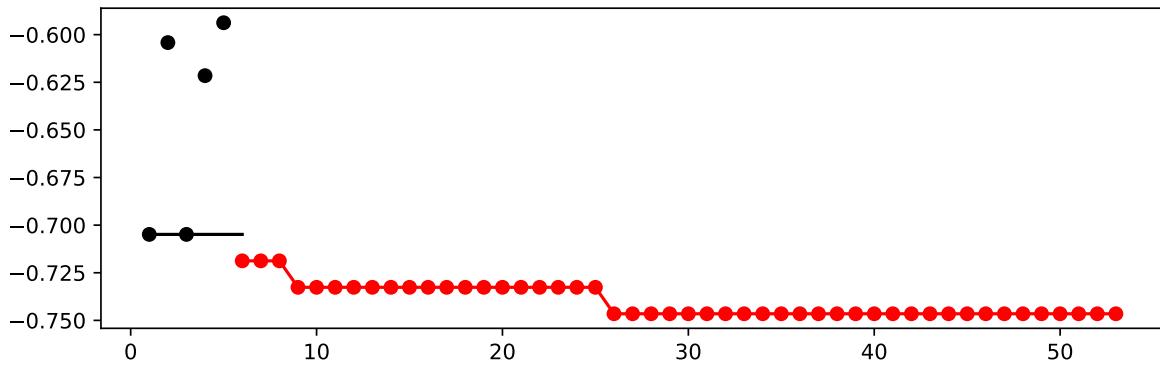


Figure 16.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

- Print the results

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

name	type	default	lower	upper	tuned	transform
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

### 16.9.1 Show variable importance

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

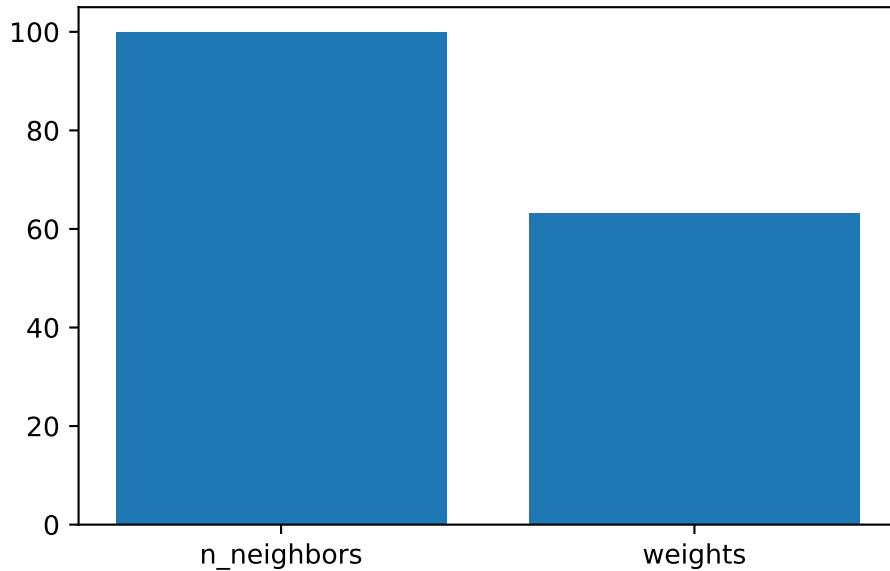


Figure 16.2: Variable importance plot, threshold 0.025.

### 16.9.2 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=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"](**values_default))
model_default

Pipeline(steps=[('nonetype', None),
                 ('kneighborsclassifier',
                  KNeighborsClassifier(leaf_size=32, n_neighbors=4))])

```

### 16.9.3 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)
```

### 16.9.4 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

```

### 16.9.5 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

```

### 16.9.6 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
```

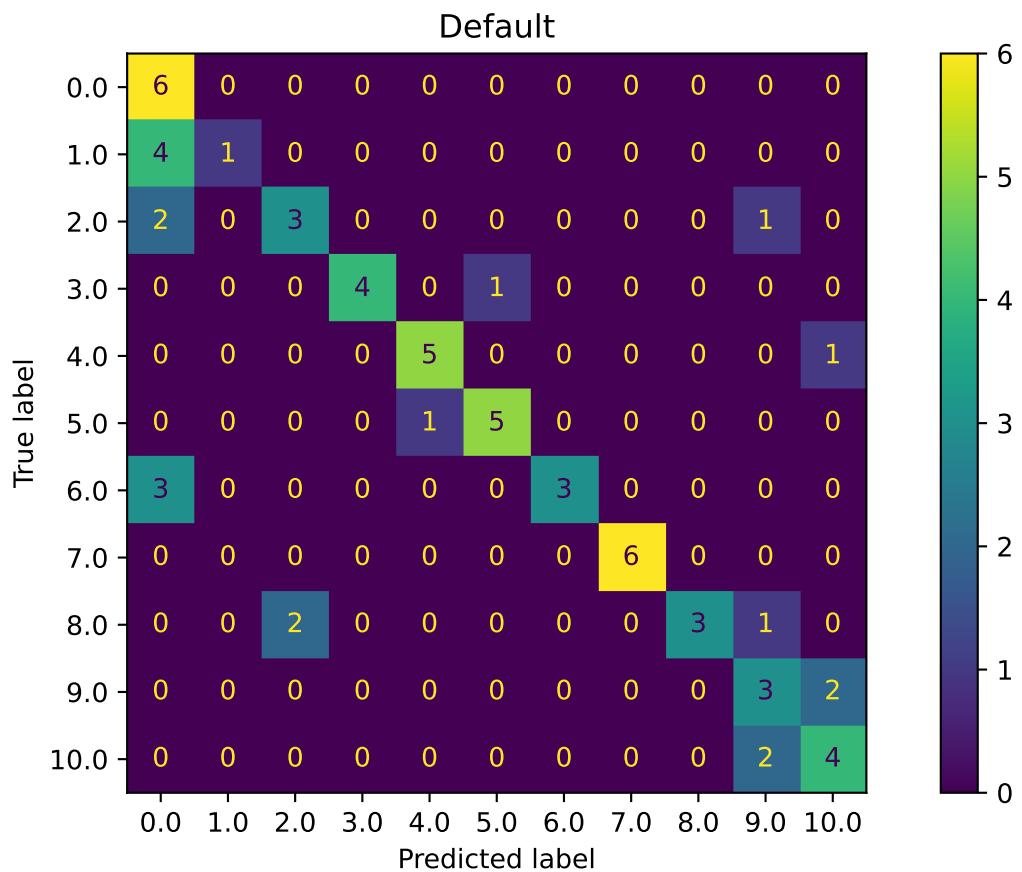
Since one single evaluation is not meaningful, we perform, similar to the evaluation of the SPOT results,  $n = 30$  runs of the default setting and calculate the mean and standard deviation of the performance metric.

```
_ = repeated_eval(30, model_default)
```

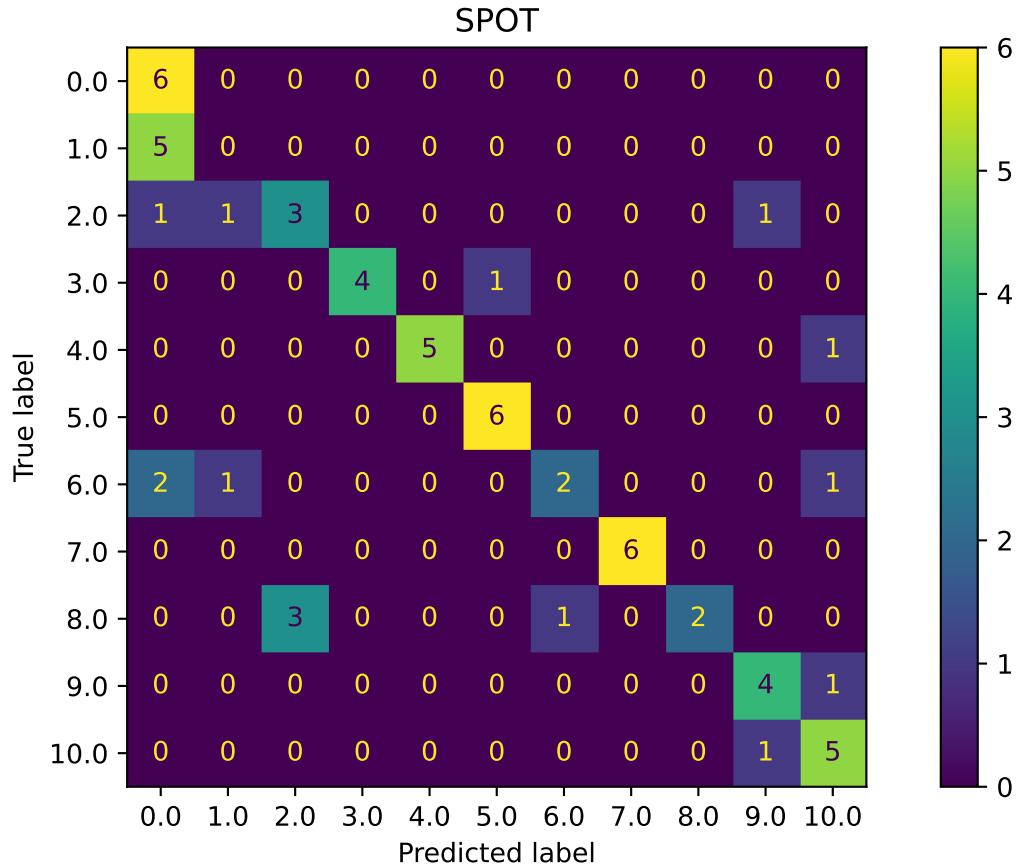
```
mean_res: 0.6878306878306877
std_res: 2.220446049250313e-16
min_res: 0.6878306878306879
max_res: 0.6878306878306879
median_res: 0.6878306878306879
```

### 16.9.7 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, -0.1666666666666666)
```

### 16.9.8 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 greater than n\_samples=10')

(nan, None)

- This is the evaluation that will be used in the comparison:

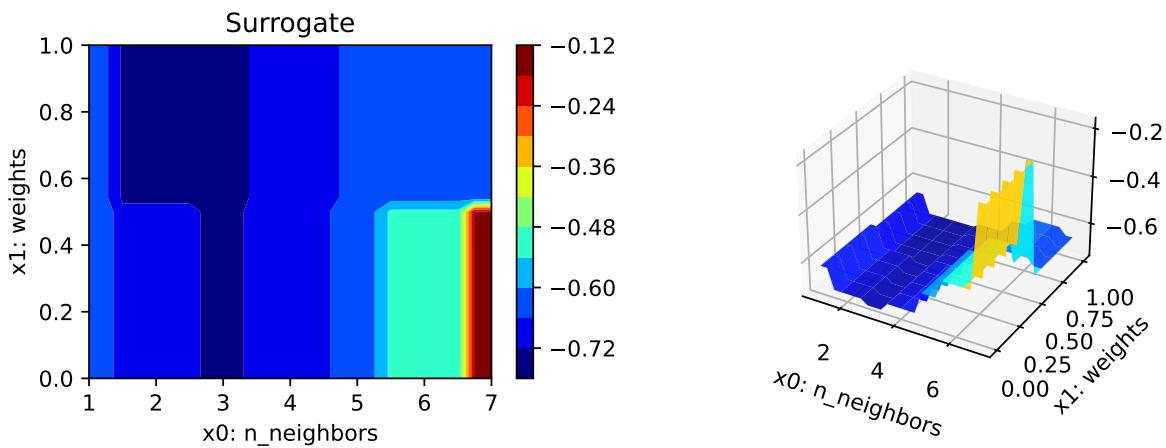
```
fun_control.update({
    "eval": "data_cv",
    "k_folds": 10,
})
evaluate_cv(model=model_spot, fun_control=fun_control, verbose=0)
```

(0.7089487179487179, None)

### 16.9.9 Detailed Hyperparameter Plots

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

n_neighbors: 100.0
weights: 63.20992884888382
```



### 16.9.10 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

### 16.9.11 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)
```

# 17 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
```

## 17.1 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.

## 17.2 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 [12.2](#), 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'
```

## 17.3 Step 3: PyTorch Data Loading

### 17.3.1 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 [17.12.2](#) illustrates how to access the data.

## 17.4 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.

## 17.5 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].

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

spotPython provides functions for modifying the hyperparameters, their bounds and factors as well as for activating and de-activating hyperparameters without re-compilation of the Python source code. These functions were described in Section 12.6.



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"]`.

## 17.7 Step 7: Data Splitting, the Objective (Loss) Function and the Metric

### 17.7.1 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 [12.7.1](#))
2. the loss function (and a metric).

 Caution: Data Splitting in Lightning

- The data splitting is handled by `Lightning`.

### 17.7.2 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.

### 17.7.3 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 17.12.3}](#).

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`.

## 17.8 Step 8: Calling the SPOT Function

### 17.8.1 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")

```

### 17.8.2 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

```

### 17.8.3 Starting the Hyperparameter Tuning

The `spotPython` hyperparameter tuning is started by calling the `Spot` function [SOURCE] as described in Section 12.8.4.

```

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()
```

```
config: {'l1': 256, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'AdamW'
```

Validate metric	DataLoader 0
hp_metric	2.263709545135498
val_acc	0.268551230430603
val_loss	2.263709545135498
valid_mapk	0.3537808656692505

```
config: {'l1': 32, 'epochs': 128, 'batch_size': 256, 'act_fn': LeakyReLU(), 'optimizer': 'Ad
```

Validate metric	DataLoader 0
hp_metric	2.2617576122283936
val_acc	0.2720848023891449
val_loss	2.2617576122283936
valid_mapk	0.3213372826576233

```
config: {'l1': 128, 'epochs': 256, 'batch_size': 8, 'act_fn': Swish(), 'optimizer': 'NAdam',
```

Validate metric	DataLoader 0
hp_metric	2.451167345046997
val_acc	0.09187278896570206
val_loss	2.451167345046997
valid_mapk	0.16377314925193787

```
config: {'l1': 64, 'epochs': 512, 'batch_size': 16, 'act_fn': Sigmoid(), 'optimizer': 'Adam'}
```

Validate metric	DataLoader 0
hp_metric	2.3177614212036133
val_acc	0.22968198359012604
val_loss	2.3177614212036133
valid_mapk	0.2924031913280487

```
config: {'l1': 64, 'epochs': 4096, 'batch_size': 64, 'act_fn': ReLU(), 'optimizer': 'Adamax'}
```

Validate metric	DataLoader 0
hp_metric	2.25834321975708
val_acc	0.2614840865135193
val_loss	2.25834321975708
valid_mapk	0.36971449851989746

```
config: {'l1': 32, 'epochs': 4096, 'batch_size': 128, 'act_fn': ReLU(), 'optimizer': 'AdamW'}
```

Validate metric	DataLoader 0
hp_metric	2.2992091178894043
val_acc	0.23674911260604858
val_loss	2.2992091178894043
valid_mapk	0.3636349141597748

```
spotPython tuning: 2.25834321975708 [-----] 4.20%
```

```
config: {'l1': 32, 'epochs': 4096, 'batch_size': 64, 'act_fn': ReLU(), 'optimizer': 'Adam',
```

Validate metric	DataLoader 0
hp_metric	2.2508015632629395
val_acc	0.27915194630622864
val_loss	2.2508015632629395
valid_mapk	0.35104164481163025

spotPython tuning: 2.2508015632629395 [####-----] 30.73%

config: {'l1': 32, 'epochs': 512, 'batch\_size': 128, 'act\_fn': ReLU(), 'optimizer': 'AdamW', 'lr': 0.001}

Validate metric	DataLoader 0
hp_metric	2.285112142562866
val_acc	0.2226148396730423
val_loss	2.285112142562866
valid_mapk	0.3453253507614136

spotPython tuning: 2.2508015632629395 [#####----] 42.49%

config: {'l1': 64, 'epochs': 256, 'batch\_size': 64, 'act\_fn': ReLU(), 'optimizer': 'Adam', 'lr': 0.001}

Validate metric	DataLoader 0
hp_metric	2.387507200241089
val_acc	0.06713780760765076
val_loss	2.387507200241089
valid_mapk	0.16718751192092896

spotPython tuning: 2.2508015632629395 [#####----] 53.64%

config: {'l1': 32, 'epochs': 8192, 'batch\_size': 128, 'act\_fn': ReLU(), 'optimizer': 'NAdam', 'lr': 0.001}

Validate metric	DataLoader 0
hp_metric	2.2326242923736572
val_acc	0.3074204921722412
val_loss	2.2326242923736572
valid_mapk	0.39719972014427185

spotPython tuning: 2.2326242923736572 [#####--] 76.50%

config: {'l1': 32, 'epochs': 8192, 'batch\_size': 256, 'act\_fn': ReLU(), 'optimizer': 'NAdam'}

Validate metric	DataLoader 0
hp_metric	2.3005590438842773
val_acc	0.23674911260604858
val_loss	2.3005590438842773
valid_mapk	0.3299093246459961

spotPython tuning: 2.2326242923736572 [#####--] 82.07%

config: {'l1': 64, 'epochs': 128, 'batch\_size': 256, 'act\_fn': LeakyReLU(), 'optimizer': 'Adad'}

Validate metric	DataLoader 0
hp_metric	2.2523937225341797
val_acc	0.2862190902233124
val_loss	2.2523937225341797
valid_mapk	0.3655478358268738

spotPython tuning: 2.2326242923736572 [#####--] 90.08%

config: {'l1': 128, 'epochs': 4096, 'batch\_size': 64, 'act\_fn': Tanh(), 'optimizer': 'AdamW'}

```
-----  
| Validate metric | DataLoader 0 | | |
|---|---|---|---|
| hp_metric | 2.2571563720703125 |  
| val_acc | 0.27915194630622864 |  
| val_loss | 2.2571563720703125 |  
| valid_mapk | 0.38902392983436584 |  
|||||
```

```
spotPython tuning: 2.2326242923736572 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x2854daf50>
```

## 17.9 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.

## 17.10 Step 10: Results

After the hyperparameter tuning run is finished, the results can be analyzed as described in Section [12.10](#).

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

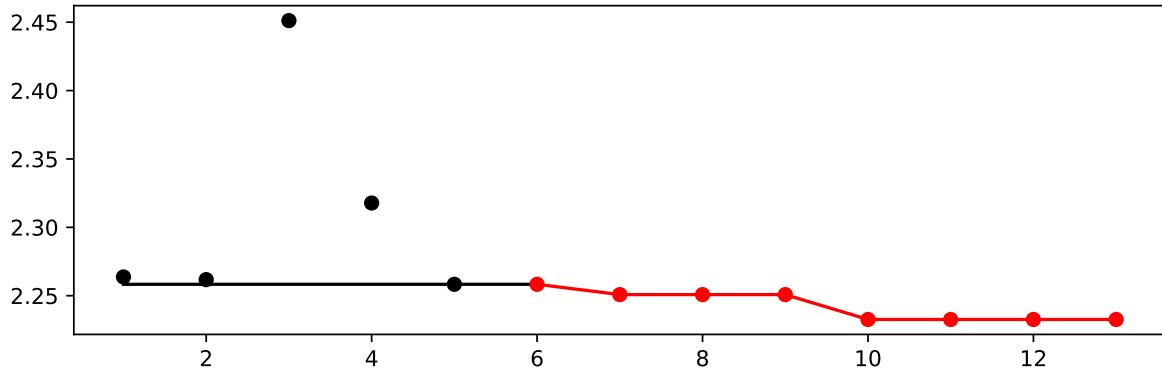


Figure 17.1: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

```
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_1
epochs	int	4	6.0	13.0	13.0	transform_1
batch_size	int	4	2.0	8.0	7.0	transform_1
act_fn	factor	ReLU	0.0	5.0	2.0	None
optimizer	factor	SGD	0.0	3.0	3.0	None
dropout_prob	float	0.01	0.0	0.25	0.07912235457426961	None
lr_mult	float	1.0	0.1	10.0	4.163993340890585	None
patience	int	2	2.0	6.0	6.0	transform_1
initialization	factor	Default	0.0	2.0	2.0	None

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

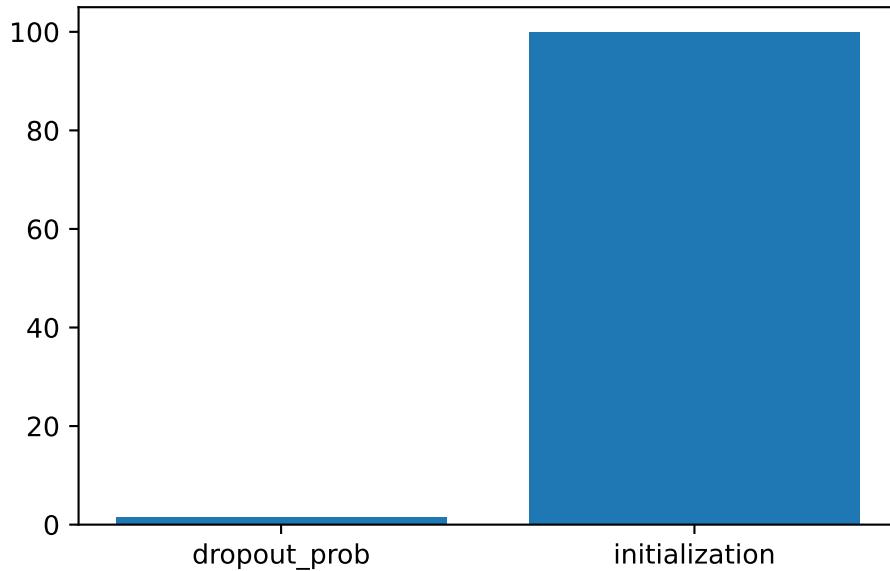


Figure 17.2: Variable importance plot, threshold 0.025.

### 17.10.1 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	2.0217232704162598
test_mapk_epoch	0.5595039129257202
val_acc	0.5190947651863098
val_loss	2.0217232704162598

(2.0217232704162598, 0.5190947651863098)

```

from spotPython.light.taintest import load_light_from_checkpoint
model_loaded = load_light_from_checkpoint(config, fun_control)

```

Loading model from runs/lightning\_logs/32\_8192\_128\_ReLU()\_NAdam\_0.07912235457426961\_4.163993

### 17.10.2 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.taintest 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	2.169114351272583
val_acc	0.3661971688270569
val_loss	2.169114351272583
valid_mapk	0.43896713852882385

train\_model result: {'valid\_mapk': 0.43896713852882385, 'val\_loss': 2.169114351272583, 'val\_a': 0.3661971688270569, 'val\_mapk': 0.43896713852882385}  
k: 1  
Train Dataset Size: 636  
Val Dataset Size: 71

Validate metric	DataLoader 0
hp_metric	2.2213735580444336
val_acc	0.3239436745643616
val_loss	2.2213735580444336
valid_mapk	0.4507042169570923

```
train_model result: {'valid_mapk': 0.4507042169570923, 'val_loss': 2.2213735580444336, 'val_a
k: 2
Train Dataset Size: 636
Val Dataset Size: 71
```

Validate metric	DataLoader 0
hp_metric	2.305535078048706
val_acc	0.23943662643432617
val_loss	2.305535078048706
valid_mapk	0.2957746386528015

```
train_model result: {'valid_mapk': 0.2957746386528015, 'val_loss': 2.305535078048706, 'val_a
k: 3
Train Dataset Size: 636
Val Dataset Size: 71
```

Validate metric	DataLoader 0
hp_metric	2.282437801361084
val_acc	0.23943662643432617
val_loss	2.282437801361084
valid_mapk	0.34741783142089844

```
train_model result: {'valid_mapk': 0.34741783142089844, 'val_loss': 2.282437801361084, 'val_a
k: 4
Train Dataset Size: 636
Val Dataset Size: 71
```

Validate metric	DataLoader 0
hp_metric	2.3464431762695312
val_acc	0.18309858441352844
val_loss	2.3464431762695312
valid_mapk	0.28169015049934387

```
train_model result: {'valid_mapk': 0.28169015049934387, 'val_loss': 2.3464431762695312, 'val_acc': 0.18309858441352844}
Train Dataset Size: 636
Val Dataset Size: 71
```

Validate metric	DataLoader 0
hp_metric	2.2693288326263428
val_acc	0.26760563254356384
val_loss	2.2693288326263428
valid_mapk	0.3779342770576477

```
train_model result: {'valid_mapk': 0.3779342770576477, 'val_loss': 2.2693288326263428, 'val_acc': 0.26760563254356384}
Train Dataset Size: 636
Val Dataset Size: 71
```

Validate metric	DataLoader 0
hp_metric	2.3574538230895996
val_acc	0.18309858441352844
val_loss	2.3574538230895996
valid_mapk	0.26056337356567383

```
train_model result: {'valid_mapk': 0.26056337356567383, 'val_loss': 2.3574538230895996, 'val_acc': 0.18309858441352844}
Train Dataset Size: 637
Val Dataset Size: 70
```

Validate metric	DataLoader 0
hp_metric	2.2719171047210693
val_acc	0.2571428716182709
val_loss	2.2719171047210693
valid_mapk	0.3499999940395355

```
train_model result: {'valid_mapk': 0.3499999940395355, 'val_loss': 2.2719171047210693, 'val_a
k: 8
Train Dataset Size: 637
Val Dataset Size: 70
```

Validate metric	DataLoader 0
hp_metric	2.3715577125549316
val_acc	0.17142857611179352
val_loss	2.3715577125549316
valid_mapk	0.3047619163990021

```
train_model result: {'valid_mapk': 0.3047619163990021, 'val_loss': 2.3715577125549316, 'val_a
k: 9
Train Dataset Size: 637
Val Dataset Size: 70
```

Validate metric	DataLoader 0
hp_metric	2.270128011703491
val_acc	0.24285714328289032
val_loss	2.270128011703491
valid_mapk	0.3857142925262451

```
train_model result: {'valid_mapk': 0.3857142925262451, 'val_loss': 2.270128011703491, 'val_a
```

0.3493527829647064

**i** Note: Evaluation for the Final Comaprison

- This is the evaluation that will be used in the comparison.

### 17.10.3 Detailed Hyperparameter Plots

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

```
dropout_prob: 1.6176864689332775
initialization: 100.0
```

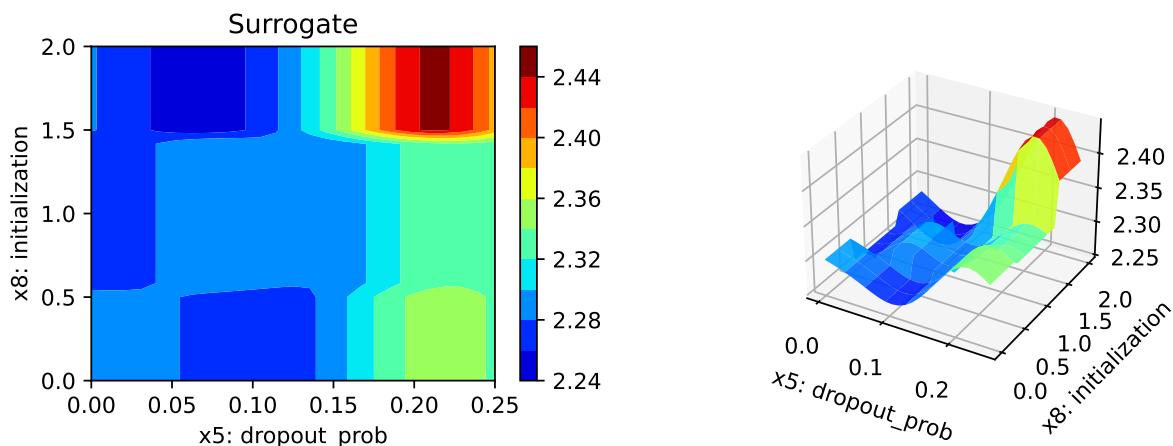


Figure 17.3: Contour plots.

### 17.10.4 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
```

### 17.10.5 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)
```

### 17.10.6 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}

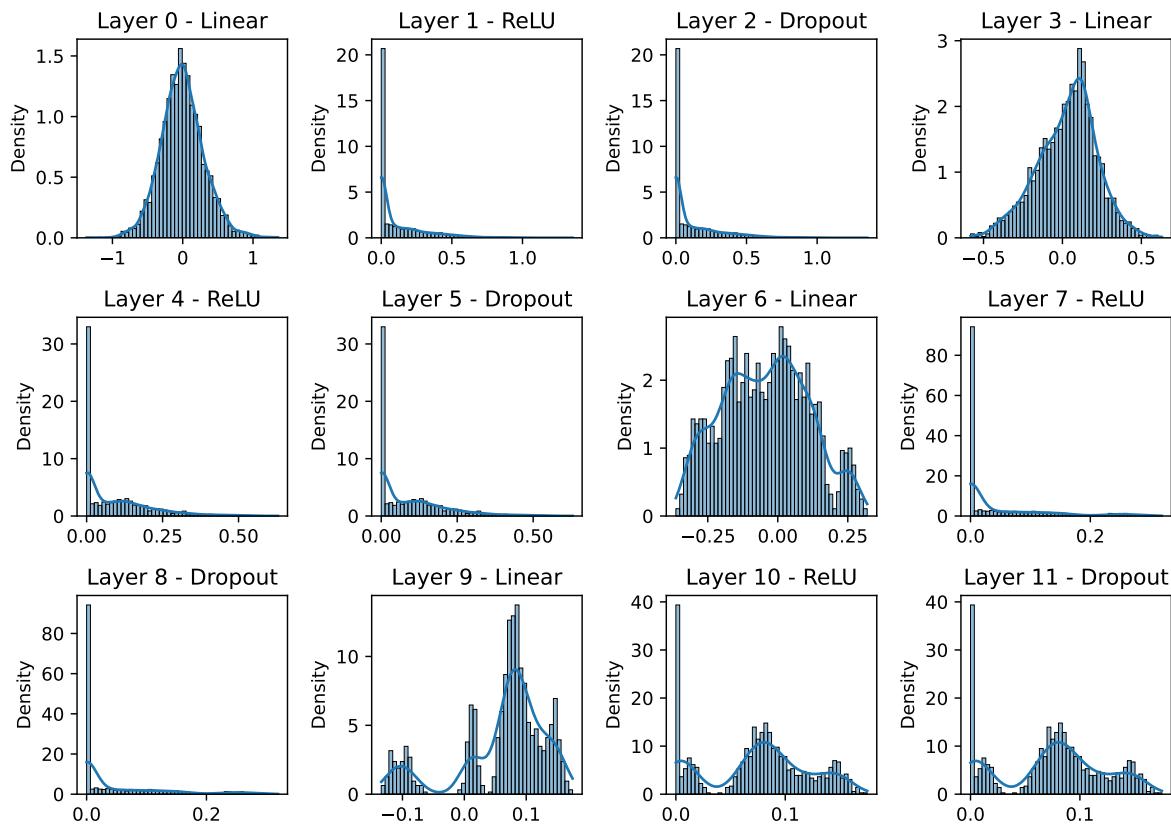
from spotPython.hyperparameters.values import get_one_config_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
config = get_one_config_from_X(X, fun_control)
model = fun_control["core_model"](**config, _L_in=64, _L_out=11)
model

NetLightBase(
    (train_mapk): MAPK()
    (valid_mapk): MAPK()
    (test_mapk): MAPK()
    (layers): Sequential(
        (0): Linear(in_features=64, out_features=32, bias=True)
```

```
(1): ReLU()
(2): Dropout(p=0.07912235457426961, inplace=False)
(3): Linear(in_features=32, out_features=16, bias=True)
(4): ReLU()
(5): Dropout(p=0.07912235457426961, inplace=False)
(6): Linear(in_features=16, out_features=16, bias=True)
(7): ReLU()
(8): Dropout(p=0.07912235457426961, inplace=False)
(9): Linear(in_features=16, out_features=8, bias=True)
(10): ReLU()
(11): Dropout(p=0.07912235457426961, 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 ReLU()



## 17.11 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	1.0
1007	1.0	0.0	0.0	1.0	1.0	0.0	1.0	1.0	1.0
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([[5.41232845e-26, 1.27095045e-05, 0.00000000e+00, ...,
       0.00000000e+00, 1.54120984e-27, 1.93529959e-29],
       [9.99993920e-01, 6.03577928e-06, 3.03779888e-26, ...,
       6.81878993e-17, 5.46532127e-15, 3.95063810e-13], ...
      ])

```

```
[1.46817777e-29, 1.97117316e-07, 9.99999762e-01, ...,
 4.56797316e-22, 1.94337737e-32, 8.91749244e-11],
...,
[0.00000000e+00, 6.26950018e-16, 5.88089470e-16, ...,
 2.32772049e-17, 0.00000000e+00, 4.82207195e-23],
[3.28134635e-15, 5.67918718e-02, 6.38429940e-01, ...,
 8.29307712e-04, 2.18537442e-21, 1.48154972e-02],
[1.39392523e-25, 9.65945560e-07, 9.99998450e-01, ...,
 7.54383891e-15, 0.00000000e+00, 8.63872707e-09]], 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)
```

## 17.12 Appendix

### 17.12.1 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.

#### 17.12.1.1 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 12.4. 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.

### 17.12.2 Taking a Look at the Data

```
import torch
from spotPython.light.csvdataset import CSVDataset
from torch.utils.data import DataLoader
from torchvision.transforms import ToTensor

# Create an instance of CSVDataset
dataset = CSVDataset(csv_file="./data/VBDP/train.csv", train=True)
# show the dimensions of the input data
print(dataset[0][0].shape)
# show the first element of the input data
print(dataset[0][0])
# show the size of the dataset
print(f"Dataset Size: {len(dataset)}")

torch.Size([64])
tensor([1., 1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 0., 1., 1., 0., 0.,
        1., 0., 1., 0., 1., 1., 1., 1., 1., 1., 0., 0., 1., 0., 0., 0., 0.,
        1., 0., 0., 0., 0., 0., 1., 0., 1., 0., 1., 0., 0., 0., 0., 1., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
Dataset Size: 707

# Set batch size for DataLoader
batch_size = 3
# Create DataLoader
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Iterate over the data in the DataLoader
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print("-----")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
```

### 17.12.3 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)
```

# A 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/>.

## A.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
```

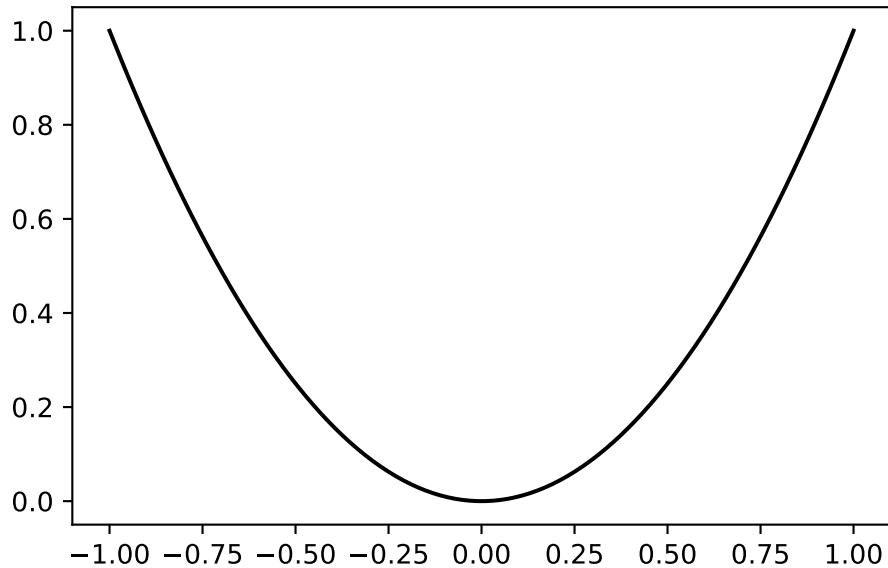
### A.1.1 The Objective Function

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`.

### A.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`

## A.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

## A.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
<code>repeats</code>	int	number of repeats of the initial samples	1	yes

## A.4 The surrogate\_control Dictionary

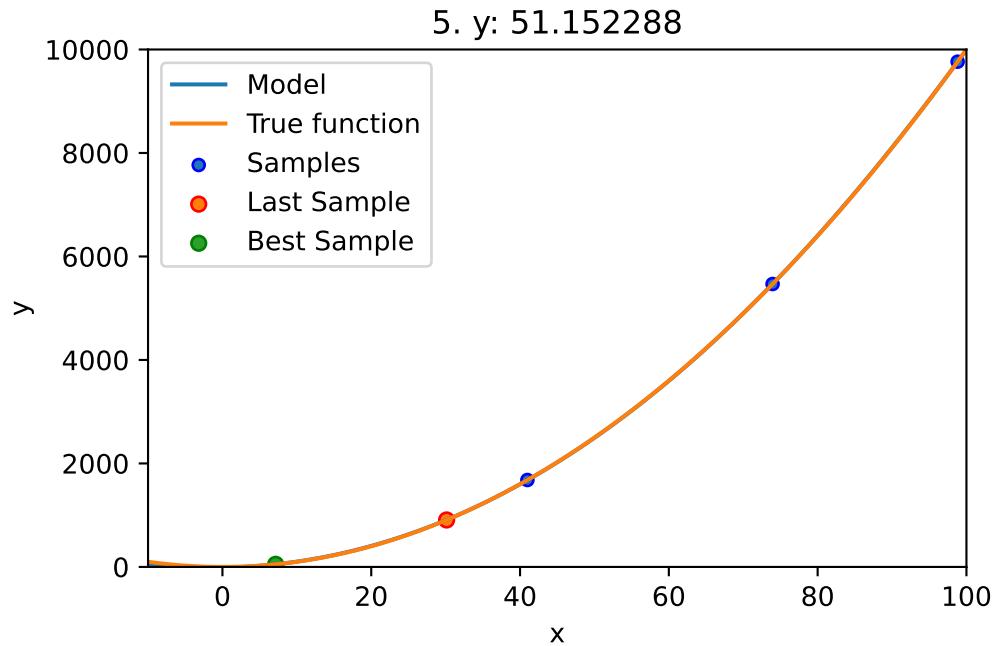
external parameter	type	description	default	mandatory
<code>noise</code>				
<code>model_optimizer</code>	object	optimizer	<code>differential_evolution</code>	
<code>model_fun_evals</code>				
<code>min_theta</code>			-3.	
<code>max_theta</code>			3.	
<code>n_theta</code>			1	
<code>n_p</code>			1	
<code>optim_p</code>			<code>False</code>	
<code>cod_type</code>			"norm"	
<code>var_type</code>				
<code>use_cod_y</code>	bool		<code>False</code>	

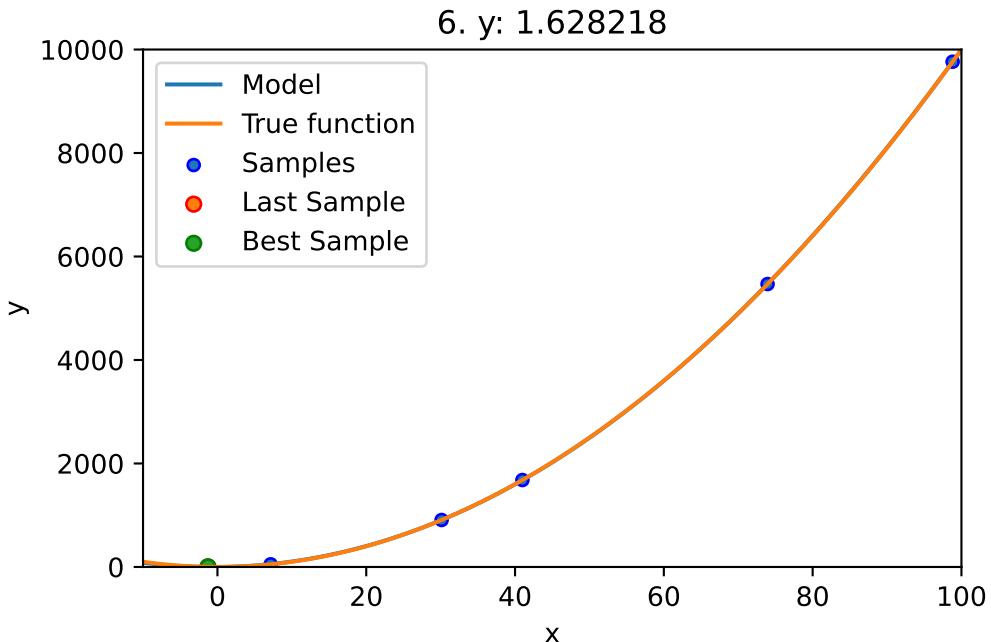
## A.5 The optimizer\_control Dictionary

external parameter	type	description	default	mandatory
<code>max_iter</code>	int	max number of iterations. Note: these are the cheap evaluations on the surrogate.	1000	no

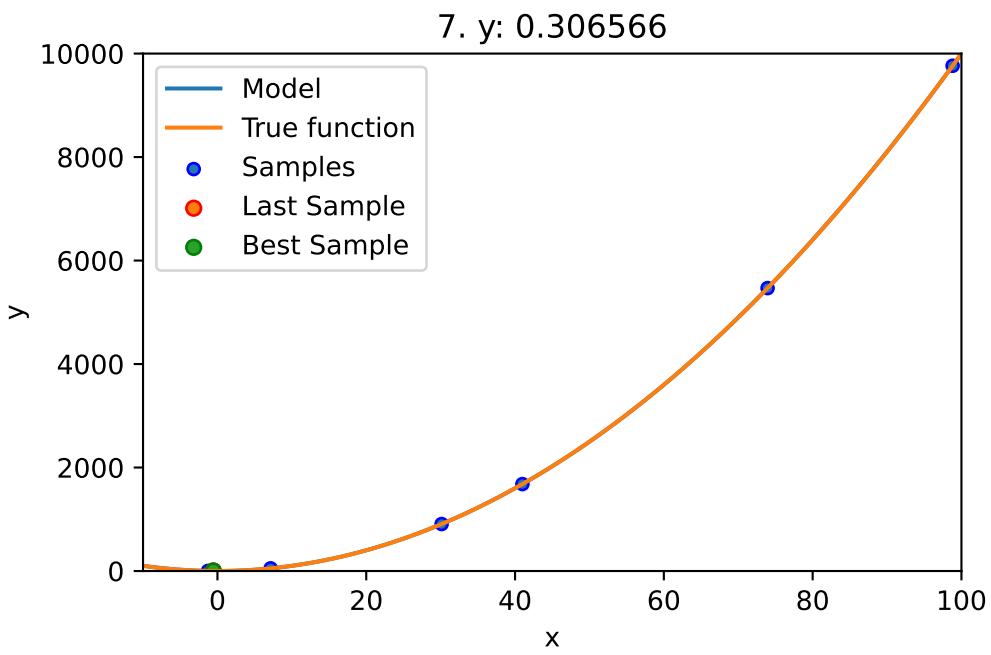
## A.6 Run

```
spot_1.run()
```





spotPython tuning: 1.6282181269484761 [#####-] 85.71%



```
spotPython tuning: 0.30656551286610595 [#####] 100.00% Done...
```

```
<spotPython.spot.spot.Spot at 0x165192fe0>
```

## A.7 Print the Results

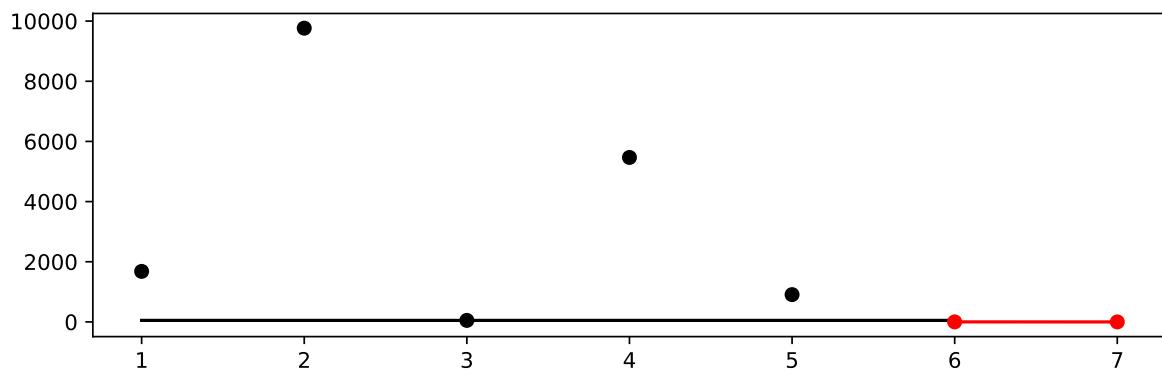
```
spot_1.print_results()
```

```
min y: 0.30656551286610595  
x0: -0.5536835855126157
```

```
[['x0', -0.5536835855126157]]
```

## A.8 Show the Progress

```
spot_1.plot_progress()
```

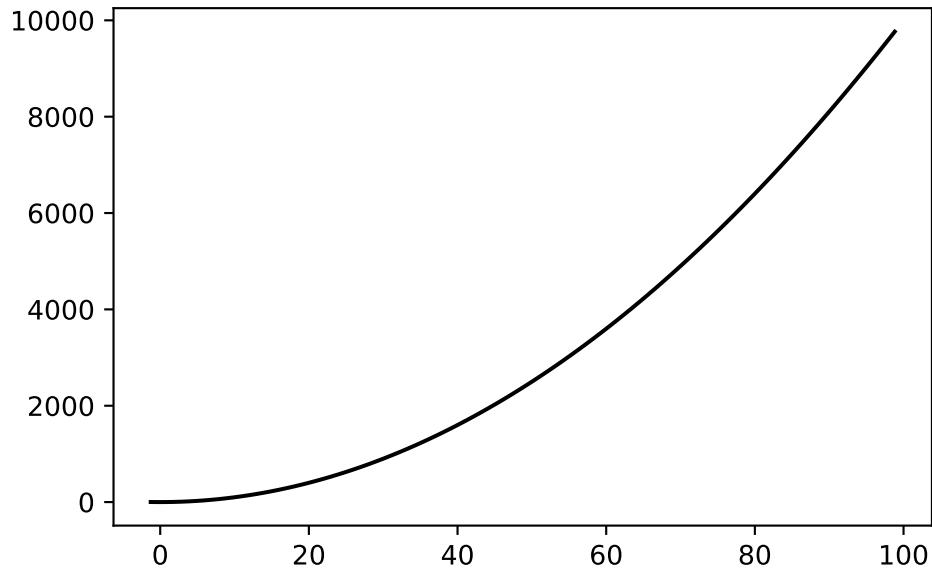


## A.9 Visualize the Surrogate

- 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>



## A.10 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]
```

## A.11 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 ]]))
```

## A.12 Surrogates

### A.12.1 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  $f$ .

## A.13 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   0.67234256         nan         nan -0.75129705  0.97550354
 0.41757584         nan  0.82585329  0.23700598 -0.49274073 -0.82319082
-0.17991251  0.1481835 ]
[-1.]
[-0.47259301]
```

```
[0.95541987]
```

```
[nan]
[0.17335968]
```

```
[nan]
```

```
[-0.58552368]
[-0.20126111]
```

```
[-0.60100809]
[-0.97897336]
```

$[-0.2748985]$   
 $[0.8359486]$

$[0.99035591]$   
 $[0.01641232]$

$[\text{nan}]$   
 $[0.5629346]$

## A.14 PyTorch: Detailed Description of the Data Splitting

### A.14.1 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

```

#### A.14.1.1 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.

#### A.14.1.2 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`.

#### A.14.1.3 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_torch()` 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.

#### A.14.1.4 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 Sourty, 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).