

# Kernel ridge regression vs Gaussian process

Lorenz Wolf

October 14, 2021

## 1 Kernel ridge regression

In this part a kernel ridge regression model with polynomial kernel  $K(x_i, x_j) = (x_i x_j + 1)^p$ ,  $p \in \mathbb{N}$  is used to construct a linear model of the form  $f(x; \theta) = \sum_{i=1}^{200} K(x, x^{(i)}) \theta_i$  with dependent variable  $y$ , where  $\{x^{(i)}\}_{i=1}^{200}$  are the training data inputs, and  $\theta_i \in \mathbb{R}$ .

The solution to the regression problem at hand is given by

$$\theta = (\mathbf{K}(\mathbf{x}, \mathbf{x}) + \lambda \mathbf{I}_N)^{-1} \mathbf{y},$$

where the kernel matrix is defined as follows

$$\mathbf{K}(\mathbf{x}, \mathbf{x}) = \begin{bmatrix} K(x^{(1)}, x^{(1)}) & K(x^{(1)}, x^{(2)}) & \dots & K(x^{(1)}, x^{(N)}) \\ K(x^{(2)}, x^{(1)}) & K(x^{(2)}, x^{(2)}) & \dots & K(x^{(2)}, x^{(N)}) \\ \vdots & \vdots & \ddots & \vdots \\ K(x^{(N)}, x^{(1)}) & K(x^{(N)}, x^{(2)}) & \dots & K(x^{(N)}, x^{(N)}) \end{bmatrix}.$$

To optimise the hyper-parameters  $p$  and  $\lambda$  grid search is performed and the pair with the smallest averaged squared error during leave-one-out cross-validation is chosen. The parameter ranges considered are  $p \in \{1, 2, \dots, 7\}$  and  $\lambda \in \{2^{-7+2k/100} \text{ for } k = 0, 1, 2, \dots, 500\}$ . The resulting average square error values are visualised in Figure 1 and a zoomed in version on  $p=5, 6$  is shown in Figure 2.

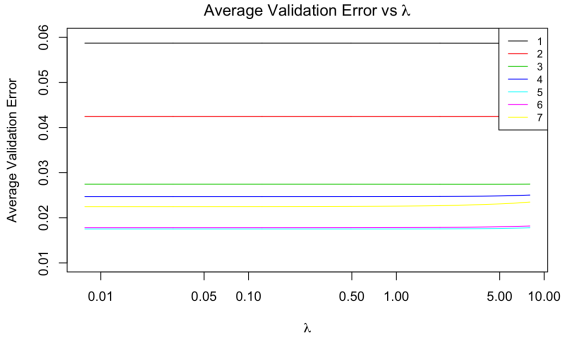


Figure 1: Average validation error vs  $\lambda$

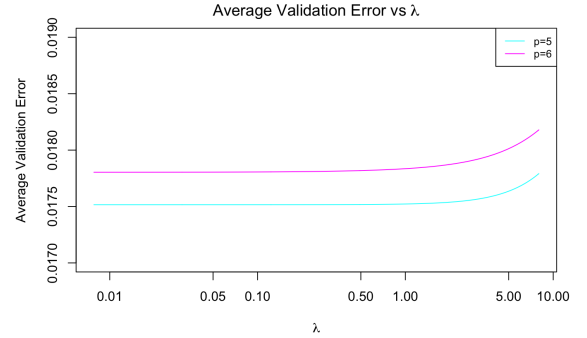


Figure 2: Average validation error vs  $\lambda$  zoomed in.

The parameters minimising the average squared error are  $p = 5$  and  $\lambda = 0.0079$ . These parameters are fixed and the model is trained on the entire train set. The predictions are then obtained by

$$y^* = \mathbf{K}(x^*, \mathbf{x}) \theta,$$

where  $\mathbf{K}(x^*, \mathbf{x})$  is given by  $\mathbf{K}(x^*, \mathbf{x}) = [K(x^*, x^{(1)}), K(x^*, x^{(2)}), \dots, K(x^*, x^{(N)})]$ . In Figure 3 and Figure 4 the model fits to train and test data, respectively, are presented. It can be observed that the model does fit the train data reasonably well, however the fit to the test set, which contains values outside the range of  $x$  values in the train data, the fit is poor. The model is then evaluated on the test data. On the test set a mean squared error of 0.0537 and a mean absolute error of 0.1825 are obtained. Compared to an MSE on the train set of 0.0162 this indicates weak generalisability of the model especially to  $x$ -values slightly outside the range of seen values. Furthermore, it can be observed that the model fit could be improved around  $x=-3.8$  and  $x=2$ .

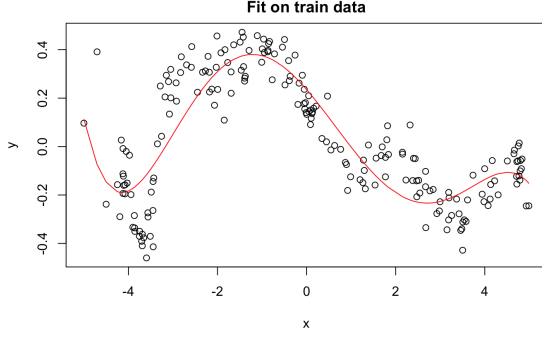


Figure 3: Fit to train set.

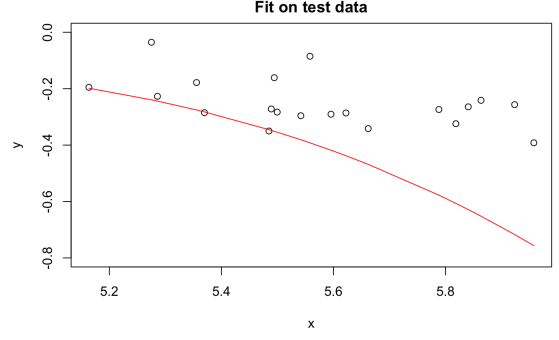


Figure 4: Fit to test set.

## Gaussian Process

When fitting a Gaussian Process a crucial part is the choice of the covariance function which encodes the assumptions about the function to be learnt. The three different kernels we will consider here are the Radial Basis Function kernel (RBF), the Rational Quadratic kernel (RQ) and the Periodic kernel (P). As we will discuss later the Periodic kernel is not appropriate. In the following parts the key properties and assumptions for each kernel will be outlined to compare the kernels quantitatively.

First, as defined in Rasmussen [3], the mean square derivative of  $f(\mathbf{x})$  in the  $i$ th direction is defined as

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \text{l.i.m}_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}$$

when the limit exists, where l.i.m denotes the limit in mean square and  $\mathbf{e}_i$  is the unit vector in the  $i$ th direction.

### RBF kernel

The RBF kernel also called Square Exponential kernel is defined by

$$k_{\text{SE}}(x, x') = \sigma^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right).$$

Here the parameter  $\ell > 0$  is the characteristic length-scale. The parameter  $\ell$  determines the length of the 'wiggles' in the function as it scales inside the exponential. Increasing  $\ell$  will elongate the 'wiggles'. The second parameter is  $\sigma$  which is part of all kernels considered. It determines how far the function will vary from the mean, i.e. increasing  $\sigma$  increases the amplitude of the function. We note that the RBF kernel assumes a positive covariance between any two points but it goes to zero quickly as  $(x - x')^2$  increases, i.e. the distance between points increases. Furthermore, samples from a GP with RBF kernel are very smooth as the covariance function is infinitely differentiable and thus the GP has mean square derivatives of all orders (see [3]). Consequently a weakness of the RBF kernel is its extreme smoothness, as it might fail to extrapolate smooth regions due to small non-smooth regions since the length-scale is determined by the smallest 'wiggles'[1].

### RQ kernel

The RQ kernel is defined by

$$k_{\text{RQ}}(x, x') = \sigma^2 \left(1 + \frac{(x - x')^2}{2\alpha\ell^2}\right)^{-\alpha},$$

with  $\alpha, \ell > 0$ . This kernel can be seen as adding infinitely many RBF kernels with different length-scales together. Thus, priors of a GP with the RQ kernel can handle functions varying smoothly across several length-scales. To adjust the variation between long scale and small scale the parameter  $\alpha$  can be chosen. With a larger  $\alpha$  the covariance decays quicker as the input distance increases until for  $\alpha \rightarrow \infty$  the kernel tends to the RBF kernel. The parameters  $\sigma$  plays the same role as for the RBF kernel. Note that the RQ kernel has the same smoothness assumption and potential issue as the RBF kernel.

## Periodic kernel

The periodic kernel is defined by

$$k_{\text{Per}}(x, x') = \sigma^2 \exp\left(-\frac{2 \sin^2(\pi |x - x'|/p)}{\ell^2}\right),$$

where  $p > 0$  determines the period of the resulting function and  $\ell$  has the same role as in the RBF kernel. As the name suggests a key assumption is that the resulting function is periodic which can be useful in case of temperature measurements over years but can also be undesirable when no periodicity is expected.

## Fit and Discussion

Before fitting the models we can make some important observations about the relevance of the different kernels for the training data at hand. Firstly, we do not have any reason to suspect periodicity and hence expect the periodic kernel to perform worse. Secondly, the strong smoothness assumptions of RBF and RQ kernel could be limiting in the fit especially around  $x=-3.8$ . Furthermore, due to the additional flexibility we expect the model with RQ kernel to yield better results than the RBF kernel, as it is a generalisation.

To fit the different models we use the TensorFlow module for distributions in Python. The kernel hyper-parameters are optimised by maximising the marginal likelihood  $p(\mathbf{y} | X, \theta)$ , i.e.

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}(p(\mathbf{y} | X, \theta)).$$

This is equivalent to minimising the negative log marginal likelihood. Letting  $\mathbf{K} = k_{\theta}(X, X)$  we can write the log marginal likelihood of the Gaussian process as

$$\log p(\mathbf{y} | \mathbf{X}, \boldsymbol{\theta}) = -\frac{1}{2} \mathbf{y}^{\top} \mathbf{K}_{\boldsymbol{\theta}}^{-1} \mathbf{y} - \frac{1}{2} \log |\mathbf{K}_{\boldsymbol{\theta}}| + \text{const}, \quad \mathbf{K}_{\boldsymbol{\theta}} := \mathbf{K} + \sigma_n^2 \mathbf{I},$$

where we have specified the prior mean function to be 0. Following the lecture slides we then have that

$$\frac{\partial \log p(\mathbf{y} | \mathbf{X}, \boldsymbol{\theta})}{\partial \theta_i} = \frac{1}{2} \operatorname{tr} \left( (\boldsymbol{\alpha} \boldsymbol{\alpha}^{\top} - \mathbf{K}_{\boldsymbol{\theta}}^{-1}) \frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \theta_i} \right)$$

$$\boldsymbol{\alpha} := \mathbf{K}_{\boldsymbol{\theta}}^{-1} \mathbf{y}$$

For the gradient descent on the negative log marginal likelihood we use an Adam optimiser with a learning rate of 0.01.[2]. All the parameters are initialised at 1, except the period with initial value of 10, and restricted to be positive as required. The gradient descent algorithm is then run for 1000 iterations. The trained parameters are reported in Table 2.

Assuming a uniform prior on the models we can also compare the log-marginal likelihoods (LML) of obtained by training each model on the entire train set (see Table 1). The periodic kernel has a significantly lower log marginal likelihood than both other kernels. The RQ kernel yields a slightly higher LLM than RBF.

To further compare models a shuffled 4-fold cross-validation is implemented using the Sklearn library. For each model and each split the MSE on the validation split is computed and the average of all folds is reported for each model (see Table 1). We find that as expected the Periodic kernel performs poorest in terms of training MSE, as well as cross validation average MSE. As for the log marginal likelihood comparison RQ improves on RBF.

In Figures 5 and 6 the fit to train and test data of the model with Rational Quadratic kernel is visualised together with 95% credible intervals. The fit on the train set looks very good, especially in regions where kernel ridge regression was struggling. Furthermore, on the test set we do not observe a strong deviation as for kernel ridge regression. However, the mean does slightly increase for the test set.

Model	P	RBF	RQ
Train MSE	0.0103	0.005	0.0048
4-CV MSE	0.0111	0.0089	0.0077
Test MSE	0.3277	0.0449	0.0194
Train LML	128.14	171.99	174.5

Table 1: MSE on train and test set and log-marginal likelihood (LML) after training on the entire train set. Additionally, 4-fold cross validation is performed and the average validation MSE is reported in row 4-CV MSE. Here only 750 optimisation iterations are performed to save computational cost.

Parameter	P	RBF	RQ
amplitude	0.215	0.2126	0.2363
length scale	0.4248	0.3429	0.4694
observation noise variance	0.0116	0.0061	0.006
scale mixture rate	na	na	0.4016
period	8.1948	na	na

Table 2: Final set of optimised parameters after training on the full train set.

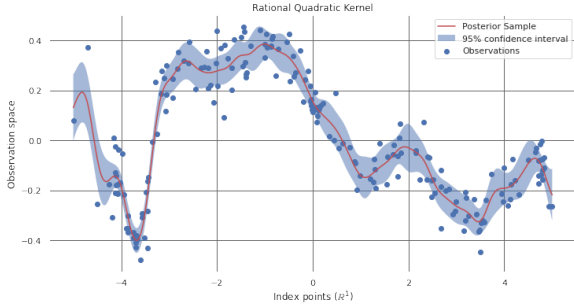


Figure 5: Fit to train set.

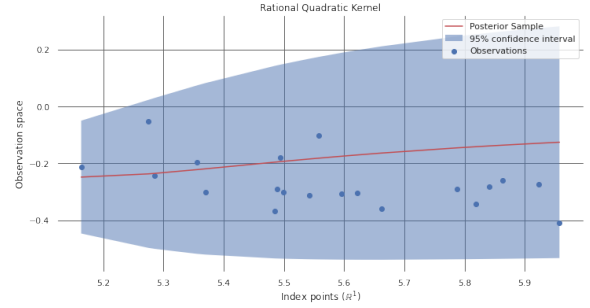


Figure 6: Fit to test set.

## References

- [1] David Duvenaud. *The Kernel Cookbook*. URL: <https://www.cs.toronto.edu/~duvenaud/cookbook/>.
- [2] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].
- [3] Carl Edward Rasmussen. “Gaussian processes for machine learning”. In: MIT Press, 2006.

# Appendix

Listing 1: Code for question 1

```
1 from google.colab import drive
2 drive.mount("/content/drive")
3
4 # basic libraries
5 import pandas as pd
6 import numpy as np
7 import matplotlib.pyplot as plt
8 import seaborn as sns
9
10 # sklearn
11 from sklearn.cluster import KMeans
12 from sklearn.metrics import silhouette_score
13 from sklearn.metrics import accuracy_score
14 from sklearn.metrics.cluster import adjusted_rand_score
15 from sklearn.metrics.cluster import adjusted_mutual_info_score
16 from sklearn.decomposition import PCA
17 from sklearn.neighbors import KNeighborsClassifier
18
19 %matplotlib inline
20
21
22
23 import warnings
24 warnings.filterwarnings("ignore")
25
26 # sklearn supervised learning
27 from sklearn.metrics import classification_report, confusion_matrix
28 from sklearn.model_selection import StratifiedKFold
29
30 # make plots nicer
31 sns.set()
32
33 # load data
34 q1_data = pd.read_csv("/content/drive/My Drive/ML Coursework 2/CW2Data.csv")
35 pd.DataFrame.head(q1_data)
36
37 # overview
38 q1_data.describe()
39 q1_data.corr()
40
41 from sklearn.ensemble import RandomForestClassifier
42 from sklearn.model_selection import StratifiedKFold
43 from sklearn.metrics import accuracy_score
44
45 X = q1_data.iloc[:, 1:]
46 y = q1_data['z']
47
48
49 # split data into predictors and target
50 from sklearn.model_selection import StratifiedShuffleSplit
51
52 sss = StratifiedShuffleSplit(n_splits=1, test_size=0.5, random_state=0)
53 # split data into train and test set
54 for train_index, test_index in sss.split(X, y):
55     X_train, X_test = X.iloc[train_index,:], X.iloc[test_index,:]
56     y_train, y_test = y[train_index], y[test_index]
```

```

57
58 # perform mean imputation, could do regression imputation mice package as well...
59 from sklearn.impute import SimpleImputer
60
61 imp = SimpleImputer(missing_values=np.nan, strategy='mean')
62 imp.fit(X_train)
63
64 X_train = imp.transform(X_train)
65 X_test = imp.transform(X_test)
66
67 #EDA
68 y_train = y_train.reset_index(drop=True)
69 X_df = pd.DataFrame(X_train)
70 X_df.columns = q1_data.columns[1:]
71 train_df = pd.concat([y_train, X_df], axis=1)
72 train_df.columns = q1_data.columns
73 train_df.describe()
74 corr_mat = train_df.corr()
75
76 # plot histograms
77 fig, axis = plt.subplots(7,4,figsize=(30, 30))
78 counter = 1
79 for ax in axis.flatten():
80     sns.histplot(train_df, x=train_df.columns[counter], ax=ax, bins=100, hue='z')
81     counter += 1
82
83 # train RF
84 from sklearn.model_selection import RandomizedSearchCV
85 # Number of trees in random forest
86 n_estimators = [int(x) for x in np.arange(2, 300, 6)]
87 # Number of features to consider at every split
88 max_features = [int(x) for x in np.arange(2, 28, 3)]
89 # Maximum number of levels in tree
90 max_depth = [int(x) for x in np.arange(1, 30, 1)]
91 max_depth.append(None)
92
93 # Create the random grid
94 random_grid = {'n_estimators': n_estimators,
95               'max_features': max_features,
96               'max_depth': max_depth}
97
98 # Use the random grid to search for best hyperparameters
99 # First create the base model to tune
100 rf = RandomForestClassifier(random_state = 0)
101 # Random search of parameters, using 5 fold cross validation,
102 # search across 200 different combinations, and use all available cores
103 rf_random = RandomizedSearchCV(estimator=rf, param_distributions=random_grid,
104                               n_iter = 200, scoring='accuracy',
105                               cv = 5, verbose=2, random_state=0, n_jobs=-1,
106                               return_train_score=True)
107
108 # Fit the random search model
109 rf_random.fit(X_train, y_train);
110
111 print(rf_random.best_params_)
112 print(rf_random.best_estimator_)
113 print(rf_random.best_score_)
114
115
116
117

```

```

118
119
120
121 # predictions for random forest and test accuracy
122 rf_rand = RandomForestClassifier(n_estimators= 146, max_features=5, max_depth= 7, ←
    random_state=0)
123 rf_rand.fit(X_train, y_train)
124 rf_pred = rf_rand.predict(X_test)
125 print('The accuracy obtained on the unseen data in the test set is:', accuracy_score←
    (y_test, rf_pred))
126
127 # training neural net
128
129 import tensorflow as tf
130 import math
131 from tensorflow.keras import backend as K
132 from tensorflow.keras.models import Sequential
133 from tensorflow.keras.layers import InputLayer, Input
134 #from tensorflow.keras.layers import Reshape
135 from tensorflow.keras.layers import Dropout, Dense, Flatten, BatchNormalization
136 from tensorflow.keras.callbacks import TensorBoard, EarlyStopping
137 from tensorflow.keras.optimizers import Adam
138 from tensorflow.keras.models import load_model
139 from tensorflow.keras import regularizers
140
141 ! pip install h5py scikit-optimize
142 import skopt
143 from skopt import gp_minimize, forest_minimize
144 from skopt.space import Real, Categorical, Integer
145 from skopt.plots import plot_convergence
146 from skopt.plots import plot_objective, plot_evaluations
147 from skopt.plots import plot_histogram, plot_objective_2D
148 from skopt.utils import use_named_args
149
150 dim_learning_rate = Real(low=1e-6, high=1e-2, prior='log-uniform',
    name='learning_rate')
151
152 dim_dropout_rate = Real(low=1e-2, high=0.5, prior='log-uniform',
    name='dropout_rate')
153
154 dim_num_dense_layers = Integer(low=1, high=8, name='num_dense_layers')
155 dim_num_dense_nodes = Integer(low=5, high=512, name='num_dense_nodes')
156 dim_activation = Categorical(categories=['relu', 'sigmoid'],
    name='activation')
157
158 dimensions = [dim_learning_rate,
159               dim_num_dense_layers,
160               dim_num_dense_nodes,
161               dim_activation,
162               dim_dropout_rate]
163 default_parameters = [1e-5, 1, 300, 'relu', 1e-1]
164
165 # split training set again into train and validations set
166 train_full_dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train))
167 train_full_dataset = train_full_dataset.shuffle(353)
168 train_full_dataset = train_full_dataset.batch(89)
169 train_full_dataset = train_full_dataset.prefetch(tf.data.experimental.AUTOTUNE)
170
171 trainval = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=0)
172 for train_index, test_index in trainval.split(X_train, y_train):
173     X_t, X_val = X_train[train_index:], X_train[test_index, :]
174     y_t, y_val = y_train[train_index], y_train[test_index]
175
176 # Load the data into training and validation Dataset objects

```



```

177 train_dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train))
178 val_dataset = tf.data.Dataset.from_tensor_slices((X_val, y_val)) # should be ←
    validation data
179
180 train_dataset = train_dataset.shuffle(353)
181
182 train_dataset = train_dataset.batch(89)
183 val_dataset = val_dataset.batch(len(y_val)) # just one batch for validation
184
185 train_dataset = train_dataset.prefetch(tf.data.experimental.AUTOTUNE)
186
187 def log_dir_name(learning_rate, num_dense_layers,
188                 num_dense_nodes, activation, dropout_rate):
189
190     # The dir-name for the TensorBoard log-dir.
191     s = "./19_logs/lr_{0:.0e}_layers_{1}_nodes_{2}_{3}_dr_{0:.0e}/"
192
193     # Insert all the hyper-parameters in the dir-name.
194     log_dir = s.format(learning_rate,
195                       num_dense_layers,
196                       num_dense_nodes,
197                       activation,
198                       dropout_rate)
199
200     return log_dir
201
202
203 l2_coeff = 1e-5
204
205
206 def create_model(learning_rate, num_dense_layers,
207                 num_dense_nodes, activation, dropout_rate):
208     """
209     Hyper-parameters:
210     learning_rate:    Learning-rate for the optimizer.
211     num_dense_layers: Number of dense layers.
212     num_dense_nodes:  Number of nodes in each dense layer.
213     activation:        Activation function for all layers.
214     """
215
216     # Start construction of a Keras Sequential model.
217     model = Sequential()
218
219     # Add an input layer which is similar to a feed_dict in TensorFlow.
220     # Note that the input-shape must be a tuple containing the image-size.
221
222     model.add(InputLayer(input_shape = (X_train.shape[-1],))) # adjust this
223     #model.add(Flatten()) #think don't need this
224
225     # Add fully-connected / dense layers.
226     # The number of layers is a hyper-parameter we want to optimize.
227     for i in range(num_dense_layers):
228
229
230         # Name of the layer. This is not really necessary
231         # because Keras should give them unique names.
232         name = 'layer_dense_{0}'.format(i+1)
233
234         # Add the dense / fully-connected layer to the model.
235         # This has two hyper-parameters we want to optimize:
236         # The number of nodes and the activation function.

```

```

237         model.add(Dense(num_dense_nodes,
238                         activation=activation,
239                         kernel_regularizer=regularizers.l2(l2_coeff),
240                         name=name))
241     # Add Dropout Layer
242     name = 'layer_dropout_{0}'.format(i+1)
243     model.add(Dropout(dropout_rate, name=name))
244
245
246
247     # Last fully-connected / dense layer with sigmoid-activation
248     # for use in classification.
249     model.add(Dense(1, activation='sigmoid')) # 2 classes
250
251     # Use the Adam method for training the network.
252     # We want to find the best learning-rate for the Adam method.
253     optimizer = Adam(lr=learning_rate)
254
255     # In Keras we need to compile the model so it can be trained.
256     model.compile(optimizer=optimizer,
257                  loss='binary_crossentropy',
258                  metrics=['binary_accuracy'])
259
260     return model
261
262
263 path_best_model = '19_best_model.h5'
264 best_accuracy = 0.0
265 @use_named_args(dimensions=dimensions)
266
267 def fitness(learning_rate, num_dense_layers,
268            num_dense_nodes, activation, dropout_rate):
269     """
270     Hyper-parameters:
271     learning_rate:    Learning-rate for the optimizer.
272     num_dense_layers: Number of dense layers.
273     num_dense_nodes:  Number of nodes in each dense layer.
274     activation:        Activation function for all layers.
275     """
276
277     # Print the hyper-parameters.
278     print('learning rate: {0:.1e}'.format(learning_rate))
279     print('num_dense_layers:', num_dense_layers)
280     print('num_dense_nodes:', num_dense_nodes)
281     print('activation:', activation)
282     print('dropout rate: {0:.1e}'.format(dropout_rate))
283     print()
284
285     # Create the neural network with these hyper-parameters.
286     model = create_model(learning_rate=learning_rate,
287                          num_dense_layers=num_dense_layers,
288                          num_dense_nodes=num_dense_nodes,
289                          activation=activation, dropout_rate=dropout_rate)
290
291     # Dir-name for the TensorBoard log-files.
292     log_dir = log_dir_name(learning_rate, num_dense_layers,
293                           num_dense_nodes, activation, dropout_rate)
294
295     # Create a callback-function for Keras which will be
296     # run after each epoch has ended during training.
297     # This saves the log-files for TensorBoard.

```

```

298 # Note that there are complications when histogram_freq=1.
299 # It might give strange errors and it also does not properly
300 # support Keras data-generators for the validation-set.
301 callback_log = TensorBoard(
302     log_dir=log_dir,
303     histogram_freq=0,
304     write_graph=True,
305     write_grads=False,
306     write_images=False)
307
308 # callback_ES = EarlyStopping(monitor = 'val_accuracy', patience=10)# probs no ↔
309     need for this as only 3 epochs
310
311 # Use Keras to train the model.
312 history = model.fit(train_dataset,
313                     epochs=10,
314                     #batch_size=128,
315                     validation_data=val_dataset,
316                     callbacks=[callback_log])
317
318 # Get the classification accuracy on the validation-set
319 # after the last training-epoch.
320 accuracy = history.history['val_binary_accuracy'][-1]
321
322 # Print the classification accuracy.
323 print()
324 print("Accuracy: {0:.2%}".format(accuracy))
325 print()
326
327 # Save the model if it improves on the best-found performance.
328 # We use the global keyword so we update the variable outside
329 # of this function.
330 global best_accuracy
331
332 # If the classification accuracy of the saved model is improved ...
333 if accuracy > best_accuracy:
334     # Save the new model to harddisk.
335     model.save(path_best_model)
336
337     # Update the classification accuracy.
338     best_accuracy = accuracy
339
340 # Delete the Keras model with these hyper-parameters from memory.
341 del model
342
343 # Clear the Keras session, otherwise it will keep adding new
344 # models to the same TensorFlow graph each time we create
345 # a model with a different set of hyper-parameters.
346 K.clear_session()
347
348 # NOTE: Scikit-optimize does minimization so it tries to
349 # find a set of hyper-parameters with the LOWEST fitness-value.
350 # Because we are interested in the HIGHEST classification
351 # accuracy, we need to negate this number so it can be minimized.
352 return -accuracy
353
354 fitness(x=default_parameters)
355 plot_convergence(search_result)
356 search_result.x
357

```

```

358 from sklearn.metrics import confusion_matrix
359
360 confusion_matrix_validation = confusion_matrix(y_test, np.array(rf_pred))
361
362 df_cm = pd.DataFrame(confusion_matrix_validation, columns=np.unique(y_test), index ←
    = np.unique(y_test))
363 df_cm.index.name = 'Actual'
364 df_cm.columns.name = 'Predicted'
365 plt.figure(figsize = (10,7))
366 plt.title('Test Data RF')
367 sns.set(font_scale=1.4)#for label size
368 sns.heatmap(df_cm, cmap="Blues", annot=True,annot_kws={"size": 16})# font size
369 print(classification_report(y_test, rf_pred))
370
371
372 final_model = create_model(learning_rate=0.0053,
373                             num_dense_layers=1,
374                             num_dense_nodes=302,
375                             activation='relu', dropout_rate=0.3)
376
377 final_model.summary()
378 callback_ES = EarlyStopping(monitor = 'val_binary_accuracy', patience=10)
379
380 # Use Keras to train the model.
381 history = final_model.fit(train_full_dataset,
382                             epochs=50,
383                             #batch_size=128,
384                             validation_data=val_dataset,
385                             callbacks=[])
386
387 # Get the classification accuracy on the validation-set
388 # after the last training-epoch.
389 accuracy = history.history['val_binary_accuracy'][-1]
390
391 # Load the data into training and validation Dataset objects
392 test_dataset = tf.data.Dataset.from_tensor_slices((X_test, y_test))
393
394 test_dataset = test_dataset.batch(len(y_test))
395 final_model.evaluate(test_dataset)
396 mlp_pred = final_model.predict(test_dataset)
397 mlp_pred[mlp_pred>0.5] = 1
398 mlp_pred[mlp_pred<0.5] = 0
399
400 confusion_matrix_validation = confusion_matrix(y_test, np.array(mlp_pred))
401
402 df_cm = pd.DataFrame(confusion_matrix_validation, columns=np.unique(y_test), index ←
    = np.unique(y_test))
403 df_cm.index.name = 'Actual'
404 df_cm.columns.name = 'Predicted'
405 plt.figure(figsize = (10,7))
406 plt.title('Test Data MLP')
407 sns.set(font_scale=1.4)#for label size
408 sns.heatmap(df_cm, cmap="Blues", annot=True,annot_kws={"size": 16})# font size
409 print(classification_report(y_test, mlp_pred))
410
411 # function to compute mc nemar's test statistics
412 def mcnemar(pred_A, pred_B, y_true):
413     errors_A = (pred_A != y_true)
414     errors_B = (pred_B != y_true)
415
416     n_A = np.sum(np.invert(errors_B[errors_A]))

```

```

417     n_B = np.sum(np.invert(errors_A[errors_B]))
418
419     score = (np.abs(n_A-n_B)-1)/np.sqrt(n_A+n_B)
420     return score
421
422     mlp_pred =mlp_pred.reshape(-1)
423     mcnemar(rf_pred, mlp_pred, y_test)
424
425 from scipy.stats import norm
426 p = 1-norm.cdf(2.1667)
427 print(p)
428
429 # reject option for random forest and confusion matrix.
430 rej = np.zeros(X_test.shape[0])
431 probs = rf_rand.predict_proba(X_test)
432 probs.shape
433 max_probs = np.max(probs, axis=1)
434 rej[1-max_probs>0.4] = -1
435 rej[1-max_probs<=0.4] = 1
436
437 classes = rf_rand.predict(X_test)
438 classes[rej==-1] = -1
439
440 y_rej = y_test[classes!=-1]
441 pred_rej = classes[classes!=-1]
442
443 confusion_matrix_validation = confusion_matrix(y_rej, np.array(pred_rej))
444
445 df_cm = pd.DataFrame(confusion_matrix_validation, columns=np.unique(y_rej), index =↵
    np.unique(y_rej))
446 df_cm.index.name = 'Actual'
447 df_cm.columns.name = 'Predicted'
448 plt.figure(figsize = (10,7))
449 plt.title('Reject option RF')
450 sns.set(font_scale=1.4)#for label size
451 sns.heatmap(df_cm, cmap="Blues", annot=True,annot_kws={"size": 16})# font size
452 print(classification_report(y_rej, pred_rej))

```

---

Listing 2: Code for question 2

```

1  library(ggplot2)
2  library(dplyr)
3  library(tidyr)
4  library(data.table)
5  #library(matrixStats)
6  library(GGally)
7  set.seed(1351308)
8
9  cw_data <- read.csv('/Users/lorenzwolf/Desktop/MSc Statistics/Electives/Machine ↵
    Learning/Coursework 2/CW2Data.csv')
10 head(cw_data)
11 summary(cw_data)
12
13 library(purrr)
14
15 get_clust_assignment <- function(dist_method, linkage_method) {
16   hclust(dist(X, method=dist_method), method=linkage_method)
17 }
18
19 # perform hierarchichal clustering using different arguments

```

```

20 hclust_args <- expand.grid(dist_method=c("euclidean", "manhattan"), linkage_method=c(
  c("complete", "single", "average", "ward.D2"))
21 hclust_list <- pmap(hclust_args, get_clust_assignment)
22
23 # plot the resulting dendrograms
24 par(mfrow=c(4,2))
25 for(i in 1:length(hclust_list)) {
26   plot(hclust_list[[i]],
27     main=sprintf("%s distance and %s linkage", hclust_args[i,1], hclust_args[i,2]),
28     xlab="State")
29   rect.hclust(hclust_list[[i]], k = 14, border = 2:n_clust) # add rectangle
30 }
31
32 library(cluster)
33 library("mice")
34
35 # find best hierarchical clustering
36 set.seed(2)
37 X <- t(scale(features))
38 #X <- scale(X)
39
40 num_clusters <- seq(2,27)
41 E_dist <- dist(X, method='euclidean')
42 M_dist <- dist(X, method='manhattan')
43 silhouette_mat <- matrix(0, nrow=length(num_clusters), ncol=length(hclust_list))
44 for(j in 1:length(num_clusters)){
45   for(i in 1:length(hclust_list)) {
46     grp <- cutree(hclust_list[[i]], k = num_clusters[j])
47
48     if (hclust_args[i,1]=="euclidean"){
49       s_score <- summary(silhouette(grp, E_dist))$avg.width
50     }
51     else {
52       s_score <- summary(silhouette(grp, M_dist))$avg.width
53     }
54     silhouette_mat[j,i] <- s_score
55   }
56 }
57 which(silhouette_mat == max(silhouette_mat), arr.ind = TRUE)
58
59 plot(2:27, silhouette_mat[,8], xlab='Number of cluster', ylab='Average silhouette
  width', main='Silhouette score vs number of clusters')
60 lines(2:27, silhouette_mat[,8], lty=2)
61 lines(c(14,14), c(0,0.2), lty=2)
62
63 # regression imputation
64 imp <- mice(scale(features), method = "norm.nob", m = 1) # Impute data
65 X <- complete(imp)
66 silhouette_mat
67
68 A <- na.omit(X)
69 km <- kmeans(A, 2, nstart=10)
70 pca <- prcomp(A, rank. = 4)
71
72
73 df <- as.data.frame(cbind(pca$x, km$cluster))
74 colnames(df)[[5]] <- "cluster_label"
75 df$cluster_label <- as.factor(df$cluster_label)
76
77 ggpairs(df, columns=1:4, aes(colour=cluster_label), progress=FALSE)

```

```

78
79 library(fossil)
80
81 # plot true labels
82 df_test <- as.data.frame(cbind(pca$x, cw_data$z))
83 colnames(df_test)[[5]] <- "cluster_label"
84 df_test$cluster_label <- as.factor(df_test$cluster_label)
85
86 ggplot(df_test, aes(x=PC1 , y=PC2, color=cluster_label)) +
87   geom_point() + ggtitle('PC2 vs PC1 with true labels')
88
89 df_test <- as.data.frame(cbind(pca$x, km$cluster))
90 colnames(df_test)[[5]] <- "cluster_label"
91 df_test$cluster_label <- as.factor(df_test$cluster_label)
92
93 ggplot(df_test, aes(x=PC1 , y=PC2, color=cluster_label)) +
94   geom_point()+ ggtitle('PC2 vs PC1 with kmeans labels')
95
96 # ran index
97 true_labels <- cw_data$z
98 ri <- rand.index(true_labels, km$cluster)
99
100 print(ri)
101
102 # adj rand index
103 adj.rand.index(true_labels+1, km$cluster)
104 # compare to random
105 # compare z to a random set
106 set.seed(1351308)
107 rand_random = replicate(400, rand.index(true_labels, sample(c(0,1), length(true_labels), replace=TRUE)))
108 hist(rand_random, breaks=50, main='Rand Index for random labels', xlab='Rand Index')
109 abline(v = ri, col='red', lty=2, lwd=3)
110
111 #kernel kmeans
112 library(kernlab)
113 kern_X <- as.matrix(X)
114 kernel_km <- kkmeans(kern_X, centers = 2, kernel = "rbfdot")
115 ri_kernel <- rand.index(true_labels, kernel_km)
116 ri_kernel
117
118 df_test <- as.data.frame(cbind(pca$x, -1*(kernel_km-3)))
119 colnames(df_test)[[5]] <- "cluster_label"
120 df_test$cluster_label <- as.factor(df_test$cluster_label)
121
122 ggplot(df_test, aes(x=PC1 , y=PC2, color=cluster_label)) +
123   geom_point()+ ggtitle('PC2 vs PC1 with kernel kmeans labels')
124
125 adj.rand.index(true_labels+1, kernel_km)

```

---

Listing 3: Code for question 3.2

---

```

1 q3 <- read.csv('train_dataQ3.csv')
2 q3_test <- read.csv('test_dataQ3.csv')
3
4 plot(q3$x, q3$y)
5
6 set.seed(1351308)
7 train_ind <- sample(1:dim(q3)[1], 200)

```

```

8 q3_train <- q3[train_ind,]
9
10 write.csv(q3_train, './q3_train.csv', row.names = FALSE)
11
12 plot(q3_train$x, q3_train$y)
13 plot(q3_test$x, q3_test$y)
14
15 # compute kernel
16 # input x is column
17 kernel <- function(x, p) (x %>% t(x) + 1)^p
18
19 # kernel ridge regression
20 loo_cv <- function(lambda, p, x, y){
21   Num_data_points <- length(y)
22   test_mse <- rep(0, Num_data_points)
23   K <- kernel(x, p)
24
25   for (i in 1:Num_data_points){
26
27     K_i <- K[-i, -i]
28
29     params <- solve(K_i + lambda * diag(dim(K_i)[2]), y[-i])
30
31     test_mse[i] <- (y[i] - sum(K[i, -i]*params))^2
32
33   }
34   return(mean(test_mse))
35 }
36
37 library(latex2exp)
38
39 p_max <- 7
40 kernel_powers <- seq(1, p_max)
41 lambda_vec <- 2^seq(-7, 3, by=0.02)
42
43 av_mse <- matrix(0, nrow=length(kernel_powers), ncol=length(lambda_vec))
44
45 for (p in kernel_powers){
46   av_mse[p,] <- sapply(lambda_vec, function(lambda) loo_cv(lambda, p, q3_train$x,
47     q3_train$y))
48 }
49
50 lambda_vec <- 2^seq(-7, 3, by=0.02)
51 length(lambda_vec)
52
53 plot(lambda_vec, av_mse[1,], log='x', ylim=c(0.01, 0.06), type='l', xlab=TeX('$\\leftarrow$
54   lambda$'), ylab='Average Validation Error', main= TeX('Average Validation Error<
55   vs $\\lambda$'), col=1)
56
57 for (p in 2:p_max){
58   lines(lambda_vec, av_mse[p,], col=p)
59 }
60
61 legend('topright', legend=c(1:p_max), col=c(1:p_max), lty=rep(1, p_max), cex=0.8)
62 #legend('topright', legend=c( TeX('p=1'), TeX('p=2'), TeX('p=3'), TeX('p=4')), col=c(
63   c(1:p_max), lty=rep(1, 4), cex=0.8)
64
65 inds <- which(av_mse == min(av_mse), arr.ind = TRUE)
66 inds
67 lambda_vec[inds[2]]
68
69 plot(lambda_vec, av_mse[5,], ylim=c(0.017, 0.019), log='x', type='l', xlab=TeX('$\\leftarrow$

```



```

        lambda$'), ylab='Average Validation Error', main= TeX('Average Validation Error↔
        vs  $\lambda$ '), col=5)
65 lines(lambda_vec, av_mse[6,], col=6)
66 legend('topright', legend=c('p=5','p=6'), col=c(5,6), lty=rep(1,2), cex=0.8)
67
68
69 #p <- 5
70 #lambda <- 0.007921558
71 p <- inds[1]
72 lambda <- lambda_vec[inds[2]]
73 K <- kernel(q3_train$x,p)
74
75 final_params <- solve(K + lambda * diag(dim(K)[2]), q3_train$y)
76
77 preds_train <- K%% final_params
78
79 # predictions on test set
80 K_star <- (q3_test$x %*% t(q3_train$x) + 1)^p
81 preds_test <- K_star %*% final_params
82
83 #plot fit on train set
84 plot(q3_train$x, q3_train$y, xlab='x', ylab='y', main='Fit on train data')
85 lines(q3_train$x[order(q3_train$x)], preds_train[order(q3_train$x)], col='red')
86 #plot fit on test set
87 plot(q3_test$x, q3_test$y, xlab='x',ylim=c(-0.8,0), ylab='y', main='Fit on test ↔
      data')
88 lines(q3_test$x[order(q3_test$x)], preds_test[order(q3_test$x)], col='red')
89
90
91 #test mse
92 test_mse <- mean((q3_test$y-preds_test)^2)
93 test_mae <- mean(abs(q3_test$y-preds_test))
94
95 test_mse
96 test_mae
97
98 mean((q3_train$y-preds_train)^2)

```

---

Listing 4: Code for question 3.3

---

```

1 import time
2
3
4 from sklearn.metrics import mean_squared_error
5 import tensorflow.compat.v2 as tf
6 import tensorflow_probability as tfp
7 tfb = tfp.bijectors
8 tfd = tfp.distributions
9 tfk = tfp.math.psd_kernels
10 tf.enable_v2_behavior()
11
12 from mpl_toolkits.mplot3d import Axes3D
13 %pylab inline
14 # Configure plot defaults
15 plt.rcParams['axes.facecolor'] = 'white'
16 plt.rcParams['grid.color'] = '#666666'
17 %config InlineBackend.figure_format = 'png'
18
19 # load data
20 q3_train = pd.read_csv("/content/drive/My Drive/ML Coursework 2/q3_train.csv") # ↔

```

```

    sampled already in R
21 q3_test = pd.read_csv("/content/drive/My Drive/ML Coursework 2/test_dataQ3.csv")
22
23 # since 0 mean assumed
24 X_train = np.array(q3_train['x']).reshape(-1, 1)
25 y_train = q3_train['y']
26 mean_y = np.mean(y_train)
27 y_train = y_train - mean_y
28
29 X_test = np.array(q3_test['x']).reshape(-1, 1)
30 y_test = q3_test['y'] - mean_y
31
32 observation_index_points_ = X_train
33 observations_ = y_train
34
35 # Exponential quadratic
36 def build_gp(amplitude, length_scale, observation_noise_variance):
37     """Defines the conditional dist. of GP outputs, given kernel parameters."""
38
39     # Create the covariance kernel, which will be shared between the prior (which we
40     # use for maximum likelihood training) and the posterior (which we use for
41     # posterior predictive sampling)
42     kernel = tfk.ExponentiatedQuadratic(amplitude, length_scale)
43
44     # Create the GP prior distribution, which we will use to train the model
45     # parameters.
46     return tfd.GaussianProcess(
47         kernel=kernel,
48         index_points=observation_index_points_,
49         observation_noise_variance=observation_noise_variance)
50
51 gp_joint_model = tfd.JointDistributionNamed({
52     'amplitude': tfd.LogNormal(loc=0., scale=np.float64(1.)),
53     'length_scale': tfd.LogNormal(loc=0., scale=np.float64(1.)),
54     'observation_noise_variance': tfd.LogNormal(loc=0., scale=np.float64(1.)),
55     'observations': build_gp,
56 })
57 # Create the trainable model parameters, which we'll subsequently optimize.
58 # Note that we constrain them to be strictly positive.
59
60 constrain_positive = tfb.Shift(np.finfo(np.float64).tiny)(tfb.Exp())
61
62 amplitude_var = tfp.util.TransformedVariable(
63     initial_value=1.,
64     bijector=constrain_positive,
65     name='amplitude',
66     dtype=np.float64)
67
68 length_scale_var = tfp.util.TransformedVariable(
69     initial_value=1.,
70     bijector=constrain_positive,
71     name='length_scale',
72     dtype=np.float64)
73
74 observation_noise_variance_var = tfp.util.TransformedVariable(
75     initial_value=1.,
76     bijector=constrain_positive,
77     name='observation_noise_variance_var',
78     dtype=np.float64)
79
80 trainable_variables = [v.trainable_variables[0] for v in

```

```

81         [amplitude_var,
82           length_scale_var,
83           observation_noise_variance_var]]
84
85 # Now we optimize the model parameters.
86 num_iters = 1000
87 optimizer = tf.optimizers.Adam(learning_rate=.01)
88
89 # Store the likelihood values during training, so we can plot the progress
90 lls_ = np.zeros(num_iters, np.float64)
91 for i in range(num_iters):
92     with tf.GradientTape() as tape:
93         #tape.watch(trainable_variables)
94         loss = -gp_joint_model.log_prob({
95             'amplitude': amplitude_var,
96             'length_scale': length_scale_var,
97             'observation_noise_variance': observation_noise_variance_var,
98             'observations': observations_
99         })
100     grads = tape.gradient(loss, trainable_variables)
101     optimizer.apply_gradients(zip(grads, trainable_variables))
102     lls_[i] = loss
103
104 print('Trained parameters:')
105 print('amplitude: {}'.format(amplitude_var._value().numpy()))
106 print('length_scale: {}'.format(length_scale_var._value().numpy()))
107 print('observation_noise_variance: {}'.format(observation_noise_variance_var._value()↵
108         ().numpy()))
109 print('marginal log likelihood: {}'.format(lls_[-1]))
110
111 # plot to check convergence
112 plt.figure(figsize=(12, 4))
113 plt.plot(lls_)
114 plt.xlabel("Training iteration")
115 plt.ylabel("Log marginal likelihood")
116 plt.show()
117
118 predictive_index_points_ = np.linspace(-5, 5, 400, dtype=np.float64)
119 predictive_index_points_ = predictive_index_points_[..., np.newaxis]
120
121 optimized_kernel = tfk.ExponentiatedQuadratic(amplitude_var, length_scale_var)
122 gprm = tfd.GaussianProcessRegressionModel(
123     kernel=optimized_kernel,
124     index_points=predictive_index_points_,
125     observation_index_points=observation_index_points_,
126     observations=observations_,
127     observation_noise_variance=observation_noise_variance_var,
128     predictive_noise_variance=0.)
129
130
131 expq_mean = gprm.mean()
132 expq_std = gprm.stddev()
133
134 plt.figure(figsize=(12, 6))
135 plt.scatter(observation_index_points_[ :, 0], observations_,
136             label='Observations')
137 plt.plot(predictive_index_points_, expq_mean, c='r',
138           label='Posterior Sample')
139 plt.fill(np.concatenate([predictive_index_points_, predictive_index_points_[:, -1]]),↵
140         ,

```

```

140         np.concatenate([expq_mean - 1.9600 * expq_std,
141                         (expq_mean + 1.9600 * expq_std)[::-1]]),
142         alpha=.5, fc='b', ec='None', label='95% confidence interval')
143 plt.xlabel(r"Index points ( $\mathbb{R}^1$ )")
144 plt.ylabel("Observation space")
145 plt.title('Square Exponential Kernel')
146 plt.legend(loc='upper right')
147 plt.show()
148
149 predictive_index_points_ = observation_index_points_
150 # Reshape to [200, 1] -- 1 is the dimensionality of the feature space.
151 predictive_index_points_ = predictive_index_points_[..., np.newaxis]
152
153 optimized_kernel = tfk.ExponentiatedQuadratic(amplitude_var, length_scale_var)
154 gprm = tfd.GaussianProcessRegressionModel(
155     kernel=optimized_kernel,
156     index_points=predictive_index_points_,
157     observation_index_points=observation_index_points_,
158     observations=observations_,
159     observation_noise_variance=observation_noise_variance_var,
160     predictive_noise_variance=0.)
161 print('train MSE: {}'.format(mean_squared_error(y_train, gprm.mean())))
162 gprm = tfd.GaussianProcessRegressionModel(
163     kernel=optimized_kernel,
164     index_points=X_test,
165     observation_index_points=observation_index_points_,
166     observations=observations_,
167     observation_noise_variance=observation_noise_variance_var,
168     predictive_noise_variance=0.)
169 print('test MSE: {}'.format(mean_squared_error(y_test, gprm.mean())))
170
171 # Rational Quadratic
172 def build_gp(amplitude, length_scale, observation_noise_variance, scale_mixture_rate):
173     """Defines the conditional dist. of GP outputs, given kernel parameters."""
174
175     # Create the covariance kernel, which will be shared between the prior (which we
176     # use for maximum likelihood training) and the posterior (which we use for
177     # posterior predictive sampling)
178     kernel = tfk.RationalQuadratic(amplitude, length_scale, scale_mixture_rate)
179
180     # Create the GP prior distribution, which we will use to train the model
181     # parameters.
182     return tfd.GaussianProcess(
183         kernel=kernel,
184         index_points=observation_index_points_,
185         observation_noise_variance=observation_noise_variance)
186
187 gp_joint_model = tfd.JointDistributionNamed({
188     'amplitude': tfd.LogNormal(loc=0., scale=np.float64(1.)),
189     'length_scale': tfd.LogNormal(loc=0., scale=np.float64(1.)),
190     'scale_mixture_rate': tfd.LogNormal(loc=0., scale=np.float64(1.)),
191     'observation_noise_variance': tfd.LogNormal(loc=0., scale=np.float64(1.)),
192     'observations': build_gp,
193 })
194
195 # Create the trainable model parameters, which we'll subsequently optimize.
196 # Note that we constrain them to be strictly positive.
197
198 constrain_positive = tfb.Shift(np.finfo(np.float64).tiny)(tfb.Exp())
199

```

```

200 amplitude_var = tfp.util.TransformedVariable(
201     initial_value=1.,
202     bijector=constrain_positive,
203     name='amplitude',
204     dtype=np.float64)
205
206 length_scale_var = tfp.util.TransformedVariable(
207     initial_value=1.,
208     bijector=constrain_positive,
209     name='length_scale',
210     dtype=np.float64)
211
212 scale_mixture_rate_var = tfp.util.TransformedVariable(
213     initial_value=1.,
214     bijector=constrain_positive,
215     name='scale_mixture_rate',
216     dtype=np.float64)
217
218 observation_noise_variance_var = tfp.util.TransformedVariable(
219     initial_value=1.,
220     bijector=constrain_positive,
221     name='observation_noise_variance_var',
222     dtype=np.float64)
223
224 trainable_variables = [v.trainable_variables[0] for v in
225     [amplitude_var,
226     length_scale_var,
227     observation_noise_variance_var,
228     scale_mixture_rate_var]]
229
230 # Now we optimize the model parameters.
231 num_iters = 1000
232 optimizer = tf.optimizers.Adam(learning_rate=.01)
233
234 # Store the likelihood values during training, so we can plot the progress
235 lls_ = np.zeros(num_iters, np.float64)
236 for i in range(num_iters):
237     with tf.GradientTape() as tape:
238         #tape.watch(trainable_variables)
239         loss = -gp_joint_model.log_prob({
240             'amplitude': amplitude_var,
241             'length_scale': length_scale_var,
242             'scale_mixture_rate': scale_mixture_rate_var,
243             'observation_noise_variance': observation_noise_variance_var,
244             'observations': observations_
245         })
246     grads = tape.gradient(loss, trainable_variables)
247     optimizer.apply_gradients(zip(grads, trainable_variables))
248     lls_[i] = loss
249
250 print('Trained parameters:')
251 print('amplitude: {}'.format(amplitude_var._value().numpy()))
252 print('length_scale: {}'.format(length_scale_var._value().numpy()))
253 print('scale_mixture_rate: {}'.format(scale_mixture_rate_var._value().numpy()))
254 print('observation_noise_variance: {}'.format(observation_noise_variance_var._value↵
255     ().numpy()))
256
257 print('marginal log likelihood: {}'.format(lls_[-1]))
258
259 predictive_index_points_ = np.linspace(-5, 5, 400, dtype=np.float64)
260 # Reshape to [200, 1] -- 1 is the dimensionality of the feature space.
261 predictive_index_points_ = predictive_index_points_[..., np.newaxis]

```

```

260
261 optimized_kernel = tfk.RationalQuadratic(amplitude_var, length_scale_var, scale_←
    mixture_rate_var)
262 gprm = tfd.GaussianProcessRegressionModel(
263     kernel=optimized_kernel,
264     index_points=predictive_index_points_,
265     observation_index_points=observation_index_points_,
266     observations=observations_,
267     observation_noise_variance=observation_noise_variance_var,
268     predictive_noise_variance=0.)
269
270 expq_mean = gprm.mean()
271 expq_std = gprm.stddev()
272
273 plt.figure(figsize=(12, 6))
274 plt.scatter(observation_index_points_[ :, 0], observations_,
275             label='Observations')
276 plt.plot(predictive_index_points_, expq_mean, c='r',
277           label='Posterior Sample')
278 plt.fill(np.concatenate([predictive_index_points_, predictive_index_points_[:, -1]]),←
    ,
279          np.concatenate([expq_mean - 1.9600 * expq_std,
280                          (expq_mean + 1.9600 * expq_std)[:, -1]]),
281          alpha=.5, fc='b', ec='None', label='95% confidence interval')
282 plt.xlabel(r"Index points ( $\mathbb{R}^1$ )")
283 plt.ylabel("Observation space")
284 plt.title('Rational Quadratic Kernel')
285 plt.legend(loc='upper right')
286 plt.show()
287
288 # plot fit on test set
289 predictive_index_points_ = np.linspace(-5, 5, 400, dtype=np.float64)
290 # Reshape to [200, 1] -- 1 is the dimensionality of the feature space.
291 predictive_index_points_ = predictive_index_points_[..., np.newaxis]
292
293 optimized_kernel = tfk.RationalQuadratic(amplitude_var, length_scale_var, scale_←
    mixture_rate_var)
294 gprm = tfd.GaussianProcessRegressionModel(
295     kernel=optimized_kernel,
296     index_points=predictive_index_points_,
297     observation_index_points=observation_index_points_,
298     observations=observations_,
299     observation_noise_variance=observation_noise_variance_var,
300     predictive_noise_variance=0.)
301
302 expq_mean = gprm.mean()
303 expq_std = gprm.stddev()
304
305 plt.figure(figsize=(12, 6))
306 plt.scatter(observation_index_points_[ :, 0], observations_,
307             label='Observations')
308 plt.plot(predictive_index_points_, expq_mean, c='r',
309           label='Posterior Sample')
310 plt.fill(np.concatenate([predictive_index_points_, predictive_index_points_[:, -1]]),←
    ,
311          np.concatenate([expq_mean - 1.9600 * expq_std,
312                          (expq_mean + 1.9600 * expq_std)[:, -1]]),
313          alpha=.5, fc='b', ec='None', label='95% confidence interval')
314 plt.xlabel(r"Index points ( $\mathbb{R}^1$ )")
315 plt.ylabel("Observation space")
316 plt.title('Rational Quadratic Kernel')

```

```

317 plt.legend(loc='upper right')
318 plt.show()
319
320 predictive_index_points_ = observation_index_points_
321 # Reshape to [200, 1] -- 1 is the dimensionality of the feature space.
322 predictive_index_points_ = predictive_index_points_[..., np.newaxis]
323
324 gprm = tfd.GaussianProcessRegressionModel(
325     kernel=optimized_kernel,
326     index_points=predictive_index_points_,
327     observation_index_points=observation_index_points_,
328     observations=observations_,
329     observation_noise_variance=observation_noise_variance_var,
330     predictive_noise_variance=0.)
331 print('train MSE: {}'.format(mean_squared_error(y_train, gprm.mean()))))
332 gprm = tfd.GaussianProcessRegressionModel(
333     kernel=optimized_kernel,
334     index_points=X_test,
335     observation_index_points=observation_index_points_,
336     observations=observations_,
337     observation_noise_variance=observation_noise_variance_var,
338     predictive_noise_variance=0.)
339 print('test MSE: {}'.format(mean_squared_error(y_test, gprm.mean()))))
340
341 def build_gp(amplitude, length_scale, observation_noise_variance, period):
342     """Defines the conditional dist. of GP outputs, given kernel parameters."""
343
344     # Create the covariance kernel, which will be shared between the prior (which we
345     # use for maximum likelihood training) and the posterior (which we use for
346     # posterior predictive sampling)
347     kernel = tfk.ExpSinSquared(
348         amplitude, length_scale, period)
349
350     # Create the GP prior distribution, which we will use to train the model
351     # parameters.
352     return tfd.GaussianProcess(
353         kernel=kernel,
354         index_points=observation_index_points_,
355         observation_noise_variance=observation_noise_variance)
356
357 gp_joint_model = tfd.JointDistributionNamed({
358     'amplitude': tfd.LogNormal(loc=0., scale=np.float64(1.)),
359     'length_scale': tfd.LogNormal(loc=0., scale=np.float64(1.)),
360     'period': tfd.LogNormal(loc=0., scale=np.float64(1.)),
361     'observation_noise_variance': tfd.LogNormal(loc=0., scale=np.float64(1.)),
362     'observations': build_gp,
363 })
364
365 # Create the trainable model parameters, which we'll subsequently optimize.
366 # Note that we constrain them to be strictly positive.
367
368 constrain_positive = tfb.Shift(np.finfo(np.float64).tiny)(tfb.Exp())
369
370 amplitude_var = tfp.util.TransformedVariable(
371     initial_value=1.,
372     bijector=constrain_positive,
373     name='amplitude',
374     dtype=np.float64)
375
376 length_scale_var = tfp.util.TransformedVariable(
377     initial_value=1.,

```

```

378     bijector=constrain_positive,
379     name='length_scale',
380     dtype=np.float64)
381
382     period_var = tfp.util.TransformedVariable(
383         initial_value=10.,
384         bijector=constrain_positive,
385         name='period',
386         dtype=np.float64)
387
388     observation_noise_variance_var = tfp.util.TransformedVariable(
389         initial_value=1.,
390         bijector=constrain_positive,
391         name='observation_noise_variance_var',
392         dtype=np.float64)
393
394     trainable_variables = [v.trainable_variables[0] for v in
395                             [amplitude_var,
396                              length_scale_var,
397                              period_var,
398                              observation_noise_variance_var]]
399
400     # Now we optimize the model parameters.
401     num_iters = 1000
402     optimizer = tf.optimizers.Adam(learning_rate=.01)
403
404     # Store the likelihood values during training, so we can plot the progress
405     lls_ = np.zeros(num_iters, np.float64)
406     for i in range(num_iters):
407         with tf.GradientTape() as tape:
408             #tape.watch(trainable_variables)
409             loss = -gp_joint_model.log_prob({
410                 'amplitude': amplitude_var,
411                 'length_scale': length_scale_var,
412                 'period': period_var,
413                 'observation_noise_variance': observation_noise_variance_var,
414                 'observations': observations_
415             })
416             grads = tape.gradient(loss, trainable_variables)
417             optimizer.apply_gradients(zip(grads, trainable_variables))
418             lls_[i] = loss
419
420     print('Trained parameters:')
421     print('amplitude: {}'.format(amplitude_var._value().numpy()))
422     print('length_scale: {}'.format(length_scale_var._value().numpy()))
423     print('period: {}'.format(period_var._value().numpy()))
424     print('observation_noise_variance: {}'.format(observation_noise_variance_var._value()↵
425             ().numpy()))
426
427     print('marginal log likelihood: {}'.format(lls_[-1]))
428
429     predictive_index_points_ = np.linspace(-5, 5, 400, dtype=np.float64)
430     # Reshape to [200, 1] -- 1 is the dimensionality of the feature space.
431     predictive_index_points_ = predictive_index_points_[..., np.newaxis]
432
433     optimized_kernel = tfk.ExpSinSquared(amplitude_var, length_scale_var, period_var)
434     gprm = tfd.GaussianProcessRegressionModel(
435         kernel=optimized_kernel,
436         index_points=predictive_index_points_,
437         observation_index_points=observation_index_points_,
438         observations=observations_,
439         observation_noise_variance=observation_noise_variance_var,

```



```

438     predictive_noise_variance=0.)
439
440 expq_mean = gprm.mean()
441 expq_std = gprm.stddev()
442
443 plt.figure(figsize=(12, 6))
444 plt.scatter(observation_index_points[:, 0], observations_,
445             label='Observations')
446 plt.plot(predictive_index_points_, expq_mean, c='r',
447           label='Posterior Sample')
448 plt.fill(np.concatenate([predictive_index_points_, predictive_index_points_[:, -1]]), ←
449          ,
450          np.concatenate([expq_mean - 1.9600 * expq_std,
451                          (expq_mean + 1.9600 * expq_std)[:, -1]]),
452          alpha=.5, fc='b', ec='None', label='95% confidence interval')
453 plt.xlabel(r"Index points ( $\mathbb{R}^1$ )")
454 plt.ylabel("Observation space")
455 plt.title('Square Exponential Kernel')
456 plt.legend(loc='upper right')
457 plt.show()
458
459 predictive_index_points_ = observation_index_points_
460 # Reshape to [200, 1] -- 1 is the dimensionality of the feature space.
461 predictive_index_points_ = predictive_index_points_[..., np.newaxis]
462 gprm = tfd.GaussianProcessRegressionModel(
463     kernel=optimized_kernel,
464     index_points=predictive_index_points_,
465     observation_index_points=observation_index_points_,
466     observations=observations_,
467     observation_noise_variance=observation_noise_variance_var,
468     predictive_noise_variance=0.)
469 print('train MSE: {}'.format(mean_squared_error(y_train, gprm.mean())))
470 gprm = tfd.GaussianProcessRegressionModel(
471     kernel=optimized_kernel,
472     index_points=X_test,
473     observation_index_points=observation_index_points_,
474     observations=observations_,
475     observation_noise_variance=observation_noise_variance_var,
476     predictive_noise_variance=0.)
477 print('test MSE: {}'.format(mean_squared_error(y_test, gprm.mean())))
478
479 ##### cross validation
480 from sklearn.model_selection import KFold
481 from sklearn.metrics import mean_squared_error
482
483 def cv(n_folds, X, y):
484     j = 0
485     mse_per = np.zeros(n_folds)
486     mse_rbf = np.zeros(n_folds)
487     mse_rq = np.zeros(n_folds)
488
489     kf = KFold(n_splits=n_folds, shuffle=True)
490
491     for train_index, test_index in kf.split(X, y):
492         X_cv_train, X_cv_test = X[train_index:], X[test_index:]
493         y_cv_train, y_cv_test = y[train_index], y[test_index]
494         X_cv_train = np.array(X_cv_train).reshape(-1, 1)
495         mean_y = np.mean(y_cv_train)
496         y_cv_train = y_cv_train - mean_y

```

```

498
499 X_cv_test = np.array(X_cv_test).reshape(-1, 1)
500 y_cv_test = y_cv_test - mean_y
501
502 # initialise kernels
503 def build_gp(amplitude, length_scale, observation_noise_variance, period):
504     """Defines the conditional dist. of GP outputs, given kernel parameters."""
505
506     # Create the covariance kernel, which will be shared between the prior (which↔
507     # we
508     # use for maximum likelihood training) and the posterior (which we use for
509     # posterior predictive sampling)
510     kernel = tfk.ExpSinSquared(
511         amplitude, length_scale, period)
512
513     # Create the GP prior distribution, which we will use to train the model
514     # parameters.
515     return tfd.GaussianProcess(
516         kernel=kernel,
517         index_points=X_cv_train,
518         observation_noise_variance=observation_noise_variance)
519
520 gp_joint_model = tfd.JointDistributionNamed({
521     'amplitude': tfd.LogNormal(loc=0., scale=np.float64(1.)),
522     'length_scale': tfd.LogNormal(loc=0., scale=np.float64(1.)),
523     'period': tfd.LogNormal(loc=0., scale=np.float64(1.)),
524     'observation_noise_variance': tfd.LogNormal(loc=0., scale=np.float64(1.)),
525     'observations': build_gp,
526 })
527
528 # Create the trainable model parameters, which we'll subsequently optimize.
529 # Note that we constrain them to be strictly positive.
530
531 constrain_positive = tfb.Shift(np.finfo(np.float64).tiny)(tfb.Exp())
532
533 amplitude_var = tfp.util.TransformedVariable(
534     initial_value=1.,
535     bijector=constrain_positive,
536     name='amplitude',
537     dtype=np.float64)
538
539 length_scale_var = tfp.util.TransformedVariable(
540     initial_value=1.,
541     bijector=constrain_positive,
542     name='length_scale',
543     dtype=np.float64)
544
545 period_var = tfp.util.TransformedVariable(
546     initial_value=10.,
547     bijector=constrain_positive,
548     name='period',
549     dtype=np.float64)
550
551 observation_noise_variance_var = tfp.util.TransformedVariable(
552     initial_value=1.,
553     bijector=constrain_positive,
554     name='observation_noise_variance_var',
555     dtype=np.float64)
556
557 trainable_variables = [v.trainable_variables[0] for v in
558     [amplitude_var,

```

```

558         length_scale_var,
559         period_var,
560         observation_noise_variance_var]]
561
562     # Now we optimize the model parameters.
563     num_iters = 750
564     optimizer = tf.optimizers.Adam(learning_rate=.01)
565
566     # Store the likelihood values during training, so we can plot the progress
567     lls_ = np.zeros(num_iters, np.float64)
568     for i in range(num_iters):
569         with tf.GradientTape() as tape:
570             #tape.watch(trainable_variables)
571             loss = -gp_joint_model.log_prob({
572                 'amplitude': amplitude_var,
573                 'length_scale': length_scale_var,
574                 'period': period_var,
575                 'observation_noise_variance': observation_noise_variance_var,
576                 'observations': y_cv_train
577             })
578             grads = tape.gradient(loss, trainable_variables)
579             optimizer.apply_gradients(zip(grads, trainable_variables))
580             lls_[i] = loss
581
582     optimized_kernel = tfk.ExpSinSquared(amplitude_var, length_scale_var, period_↵
583                                         var)
584     gprm = tfd.GaussianProcessRegressionModel(
585         kernel=optimized_kernel,
586         index_points=X_cv_test,
587         observation_index_points=X_cv_train,
588         observations=y_cv_train,
589         observation_noise_variance=observation_noise_variance_var,
590         predictive_noise_variance=0.)
591     mse_per[j] = mean_squared_error(y_cv_test, gprm.mean())
592
593     #RQ kernel
594
595     def build_gp(amplitude, length_scale, observation_noise_variance, scale_mixture_↵
596                 _rate):
597         """Defines the conditional dist. of GP outputs, given kernel parameters."""
598
599         # Create the covariance kernel, which will be shared between the prior (which↵
600         # we
601         # use for maximum likelihood training) and the posterior (which we use for
602         # posterior predictive sampling)
603         kernel = tfk.RationalQuadratic(amplitude, length_scale, scale_mixture_rate)
604
605         # Create the GP prior distribution, which we will use to train the model
606         # parameters.
607         return tfd.GaussianProcess(
608             kernel=kernel,
609             index_points=X_cv_train,
610             observation_noise_variance=observation_noise_variance)
611
612     gp_joint_model = tfd.JointDistributionNamed({
613         'amplitude': tfd.LogNormal(loc=0., scale=np.float64(1.)),
614         'length_scale': tfd.LogNormal(loc=0., scale=np.float64(1.)),
615         'scale_mixture_rate': tfd.LogNormal(loc=0., scale=np.float64(1.)),
616         'observation_noise_variance': tfd.LogNormal(loc=0., scale=np.float64(1.)),
617         'observations': build_gp,

```

```

616     })
617
618     # Create the trainable model parameters, which we'll subsequently optimize.
619     # Note that we constrain them to be strictly positive.
620
621     constrain_positive = tfb.Shift(np.finfo(np.float64).tiny)(tfb.Exp())
622
623     amplitude_var = tfp.util.TransformedVariable(
624         initial_value=1.,
625         bijector=constrain_positive,
626         name='amplitude',
627         dtype=np.float64)
628
629     length_scale_var = tfp.util.TransformedVariable(
630         initial_value=1.,
631         bijector=constrain_positive,
632         name='length_scale',
633         dtype=np.float64)
634
635     scale_mixture_rate_var = tfp.util.TransformedVariable(
636         initial_value=1.,
637         bijector=constrain_positive,
638         name='scale_mixture_rate',
639         dtype=np.float64)
640
641     observation_noise_variance_var = tfp.util.TransformedVariable(
642         initial_value=1.,
643         bijector=constrain_positive,
644         name='observation_noise_variance_var',
645         dtype=np.float64)
646
647     trainable_variables = [v.trainable_variables[0] for v in
648                             [amplitude_var,
649                              length_scale_var,
650                              observation_noise_variance_var,
651                              scale_mixture_rate_var]]
652     # Now we optimize the model parameters.
653     num_iters = 750
654     optimizer = tf.optimizers.Adam(learning_rate=.01)
655
656     # Store the likelihood values during training, so we can plot the progress
657     lls_ = np.zeros(num_iters, np.float64)
658     for i in range(num_iters):
659         with tf.GradientTape() as tape:
660             #tape.watch(trainable_variables)
661             loss = -gp_joint_model.log_prob({
662                 'amplitude': amplitude_var,
663                 'length_scale': length_scale_var,
664                 'scale_mixture_rate': scale_mixture_rate_var,
665                 'observation_noise_variance': observation_noise_variance_var,
666                 'observations': y_cv_train
667             })
668             grads = tape.gradient(loss, trainable_variables)
669             optimizer.apply_gradients(zip(grads, trainable_variables))
670             lls_[i] = loss
671
672     optimized_kernel = tfk.RationalQuadratic(amplitude_var, length_scale_var, scale↵
673                                               _mixture_rate_var)
674     gprm = tfd.GaussianProcessRegressionModel(
675         kernel=optimized_kernel,
676         index_points=X_cv_test,

```

```

676     observation_index_points=X_cv_train,
677     observations=y_cv_train,
678     observation_noise_variance=observation_noise_variance_var,
679     predictive_noise_variance=0.)
680 mse_rq[j] = mean_squared_error(y_cv_test, gprm.mean())
681
682 def build_gp(amplitude, length_scale, observation_noise_variance):
683     """Defines the conditional dist. of GP outputs, given kernel parameters."""
684
685     # Create the covariance kernel, which will be shared between the prior (which↔
686     # we
687     # use for maximum likelihood training) and the posterior (which we use for
688     # posterior predictive sampling)
689     kernel = tfk.ExponentiatedQuadratic(amplitude, length_scale)
690
691     # Create the GP prior distribution, which we will use to train the model
692     # parameters.
693     return tfd.GaussianProcess(
694         kernel=kernel,
695         index_points=X_cv_train,
696         observation_noise_variance=observation_noise_variance)
697
698 gp_joint_model = tfd.JointDistributionNamed({
699     'amplitude': tfd.LogNormal(loc=0., scale=np.float64(1.)),
700     'length_scale': tfd.LogNormal(loc=0., scale=np.float64(1.)),
701     'observation_noise_variance': tfd.LogNormal(loc=0., scale=np.float64(1.)),
702     'observations': build_gp,
703 })
704
705 # Create the trainable model parameters, which we'll subsequently optimize.
706 # Note that we constrain them to be strictly positive.
707
708 constrain_positive = tfb.Shift(np.finfo(np.float64).tiny)(tfb.Exp())
709
710 amplitude_var = tfp.util.TransformedVariable(
711     initial_value=1.,
712     bijector=constrain_positive,
713     name='amplitude',
714     dtype=np.float64)
715
716 length_scale_var = tfp.util.TransformedVariable(
717     initial_value=1.,
718     bijector=constrain_positive,
719     name='length_scale',
720     dtype=np.float64)
721
722 observation_noise_variance_var = tfp.util.TransformedVariable(
723     initial_value=1.,
724     bijector=constrain_positive,
725     name='observation_noise_variance_var',
726     dtype=np.float64)
727
728 trainable_variables = [v.trainable_variables[0] for v in
729     [amplitude_var,
730     length_scale_var,
731     observation_noise_variance_var]]
732
733 # Now we optimize the model parameters.
734 num_iters = 750
735 optimizer = tf.optimizers.Adam(learning_rate=.01)

```

```

736
737 # Store the likelihood values during training, so we can plot the progress
738 lls_ = np.zeros(num_iters, np.float64)
739 for i in range(num_iters):
740     with tf.GradientTape() as tape:
741         #tape.watch(trainable_variables)
742         loss = -gp_joint_model.log_prob({
743             'amplitude': amplitude_var,
744             'length_scale': length_scale_var,
745             'observation_noise_variance': observation_noise_variance_var,
746             'observations': y_cv_train
747         })
748         grads = tape.gradient(loss, trainable_variables)
749         optimizer.apply_gradients(zip(grads, trainable_variables))
750         lls_[i] = loss
751     optimized_kernel = tfk.ExponentiatedQuadratic(amplitude_var, length_scale_var)
752
753     gprm = tfd.GaussianProcessRegressionModel(
754         kernel=optimized_kernel,
755         index_points=X_cv_test,
756         observation_index_points=X_cv_train,
757         observations=y_cv_train,
758         observation_noise_variance=observation_noise_variance_var,
759         predictive_noise_variance=0.)
760     mse_rbf[j] = mean_squared_error(y_cv_test, gprm.mean())
761
762     j += 1
763 # average all test mse's
764 av_per = np.mean(mse_per)
765 av_rbf = np.mean(mse_rbf)
766 av_rq = np.mean(mse_rq)
767 return (av_per, av_rbf, av_rq)
768
769 av_cv = cv(4, X_train, y_train)

```

---