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

The solution to the regression problem at hand is given by

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

where the kernel matrix is defined as follows

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

To optimise the hyper-parameters p and λ 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 $p \in \{1, 2, \ldots, 7\}$ and $\lambda \in \left\{2^{-7+2k/100} \text{ for } k = 0, 1, 2, \ldots 500\right\}$. 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.

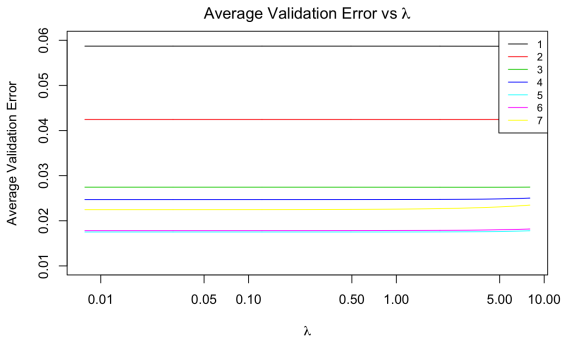

Figure 1: Average validation error vs λ

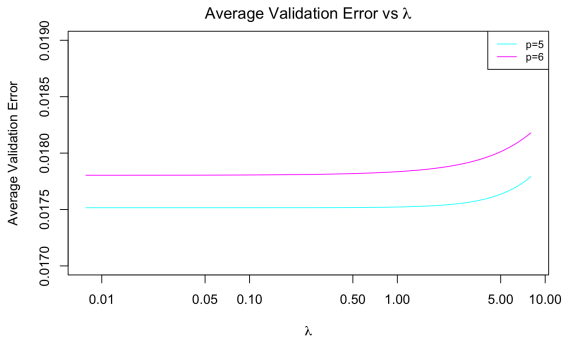

Figure 2: Average validation error vs λ zoomed in.

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

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

where $\mathbf{K}\left(x^*, \mathbf{x}\right)$ is given by $\mathbf{K}\left(x^*, \mathbf{x}\right) = \left[K\left(x^*, x^{(1)}\right), K\left(x^*, x^{(2)}\right), \ldots, K\left(x^*, x^{(N)}\right)\right]$ 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.

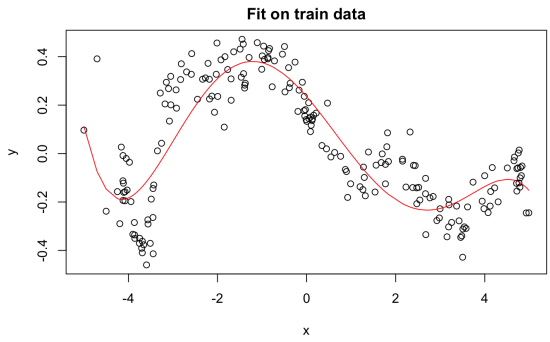

Figure 3: Fit to train set.

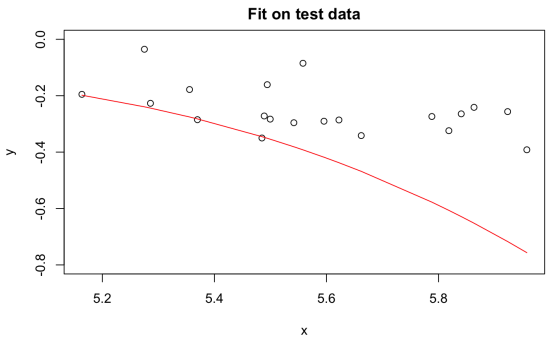

Figure 4: Fit to test set.

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 ith direction is defined as

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

when the limit exists, where l.i.m denotes the limit in mean square and e_i is the unit vector in the ith direction.

RBF kernel

The RBF kernel also called Square Exponential kernel is defined by

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

Here the parameter $\ell > 0$ is the characteristic length-scale. The parameter ℓ determines the length of the 'wiggles' in the function as it scales inside the exponential. Increasing ℓ will elongate the 'wiggles'. The second parameter is σ which is part of all kernels considered. It determines how far the function will vary from the mean, i.e. increasing σ 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 $\left(x-x'\right)^2$ 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 'wiggle'[1].

RQ kernel

The RQ kernel is defined by

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

with α, $\ell > 0$. 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 α can be chosen. With a larger α the covariance decays quicker as the input distance increases until for $\alpha \to \infty$ the kernel tends to the RBF kernel. The parameters σ 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 ℓ 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 hyperparameters are optimised by maximising the marginal likelihood $p(\mathbf{y} \mid X, \theta)$, i.e.

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

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

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

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

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

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.

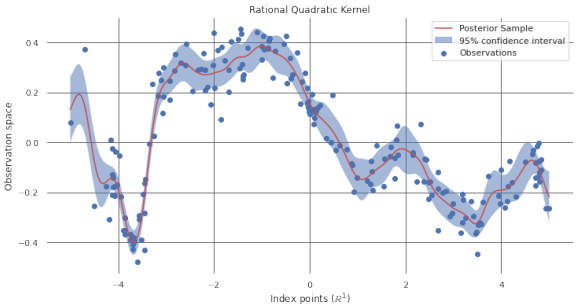

Figure 5: Fit to train set.

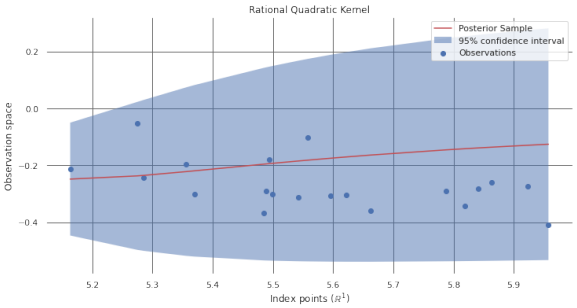

Figure 6: Fit to test set.

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

```
from google.colab import drive
drive.mount("/content/drive")

# basic libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# sklearn
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.metrics import accuracy_score
from sklearn.metrics.cluster import adjusted_rand_score
from sklearn.metrics.cluster import adjusted_mutual_info_score
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier

%matplotlib inline



import warnings
warnings.filterwarnings("ignore")

# sklearn supervised learning
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import StratifiedKFold

# make plots nicer
sns.set()

# load data
q1_data = pd.read_csv("/content/drive/My Drive/ML Coursework 2/CW2Data.csv")
pd.DataFrame.head(q1_data)

# overview
q1_data.describe()
q1_data.corr()

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score

X = q1_data.iloc[:, 1:]
y = q1_data['z']


# split data into predictors and target
from sklearn.model_selection import StratifiedShuffleSplit

sss = StratifiedShuffleSplit(n_splits=1, test_size=0.5, random_state=0)
# split data into train and test set
for train_index, test_index in sss.split(X, y):
      X_train, X_test = X.iloc[train_index,:], X.iloc[test_index,:]
      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',
151                          name='learning_rate')
152 dim_dropout_rate = Real(low=1e-2, high=0.5, prior='log-uniform',
153                          name='dropout_rate')
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'],
157                              name='activation')
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
```

```
train_dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train))
val_dataset = tf.data.Dataset.from_tensor_slices((X_val, y_val)) # should be validation data

train_dataset = train_dataset.shuffle(353)

train_dataset = train_dataset.batch(89)
val_dataset = val_dataset.batch(len(y_val)) # just one batch for validation

train_dataset = train_dataset.prefetch(tf.data.experimental.AUTOTUNE)

def log_dir_name(learning_rate, num_dense_layers,
                 num_dense_nodes, activation, dropout_rate):

    # The dir-name for the TensorBoard log-dir.
    s = "./19_logs/lr_{0:.0e}_layers_{1}_nodes_{2}_{3}_dr_{0:.0e}/"

    # Insert all the hyper-parameters in the dir-name.
    log_dir = s.format(learning_rate,
                       num_dense_layers,
                       num_dense_nodes,
                       activation,
                       dropout_rate)

    return log_dir


l2_coeff = 1e-5


def create_model(learning_rate, num_dense_layers,
                 num_dense_nodes, activation, dropout_rate):
    """
    Hyper-parameters:
    learning_rate:     Learning-rate for the optimizer.
    num_dense_layers:  Number of dense layers.
    num_dense_nodes:   Number of nodes in each dense layer.
    activation:        Activation function for all layers.
    """

    # Start construction of a Keras Sequential model.
    model = Sequential()

    # Add an input layer which is similar to a feed_dict in TensorFlow.
    # Note that the input-shape must be a tuple containing the image-size.

    model.add(InputLayer(input_shape = (X_train.shape[-1],)))# adjust this
    #model.add(Flatten()) #think don't need this

    # Add fully-connected / dense layers.
    # The number of layers is a hyper-parameter we want to optimize.
    for i in range(num_dense_layers):


        # Name of the layer. This is not really necessary
        # because Keras should give them unique names.
        name = 'layer_dense_{0}'.format(i+1)

        # Add the dense / fully-connected layer to the model.
        # This has two hyper-parameters we want to optimize:
        # The number of nodes and the activation function.
```

```
        model.add(Dense(num_dense_nodes,
                        activation=activation,
                        kernel_regularizer=regularizers.l2(l2_coeff),
                        name=name))
        # Add Dropout Layer
        name = 'layer_dropout_{0}'.format(i+1)
        model.add(Dropout(dropout_rate, name=name))



    # Last fully-connected / dense layer with sigmoid-activation
    # for use in classification.
    model.add(Dense(1, activation='sigmoid')) # 2 classes

    # Use the Adam method for training the network.
    # We want to find the best learning-rate for the Adam method.
    optimizer = Adam(lr=learning_rate)

    # In Keras we need to compile the model so it can be trained.
    model.compile(optimizer=optimizer,
                  loss='binary_crossentropy',
                  metrics=['binary_accuracy'])

    return model


path_best_model = '19_best_model.h5'
best_accuracy = 0.0
@use_named_args(dimensions=dimensions)

def fitness(learning_rate, num_dense_layers,
            num_dense_nodes, activation, dropout_rate):
    """
    Hyper-parameters:
    learning_rate:     Learning-rate for the optimizer.
    num_dense_layers:  Number of dense layers.
    num_dense_nodes:   Number of nodes in each dense layer.
    activation:        Activation function for all layers.
    """

    # Print the hyper-parameters.
    print('learning rate: {0:.1e}'.format(learning_rate))
    print('num_dense_layers:', num_dense_layers)
    print('num_dense_nodes:', num_dense_nodes)
    print('activation:', activation)
    print('dropout rate: {0:.1e}'.format(dropout_rate))
    print()

    # Create the neural network with these hyper-parameters.
    model = create_model(learning_rate=learning_rate,
                         num_dense_layers=num_dense_layers,
                         num_dense_nodes=num_dense_nodes,
                         activation=activation, dropout_rate=dropout_rate)

    # Dir-name for the TensorBoard log-files.
    log_dir = log_dir_name(learning_rate, num_dense_layers,
                           num_dense_nodes, activation, dropout_rate)

    # Create a callback-function for Keras which will be
    # run after each epoch has ended during training.
    # This saves the log-files for TensorBoard.
```

```
    # Note that there are complications when histogram_freq=1.
    # It might give strange errors and it also does not properly
    # support Keras data-generators for the validation-set.
    callback_log = TensorBoard(
        log_dir=log_dir,
        histogram_freq=0,
        write_graph=True,
        write_grads=False,
        write_images=False)

    # callback_ES = EarlyStopping(monitor = 'val_accuracy', patience=10)# probs no ↩
       need for this as only 3 epochs

    # Use Keras to train the model.
    history = model.fit(train_dataset,
                        epochs=10,
                        #batch_size=128,
                        validation_data=val_dataset,
                        callbacks=[callback_log])

    # Get the classification accuracy on the validation-set
    # after the last training-epoch.
    accuracy = history.history['val_binary_accuracy'][-1]

    # Print the classification accuracy.
    print()
    print("Accuracy: {0:.2%}".format(accuracy))
    print()

    # Save the model if it improves on the best-found performance.
    # We use the global keyword so we update the variable outside
    # of this function.
    global best_accuracy

    # If the classification accuracy of the saved model is improved ...
    if accuracy > best_accuracy:
        # Save the new model to harddisk.
        model.save(path_best_model)

        # Update the classification accuracy.
        best_accuracy = accuracy

    # Delete the Keras model with these hyper-parameters from memory.
    del model

    # Clear the Keras session, otherwise it will keep adding new
    # models to the same TensorFlow graph each time we create
    # a model with a different set of hyper-parameters.
    K.clear_session()

    # NOTE: Scikit-optimize does minimization so it tries to
    # find a set of hyper-parameters with the LOWEST fitness-value.
    # Because we are interested in the HIGHEST classification
    # accuracy, we need to negate this number so it can be minimized.
    return -accuracy


fitness(x=default_parameters)
plot_convergence(search_result)
search_result.x

```

```
from sklearn.metrics import confusion_matrix

confusion_matrix_validation = confusion_matrix(y_test, np.array(rf_pred))

df_cm = pd.DataFrame(confusion_matrix_validation, columns=np.unique(y_test), index = np.unique(y_test))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
plt.figure(figsize = (10,7))
plt.title('Test Data RF')
sns.set(font_scale=1.4)#for label size
sns.heatmap(df_cm, cmap="Blues", annot=True,annot_kws={"size": 16})# font size
print(classification_report(y_test, rf_pred))


final_model = create_model(learning_rate=0.0053,
                           num_dense_layers=1,
                           num_dense_nodes=302,
                           activation='relu', dropout_rate=0.3)

final_model.summary()
callback_ES = EarlyStopping(monitor = 'val_binary_accuracy', patience=10)

# Use Keras to train the model.
history = final_model.fit(train_full_dataset,
                      epochs=50,
                      #batch_size=128,
                      validation_data=val_dataset,
                      callbacks=[])

# Get the classification accuracy on the validation-set
# after the last training-epoch.
accuracy = history.history['val_binary_accuracy'][-1]

# Load the data into training and validation Dataset objects
test_dataset = tf.data.Dataset.from_tensor_slices((X_test, y_test))

test_dataset = test_dataset.batch(len(y_test))
final_model.evaluate(test_dataset)
mlp_pred = final_model.predict(test_dataset)
mlp_pred[mlp_pred>0.5] = 1
mlp_pred[mlp_pred<0.5] = 0

confusion_matrix_validation = confusion_matrix(y_test, np.array(mlp_pred))

df_cm = pd.DataFrame(confusion_matrix_validation, columns=np.unique(y_test), index = np.unique(y_test))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
plt.figure(figsize = (10,7))
plt.title('Test Data MLP')
sns.set(font_scale=1.4)#for label size
sns.heatmap(df_cm, cmap="Blues", annot=True,annot_kws={"size": 16})# font size
print(classification_report(y_test, mlp_pred))

# function to compute mc nemar's test statistics
def mcnemar(pred_A, pred_B, y_true):
  errors_A = (pred_A != y_true)
  errors_B = (pred_B != y_true)

  n_A = np.sum(np.invert(errors_B[errors_A]))
```

```
  n_B = np.sum(np.invert(errors_A[errors_B]))

  score = (np.abs(n_A-n_B)-1)/np.sqrt(n_A+n_B)
  return score

  mlp_pred =mlp_pred.reshape(-1)
  mcnemar(rf_pred, mlp_pred, y_test)

from scipy.stats import norm
p = 1-norm.cdf(2.1667)
print(p)

# reject option for random forest and confusion matrix.
rej = np.zeros(X_test.shape[0])
probs = rf_rand.predict_proba(X_test)
probs.shape
max_probs = np.max(probs, axis=1)
rej[1-max_probs>0.4] = -1
rej[1-max_probs<=0.4] = 1

classes = rf_rand.predict(X_test)
classes[rej==-1] = -1

y_rej = y_test[classes!=-1]
pred_rej = classes[classes!=-1]

confusion_matrix_validation = confusion_matrix(y_rej, np.array(pred_rej))

df_cm = pd.DataFrame(confusion_matrix_validation, columns=np.unique(y_rej), index =↩
     np.unique(y_rej))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
plt.figure(figsize = (10,7))
plt.title('Reject option RF')
sns.set(font_scale=1.4)#for label size
sns.heatmap(df_cm, cmap="Blues", annot=True,annot_kws={"size": 16})# font size
print(classification_report(y_rej, pred_rej))
```

Listing 2: Code for question 2

```
library(ggplot2)
library(dplyr)
library(tidyr)
library(data.table)
#library(matrixStats)
library(GGally)
set.seed(1351308)

cw_data <- read.csv('/Users/lorenzwolf/Desktop/MSc Statistics/Electives/Machine ↩
    Learning/Coursework 2/CW2Data.csv')
head(cw_data)
summary(cw_data)

library(purrr)

get_clust_assignment <- function(dist_method, linkage_method) {
  hclust(dist(X, method=dist_method), method=linkage_method)
}

# perform hierarchichal clustering using different arguments
```

```
hclust_args <- expand.grid(dist_method=c("euclidean", "manhattan"), linkage_method=c("complete", "single", "average", "ward.D2"))
hclust_list <- pmap(hclust_args, get_clust_assignment)

# plot the resulting dendrograms
par(mfrow=c(4,2))
for(i in 1:length(hclust_list)) {
  plot(hclust_list[[i]],
       main=sprintf("%s distance and %s linkage", hclust_args[i,1], hclust_args[i,2]),
       xlab="State")
  rect.hclust(hclust_list[[i]], k = 14, border = 2:n_clust) # add rectangle
}

library(cluster)
library("mice")

# find best hierarchical clustering
set.seed(2)
X <- t(scale(features))
#X <- scale(X)

num_clusters <- seq(2,27)
E_dist <- dist(X, method='euclidean')
M_dist <- dist(X, method='manhattan')
silhouette_mat <- matrix(0, nrow=length(num_clusters), ncol=length(hclust_list))
for(j in 1:length(num_clusters)){
  for(i in 1:length(hclust_list)) {
    grp <- cutree(hclust_list[[i]], k = num_clusters[j])

    if (hclust_args[i,1]=="euclidean"){
      s_score <- summary(silhouette(grp, E_dist))$avg.width
    }
    else {
      s_score <- summary(silhouette(grp, M_dist))$avg.width
    }
    silhouette_mat[j,i] <- s_score
  }
}
which(silhouette_mat == max(silhouette_mat), arr.ind = TRUE)

plot(2:27, silhouette_mat[,8], xlab='Number of cluster', ylab='Average silhouette width', main='Silhouette score vs number of clusters')
lines(2:27, silhouette_mat[,8], lty=2)
lines(c(14,14), c(0,0.2), lty=2)

# regression imputation
imp <- mice(scale(features), method = "norm.nob", m = 1) # Impute data
X <- complete(imp)
silhouette_mat

A <- na.omit(X)
km <- kmeans(A, 2, nstart=10)
pca <- prcomp(A, rank. = 4)


df <- as.data.frame(cbind(pca$x, km$cluster))
colnames(df)[[5]] <- "cluster_label"
df$cluster_label <- as.factor(df$cluster_label)

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)
```

```
q3_train <- q3[train_ind,]

write.csv(q3_train,'./q3_train.csv', row.names = FALSE)

plot(q3_train$x, q3_train$y)
plot(q3_test$x, q3_test$y)

# compute kernel
# input x is column
kernel <- function(x, p) (x %*% t(x) + 1)^p

# kernel ridge regression
loo_cv <- function(lambda,p, x, y){
  Num_data_points <- length(y)
  test_mse <- rep(0,Num_data_points)
  K <- kernel(x,p)

  for (i in 1:Num_data_points){

    K_i <- K[-i,-i]

    params <- solve(K_i + lambda * diag(dim(K_i)[2]), y[-i])

    test_mse[i] <- (y[i] - sum(K[i,-i]*params))^2

  }
  return(mean(test_mse))
}

library(latex2exp)

p_max<-7
kernel_powers <- seq(1,p_max)
lambda_vec <- 2^seq(-7,3,by=0.02)

av_mse <- matrix(0, nrow=length(kernel_powers), ncol=length(lambda_vec))

for (p in kernel_powers){
  av_mse[p,] <- sapply(lambda_vec, function(lambda) loo_cv(lambda, p, q3_train$x, ↩
      q3_train$y))
}

lambda_vec <- 2^seq(-7,3,by=0.02)
length(lambda_vec)

plot(lambda_vec, av_mse[1,], log='x', ylim=c(0.01,0.06), type='l', xlab=TeX('$\\↩
    lambda$'), ylab='Average Validation Error', main= TeX('Average Validation Error↩
     vs $\\lambda$'), col=1)
for (p in 2:p_max){
  lines(lambda_vec, av_mse[p,], col=p)
}
legend('topright', legend=c(1:p_max), col=c(1:p_max), lty=rep(1,p_max), cex=0.8)
#legend('topright', legend=c( TeX('p=1'), TeX('p=2'), TeX('p=3'), TeX('p=4')), col=↩
    c(1:p_max), lty=rep(1,4), cex=0.8)


inds <- which(av_mse == min(av_mse), arr.ind = TRUE)
inds
lambda_vec[inds[2]]

plot(lambda_vec, av_mse[5,], ylim=c(0.017, 0.019), log='x', type='l', xlab=TeX('$\\↩
```

```
    lambda$'), ylab='Average Validation Error', main= TeX('Average Validation Error vs $\\lambda$'), col=5)
lines(lambda_vec, av_mse[6,], col=6)
legend('topright', legend=c('p=5','p=6'), col=c(5,6), lty=rep(1,2), cex=0.8)


#p <- 5
#lambda <- 0.007921558
p <- inds[1]
lambda <- lambda_vec[inds[2]]
K <- kernel(q3_train$x,p)

final_params <- solve(K + lambda * diag(dim(K)[2]), q3_train$y)

preds_train <- K%*% final_params

# predictions on test set
K_star <- (q3_test$x %*% t(q3_train$x) + 1)^p
preds_test <- K_star %*% final_params

#plot fit on train set
plot(q3_train$x, q3_train$y, xlab='x', ylab='y', main='Fit on train data')
lines(q3_train$x[order(q3_train$x)], preds_train[order(q3_train$x)], col='red')
#plot fit on test set
plot(q3_test$x, q3_test$y, xlab='x',ylim=c(-0.8,0), ylab='y', main='Fit on test data')
lines(q3_test$x[order(q3_test$x)], preds_test[order(q3_test$x)], col='red')


#test mse
test_mse <- mean((q3_test$y-preds_test)^2)
test_mae <- mean(abs(q3_test$y-preds_test))

test_mse
test_mae

mean((q3_train$y-preds_train)^2)
```

Listing 4: Code for question 3.3

```
import time


from sklearn.metrics import mean_squared_error
import tensorflow.compat.v2 as tf
import tensorflow_probability as tfp
tfb = tfp.bijectors
tfd = tfp.distributions
tfk = tfp.math.psd_kernels
tf.enable_v2_behavior()

from mpl_toolkits.mplot3d import Axes3D
%pylab inline
# Configure plot defaults
plt.rcParams['axes.facecolor'] = 'white'
plt.rcParams['grid.color'] = '#666666'
%config InlineBackend.figure_format = 'png'

# load data
q3_train = pd.read_csv("/content/drive/My Drive/ML Coursework 2/q3_train.csv") #
```

```
    sampled already in R
q3_test = pd.read_csv("/content/drive/My Drive/ML Coursework 2/test_dataQ3.csv")

# since 0 mean assumed
X_train = np.array(q3_train['x']).reshape(-1, 1)
y_train = q3_train['y']
mean_y = np.mean(y_train)
y_train = y_train - mean_y

X_test = np.array(q3_test['x']).reshape(-1, 1)
y_test = q3_test['y'] -mean_y

observation_index_points_ = X_train
observations_ = y_train

# Exponential quadratic
def build_gp(amplitude, length_scale, observation_noise_variance):
  """Defines the conditional dist. of GP outputs, given kernel parameters."""

  # Create the covariance kernel, which will be shared between the prior (which we
  # use for maximum likelihood training) and the posterior (which we use for
  # posterior predictive sampling)
  kernel = tfk.ExponentiatedQuadratic(amplitude, length_scale)

  # Create the GP prior distribution, which we will use to train the model
  # parameters.
  return tfd.GaussianProcess(
      kernel=kernel,
      index_points=observation_index_points_,
      observation_noise_variance=observation_noise_variance)

gp_joint_model = tfd.JointDistributionNamed({
    'amplitude': tfd.LogNormal(loc=0., scale=np.float64(1.)),
    'length_scale': tfd.LogNormal(loc=0., scale=np.float64(1.)),
    'observation_noise_variance': tfd.LogNormal(loc=0., scale=np.float64(1.)),
    'observations': build_gp,
})
# Create the trainable model parameters, which we'll subsequently optimize.
# Note that we constrain them to be strictly positive.

constrain_positive = tfb.Shift(np.finfo(np.float64).tiny)(tfb.Exp())

amplitude_var = tfp.util.TransformedVariable(
    initial_value=1.,
    bijector=constrain_positive,
    name='amplitude',
    dtype=np.float64)

length_scale_var = tfp.util.TransformedVariable(
    initial_value=1.,
    bijector=constrain_positive,
    name='length_scale',
    dtype=np.float64)

observation_noise_variance_var = tfp.util.TransformedVariable(
    initial_value=1.,
    bijector=constrain_positive,
    name='observation_noise_variance_var',
    dtype=np.float64)

trainable_variables = [v.trainable_variables[0] for v in
```

```
                        [amplitude_var,
                        length_scale_var,
                        observation_noise_variance_var]]

# Now we optimize the model parameters.
num_iters = 1000
optimizer = tf.optimizers.Adam(learning_rate=.01)

# Store the likelihood values during training, so we can plot the progress
lls_ = np.zeros(num_iters, np.float64)
for i in range(num_iters):
  with tf.GradientTape() as tape:
    #tape.watch(trainable_variables)
    loss = -gp_joint_model.log_prob({
      'amplitude': amplitude_var,
      'length_scale': length_scale_var,
      'observation_noise_variance': observation_noise_variance_var,
      'observations': observations_
  })
  grads = tape.gradient(loss, trainable_variables)
  optimizer.apply_gradients(zip(grads, trainable_variables))
  lls_[i] = loss

print('Trained parameters:')
print('amplitude: {}'.format(amplitude_var._value().numpy()))
print('length_scale: {}'.format(length_scale_var._value().numpy()))
print('observation_noise_variance: {}'.format(observation_noise_variance_var._value()
    ().numpy()))
print('marginal log likelihood: {}'.format(lls_[-1]))

# plot to check convergence
plt.figure(figsize=(12, 4))
plt.plot(lls_)
plt.xlabel("Training iteration")
plt.ylabel("Log marginal likelihood")
plt.show()


predictive_index_points_ = np.linspace(-5, 5, 400, dtype=np.float64)
predictive_index_points_ = predictive_index_points_[..., np.newaxis]

optimized_kernel = tfk.ExponentiatedQuadratic(amplitude_var, length_scale_var)
gprm = tfd.GaussianProcessRegressionModel(
    kernel=optimized_kernel,
    index_points=predictive_index_points_,
    observation_index_points=observation_index_points_,
    observations=observations_,
    observation_noise_variance=observation_noise_variance_var,
    predictive_noise_variance=0.)


expq_mean = gprm.mean()
expq_std = gprm.stddev()

plt.figure(figsize=(12, 6))
plt.scatter(observation_index_points_[:, 0], observations_,
            label='Observations')
plt.plot(predictive_index_points_, expq_mean, c='r',
           label='Posterior Sample')
plt.fill(np.concatenate([predictive_index_points_, predictive_index_points_[::-1]])
    ,
```

```
        np.concatenate([expq_mean - 1.9600 * expq_std,
                        (expq_mean + 1.9600 * expq_std)[::-1]]),
        alpha=.5, fc='b', ec='None', label='95% confidence interval')
plt.xlabel(r"Index points ($\mathbb{R}^1$)")
plt.ylabel("Observation space")
plt.title('Square Exponential Kernel')
plt.legend(loc='upper right')
plt.show()

predictive_index_points_ = observation_index_points_
# Reshape to [200, 1] -- 1 is the dimensionality of the feature space.
predictive_index_points_ = predictive_index_points_[..., np.newaxis]

optimized_kernel = tfk.ExponentiatedQuadratic(amplitude_var, length_scale_var)
gprm = tfd.GaussianProcessRegressionModel(
    kernel=optimized_kernel,
    index_points=predictive_index_points_,
    observation_index_points=observation_index_points_,
    observations=observations_,
    observation_noise_variance=observation_noise_variance_var,
    predictive_noise_variance=0.)
print('train MSE: {}'.format(mean_squared_error(y_train, gprm.mean())))
gprm = tfd.GaussianProcessRegressionModel(
    kernel=optimized_kernel,
    index_points=X_test,
    observation_index_points=observation_index_points_,
    observations=observations_,
    observation_noise_variance=observation_noise_variance_var,
    predictive_noise_variance=0.)
print('test MSE: {}'.format(mean_squared_error(y_test, gprm.mean())))

# Rational Quadratic
def build_gp(amplitude, length_scale, observation_noise_variance, scale_mixture_rate):
  """Defines the conditional dist. of GP outputs, given kernel parameters."""

  # Create the covariance kernel, which will be shared between the prior (which we
  # use for maximum likelihood training) and the posterior (which we use for
  # posterior predictive sampling)
  kernel = tfk.RationalQuadratic(amplitude, length_scale, scale_mixture_rate)

  # Create the GP prior distribution, which we will use to train the model
  # parameters.
  return tfd.GaussianProcess(
      kernel=kernel,
      index_points=observation_index_points_,
      observation_noise_variance=observation_noise_variance)

gp_joint_model = tfd.JointDistributionNamed({
    'amplitude': tfd.LogNormal(loc=0., scale=np.float64(1.)),
    'length_scale': tfd.LogNormal(loc=0., scale=np.float64(1.)),
    'scale_mixture_rate': tfd.LogNormal(loc=0., scale=np.float64(1.)),
    'observation_noise_variance': tfd.LogNormal(loc=0., scale=np.float64(1.)),
    'observations': build_gp,
})

# Create the trainable model parameters, which we'll subsequently optimize.
# Note that we constrain them to be strictly positive.

constrain_positive = tfb.Shift(np.finfo(np.float64).tiny)(tfb.Exp())

```

```
amplitude_var = tfp.util.TransformedVariable(
    initial_value=1.,
    bijector=constrain_positive,
    name='amplitude',
    dtype=np.float64)

length_scale_var = tfp.util.TransformedVariable(
    initial_value=1.,
    bijector=constrain_positive,
    name='length_scale',
    dtype=np.float64)

scale_mixture_rate_var = tfp.util.TransformedVariable(
    initial_value=1.,
    bijector=constrain_positive,
    name='scale_mixture_rate',
    dtype=np.float64)

observation_noise_variance_var = tfp.util.TransformedVariable(
    initial_value=1.,
    bijector=constrain_positive,
    name='observation_noise_variance_var',
    dtype=np.float64)

trainable_variables = [v.trainable_variables[0] for v in
                       [amplitude_var,
                       length_scale_var,
                       observation_noise_variance_var,
                        scale_mixture_rate_var]]

# Now we optimize the model parameters.
num_iters = 1000
optimizer = tf.optimizers.Adam(learning_rate=.01)

# Store the likelihood values during training, so we can plot the progress
lls_ = np.zeros(num_iters, np.float64)
for i in range(num_iters):
  with tf.GradientTape() as tape:
    #tape.watch(trainable_variables)
    loss = -gp_joint_model.log_prob({
      'amplitude': amplitude_var,
      'length_scale': length_scale_var,
      'scale_mixture_rate': scale_mixture_rate_var,
      'observation_noise_variance': observation_noise_variance_var,
      'observations': observations_
  })
  grads = tape.gradient(loss, trainable_variables)
  optimizer.apply_gradients(zip(grads, trainable_variables))
  lls_[i] = loss

print('Trained parameters:')
print('amplitude: {}'.format(amplitude_var._value().numpy()))
print('length_scale: {}'.format(length_scale_var._value().numpy()))
print('scale_mixture_rate: {}'.format(scale_mixture_rate_var._value().numpy()))
print('observation_noise_variance: {}'.format(observation_noise_variance_var._value↩
    ().numpy()))
print('marginal log likelihood: {}'.format(lls_[-1]))

predictive_index_points_ = np.linspace(-5, 5, 400, dtype=np.float64)
# Reshape to [200, 1] -- 1 is the dimensionality of the feature space.
predictive_index_points_ = predictive_index_points_[..., np.newaxis]
```

```
optimized_kernel = tfk.RationalQuadratic(amplitude_var, length_scale_var, scale_mixture_rate_var)
gprm = tfd.GaussianProcessRegressionModel(
    kernel=optimized_kernel,
    index_points=predictive_index_points_,
    observation_index_points=observation_index_points_,
    observations=observations_,
    observation_noise_variance=observation_noise_variance_var,
    predictive_noise_variance=0.)

expq_mean = gprm.mean()
expq_std = gprm.stddev()

plt.figure(figsize=(12, 6))
plt.scatter(observation_index_points_[:, 0], observations_,
            label='Observations')
plt.plot(predictive_index_points_, expq_mean, c='r',
           label='Posterior Sample')
plt.fill(np.concatenate([predictive_index_points_, predictive_index_points_[::-1]]),
         np.concatenate([expq_mean - 1.9600 * expq_std,
                         (expq_mean + 1.9600 * expq_std)[::-1]]),
         alpha=.5, fc='b', ec='None', label='95% confidence interval')
plt.xlabel(r"Index points ($\mathbb{R}^1$)")
plt.ylabel("Observation space")
plt.title('Rational Quadratic Kernel')
plt.legend(loc='upper right')
plt.show()

# plot fit on test set
predictive_index_points_ = np.linspace(-5, 5, 400, dtype=np.float64)
# Reshape to [200, 1] -- 1 is the dimensionality of the feature space.
predictive_index_points_ = predictive_index_points_[..., np.newaxis]

optimized_kernel = tfk.RationalQuadratic(amplitude_var, length_scale_var, scale_mixture_rate_var)
gprm = tfd.GaussianProcessRegressionModel(
    kernel=optimized_kernel,
    index_points=predictive_index_points_,
    observation_index_points=observation_index_points_,
    observations=observations_,
    observation_noise_variance=observation_noise_variance_var,
    predictive_noise_variance=0.)

expq_mean = gprm.mean()
expq_std = gprm.stddev()

plt.figure(figsize=(12, 6))
plt.scatter(observation_index_points_[:, 0], observations_,
            label='Observations')
plt.plot(predictive_index_points_, expq_mean, c='r',
           label='Posterior Sample')
plt.fill(np.concatenate([predictive_index_points_, predictive_index_points_[::-1]]),
         np.concatenate([expq_mean - 1.9600 * expq_std,
                         (expq_mean + 1.9600 * expq_std)[::-1]]),
         alpha=.5, fc='b', ec='None', label='95% confidence interval')
plt.xlabel(r"Index points ($\mathbb{R}^1$)")
plt.ylabel("Observation space")
plt.title('Rational Quadratic Kernel')
```

```
plt.legend(loc='upper right')
plt.show()

predictive_index_points_ = observation_index_points_
# Reshape to [200, 1] -- 1 is the dimensionality of the feature space.
predictive_index_points_ = predictive_index_points_[..., np.newaxis]

gprm = tfd.GaussianProcessRegressionModel(
    kernel=optimized_kernel,
    index_points=predictive_index_points_,
    observation_index_points=observation_index_points_,
    observations=observations_,
    observation_noise_variance=observation_noise_variance_var,
    predictive_noise_variance=0.)
print('train MSE: {}'.format(mean_squared_error(y_train, gprm.mean())))
gprm = tfd.GaussianProcessRegressionModel(
    kernel=optimized_kernel,
    index_points=X_test,
    observation_index_points=observation_index_points_,
    observations=observations_,
    observation_noise_variance=observation_noise_variance_var,
    predictive_noise_variance=0.)
print('test MSE: {}'.format(mean_squared_error(y_test, gprm.mean())))

def build_gp(amplitude, length_scale, observation_noise_variance, period):
  """Defines the conditional dist. of GP outputs, given kernel parameters."""

  # Create the covariance kernel, which will be shared between the prior (which we
  # use for maximum likelihood training) and the posterior (which we use for
  # posterior predictive sampling)
  kernel = tfk.ExpSinSquared(
    amplitude, length_scale, period)

  # Create the GP prior distribution, which we will use to train the model
  # parameters.
  return tfd.GaussianProcess(
      kernel=kernel,
      index_points=observation_index_points_,
      observation_noise_variance=observation_noise_variance)

gp_joint_model = tfd.JointDistributionNamed({
    'amplitude': tfd.LogNormal(loc=0., scale=np.float64(1.)),
    'length_scale': tfd.LogNormal(loc=0., scale=np.float64(1.)),
    'period': tfd.LogNormal(loc=0., scale=np.float64(1.)),
    'observation_noise_variance': tfd.LogNormal(loc=0., scale=np.float64(1.)),
    'observations': build_gp,
})

# Create the trainable model parameters, which we'll subsequently optimize.
# Note that we constrain them to be strictly positive.

constrain_positive = tfb.Shift(np.finfo(np.float64).tiny)(tfb.Exp())

amplitude_var = tfp.util.TransformedVariable(
    initial_value=1.,
    bijector=constrain_positive,
    name='amplitude',
    dtype=np.float64)

length_scale_var = tfp.util.TransformedVariable(
    initial_value=1.,
```

```
    bijector=constrain_positive,
    name='length_scale',
    dtype=np.float64)

period_var = tfp.util.TransformedVariable(
    initial_value=10.,
    bijector=constrain_positive,
    name='period',
    dtype=np.float64)

observation_noise_variance_var = tfp.util.TransformedVariable(
    initial_value=1.,
    bijector=constrain_positive,
    name='observation_noise_variance_var',
    dtype=np.float64)

trainable_variables = [v.trainable_variables[0] for v in
                       [amplitude_var,
                       length_scale_var,
                       period_var,
                       observation_noise_variance_var]]

# Now we optimize the model parameters.
num_iters = 1000
optimizer = tf.optimizers.Adam(learning_rate=.01)

# Store the likelihood values during training, so we can plot the progress
lls_ = np.zeros(num_iters, np.float64)
for i in range(num_iters):
  with tf.GradientTape() as tape:
    #tape.watch(trainable_variables)
    loss = -gp_joint_model.log_prob({
      'amplitude': amplitude_var,
      'length_scale': length_scale_var,
      'period': period_var,
      'observation_noise_variance': observation_noise_variance_var,
      'observations': observations_
  })
  grads = tape.gradient(loss, trainable_variables)
  optimizer.apply_gradients(zip(grads, trainable_variables))
  lls_[i] = loss

print('Trained parameters:')
print('amplitude: {}'.format(amplitude_var._value().numpy()))
print('length_scale: {}'.format(length_scale_var._value().numpy()))
print('period: {}'.format(period_var._value().numpy()))
print('observation_noise_variance: {}'.format(observation_noise_variance_var._value↩
    ().numpy()))
print('marginal log likelihood: {}'.format(lls_[-1]))

predictive_index_points_ = np.linspace(-5, 5, 400, dtype=np.float64)
# Reshape to [200, 1] -- 1 is the dimensionality of the feature space.
predictive_index_points_ = predictive_index_points_[..., np.newaxis]

optimized_kernel = tfk.ExpSinSquared(amplitude_var, length_scale_var, period_var)
gprm = tfd.GaussianProcessRegressionModel(
    kernel=optimized_kernel,
    index_points=predictive_index_points_,
    observation_index_points=observation_index_points_,
    observations=observations_,
    observation_noise_variance=observation_noise_variance_var,
```

```
    predictive_noise_variance=0.)

expq_mean = gprm.mean()
expq_std = gprm.stddev()

plt.figure(figsize=(12, 6))
plt.scatter(observation_index_points_[:, 0], observations_,
            label='Observations')
plt.plot(predictive_index_points_, expq_mean, c='r',
           label='Posterior Sample')
plt.fill(np.concatenate([predictive_index_points_, predictive_index_points_[::-1]]),
    np.concatenate([expq_mean - 1.9600 * expq_std,
                    (expq_mean + 1.9600 * expq_std)[::-1]]),
         alpha=.5, fc='b', ec='None', label='95% confidence interval')
plt.xlabel(r"Index points ($\mathbb{R}^1$)")
plt.ylabel("Observation space")
plt.title('Square Exponential Kernel')
plt.legend(loc='upper right')
plt.show()

predictive_index_points_ = observation_index_points_
# Reshape to [200, 1] -- 1 is the dimensionality of the feature space.
predictive_index_points_ = predictive_index_points_[..., np.newaxis]
gprm = tfd.GaussianProcessRegressionModel(
    kernel=optimized_kernel,
    index_points=predictive_index_points_,
    observation_index_points=observation_index_points_,
    observations=observations_,
    observation_noise_variance=observation_noise_variance_var,
    predictive_noise_variance=0.)
print('train MSE: {}'.format(mean_squared_error(y_train, gprm.mean())))
gprm = tfd.GaussianProcessRegressionModel(
    kernel=optimized_kernel,
    index_points=X_test,
    observation_index_points=observation_index_points_,
    observations=observations_,
    observation_noise_variance=observation_noise_variance_var,
    predictive_noise_variance=0.)
print('test MSE: {}'.format(mean_squared_error(y_test, gprm.mean())))


#### cross validation
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

def cv(n_folds, X, y):
  j = 0
  mse_per = np.zeros(n_folds)
  mse_rbf = np.zeros(n_folds)
  mse_rq = np.zeros(n_folds)


  kf = KFold(n_splits=n_folds, shuffle=True)

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

```
                        length_scale_var,
                        period_var,
                        observation_noise_variance_var]]

    # Now we optimize the model parameters.
  num_iters = 750
  optimizer = tf.optimizers.Adam(learning_rate=.01)

  # Store the likelihood values during training, so we can plot the progress
  lls_ = np.zeros(num_iters, np.float64)
  for i in range(num_iters):
    with tf.GradientTape() as tape:
      #tape.watch(trainable_variables)
      loss = -gp_joint_model.log_prob({
        'amplitude': amplitude_var,
        'length_scale': length_scale_var,
        'period': period_var,
        'observation_noise_variance': observation_noise_variance_var,
        'observations': y_cv_train
    })
    grads = tape.gradient(loss, trainable_variables)
    optimizer.apply_gradients(zip(grads, trainable_variables))
    lls_[i] = loss

  optimized_kernel = tfk.ExpSinSquared(amplitude_var, length_scale_var, period_↩
      var)
  gprm = tfd.GaussianProcessRegressionModel(
    kernel=optimized_kernel,
    index_points=X_cv_test,
    observation_index_points=X_cv_train,
    observations=y_cv_train,
    observation_noise_variance=observation_noise_variance_var,
    predictive_noise_variance=0.)
  mse_per[j] = mean_squared_error(y_cv_test, gprm.mean())


  #RQ kernel

  def build_gp(amplitude, length_scale, observation_noise_variance, scale_mixture↩
      _rate):
    """Defines the conditional dist. of GP outputs, given kernel parameters."""

    # Create the covariance kernel, which will be shared between the prior (which↩
        we
    # use for maximum likelihood training) and the posterior (which we use for
    # posterior predictive sampling)
    kernel = tfk.RationalQuadratic(amplitude, length_scale, scale_mixture_rate)

    # Create the GP prior distribution, which we will use to train the model
    # parameters.
    return tfd.GaussianProcess(
        kernel=kernel,
        index_points=X_cv_train,
        observation_noise_variance=observation_noise_variance)

  gp_joint_model = tfd.JointDistributionNamed({
      'amplitude': tfd.LogNormal(loc=0., scale=np.float64(1.)),
      'length_scale': tfd.LogNormal(loc=0., scale=np.float64(1.)),
      'scale_mixture_rate': tfd.LogNormal(loc=0., scale=np.float64(1.)),
      'observation_noise_variance': tfd.LogNormal(loc=0., scale=np.float64(1.)),
      'observations': build_gp,
```

```
    })

    # Create the trainable model parameters, which we'll subsequently optimize.
    # Note that we constrain them to be strictly positive.

    constrain_positive = tfb.Shift(np.finfo(np.float64).tiny)(tfb.Exp())

    amplitude_var = tfp.util.TransformedVariable(
        initial_value=1.,
        bijector=constrain_positive,
        name='amplitude',
        dtype=np.float64)

    length_scale_var = tfp.util.TransformedVariable(
        initial_value=1.,
        bijector=constrain_positive,
        name='length_scale',
        dtype=np.float64)

    scale_mixture_rate_var = tfp.util.TransformedVariable(
        initial_value=1.,
        bijector=constrain_positive,
        name='scale_mixture_rate',
        dtype=np.float64)

    observation_noise_variance_var = tfp.util.TransformedVariable(
        initial_value=1.,
        bijector=constrain_positive,
        name='observation_noise_variance_var',
        dtype=np.float64)

    trainable_variables = [v.trainable_variables[0] for v in
                           [amplitude_var,
                           length_scale_var,
                           observation_noise_variance_var,
                             scale_mixture_rate_var]]
    # Now we optimize the model parameters.
    num_iters = 750
    optimizer = tf.optimizers.Adam(learning_rate=.01)

  # Store the likelihood values during training, so we can plot the progress
    lls_ = np.zeros(num_iters, np.float64)
    for i in range(num_iters):
      with tf.GradientTape() as tape:
        #tape.watch(trainable_variables)
        loss = -gp_joint_model.log_prob({
          'amplitude': amplitude_var,
          'length_scale': length_scale_var,
          'scale_mixture_rate': scale_mixture_rate_var,
          'observation_noise_variance': observation_noise_variance_var,
          'observations': y_cv_train
      })
      grads = tape.gradient(loss, trainable_variables)
      optimizer.apply_gradients(zip(grads, trainable_variables))
      lls_[i] = loss

    optimized_kernel = tfk.RationalQuadratic(amplitude_var, length_scale_var, scale↩
        _mixture_rate_var)
    gprm = tfd.GaussianProcessRegressionModel(
      kernel=optimized_kernel,
      index_points=X_cv_test,
```

```
      observation_index_points=X_cv_train,
      observations=y_cv_train,
      observation_noise_variance=observation_noise_variance_var,
      predictive_noise_variance=0.)
    mse_rq[j] = mean_squared_error(y_cv_test, gprm.mean())

    def build_gp(amplitude, length_scale, observation_noise_variance):
      """Defines the conditional dist. of GP outputs, given kernel parameters."""

      # Create the covariance kernel, which will be shared between the prior (which we
      # use for maximum likelihood training) and the posterior (which we use for
      # posterior predictive sampling)
      kernel = tfk.ExponentiatedQuadratic(amplitude, length_scale)

      # Create the GP prior distribution, which we will use to train the model
      # parameters.
      return tfd.GaussianProcess(
          kernel=kernel,
          index_points=X_cv_train,
          observation_noise_variance=observation_noise_variance)

    gp_joint_model = tfd.JointDistributionNamed({
        'amplitude': tfd.LogNormal(loc=0., scale=np.float64(1.)),
        'length_scale': tfd.LogNormal(loc=0., scale=np.float64(1.)),
        'observation_noise_variance': tfd.LogNormal(loc=0., scale=np.float64(1.)),
        'observations': build_gp,
    })


    # Create the trainable model parameters, which we'll subsequently optimize.
    # Note that we constrain them to be strictly positive.

    constrain_positive = tfb.Shift(np.finfo(np.float64).tiny)(tfb.Exp())

    amplitude_var = tfp.util.TransformedVariable(
        initial_value=1.,
        bijector=constrain_positive,
        name='amplitude',
        dtype=np.float64)

    length_scale_var = tfp.util.TransformedVariable(
        initial_value=1.,
        bijector=constrain_positive,
        name='length_scale',
        dtype=np.float64)

    observation_noise_variance_var = tfp.util.TransformedVariable(
        initial_value=1.,
        bijector=constrain_positive,
        name='observation_noise_variance_var',
        dtype=np.float64)

    trainable_variables = [v.trainable_variables[0] for v in
                           [amplitude_var,
                           length_scale_var,
                           observation_noise_variance_var]]

    # Now we optimize the model parameters.
    num_iters = 750
    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)
```