Supervised Learning (COMP0078) Report

Yoga Advaith Veturi zchayav@ucl.ac.uk
Maki Egawa zcabmeg@ucl.ac.uk

November 16, 2020

1 PART I

The source code for this section is provided in `p1_sourcecode.ipynb`, which is attached with the submission, or is accessible via this clickable link. For convenience however, relevant code snippets are also attached inline with the explanations.

1.1 Linear Regression

1. (a) We are provided with a toy data set

$$\mathcal{D} = \{(1,3),(2,2),(3,0),(4,5)\}$$

where each pair (x_i, y_i) contains an input $x_i \in \mathbb{R}^n$ and output $y_i \in \mathbb{R}$. The goal is to fit $\mathcal{D}$ using a linear combination of polynomial basis functions

$$\{\phi_1(x) = 1, \phi_2(x) = x^2\, \phi_3(x) = x^3, ..., \phi_k(x) = x^{k-1}\}$$

of dimension k, where each basis function $\phi_k : \mathbb{R}^n \to \mathbb{R}$. The fitting is performed for dimensions $k = 1, 2, 3, 4$.

To perform the curve fitting, we first defined a function Φ that takes the vector of inputs $\mathbf{x}$ and transforms it into an $(\mathrm{m} \times \mathrm{k})$ feature map $\Phi(\mathbf{x})$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \longrightarrow \Phi(\mathbf{x}) = \begin{bmatrix} \phi_1(x_1) & \dots & \phi_k(x_1) \\ \phi_1(x_2) & \dots & \phi_k(x_2) \\ \vdots & \ddots & \vdots \\ \phi_1(x_m) & \dots & \phi_k(x_m) \end{bmatrix}$$

where each $\phi_k(x)$ is a single polynomial basis function. The following code executes this, provided the value of k.

```
#Computes the polynomial basis
def phi(x, k):
  """ Computes a single basis function $x^{k-1}$, given
    an input vector x of shape (m, 1) and the value k.
   m is the number of training examples """
  return x**(k-1)

#This function creates the feature map PHI()
def PHI(x, k):
  """ Creates the feature map, which is a matrix of
   shape (m, k). Each column is the kth basis function
   applied on
  the input vector x of shape (m, 1). m is the number
   of training examples """

  #Number of training examples
  m = x.shape[0]

  #Initialize a feature map of shape (m, k) with 0s in
   it.
  feature_map = np.zeros((m, k))

  #In every column, replace the zeros with the basis
   function k computed on input vector x
  for i in range(0, k):
    feature_map[:, i] = phi(x.squeeze(), i+1)

  return feature_map

#Compute feature maps for k = {1,...,4}
x_tr_k1 = PHI(x, k=1)
```

```
x_tr_k2 = PHI(x, k=2)
x_tr_k3 = PHI(x, k=3)
x_tr_k4 = PHI(x, k=4)
```

With the transformed feature map $\Phi(\mathbf{x})$, we then computed the weights $\mathbf{w} \in \mathbb{R}^k$ that minimises the sum of squared errors defined by

$$SSE = \sum_{i=1}^{m} (y_i - \mathbf{w} \cdot \phi(x))^2 = (\Phi\mathbf{w} - y)^T(\Phi\mathbf{w} - y)$$

Setting the gradient $\nabla_{\mathbf{w}}(SSE) = 0$ and solving for $\mathbf{w}$, the least squares solution is

$$\mathbf{w} = (\Phi^T\Phi)^{-1}\Phi^T y$$

The following code computes the weights for $k = 1, 2, 3, 4$.

```
#Compute weights for the transformed data using least
    squares regression
w_k1 = np.linalg.inv(x_tr_k1.T @ x_tr_k1) @ x_tr_k1.T @
    y
w_k2 = np.linalg.inv(x_tr_k2.T @ x_tr_k2) @ x_tr_k2.T @
    y
w_k3 = np.linalg.inv(x_tr_k3.T @ x_tr_k3) @ x_tr_k3.T @
    y
w_k4 = np.linalg.inv(x_tr_k4.T @ x_tr_k4) @ x_tr_k4.T @
    y
```

(b) The following polynomial equations were obtained, corresponding to $k = 1, 2, 3, 4$:

$$y = 2.5$$

$$y = 1.5 + 0.4x$$

$$y = 9.0 - 7.1x + 1.5x^2$$

$$y = -5.0 + 15.17x - 8.5x^2 + 1.33x^3$$

These are visualized in Figure 1.

(c) For each fitted curve, the mean squared error was computed using $MSE = \frac{SSE}{m}$. The following code executes the MSE function.

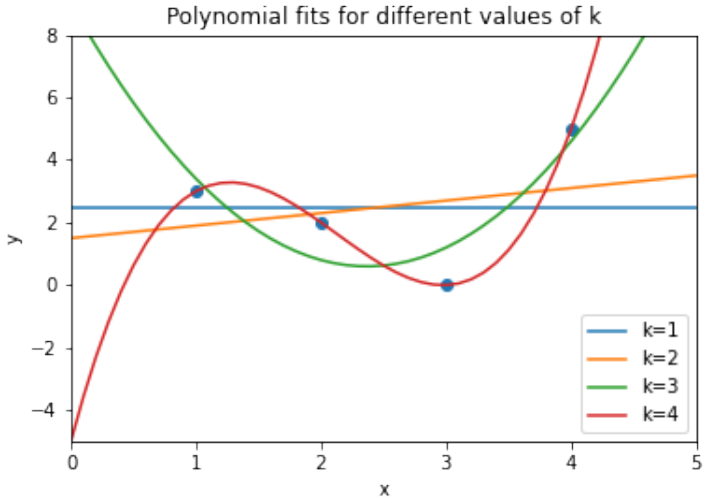


Figure 1: Fitted curves for k=1,2,3,4

```
def MSE(x_values, y_values, weights):
  """
  Computes the MSE given the x_values matrix (m, n),
   y_values vector of shape (m, 1) and weights vector
   of shape (n, 1).
  m is the number of training examples, n is the number
    of input features.
  """
  m = x_values.shape[0]
  SSE = ((x_values @ weights) - y_values).T @ ((
   x_values @ weights) - y_values)
  MSE = SSE / m
  return MSE

#Print MSE values for each of the polynomial bases
print("MSE for k = 1 : " + str(MSE(x_tr_k1, y, w_k1).
    item()))
print("MSE for k = 2 : " + str(MSE(x_tr_k2, y, w_k2).
    item()))
print("MSE for k = 3 : " + str(MSE(x_tr_k3, y, w_k3).
    item()))
print("MSE for k = 4 : " + str(MSE(x_tr_k4, y, w_k4).
    item()))
```

The MSE values are presented in Table 1 below.

k	MSE
1	3.25
2	3.05
3	0.80
4	3.49e-23

Table 1: Mean Squared Error for fitted curves $k = 1, 2, 3, 4$

2. We now fit models on data generated from the random function

$$g_{\sigma=0.07}(x) \coloneqq \sin^2 2\pi x + \epsilon$$

where ϵ is Gaussian noise with a mean of 0 and variance $\sigma^2 = 0.07^2$.

(a) i. A training set of 30 points was generated by sampling random points uniformly in the interval $[0, 1]$. These points were passed through g_σ and on each call of the function, Gaussian noise ϵ is added. The following code implements this:

```
def g_sigma(x, sigma):
  """
  Random function of form g_sigma(x) = sin^2(2*pi*x
   ) + epsilon where epsilon is a Gaussian
   distributed random variable
  with mean 0 and sigma^2 variance.

  Takes an input x and returns the value g_sigma(x)
    with an added noise value
  generated from epsilon.
  """

  #Compute function on the point x
  function = np.square(np.sin(2 * np.pi * x))
  #Generate a single noise value for that point x
  noise = np.random.normal(0.0, sigma)

  return function + noise

def generate_data(interval, size, sigma):
  """
  Generates a random sample {(x_1, g_1),...,(x_m,
   g_m)}. First samples a single x value from the
   interval
```

```
  and then computes g_sigma = sin^2(2pi*x) +
   epsilon, where epsilon is random gaussian noise.

  Args
  ----
  interval - tuple of (lower_bound, upper_bound)
  size - number of points to generate
  sigma - standard deviation of the noise

  Returns
  -------
  NumPy arrays of the generated input and output
   data
  """

  #This will store the x_values and g_sigma(x)
   values
  x_values = []
  g_values = []

  for i in range(size):

    #Generate a single point
    x = np.random.uniform(interval[0], interval[1])
    #Create the output value using the random
   function
    g = g_sigma(x, sigma=sigma)
    #Append the values to the list
    x_values.append(x)
    g_values.append(g)

  return np.array(x_values), np.array(g_values)

#Generate training data - set a seed for ensuring
   reproducible results
np.random.seed(1)
x_train, g_train = generate_data(interval=(0, 1),
   size=30, sigma=0.07)
```

The generated data set is plotted along with the deterministic function $\sin^2(2\pi x)$ in Figure 2a.

ii. This data set is then fit using polynomial bases of dimension $k = 2, 5, 10, 14, 18$ using the same process followed in 1a. The x values are first transformed into the feature map Φ, then using

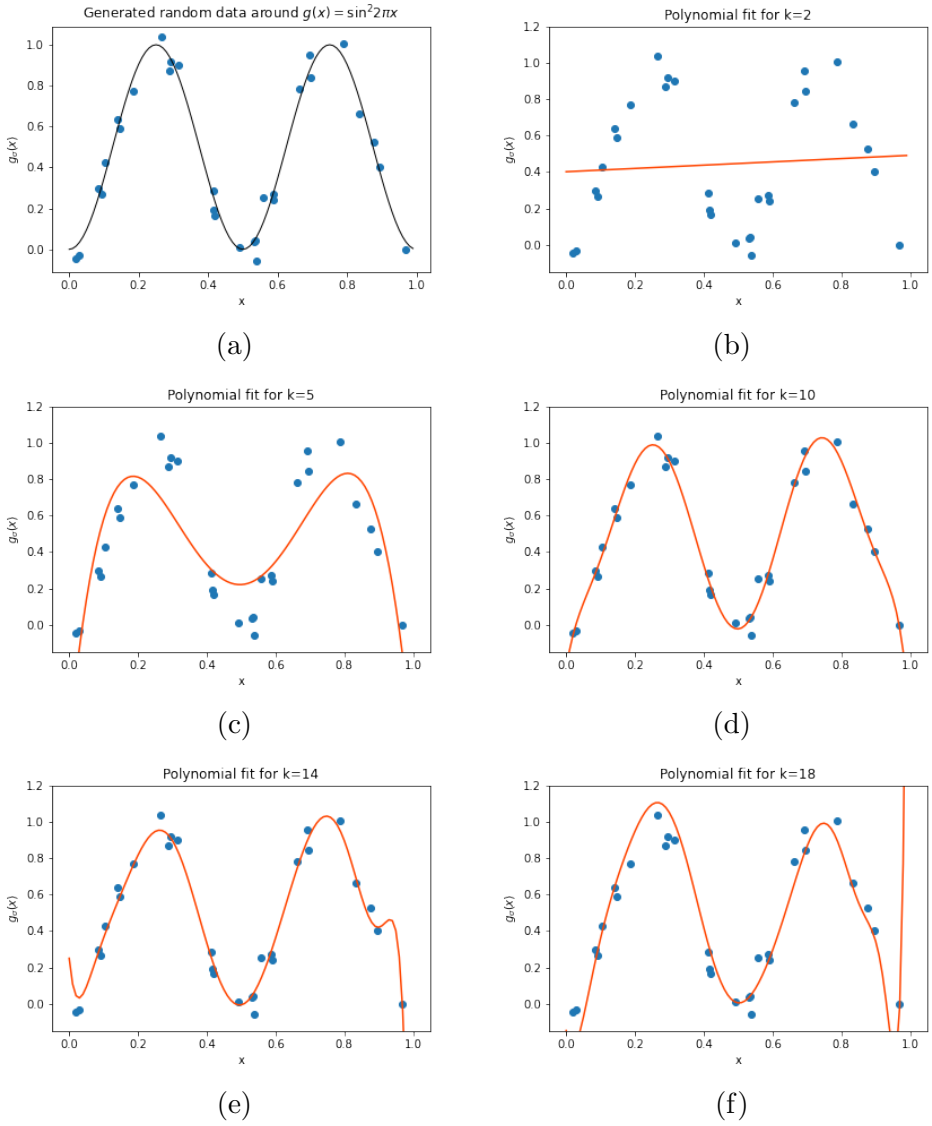


Figure 2: (a) Generated data (blue points) with the deterministic function $sin^2(2\pi x)$ overlapped (black curve). All fitted curves are shown in orange. (b) Fitted curve for $k = 2$. (c) Fitted curve for $k = 5$. (d) Fitted curve for $k = 10$. (e) Fitted curve for $k = 14$. (f) Fitted curve for $k = 18$.

least squares regression, the weights $\mathbf{w}$ are computed for each k. These fitted curves are presented in Figure 2(b-f).

(b) The training error denoted te_k was computed by performing linear regression on the training points for every value of $k = 1, ..., 18$ and calculating the MSE on the training points using the *MSE()* function. The natural log of the training error is plotted in Figure 3(a). It is noticeable that the function is decreasing, however around epoch 16, there is a rise, which could be attributed to numerical errors.

(c) A test set of 1000 points was generated using the *generate_data*() function described earlier. The test set error, denoted tse_k was computed by calculating the MSE on the test points using the weights computed on the training points. The natural log of the test set error for $k = 1, ..., 18$ is plotted in Figure 3(b).

(d) The previous results correspond to one type of generated training and test set. To see these errors smoothed out for different types of training and test sets, the previous steps in 2b and 2c were repeated for 100 runs and the log of the average train and test set error over the runs were computed. The results are plotted in Figure 3(c).

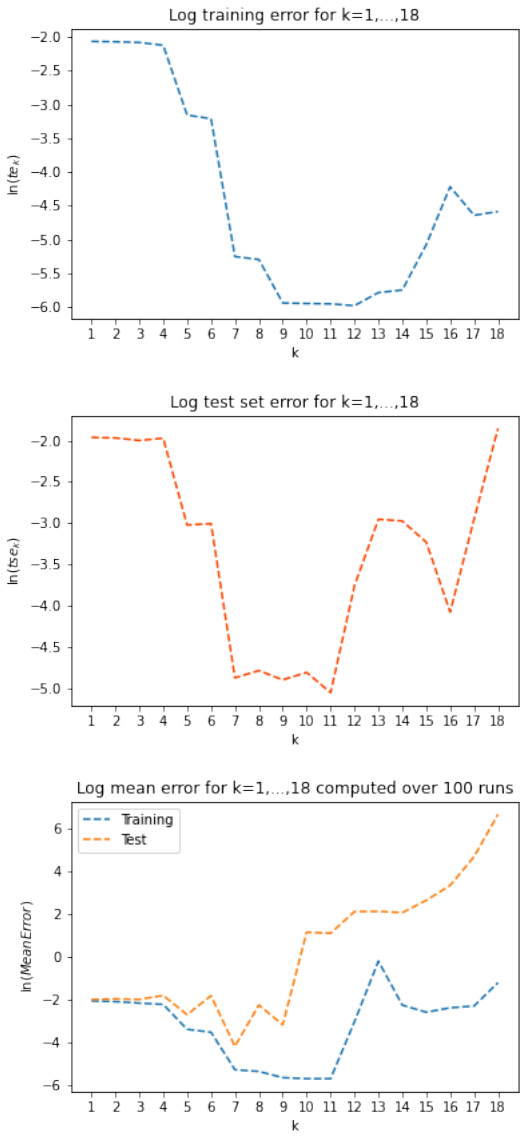


Figure 3: MSE values for $k = 1, ..., 18$ (a) Log training error (b) Log test set error (c) Log training and test set error, averaged over 100 runs

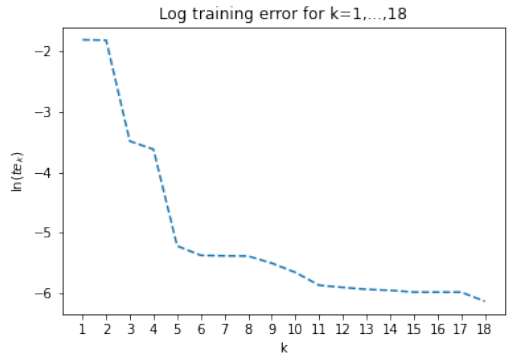
3. The experiments performed in 2b, 2c and 2d are repeated, now using a different basis

$$\{\sin 1\pi x, \sin 2\pi x, \cdots, \sin k\pi x\}$$

We modify the functions for computing the feature map as follows:

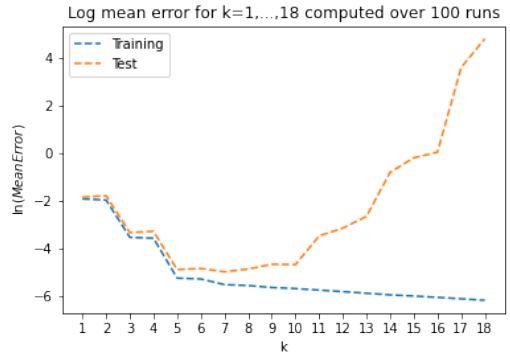
```
#Computes the basis function
def phi_v2(x, k):
  """ Computes a single basis function, given input vector
   x of shape (m, 1) and value k. m is the number of
   training examples """
  return np.sin(k*np.pi*x)

#This function creates the feature map
def PHI_v2(x, k):
  """ Creates the feature map, which is a matrix of shape (
   m, k). Each column is the kth basis function applied on
  the input vector x of shape (m, 1). m is the number of
   training examples """

  #Extract the number of training examples
  m = x.shape[0]

  #Initialize feature map matrix to matrix of zeros
  feature_map = np.zeros((m, k))

  #Iterative over each column
  for i in range(0, k):
    #For each column of the feature map matrix, replace the
    0s with the computed kth basis function values
    feature_map[:, i] = phi_v2(x.squeeze(), i+1)
  return feature_map
```

The log te_k and tse_k for $k = 1, ..., 18$ are plotted in Figure 4a and 4b respectively. The entire operation is performed for 100 runs, using different generated training and test points. The log of the average te_k and tse_k is visualized in Figure 4c.



(a)

(b)



(c)

Figure 4: (a) Log training error for $k = 1, ..., 18$ (b) Log test error for $k = 1, ..., 18$ (c) Log of average train and test error over 100 runs.

1.2 Filtered Boston Housing and Kernels

4. In this section, we were provided with a modified version of the Boston Housing data set. Below is a quick inspection of the first few rows of this data.

CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	LSTAT	MEDV
0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	4.98	24.0
0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	9.14	21.6
0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	4.03	34.7
0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	2.94	33.4
0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	5.33	36.2

Table 2: Inspection of first 5 rows of Boston Housing data set

The goal was to predict the median house price (MEDV) using the remaining 12 variables. To do so, we performed multiple experiments with different variants of linear regression. These experiments involved splitting the data into a training set, which contained 2/3 of the examples, and a test set with 1/3 of the examples. The following function implements the data splitting. Note that the data can be shuffled before splitting in order to generate random splits.

```
def split_data(inputs, targets, test_proportion, shuffle=
    None):
  """
  Splits the data into training and test sets.

  Args
  ----
  inputs : NumPy array of input data. Should be of shape (#
    examples, # features)
  targets : NumPy array of target data. Should be of shape
   (# examples, 1)
  test_proportion : Value between 0 and 1 which specifies
   how much of the data to use for testing.
  shuffle : Optional. Set to True if you want the data
   shuffled and then split.
  seed : Optional. Set for reproducible results.

  Returns
  -------
  train_X : NumPy array of training examples. Should be of
   shape (# examples, # features)
```

```
    train_Y : NumPy array of training targets. Should be of shape (# examples, 1)
    test_X : NumPy array of testing examples. Should be of shape (# examples, # features)
    test_Y : NumPy array of testing targets. Should be of shape (# examples, 1)
    """

    #Stores the number of data points
    nData = inputs.shape[0]

    # Shuffle data
    if shuffle:
      #Generate a shuffled version of the array indices
      shuffled_indices = np.random.permutation(nData)
      #Shuffle the inputs as per in the array of shuffled indices
      shuffled_inputs = inputs[shuffled_indices, :]
      shuffled_targets = targets[shuffled_indices, :]
    else:
      #If shuffle is set to False then we just work with the data in its original order
      shuffled_inputs = inputs
      shuffled_targets = targets

    # Calculate the split index based on the specified proportions
    split_index = int((1 - test_proportion) * nData)

    # Select the examples up to the split index to be used as training set
    train_X = shuffled_inputs[:split_index]
    train_Y = shuffled_targets[:split_index]
    # Select the examples from the split index onwards to be used as the test set
    test_X = shuffled_inputs[split_index:]
    test_Y = shuffled_targets[split_index:]

    return train_X, train_Y, test_X, test_Y
```

(a) We first performed Naive regression. The data set was split into train and test sets using the splitting function. A vector of ones was created with the same length as the training set and another for the test set. Using the training ones vector and the training outputs, we computed the weights with linear regression. Finally, the train and test MSE was computed using the $MSE()$ function. The code implementation for this is provided below. Note that it is performed over 20 runs for 20 random train/test splits.

```
#For reproducibility of results
np.random.seed(101010)

#Stores the number of runs
runs = 20

#These will store the total train and test MSE over the 20 runs
train_MSE = 0
test_MSE = 0

#Iterate over runs
for i in range(runs):

  #Generate samples
  xtrain, ytrain, xtest, ytest = split_data(inputs=X, targets=Y, test_proportion=1/3, shuffle=True)

  train_ones = np.ones((xtrain.shape[0], 1))
  test_ones = np.ones((xtest.shape[0], 1))

  #Perform Naive Regression
  w_naive = np.linalg.inv(train_ones.T @ train_ones) @ train_ones.T @ ytrain

  #Calculate and aggregate train and test MSE
  train_MSE += MSE(train_ones, ytrain, w_naive)
  test_MSE += MSE(test_ones, ytest, w_naive)

  print(">> Run {} Completed.".format(i+1))
```

The average train and test MSE over 20 runs is as follows:

```
Average Train MSE = 86.05624289198637
Average Test MSE = 81.51650453132758
```

(b) With Naive Regression, we are essentially fitting the data using a constant function and asking the question "What would the model predict in the absence of any features?". The interpretation of this constant function is as follows:

We can define the training inputs and outputs as the vectors X_{train} and Y_{train}

$$X_{train} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}, Y_{train} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

where m is the number of training examples. If we used these vectors to compute the weights w_{naive} then we would get the following solution:

$$w_{naive} = (X_{train}^T X_{train})^{-1} X_{train}^T Y_{train} = \frac{1}{m} \sum_{i=1}^{m} y_i$$

Hence, the naive weights is essentially the mean of the training output values, which is what the model would predict for a given input. To validate this, we compared the output of the w_{naive} with the mean of Y_{train} on a single run and found them to be the same:

```
w_naive value = [[23.11780415]]
Mean of Y_train values = 23.11780415430267
```

(c) We next performed Linear Regression over single attributes. A column of ones was added to the matrix of data in order to account for the bias term in the linear regression. The following function adds the ones column:

```
#Function to add ones to the training set as a way to account for the bias term in the weights
def add_bias(data):
  """
  Adds a column of ones into the dataset in order to account for the bias term.

  Args
  ----
  data : a NumPy array of the input data. Can be a matrix of shape (m, n) or a single input vector x_i of shape (m, 1). m is the number of training examples
  while n is the number of input features.

  Returns
  -------
  data_with_bias : a NumPy array of data with 1s added. If data is a matrix, then a column os 1s is added to the first column.
  If data is a single vector x_i, then the first element of x_i is 1.
  """

  #If the data is a vector
  if data.shape[1] == 1:
    #Create a one
    ones = np.ones((data.shape[1], ))
    #Insert 1 as the first value for the vector
    data_with_bias = np.insert(data, 0, ones, axis=0)

  #If data is a matrix
  else:
    #Create a vector ones
    ones = np.ones((data.shape[0], ))
    #Stack this vector of ones to the beginning of the matrix
    data_with_bias = np.insert(data, 0, ones, axis=1)

  return data_with_bias

#Convert the pandas dataframe into matrix
data = dataframe.values

#Add a column of 1's to the data matrix to account for
```

```
    bias term
data_with_bias = add_bias(data)

#Split the data matrix into a matrix of the inputs and
    the vector of outputs
X = data_with_bias[:, :13]
Y = data_with_bias[:, 13].reshape(-1, 1)
```

Using the *split_data*() function, the data was shuffled and then split into training and testing sets. For training, the individual columns of the training data matrix, which represent the different attributes, were selected along with the "ones" column and the weights were computed using linear regression for these "sub-matrices". The training and test MSE was then computed using the *MSE()* function. We report the results averaged over 20 runs for 20 random train/test splits in Table 3 below.

Method	MSE Train	MSE Test
Attribute 1	69.568609	77.953024
Attribute 2	72.015690	76.825691
Attribute 3	62.422693	69.677222
Attribute 4	80.442598	85.280471
Attribute 5	67.203539	73.004103
Attribute 6	42.845189	45.544419
Attribute 7	70.498032	76.720579
Attribute 8	77.077643	83.831789
Attribute 9	70.034179	76.850144
Attribute 10	63.711957	70.728458
Attribute 11	61.489686	65.364829
Attribute 12	37.487598	40.817604

Table 3: Train and Test MSE for Linear regression over single attributes. Averaged over 20 runs.

(d) The next experiment was linear regression over all attributes together. We followed the exact same procedure as in the single attributes, except now we computed the weights using the entire training data matrix. We report the training and test MSE averaged over

20 runs for 20 random train/test splits as follows.

```
Average Train MSE = 21.414764727427052
Average Test MSE  = 25.922665880996654
```

It can be seen that our train and test MSE for linear regression over all attributes is smaller than the regressors for single attributes. This is expected as our models incorporated more features, allowing for better prediction of the output values.

1.3 Kernelized Ridge Regression

5. In this section, we perform kernel ridge regression over the training set with ℓ examples. This involved computing the dual solution $\boldsymbol{\alpha}^*$ that optimises the equation

$$\boldsymbol{\alpha}^* = \arg\min_{\alpha \in \mathbb{R}^\ell} \frac{1}{\ell} \sum_{i=1}^{\ell} (\sum_{j=1}^{\ell} \alpha_j K_{i,j} - y_i)^2 + \gamma \boldsymbol{\alpha}^T \mathbf{K} \boldsymbol{\alpha} \tag{1}$$

where $K(x_i, x_j)$ (abbreviated $K_{i,j}$) is the Gaussian kernel computed using x_i and x_j:

$$K(x_i, x_j) = \exp\left(-\frac{\| x_i - x_j \|^2}{2\sigma^2}\right) \tag{2}$$

$\mathbf{K}$ is an $(\ell \times \ell)$ matrix containing the entries $K_{i,j}$.

The dual solution is thus defined as

$$\boldsymbol{\alpha}^* = (\mathbf{K} + \gamma \ell \mathbf{I}_\ell)^{-1} \mathbf{y} \tag{3}$$

where γ is a regularization parameter, $\mathbf{I}_\ell$ is an $(\ell \times \ell)$ identity matrix and $\mathbf{y}$ is the vector of training outputs.

To evaluate the regression function on a test point, the formula is

$$y_{test} = \sum_{i=1}^{\ell} \alpha_i^* K(x_i, x_{test}) \tag{4}$$

(a) In order to perform kernel ridge regression, the data set was split into training and testing sets (2/3:1/3) using the splitting function. As can be seen in equations 1 and 2, there are two additional hyperparameters σ and γ. In order to select the best values for these hyperparameters, we first performed kernel ridge regression on the training set using five-fold cross-validation. The chosen range of values for σ and γ were $[2^7, 2^{7.5}, ..., 2^{13}]$ and $[2^{-40}, 2^{-39}, ..., 2^{-26}]$ respectively.

The following are the steps for 5-fold cross-validation on a single pair of (σ, γ). We performed this over all pairs of hyperparameters.

i. Shuffle the training set. (To reduce bias)
ii. Divide into K=5 groups.
iii. Select first group as the "validation" set and use the remaining K-1 groups as the training set.
iv. Compute the dual solution $\boldsymbol{\alpha}^*$ using the training set.
v. Compute predictions for validation data.
vi. Compute MSE for validation set and record value.
vii. Repeat steps (iii-vi) on all groups.

In order to perform steps (iv) and (v), three components were crucial: the computation of the kernel matrix $\mathbf{K}$, the dual solution $\boldsymbol{\alpha}^*$ and the evaluation function (4). The code for these three components are presented below. Note that with the computation of $\mathbf{K}$ and the evaluation function, the simple method would be to use for-loops. However, this can be computationally expensive, hence we developed a vectorized implementation, for which a detailed explanation is provided in Appendix 3.1.

```
def gaussian_kernel_matrix(X, sigma):
  """
  Computes the gaussian kernel matrix on a given set of vectors.

  Args
  ----
  X : NumPy array of data. Should be of shape (l, n) where l is number of examples and n is number of features
  sigma : sigma value of Gaussian kernel.

  Returns
  -------
  K : a symmetric positive definite matrix of shape (l, l)
  """

  #Number of samples
  l = X.shape[0]

  # Extract the diagonals of XX^T into a vector and stack it l times into lxl matrix B
  inner_product_vector = np.diagonal(X@X.T).reshape(-1, 1)

  #Compute matrix of just the dot products between x_i and x_j
  inner_product_matrix = inner_product_vector - 2 * (X@X.T) + inner_product_vector.T

  #Compute gaussian kernel matrix
  K = np.exp((-1/(2*sigma**2)) * inner_product_matrix)

  #Checkpoint to ensure matrix K is correct dimensions
  assert K.shape == (l, l), "K matrix should be of shape ({}, {})".format(l, l)

  return K

def kernel_ridge_regression(X, y, sigma, gamma):
  """
  Compute the solution alpha* for kernelized ridge regression.

```

```
    Args
    ----
    X : NumPy array of data. Should be of shape (l, n)
     where l is number of examples and n is number of
     input features
    y : NumPy array of target outputs. Should be of shape
      (l, 1)
    sigma : sigma value for Gaussian kernel
    gamma : regularization parameter for ridge regression

    Returns
    -------
    alpha_star : (l, 1) vector which represents the dual
     solution for ridge regression.

    """

    #Extract number of samples
    l = X.shape[0]

    #Compute kernel
    K = gaussian_kernel_matrix(X, sigma) #kernel_matrix(X
     , X, sigma=sigma)

    #Create identity matrix used in the formula
    I = np.eye(l)

    #Compute alpha_star value using formula
    try:
      alpha_star = np.linalg.inv(K + gamma*l*I) @ y
    except np.linalg.LinAlgError:
      alpha_star = np.linalg.pinv(K + gamma*l*I) @ y

    #Checkpoint to ensure the alpha is correct dimensions
    assert alpha_star.shape == (l, 1), "Alpha star should
      be of shape ({}, {})".format(l, 1)

    return alpha_star

def evaluate(alpha, X_tr, X_te, sigma):
    """
    Vectorized implementation of kernel ridge regression
     evaluation function on a test point.

    Args
```

```
    ----
    alpha : NumPy array of the dual solution alpha.
     Should be of shape (l, 1) where l is the number of
     training examples.
    X_tr : NumPy array of the training set inputs. Should
      be of shape (l, n)
    X_te : NumPy array of the test set inputs. Should be
     of shape (m, n) where m is number of test examples.
    sigma : Sigma value of the Gaussian kernel.

    Returns
    -------
    pred : NumPy array of predictions for the test data.
     Should be of shape (m, 1)
    """

    #Extract dimensions of data
    l = X_tr.shape[0]
    m = X_te.shape[0]

    #Evaluate regression model at all test points
    inner_product_matrix = np.diagonal(X_tr@X_tr.T).
     reshape(1, -1) - 2 * (X_te @ X_tr.T) + np.diagonal(
     X_te@X_te.T).reshape(-1, 1)
    K = np.exp((-1/(2*sigma**2)) * inner_product_matrix)
    pred = K @ alpha

    assert pred.shape == (m, 1), "Predictions should be
     of shape ({}, 1)".format(m, 1)

    return pred
```

The final implementation of the entire 5-fold cross-validation is as follows:

```
def train_with_kfoldCV(k, data, kernel_params, shuffle=
    True, verbose=True):
    """
    Performs k-fold cross validation on a given dataset.

    Args
    ----
    k - number of folds of cross-validation
    data - tuple of training input and output data.
     Training input must be NumPy array of shape (#
     examples, # features) while output must be of shape
```

```
    (# examples, 1)
    kernel_params - tuple of (sigma, gamma) parameters
     for computing the Gaussian kernel.
    shuffle - If True, then data is shuffled
    seed - to ensure reproducible results
    verbose - Set to True if you wish to see printed
     notifications.

    Returns
    -------
    cv_error - the validation set error averaged over the
      k folds of cross validation for the specified
     kernel parameters
    """

    #Extract data
    x, y = data

    #Extract gaussian kernel params
    sigma, gamma = kernel_params

    #Extract dimensions
    nSamples, nFeatures = x.shape

    #Shuffle dataset randomly
    if shuffle:
      perm = np.random.permutation(nSamples)
      x_shuffled = x[perm, :]
      y_shuffled = y[perm, :]
    else:
      x_shuffled = x
      y_shuffled = y

    # print("X_shuffled = ", x_shuffled)

    #Split data into k-groups
    x_groups = np.array_split(x_shuffled, k)
    y_groups = np.array_split(y_shuffled, k)

    #Stores the cross validation MSE which is the MSE
     calculated over the validation sets generated during
      each fold of k-fold CV
    cv_mse = 0

    #Iterate over each cross-validation group
```

```
  for i in range(len(x_groups)):

    #Use the selected group as "validation" set
    val_inputs, val_outputs = x_groups[i], y_groups[i]

    #Use rest of groups as training set
    train_inputs = np.vstack([x_groups[j] for j in range(len(x_groups)) if j != i])
    train_outputs = np.vstack([y_groups[j] for j in range(len(x_groups)) if j != i])

    #Fit model on train set
    alpha_star = kernel_ridge_regression(X=train_inputs, y=train_outputs, sigma=sigma, gamma=gamma)

    #Generate predictions on "validation" set
    val_predictions = evaluate(alpha=alpha_star, X_tr=train_inputs, X_te=val_inputs, sigma=sigma)

    #Compute validation errors
    cv_mse += (1/len(val_outputs)) * (val_outputs - val_predictions).T @ (val_outputs - val_predictions)
    if verbose:
      print(">> Cross-validation Fold {} Completed.".format(i+1))

  #Average the errors
  cv_error = cv_mse.item() / k

  if verbose:
    print(">> MEAN CROSS-VALIDATION ERROR = {}\n".format(cv_error))

  #Summarize performance using sample of model eval scores
  return cv_error

#Create parameter space
sigma_values = 2**np.arange(7, 13.5, 0.5)
gamma_values = (2**np.arange(40.0, 25.0, -1))**-1

#Represent all (sigma, gamma) pairs using meshgrid. This will be helpful when creating the 3D plot
gamma_array, sigma_array = np.meshgrid(gamma_values, sigma_values)
```

```
82
83 #This grid will store the cross validation error values
84 errors = np.zeros((sigma_values.shape[0], gamma_values.
       shape[0]))
85
86 #Split data into train and test set - for
       reproducibility, I've set a seed
87 np.random.seed(100000)
88 X_train, Y_train, X_test, Y_test = split_data(inputs=X
       [:, 1:], targets=Y, test_proportion=1/3, shuffle=
       True)
89 train_data = (X_train, Y_train)
90
91 #Train the model for all the different values of sigma
       and gamma
92 for i in range(errors.shape[0]):
93   for j in range(errors.shape[1]):
94
95     #Create the sigma and gamma pair
96     sigma = sigma_values[i]
97     gamma = gamma_values[j]
98
99     print("Sigma = {}, Gamma = {}".format(sigma, gamma)
       )
100
101    #Perform 5-fold cross validation. Note - Because we
       are adding a shuffle operation inside crossval
       function, every pair of sigma and gamma may train on
       different examples
102    #To ensure that each (sigma, gamma) pair is used on
       the same set of training and test examples on each
       fold, I've seed a seed
103    np.random.seed(130399)
104    cv_error = train_with_kfoldCV(k=5, data=train_data,
       kernel_params=(sigma, gamma), shuffle=True, verbose
       =True)
105
106    #Add this error value into our errors matrix
107    errors[i, j] = cv_error
108
109 print("Ridge Regression Completed!")
```

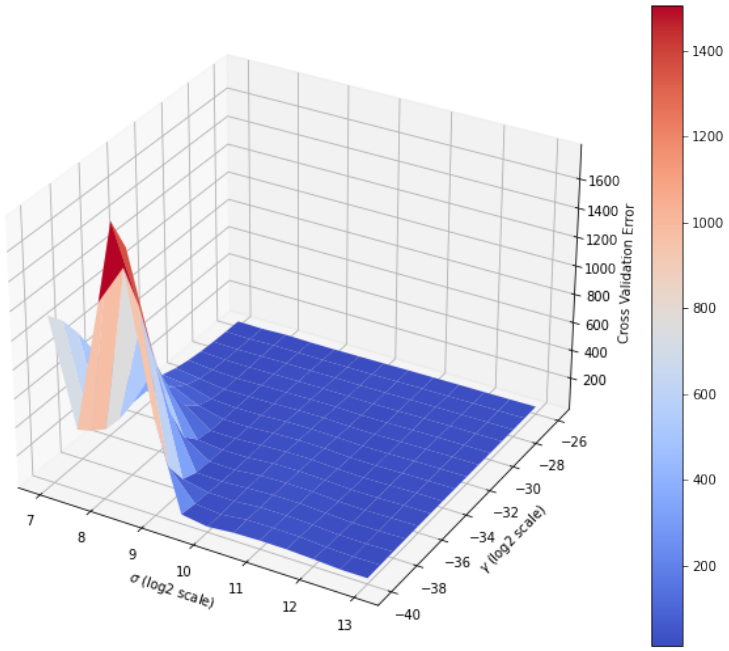


Figure 5: Surface plot of cross validation error as a function of σ and γ values. For convenience, the σ and γ values are plotted on a log-2 scale.

(b) The cross-validation error as a function of the σ and γ values is plotted as a surface plot in Figure 5.

(c) The best (σ, γ) pair was defined to be the one that had the least cross validation MSE. This was found to be $\sigma = 2^{9.5}$ and $\gamma = 2^{-36.0}$. Using these values, we again performed kernel ridge regression, this time using the training set and testing set. The train and test MSE for these hyperparameters is as follows:

```
Train MSE = 5.035038540967276
Test MSE  = 16.479302486525636
```

(d) Finally, all the different experiments from 4(a,c,d) and 5(a,c) were repeated for 20 random train/test splits to smooth out the results. For convenience, all the experiments for Naive Regression, Linear Regression and Kernel Ridge Regression were converted into functions, which are presented in Appendix 3.2 of the report.

The summarized results are presented in the following table.

Method	MSE Train	MSE Test
Naive Regression	83.517 ± 3.055	86.421 ± 6.143
attribute 1	69.581 ± 2.623	77.042 ± 5.180
attribute 2	72.068 ± 3.240	76.581 ± 6.554
attribute 3	63.221 ± 3.033	67.842 ± 6.142
attribute 4	81.172 ± 3.244	84.269 ± 6.907
attribute 5	67.346 ± 3.129	72.556 ± 6.382
attribute 6	44.215 ± 2.740	42.725 ± 5.437
attribute 7	70.518 ± 3.365	76.685 ± 6.933
attribute 8	77.569 ± 3.298	82.644 ± 6.792
attribute 9	70.002 ± 3.143	76.719 ± 6.460
attribute 10	63.563 ± 3.278	70.825 ± 6.664
attribute 11	61.310 ± 2.796	65.757 ± 5.560
attribute 12	37.638 ± 1.791	40.392 ± 3.651
all attributes	21.736 ± 1.472	25.097 ± 2.926
Kernel Ridge Regression	7.229 ± 1.206	13.738 ± 1.577

Table 4: Summarized results for all regression experiments averaged over 20 runs. Results reported as mean ± standard deviation.

2 PART II

2.1 Questions

6. Bayes estimator

(a) Let $\hat{y} \coloneqq f(x)$.
Deriving the optimal solution (Bayes estimator) f^*,

$$\mathcal{E}(f) \coloneqq \sum_{x \in X} \sum_{y \in Y} [y \neq f(x)] c_y p(x, y)$$

and $\mathcal{E}(f)$ is the expected error of an arbitrary predictor f.
Separating x and y:

$$\mathcal{E}(f) = \sum_{x \in X} [\sum_{y \in Y} [y \neq f(x)] c_y p(y|x)] p(x)$$

which can be re-written as:

$$\mathcal{E}(f) = \sum_{x \in X} [\sum_{y \neq f(x)} c_y p(y|x)] p(x)$$

Now, let us find the value of f^* at a specific point, x' by setting $x \coloneqq x'$. Then we have:

$$\begin{aligned}
\mathcal{E}(f(x')) &= [\sum_{y \neq f(x')} c_y p(y|x')] p(x') \\
&= \sum_{y \neq f(x')} c_y p(y|x') \qquad \text{since } p(x') \text{ is constant} \\
&= 1 - \sum_{y = f(x')} c_y p(y|x')
\end{aligned}$$

To find the minimum of $\mathcal{E}$, we can maximise $c_y p(y|x')$ i.e. argmax $c_y p(y|x')$.
Therefore,

$$f^* = \text{argmax } c_{y=f(x)} p(y = f(x)|x)$$

where the loss is 0 if we predict correctly and c_y otherwise.

(b) i.

$$L_F(y,\hat{y}) = \hat{y}^2 - y^2 + (y-\hat{y})2y = \hat{y}^2 - y^2 + 2y^2 - 2y\hat{y} = y^2 - 2y\hat{y} + \hat{y}^2 = (y-\hat{y})^2$$

This is clearly a square loss function.

ii. Proof of "⇒":
if $y = \hat{y}$,

$$L_F(\hat{y},\hat{y}) = F(\hat{y}) - F(\hat{y}) + (\hat{y}-\hat{y})F'(\hat{y}) = 0 + 0F'(\hat{y}) = 0$$

Hence, we have shown that $y = \hat{y} \rightarrow L_F(y,\hat{y}) = 0$.

Proof of "⇐":
if $L_F(y,\hat{y}) = 0$,

$$L_F(y,\hat{y}) = F(\hat{y}) - F(y) + (y-\hat{y})F'(y) = 0$$

$$\Rightarrow F(y) = F(\hat{y}) + (y-\hat{y})F'(y) \tag{1}$$

Since F is strictly convex,

$$F(\alpha p + (1-\alpha)q) < \alpha F(p) + (1-\alpha)F(q) \quad \forall p, q \in \mathbb{R}, \alpha \in (0,1)$$

$$\Rightarrow F(q + \alpha(p-q)) < F(q) + \alpha(F(p) - F(q))$$

$$\Rightarrow F(p) - F(q) > \frac{F(q+\alpha(p-q)) - F(q)}{\alpha}$$

As $\alpha \rightarrow 0$, we obtain:

$$F(p) - F(q) > F'(q)(p-q)$$

$$\Rightarrow F(p) > F'(q)(p-q) + F(q) \tag{2}$$

Replacing p, q with $y, \hat{y}$ respectively, we get:

$$F(y) > F'(\hat{y})(y-\hat{y}) + F(\hat{y})$$

Therefore, the only way 1 holds true is if $y = \hat{y}$:

$$F(\hat{y}) = 0F'(\hat{y}) + F(\hat{y}) = F(\hat{y})$$

Hence, we have shown that $L_F(y,\hat{y}) = 0 \rightarrow y = \hat{y}$.

iii. Replace p, q in equation 2 in 6(b)ii with $\hat{y}, y$ respectively. Then we get:

$$\begin{aligned} & F(\hat{y}) > F'(y)(\hat{y} - y) + F(y) \\ \Rightarrow\ & F(\hat{y}) - F'(y)(\hat{y} - y) - F(y) > 0 \\ \Rightarrow\ & F(\hat{y}) - F(y) + (y - \hat{y})F'(y) > 0 \\ \Rightarrow\ & L_F(y, \hat{y}) > 0 \end{aligned}$$

Thus, we have showed that $L_F(y, \hat{y}) > 0$.
Now, to show $L_F(y, \hat{y}) = 0$, we can use the proof from 6(b)ii that $L_F(y, \hat{y}) = 0$ when $y = \hat{y}$.
Therefore, we have proved that

$$L_F(y, \hat{y}) \geq 0 \qquad\qquad \forall y, \hat{y} \in \mathbb{R}$$

iv. Let $\hat{y} \coloneqq f(x)$.
Deriving the optimal solution (Bayes estimator) f^* in the continuous case,

$$\mathcal{E}(f) \coloneqq \int_{x \in X} \int_{y \in Y} [F(f(x)) - F(y) + (y - f(x))F'(y)]\, dP(x, y)$$

and $\mathcal{E}(f)$ is the expected error of an arbitrary predictor f.
Separating x and y:

$$\mathcal{E}(f) = \int_{x \in X} [\int_{y \in Y} [F(f(x)) - F(y) + (y - f(x))F'(y)]\, dP(y|x)]\, dP(x)$$

Now, let us find the value of f^* at a specific point, x' by setting $x \coloneqq x'$. Then we have:

$$\begin{aligned} \mathcal{E}(f(x')) &= [\int_{y \in Y} [F(f(x')) - F(y) + (y - f(x'))F'(y)]\, dP(y|x')]\, dP(x') \\ &= \int_{y \in Y} [F(f(x')) - F(y) + (y - f(x'))F'(y)]\, dP(y|x') \\ &\qquad\qquad \text{since } dP(x') \text{ is constant} \end{aligned}$$

Let $e \coloneqq \mathcal{E}(f(x'))$ and $z \coloneqq f(x')$. Replace "=" with "$\propto$".

$$e \propto \int_{y \in Y} [F(z) - F(y) + (y - z)F'(y)]\, dP(y|x')$$

To find the minimum, we differentiate e w.r.t. z:

$$\frac{\partial e}{\partial z} = \int_{y \in Y} [F'(z) - F'(y)]\, dP(y|x') \quad \text{using derivative rules}$$

Setting it equal to zero and solving,

$$\begin{aligned}
0 &= \int_{y \in Y} F'(z)\, dP(y|x') - \int_{y \in Y} F'(y)\, dP(y|x') \\
&= F'(z) - \int_{y \in Y} F'(y)\, dP(y|x') \\
&\qquad \text{since } F'(z) \text{ is independent of } dP(y|x') \text{ and so } \int_{y \in Y} dP(y|x') = 1 \\
\Rightarrow F'(z) &= \int_{y \in Y} F'(y)\, dP(y|x') \qquad (1) \\
\Rightarrow F(z) &= \int_{y \in Y} F'(y)\, dP(y|x') z + c \qquad \text{where } c \text{ denotes constant} \\
\Rightarrow z &= \frac{F(z) - c}{\int_{y \in Y} F'(y)\, dP(y|x')} \\
\Rightarrow z &= \frac{F(z) - c}{\int_{y \in Y} F'(y) p(y|x')\, dy}
\end{aligned}$$

and by using the conditional distribution function, we can find the Bayes estimator with respect to the the probability mass function $p(x, y)$:

$$f^* = \frac{F(f(x)) - c}{\sum_{y \in Y} F'(y) p(y|x)} \qquad \text{for some constant } c$$

In the special case of $F(x) = |x|^p$, using equation 1 and $F'(y) =$

$p|y|^{p-1}$:

$$
\begin{aligned}
p|z|^{p-1} &= \int_{y\in Y} p|y|^{p-1}\, dP(y|x') \\
\Rightarrow |z|^{p-1} &= \int_{y\in Y} |y|^{p-1}\, dP(y|x') \\
\Rightarrow |z| &= (\int_{y\in Y} |y|^{p-1}\, dP(y|x'))^{\frac{1}{p-1}} \\
\Rightarrow f^* &= (\sum_{y\in Y} |y|^{p-1} p(y|x))^{\frac{1}{p-1}}
\end{aligned}
$$

again by using the conditional distribution function

7. Kernel modification

   (a) By the theorem, K_c is positive semidefinite if and only if $K_c(\boldsymbol{x}, \boldsymbol{z}) = \langle \phi(x), \phi(z) \rangle$, where $x, z \in \mathbb{R}^n, c \in \mathbb{R}$.
   So assuming that $K_c(\boldsymbol{x}, \boldsymbol{z}) = \langle \phi(x), \phi(z) \rangle$, then we have:

$$
\begin{aligned}
\sum_{i,j=1}^{m} c_i c_j (c + \sum_{k=1}^{n} x_{i_k} x_{j_k}) &= c \sum_{i,j} c_i c_j + \sum_{i,j} \sum_{k} c_i c_j x_{i_k} x_{j_k} \\
&= \sum_{k} \sum_{i} c_i x_{i_k} \sum_{j} c_j x_{j_k} + c(\sum_{i} c_i \sum_{j} c_j) \\
&= \sum_{k} (\sum_{i} c_i x_{i_k})^2 + c(\sum_{i} c_i)^2
\end{aligned}
$$

   Since $\sum_k (\sum_i c_i x_{i_k})^2 \geq 0$ and $(\sum_i c_i)^2 \geq 0$, for the entire equation to be ≥ 0, c must be ≥ 0.
   Hence, K_c is a positive semidefinite kernel for $c \geq 0$. The proof for this has already been done above.

   (b) c in K_c acts like a bias term, b in linear regression equation, $\boldsymbol{y} = X\boldsymbol{w} + b$. It allows us to get the best fit offset from the origin. Without it, we are forcing the line to go through the origin which might not provide the best fit and bias other parameters.

8. Under a scenario in which m, a number of data is small, we select β to be a function of $\boldsymbol{x}_i \in \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$, and the test point $\boldsymbol{t}$:

$$\beta = \hat{\beta}(\boldsymbol{x}_i, \boldsymbol{t}) = \begin{cases} \infty & \text{if } \boldsymbol{x}_i \text{ is the nearest point to } \boldsymbol{t} \\ -\infty & \text{otherwise} \end{cases}$$

Here, the nearest point $\boldsymbol{x}_i$ to $\boldsymbol{t}$ can be found by calculating the distance between $\boldsymbol{x}_i$ and $\boldsymbol{t}$ i.e. $\|\boldsymbol{x}_i - \boldsymbol{t}\|$.

Argument:
One of differences between this trained linear classifier and 1-NN classifier is that the former considers all the data points whereas the latter considers one data point that is closest to the test point. To make the trained linear classifier behave like a 1-Nearest Neighbor classifier i.e. to make the trained linear classifier to take only one nearest data point to the test point into account, we want only one nearest neighbour to receive the highest weight and the rest to receive the lowest possible weight so that they would almost be 'ignored'. This is where the Gaussian kernel comes into play to act as that weight. Since the Gaussian kernel output depends on β, we can define β to return a value such that the kernel function output is the largest with an input, a closest data point to the test point and smallest with other inputs. Since the kernel function decays exponentially, we know that $exp(-\beta\|\boldsymbol{x} - \boldsymbol{t}\|^2) \to \infty$ as $-\beta\|\boldsymbol{x} - \boldsymbol{t}\|^2 \to -\infty$ and $exp(-\beta\|\boldsymbol{x} - \boldsymbol{t}\|^2) \to 0$ as $-\beta\|\boldsymbol{x} - \boldsymbol{t}\|^2 \to \infty$ which means that when $\boldsymbol{x}$ is the nearest neighbour to $\boldsymbol{t}$, β needs to return ∞ and $-\infty$ otherwise. Now, if we assume $\boldsymbol{x}_c$ is a nearest point to $\boldsymbol{t}$ then we would have $f(\boldsymbol{t}) \approx \alpha_c K_\beta(\boldsymbol{x}_c, \boldsymbol{t})$ for $c \in \{1, \ldots, m\}$ since $\alpha_i K_\beta(\boldsymbol{x}_c, \boldsymbol{t}) \approx 0$ for all $i \neq c$. So $sign(f(\boldsymbol{t}))$ would be equal to y_c.

9. Let us represent:

   - a n × n matrix, C, as a board configuration indicating non-empty holes as 1s and empty holes as 0s. This matrix will be initialised to an initial board configuration, G.
   - a holes hit as a n × n matrix, A, representing holes being hit in row i and column j. For example, given n = 4:

$$A_{23} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Given above, we can represent a n $\times$ n matrix, C', showing a new board configuration after one hit, A_{ij} to the previous board before A_{ij}, C:

$$C' = A_{ij} + C$$

where each entry is in modulo 2. This means that $1 + 1 = 2 = 0 \pmod 2$, which is correct as any moles currently appearing will hide after a hit. Similarly, $1 + 0 = 1 = 1 \pmod 2$ is also correct as new moles pop up when empty holes are hit.

This means that if a sequence of holes, that you can hit to empty the board, exists then there is a combination of matrices, when added to the initial board configuration, G, that produces a zero matrix representing an empty board:

$$G + \sum_{i,j=1}^{n} \alpha_{ij} A_{ij} = 0 \tag{1}$$

where $\alpha_{ij} \in \{0, 1\}$ is a coefficient deciding whether to hit the hole or not. Therefore, the task would be to find such a sequence of α_{ij} to determine whether the solution exists or not. In other words, a solution exists if there is a sequence of hit patterns to whack all the moles in G. Re-arranging the equation (1) above, we obtain:

$$G = -\sum_{i,j=1}^{n} \alpha_{ij} A_{ij} = \sum_{i,j=1}^{n} \alpha_{ij} A_{ij} \qquad \text{since -1 = 1 (mod 2)} \tag{2}$$

which can be expanded to:

$$G = \alpha_{11}\begin{pmatrix} 1 & 1 & 0 & \dots & 0 \\ 1 & 0 & 0 & \dots & 0 \\ 0 & & \dots & & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & \dots & & 0 \end{pmatrix} + \alpha_{12}\begin{pmatrix} 1 & 1 & 1 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & & \dots & & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & \dots & & 0 \end{pmatrix} + \dots$$

$$+ \alpha_{nn}\begin{pmatrix} 0 & & \dots & & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & \dots & & 0 \\ 0 & \dots & 0 & 0 & 1 \\ 0 & \dots & 0 & 1 & 1 \end{pmatrix}$$

Here, we can notice that each entry in G can be represented as a linear equation. For example:

$$G_{22} = \alpha_{11} + \alpha_{21} + \dots + \alpha_{32}$$

This tells us that we can represent G as a $n^2 \times 1$ vector of its entries as follows:

$$G = \begin{bmatrix} G_{11} \\ G_{12} \\ \vdots \\ G_{nn} \end{bmatrix}$$

Thus, the equation (2) can be re-written as a system of linear equations, $\boldsymbol{A}\vec{\alpha} = \vec{G}$:

$$\big[vec(A_{11})vec(A_{12})\dots vec(A_{nn})\big] \cdot \begin{bmatrix} \alpha_{11} \\ \alpha_{12} \\ \vdots \\ \alpha_{nn} \end{bmatrix} = \begin{bmatrix} G_{11} \\ G_{12} \\ \vdots \\ G_{nn} \end{bmatrix}$$

where $vec(A_{ij})$ denotes a vectorised matrix A_{ij}

To find $\vec{\alpha}$, we can use Gaussian Elimination (row reduction). Given $\vec{\alpha}$, to obtain the sequence of hit patterns, we can find each solution's (α_{ij}) corresponding A_{ij} where $\alpha_{ij} = 1$. Convert each A_{ij} to an integer labelling each hole on the board as shown in the question and we are done.

We can show that the algorithm is polynomial in n:
First, computing each A_{ij} in $\boldsymbol{A}$ takes $\mathrm{O}(n^2)$ and since there are n^2 number of A_{ij}s, overall, computing $\boldsymbol{A}$ takes $\mathrm{O}(n^4)$. Performing the Gaussian elimination takes $\mathrm{O}(n^6)$, which dominates the runtime of the entire algorithm. Hence, we have showed that the algorithm is polynomial in n.

3 Appendices

3.1 Vectorized Implementation of Kernel Ridge Regression

Here, we present the mathematical formulation for the vectorized implementation of kernel ridge regression. This allows us to avoid using for-loops, thus improving efficiency of training.

3.1.1 Kernel Matrix

We first vectorize the computation of the kernel matrix $\mathbf{K}$, which is required to compute the dual solution α^*. To do this, we begin with the original description of $\mathbf{K}$:

$$\mathbf{K} = \begin{bmatrix} K(x_1, x_1) & \cdots & K(x_1, x_\ell) \\ \vdots & \ddots & \vdots \\ K(x_\ell, x_1) & \cdots & K(x_\ell, x_\ell) \end{bmatrix}$$

$$K(x_i, x_j) = \exp\left(-\frac{\| x_i - x_j \|^2}{2\sigma^2}\right) = \exp\left(-\frac{\langle x_i - x_j, x_i - x_j \rangle}{2\sigma^2}\right)$$

Note that ℓ is the number of training examples, thus $\mathbf{K}$ is an $(\ell \times \ell)$ matrix. Using the NumPy framework, we can reformulate $\mathbf{K}$ as:

$$\mathbf{K} = \text{np.exp}\left(-\frac{1}{2\sigma^2}\begin{bmatrix} \langle x_1 - x_1, x_1 - x_1 \rangle & \cdots & \langle x_1 - x_\ell, x_1 - x_\ell \rangle \\ \vdots & \ddots & \vdots \\ \langle x_\ell - x_1, x_\ell - x_1 \rangle & \cdots & \langle x_\ell - x_\ell, x_\ell - x_\ell \rangle \end{bmatrix}\right)$$

The goal now is to breakdown the matrix in the above expression into a combination of simpler matrices. Note that a general inner product in this matrix can be expanded as follows:

$$\langle x_i - x_j, x_i - x_j \rangle = \langle x_i, x_i \rangle - 2\langle x_i, x_j \rangle + \langle x_j, x_j \rangle \tag{5}$$

We can use this property to redefine the matrix as follows:

$$\begin{bmatrix} \langle x_1 - x_1, x_1 - x_1 \rangle & \cdots & \langle x_1 - x_\ell, x_1 - x_\ell \rangle \\ \vdots & \ddots & \vdots \\ \langle x_\ell - x_1, x_\ell - x_1 \rangle & \cdots & \langle x_\ell - x_\ell, x_\ell - x_\ell \rangle \end{bmatrix} = \mathbf{B} - 2\mathbf{X}\mathbf{X}^T + \mathbf{B}^T \quad (6)$$

where

$$\mathbf{B} = \begin{bmatrix} \langle x_1, x_1 \rangle & \cdots & \langle x_1, x_1 \rangle \\ \vdots & \ddots & \vdots \\ \langle x_\ell, x_\ell \rangle & \cdots & \langle x_1, x_1 \rangle \end{bmatrix}, \mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_\ell^T \end{bmatrix} \quad (7)$$

Essentially what we have done is break down each element of the LHS of 6 into the 3 terms as per equation 5 and then the first, second and third terms are separated into individual matrices. The second matrix consists of the terms $-2\langle x_i - x_j, x_i - x_j \rangle$, which can be simplified to the matrix $-2\mathbf{X}\mathbf{X}^T$ where $\mathbf{X}$ is just the $(l \times n)$ training data matrix. It can also be noticed that the first and third matrices are just transposes (hence $\mathbf{B}$ and $\mathbf{B}^T$) of each other and contain the diagonal terms of the $\mathbf{X}\mathbf{X}^T$ matrix stacked ℓ times as rows or columns. When implementing in NumPy however, we could just represent $\mathbf{B}$ as a single column of the diagonals of $\mathbf{X}\mathbf{X}^T$. Due to broadcasting, we can expect that the vector and matrix additions will still work. Thus, we summarize the $\mathbf{K}$ reformulation as

$$\mathbf{K} = \text{np.exp}\left(-\frac{1}{2\sigma^2}(\mathbf{B} - 2\mathbf{X}\mathbf{X}^T + \mathbf{B}^T)\right)$$

3.1.2 Evaluation function

The next component is vectorizing the evaluation function. This function has the form

$$y_{test} = \sum_{i=1}^{l} \alpha_i^* K(x_i, x_{test})$$

where y_{test} is the predicted value for a test example x_{test}. Using the same logic as in the kernel matrix reformulation, we get

$$y_{test} = \mathbf{K}_{eval}\alpha^*$$

where

$$\mathbf{K}_{eval} = \text{np.exp}\left(-\frac{1}{2\sigma^2}(\mathbf{B}^T - 2\mathbf{X_{test}}\mathbf{X}^T + \mathbf{C})\right)$$

where $\mathbf{X}_{test}$ is an $(m \times n)$ testing data matrix, $\mathbf{B}^T$ is same as defined in 7 and $\mathbf{C}$ is defined as the diagonals of matrix $\mathbf{X_{test}}\mathbf{X_{test}}^T$. Note however that the $\mathbf{K_{eval}}$ matrix is not symmetric matrix like the kernel matrix; it is of dimensions $(m \times l)$.

3.2 Functions for regression experiments on Boston Housing Dataset

```
def naive_regression_eval(num_train_samples, num_test_samples
    , train_y, test_y, results_dict):
  """
  Performs Naive Regression on given data and saves results.

  Args
  ----
  num_train_samples : number of training samples. This will
   be used to create a ones vector for the training set.
  num_test_samples : number of testing samples. This will be
   used to create a ones vector for the testing set.
  train_y : NumPy array of training output values. Should be
   of shape (num_train_samples, 1)
  test_y : NumPy array of testing output values. Should be of
    shape (num_test_samples, 1)
  results_dict : Dictionary in which to save the results.

  Returns
  -------
  None.

  """

  #Add the key into the results dictionary
  key = "Naive Regression"
  if not key in results_dict.keys():
    results_dict[key] = {"Train MSE":[], "Test MSE": []}

  #Create a ones vector for train and test data
  train_ones = np.ones((num_train_samples, 1))
  test_ones = np.ones((num_test_samples, 1))
```

```
27
28    #Compute the weights using training set - in case we
       encounter singular matrices, Ive added the psuedoinverse
       as backup
29    try:
30      w = np.linalg.inv(train_ones.T @ train_ones) @ train_ones
       .T @ train_y
31    except np.linalg.LinAlgError:
32      w = np.linalg.pinv(train_ones.T @ train_ones) @
       train_ones.T @ train_y
33
34    #Add results into the dictionary
35    results_dict[key]["Train MSE"].append(MSE(train_ones,
       train_y, w).item())
36    results_dict[key]["Test MSE"].append(MSE(test_ones, test_y,
        w).item())
37
38    print(">> Naive Regression done.")
39
40  def linear_regression_eval(train_x, test_x, train_y, test_y,
       results_dict, single=False):
41    """
42    Performs linear regression on the given data and saves
       results.
43
44    Args
45    ----
46    train_x : NumPy array of training inputs. Should be of
       shape (m, n) where m is number of training examples and n
       is number of input features
47    test_x : NumPy array of test inputs. Should be of shape (1,
        n) where l is number of testing examples and n is number
       of input features
48    train_y : NumPy array of training outputs. Should be of
       shape (m, 1) where m is number of training examples
49    test_y : NumPy array of testing inputs. Should be of shape
       (l, 1) where l is number of training examples
50    results_dict : Dictionary in which to save the results.
51    single : Set to True if you wish to perform on single
       attributes.
52
53    Returns
54    -------
55    None.
56    """
```

```
57
58  if single:
59    num_features = train_x.shape[1] - 1
60
61    #Iterate over the features
62    for i in range(num_features):
63      #Create a key for the attribute and add it into the
     results dictionary if it doesn't exist
64      key = "attribute {}".format(i+1)
65      if not key in results_dict.keys():
66        results_dict[key] = {"Train MSE": [], "Test MSE": []}
67
68      #Pick out a the specific feature and the bias column
69      train_x_subset = train_x[:, [0, i+1]]
70      test_x_subset = test_x[:, [0, i+1]]
71
72      #Checkpoint to make sure the dimensions are correct
73      assert train_x_subset.shape == (train_x.shape[0], 2)
74
75      #Computed least squares solution - In case we encounter
     singular matrices, I have used the pseudoinverse as
     backup
76      try:
77        w = np.linalg.inv(train_x_subset.T @ train_x_subset)
     @ train_x_subset.T @ train_y
78      except np.linalg.LinAlgError:
79        w = np.linalg.pinv(train_x_subset.T @ train_x_subset)
     @ train_x_subset.T @ train_y
80
81      # Compute the train and test MSE and add it into the
     results_dict
82      results_dict[key]["Train MSE"].append(MSE(
     train_x_subset, train_y, w).item())
83      results_dict[key]["Test MSE"].append(MSE(test_x_subset,
     test_y, w).item())
84
85      print(">> Linear regression for {} done.".format(key))
86
87  else:
88    #Create a key for the attribute and add it into the
     results dictionary if it doesn't exist
89    key = "all attributes"
90    if not key in results_dict.keys():
91      results_dict[key] = {"Train MSE": [], "Test MSE": []}
92
```

```
93      #Compute the weights using training set - in case we
        encounter singular matrices, Ive added the psuedoinverse
        as backup
94      try:
95        w = np.linalg.inv(train_x.T @ train_x) @ train_x.T @
        train_y
96      except np.linalg.LinAlgError:
97        w = np.linalg.pinv(train_x.T @ train_x) @ train_x.T @
        train_y
98
99      #Compute the train and test MSE and add it into the
        results dictionary
100     results_dict[key]["Train MSE"].append(MSE(train_x,
        train_y, w).item())
101     results_dict[key]["Test MSE"].append(MSE(test_x, test_y,
        w).item())
102
103     print(">> Linear Regression for {} done.".format(key))
104
105 def ridge_regression_eval(train_x, train_y, test_x, test_y,
        sigma_values, gamma_values, results_dict):
106   """
107   Performs kernelized ridge regression on the given data and
        saves results.
108
109   Args
110   ----
111   train_x : NumPy array of training inputs. Should be of
        shape (m, n) where m is number of training examples and n
        is number of input features
112   test_x : NumPy array of test inputs. Should be of shape (l,
        n) where l is number of testing examples and n is number
        of input features
113   train_y : NumPy array of training outputs. Should be of
        shape (m, 1) where m is number of training examples
114   test_y : NumPy array of testing inputs. Should be of shape
        (l, 1) where l is number of training examples
115   sigma_values : NumPy 1D array of possible sigma values.
116   gamma_values : NumPY 1D array of possible gamma values.
117
118   Returns
119   -------
120   None.
121   """
122
```

```
#Generate a seed value which we will use inside the for loop.
seed=np.random.randint(100)

#Create a key for the attribute and add it into the results dictionary if it doesn't exist
key = "Kernel Ridge Regression"
if not key in results_dict.keys():
  results_dict[key] = {"Train MSE": [], "Test MSE": []}

#The best parameters will be saved in the order [best_sigma, best_gamma]
best_parameters = {"sigma":0, "gamma":0}

#Initialize the smallest error to be some large number. This will be used to update our best parameters
min_error = 10000000000000000000

#Iterate over entire parameter space
for i in range(len(sigma_values)):
  for j in range(len(gamma_values)):

    #Create the sigma and gamma pair
    sigma = sigma_values[i]
    gamma = gamma_values[j]

    #Perform 5-fold cross validation.
    # Note - The train_with_kfoldCV() function performs a kfoldCV for just one sigma and gamma.
    # In the function, I've added a shuffle operation BEFORE splitting the data into the k groups.
    # This means that everytime we use a new sigma and gamma, the group compositions may differ due to the shuffle.
    # We need to ensure that the group compositions after shuffling are the same everytime so that our parameters sigma and gamma are comparable.
    np.random.seed(seed)
    mean_cv_error = train_with_kfoldCV(k=5, data=(train_x, train_y), kernel_params=(sigma, gamma), shuffle=True, verbose=False)

    #If the mean_cv_error is smaller than the minimum error, then we can update our best parameters
    if mean_cv_error < min_error:
```

```
        min_error = mean_cv_error
        best_parameters['sigma'] = sigma
        best_parameters['gamma'] = gamma

  #Compute dual regression solution using training set
  alpha_star = kernel_ridge_regression(X=train_x, y=train_y,
   sigma=best_parameters['sigma'], gamma=best_parameters['
   gamma'])

  #Compute predictions on train set: The "test" points here
   are just the training set again
  train_y_pred = evaluate(alpha=alpha_star, X_tr=train_x,
   X_te=train_x, sigma=best_parameters['sigma'])
  #Compute predictions on test set
  test_y_pred = evaluate(alpha=alpha_star, X_tr=train_x, X_te
   =test_x, sigma=best_parameters['sigma'])

  #Compute train and test MSE
  train_MSE = (1/len(train_y)) * (train_y_pred - train_y).T @
    (train_y_pred - train_y)
  test_MSE = (1/len(test_y)) * (test_y_pred - test_y).T @ (
   test_y_pred - test_y)

  results_dict[key]["Train MSE"].append(train_MSE)
  results_dict[key]["Test MSE"].append(test_MSE)

  print(">> Kernel Ridge Regression done.\n")

#Initialize a final results dictionary which we shall update
   with all the errors
final_results = dict()

#Store number of runs here
runs = 20

#Generate a seed for reproducible results
np.random.seed(1753295)

#PROBLEM: INSPITE OF SETTING SEED HERE, THE FUNCTION IS
   GENERATING SAME TRAINING SET EVERYTIME!! HOW TO FIX IT?

#Iterate over runs
for i in range(runs):

  print("Run {}".format(i+1))
```

```
191
192   #Generate a fresh batch of train/test data
193   train_X, train_Y, test_X, test_Y = split_data(inputs=X,
       targets=Y, test_proportion=1/3, shuffle=True)
194   # print("Training set = ", train_X)
195
196   #PERFORM NAIVE REGRESSION AND UPDATE RESULTS
197   naive_regression_eval(num_train_samples=len(train_X),
       num_test_samples=len(test_X), train_y=train_Y, test_y=
       test_Y, results_dict=final_results)
198
199   #PERFORM LINEAR REGRESSION ON SINGLE ATTRIBUTES AND UPDATE
       RESULTS
200   linear_regression_eval(train_x=train_X, test_x=test_X,
       train_y=train_Y, test_y=test_Y, results_dict=final_results
       , single=True)
201
202   #PERFORM LINEAR REGRESSION ON ALL ATTRIBUTES AND UPDATE
       RESULTS
203   linear_regression_eval(train_x=train_X, test_x=test_X,
       train_y=train_Y, test_y=test_Y, results_dict=final_results
       , single=False)
204
205   #PERFORM KERNELIZED RIDGE REGRESSION ON ALL ATTRIBUTES AND
       UPDATE RESULTS
206   #Create parameter space
207   sigma_values = 2**np.arange(7, 13.5, 0.5)
208   gamma_values = (2**np.arange(40.0, 25.0, -1))**-1
209   # sigma_values = 2**np.arange(7, 8.0, 0.5)
210   # gamma_values = (2**np.arange(40.0, 38.0, -1))**-1
211   ridge_regression_eval(train_x=train_X[:, 1:], train_y=
       train_Y,
212                         test_x=test_X[:, 1:], test_y=test_Y,
213                         sigma_values=sigma_values,
214                         gamma_values=gamma_values,
215                         results_dict=final_results)
216
217 print("Evaluation Completed.")
```