

ECE521 Winter 2017: Assignment 1

FuYuan Tee, (999295837) * Chee Loong Soon, (999295793) †

February 8th, 2017

Contents

1	k-Nearest Neighbour	3
1.1	Geometry of k-NN	3
1.1.1	Describe 1D Dataset	3
1.1.2	Curse of Dimensionality	4
1.2	Euclidean Distance Function	6
1.2.1	Inner Product	6
1.2.2	Pairwise Distances	7
1.3	Making Predictions	8
1.3.1	Choosing Nearest Neighbour	8
1.3.2	Prediction	9
1.4	Soft kNN and Gaussian Processes	13
1.4.1	Soft Decisions and Gaussian Process Regression	15
1.4.2	Conditional Distribution of a Gaussian	16
2	Linear and Logistic Regression	19
2.1	Geometry of Linear Regression	19
2.1.1	Convex Function	19

*Equal Contribution (50%), fuyuan.tee@mail.utoronto.ca

†Equal Contribution (50%), cheeloong.soon@mail.utoronto.ca

2.1.2	DeNormalization	23
2.1.3	Regularization	26
2.1.4	Binary Classifiers for Multi-class Classification	26
2.2	Stochastic Gradient Descent	28
2.2.1	Tuning Learning Rate, η	28
2.2.2	Mini-batch Size	30
2.2.3	Generalization	33
3	Appendices	35
3.1	Entire Code 1: Chee Loong Soon's version	35
3.2	Entire Code 2: FuYuan Tee's version	51
3.2.1	Question 1: k-Nearest Neighbour	51
3.2.2	Question 2.2: Stochastic Gradient Descent	58
3.2.3	Question 2.2.1: Tuning the learning rate	61
3.2.4	Question 2.2.2: Effect of the mini-batch size	66
3.2.5	Question 2.2.3: Generalization	73

1 k-Nearest Neighbour

1.1 Geometry of k-NN

1.1.1 Describe 1D Dataset

An example of a 1-D dataset with two classes in which k-NN produces an accuracy that is periodic with k is illustrated in Figure 1. The data point comes from the training set itself. The data points are equally distant from each adjacent data point.

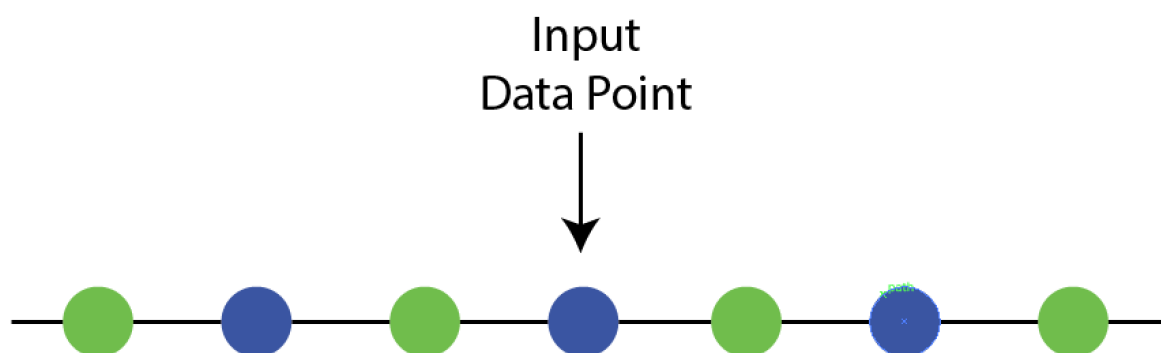


Figure 1: 1-D Dataset Illustration with classification accuracy that is periodic to k .

For such a dataset, the classification accuracy of the data point can be summarised in the Table 1 below. The accuracy follows a periodic function of $50\% \cdot \sin\left(\frac{\pi}{2}k\right) + 50\%$.

Table 1: k and prediction accuracy for two periods								
k	1	2	3	4	5	6	7	8
Prediction Accuracy (%)	100	50	0	50	100	50	0	50

1.1.2 Curse of Dimensionality

Proving equation 1,

$$\mathbf{var}\left(\frac{\|x^{(i)} - x^{(j)}\|_2^2}{\mathbf{E}[\|x^{(i)} - x^{(j)}\|_2^2]}\right) = \frac{N+2}{N} - 1 \quad (1)$$

We can utilise equation 2 from Probability Theory,

$$\mathbf{var}[x] = \mathbf{E}[x^2] - \mathbf{E}[x]^2 \quad (2)$$

Below are the given equations,

$$x \in \mathbb{R}^n \quad (3)$$

$$\Pr(X) \sim \prod_{n=1}^N \mathcal{N}(x_n, 0 \sigma^2) \quad (4)$$

where n represents the n^{th} dimension.

N represents the number of training data.

$$d_n = x_n^i - x_n^j \quad (5)$$

where i represents the i^{th} training data.

j represents the j^{th} training data.

$$\Pr(d_n) \sim \mathcal{N}(d_n; 0, 2\sigma^2) \quad (6)$$

$$\mathbf{E}[d_n^2 d_m^2] = \mathbf{E}[d_n^2] \mathbf{E}[d_m^2] \quad (7)$$

$$\mathbf{E}[d_n^4] = 3(\sqrt{2}\sigma)^4 = 12\sigma^4 \quad (8)$$

From equations 4 and 2, it is implied that

$$\mathbf{E}[x_n] = 0 \quad (9)$$

$$\mathbf{var}[x_n] = \sigma^2 = \mathbf{E}[x_n^2] \quad (10)$$

From equations 6 and 2, it is implied

$$\mathbf{E}[d_n] = 0 \quad (11)$$

$$\text{var}[d_n] = 2\sigma^2 = \mathbf{E}[d_n^2] \quad (12)$$

From equations 7 and 12,

$$\mathbf{E}[d_n^2 d_m^2] = \mathbf{E}[d_n^2] \mathbf{E}[d_m^2] = (2\sigma^2)(2\sigma^2) = 4\sigma^4 \quad (13)$$

From equation 1 and 5,

$$\|x^{(i)} - x^{(j)}\|_2^2 = \sum_{n=1}^N (x_n^{(i)} - x_n^{(j)})^2 = \sum_{n=1}^N d_n^2 \quad (14)$$

Substituting equations 1, 14 into 2,

$$\text{var} \left(\frac{\sum_{n=1}^N d_n^2}{\mathbf{E}[\sum_{n=1}^N d_n^2]} \right) = \mathbf{E} \left[\left(\frac{\sum_{n=1}^N d_n^2}{\mathbf{E}[\sum_{n=1}^N d_n^2]} \right)^2 \right] - \mathbf{E} \left[\left(\frac{\sum_{n=1}^N d_n^2}{\mathbf{E}[\sum_{n=1}^N d_n^2]} \right) \right]^2 \quad (15)$$

Looking into the first term of the Right Hand Side (RHS) of equation 15,

$$\begin{aligned} \mathbf{E} \left[\left(\frac{\sum_{n=1}^N d_n^2}{\mathbf{E}[\sum_{n=1}^N d_n^2]} \right)^2 \right] &= \mathbf{E} \left[\left(\frac{\sum_{n=1}^N d_n^2}{\sum_{n=1}^N \mathbf{E}[d_n^2]} \right)^2 \right] = \mathbf{E} \left[\left(\frac{\sum_{n=1}^N d_n^2}{\sum_{n=1}^N 2\sigma^2} \right)^2 \right] \\ &= \mathbf{E} \left[\left(\frac{\sum_{n=1}^N d_n^2}{2N\sigma^2} \right)^2 \right] = \mathbf{E} \left[\left(\frac{\sum_{n=1}^N d_n^2}{2N\sigma^2} \right)^2 \right] = \mathbf{E} \left[\frac{(\sum_{n=1}^N d_n^2)^2}{(2N\sigma^2)^2} \right] \\ &= \mathbf{E} \left[\frac{(\sum_{n=1}^N d_n^2)^2}{4N^2\sigma^4} \right] = \mathbf{E} \left[\frac{\sum_{n=1}^N \sum_{m=1}^N d_n^2 d_m^2}{4N^2\sigma^4} \right] = \mathbf{E} \left[\frac{\sum_{n=1}^N \sum_{m=1}^N d_n^2 d_m^2}{4N^2\sigma^4} \right] \\ &= \frac{\mathbf{E}[\sum_{n=1}^N \sum_{m=1}^N d_n^2 d_m^2]}{4N^2\sigma^4} \\ &= \frac{\mathbf{E}[\sum_{n=1}^N d_n^4 + 2 \sum_{n=1}^{N-1} \sum_{m=n+1}^N d_n^2 d_m^2]}{4N^2\sigma^4} \\ &= \frac{\mathbf{E}[\sum_{n=1}^N d_n^4] + \mathbf{E}[2 \sum_{n=1}^{N-1} \sum_{m=n+1}^N d_n^2 d_m^2]}{4N^2\sigma^4} \\ &= \frac{\sum_{n=1}^N \mathbf{E}[d_n^4] + \mathbf{E}[2 \sum_{n=1}^{N-1} \sum_{m=n+1}^N d_n^2 d_m^2]}{4N^2\sigma^4} \\ &= \frac{\sum_{n=1}^N 12\sigma^4 + 2 \sum_{n=1}^{N-1} \sum_{m=n+1}^N \mathbf{E}[d_n^2 d_m^2]}{4N^2\sigma^4} \\ &= \frac{12N\sigma^4 + 2 \sum_{n=1}^{N-1} \sum_{m=n+1}^N 4\sigma^4}{4N^2\sigma^4} \\ &= \frac{12N\sigma^4 + 2 \binom{N}{2} 4\sigma^4}{4N^2\sigma^4} = \frac{12N\sigma^4 + 2 \left(\frac{N(N-1)}{2} \right) 4\sigma^4}{4N^2\sigma^4} = \frac{4N^2\sigma^4 + 8N\sigma^4}{4N^2\sigma^4} = \frac{N+2}{N} \end{aligned} \quad (16)$$

Looking into the second term of the RHS of equation 15,

$$\begin{aligned}
\mathbf{E} \left[\left(\frac{\sum_{n=1}^N d_n^2}{\mathbf{E}[\sum_{n=1}^{\infty} d_n^2]} \right) \right]^2 &= \mathbf{E} \left[\left(\frac{\sum_{n=1}^N d_n^2}{\sum_{n=1}^N \mathbf{E}[d_n^2]} \right) \right]^2 = \mathbf{E} \left[\left(\frac{\sum_{n=1}^N d_n^2}{\sum_{n=1}^N 2\sigma^2} \right) \right]^2 \\
&= \mathbf{E} \left[\left(\frac{\sum_{n=1}^N d_n^2}{2N\sigma^2} \right) \right]^2 = \left(\frac{\mathbf{E}[\sum_{n=1}^N d_n^2]}{2N\sigma^2} \right)^2 = \left(\frac{\sum_{n=1}^N \mathbf{E}[d_n^2]}{2N\sigma^2} \right)^2 \\
&= \left(\frac{N2\sigma^2}{2N\sigma^2} \right)^2 = 1^2 = 1
\end{aligned} \tag{17}$$

Therefore, combining both terms from the RHS of equation 15 as calculated in equations 16 and 17 results in

$$\frac{N+2}{N} - 1 \tag{18}$$

which proves equation 1.

To show that equation 18 vanishes as $N \rightarrow \infty$,

$$\lim_{N \rightarrow \infty} \left(\frac{N+2}{N} - 1 \right) = \lim_{N \rightarrow \infty} \left(\frac{N+2-N}{N} \right) = \lim_{N \rightarrow \infty} \left(\frac{2}{N} \right) = 0 \tag{19}$$

This proves that the variance vanishes which means that a test data point will be equally close to all training examples in a high dimensional space.

1.2 Euclidean Distance Function

1.2.1 Inner Product

All input vectors have same magnitude in the training set. $\|x^{(1)}\|_2^2 = \dots = \|x^{(M)}\|_2^2$

To show that in order to find nearest neighbor of a test point x^* among the training set, it is sufficient to just compare and rank the negative inner product between the training and the test data, $x^{(M)T} x^*$.

In a 2-Dimensional case, if all input vectors have the same magnitude, that defines a circle in the training set. Taking the inner product between two vectors, $x^{(M)T} x^*$ would simply be calculating the angle between the two vectors. This is similar to performing

a cosine similarity calculation. Therefore, ranking based on negative inner product would be looking for the smallest angle between any 2 input vectors.

Now, extending this intuition to a M-Dimensional case. It is a M-dimensional hypersphere where all the vectors extend to the surface of the hypersphere as they have the same magnitude. Taking the negative inner product, $x^{(M)^T} x^*$ and ranking them would be finding the minimum angle between any of the 2 points on the surface, which is its nearest neighbor.

This can be illustrated below in equation 20, where R is the radius of the hypersphere and all vectors have a magnitude equal to this radius.

$$\begin{aligned}
 \|x^{(*)}\|_2^2 &= \sum_{n=1}^N (x_n^{(*)})^2 = R^2 \\
 \|x^{(1)}\|_2^2 &= \sum_{n=1}^N (x_n^{(1)})^2 = R^2 \\
 &\vdots \\
 \|x^{(M)}\|_2^2 &= \sum_{n=1}^N (x_n^{(M)})^2 = R^2
 \end{aligned} \tag{20}$$

1.2.2 Pairwise Distances

Code snippets are included below:

```

1 {python}
2 # 1.2 Euclidean Distance Function
3 # 1.2.2 Pairwise Distances
4 # Write a vectorized Tensorflow Python function that
   ↪ implements
5 # the pairwise squared Euclidean distance function
6 # for two input matrices.
7 # No Loops and makes use of Tensorflow broadcasting.
8 def PairwiseDistances(X, Z):
9     """

```

```

10     input:
11         X is a matrix of size (B x N)
12         Z is a matrix of size (C x N)
13     output:
14         D = matrix of size (B x C) containing
15         the pairwise Euclidean distances
16     """
17     B = X.get_shape().as_list()[0]
18     N = X.get_shape().as_list()[1]
19     C = Z.get_shape().as_list()[0]
20     # Ensure the N dimensions are consistent
21     assert N == Z.get_shape().as_list()[1]
22     # Reshape to make use of broadcasting in Python
23     X = tf.reshape(X, [B, 1, N])
24     Z = tf.reshape(Z, [1, C, N])
25     # The code below automatically does broadcasting.
26     # Calculates the
27     # pairwise squared Euclidean distance function
28     D = tf.reduce_sum(tf.square(tf.sub(X, Z)), 2)
29     return D

```

1.3 Making Predictions

1.3.1 Choosing Nearest Neighbour

Code snippets are included below:

```

1 {python}
2 # 1.3 Making Predictions
3 # 1.3.1 Choosing nearest neighbours

```



```

4  # Write a vectorized Tensorflow Python function that
5  # takes a pairwise distance matrix
6  # and returns the responsibilities of the
7  # training examples to a new test data point.
8  # It should not contain loops.
9  # Use tf.nn.top_k
10 def ChooseNearestNeighbours(D, K):
11     """
12     input:
13         D is a matrix of size (B x C)
14         K is the top K responsibilities for each test input
15     output:
16         topK are the value of the squared distances
17         for the top K values
18         indices are the index of the location of
19         these top K squared distances.
20     """
21     # Take topK of negative distances since
22     # closer data ranks higher.
23     topK, indices = tf.nn.top_k(tf.neg(D), K)
24     return topK, indices

```

1.3.2 Prediction

Code snippets are included below:

```

1  {python}
2  # 1.3.2 Prediction
3  # Compute the k-NN prediction with K = {1, 3, 5, 50}
4  # For each value of K, compute and report:

```

```

5      # training MSE loss
6      # validation MSE loss
7      # test MSE loss
8  # Choose best k using validation error = 50
9  def PredictKnn(trainData , testData, trainTarget, testTarget,
10 ↪ K) :
11      """
12      input:
13          trainData: Data for training KNN
14          testData: Data used in testing
15          trainTarget: Targets used to create prediction.
16          testTarget: Targets used to calculate loss.
17      output:
18          loss: The mean squared loss of the prediction.
19      """
20      D = PairwiseDistances(testData, trainData)
21      topK, indices = ChooseNearestNeighbours(D, K)
22      # Select the proper outputs to be averaged
23      # from the target values and average them
24      trainTargetSelectedAveraged = tf.reduce_mean( \
25          tf.gather(trainTarget, indices), 1)
26      # Calculate the loss from the actual values
27      loss = tf.reduce_mean(tf.square(tf.sub( \
28          trainTargetSelectedAveraged, testTarget)))
29      return loss
30
31  # Plot the prediction function for x = [0, 11]
32  def PredictedValues(x, trainData, trainTarget, K):
33      """
34      Plot the predicted values

```

```

34     input:
35         x = test target to plot and predict
36     """
37     D = PairwiseDistances(x, trainData)
38     topK, indices = ChooseNearestNeighbours(D, K)
39     predictedValues = tf.reduce_mean( \
40         tf.gather(trainTarget, indices), 1)
41     return predictedValues

```

Table 2: KNN and Loss

k	Training MSE Loss	Validation MSE Loss	Test MSE Loss
1	0.000	0.272	0.311
3	0.105	0.326	0.145
5	0.119	0.310	0.178
50	1.248	1.229	0.707

The best value of k is based on one that gives the lowest validation MSE loss. In this case, the best k is found to be $k = 1$.

By inspecting the plot, a k value of 3 or 5 would be picked, instead of $k = 1$. The reason for this is simply because $k = 1$ is overfitting the training data, and would have modeled noises in the data. This is confirmed when comparing between the Test MSE losses for different values of k .

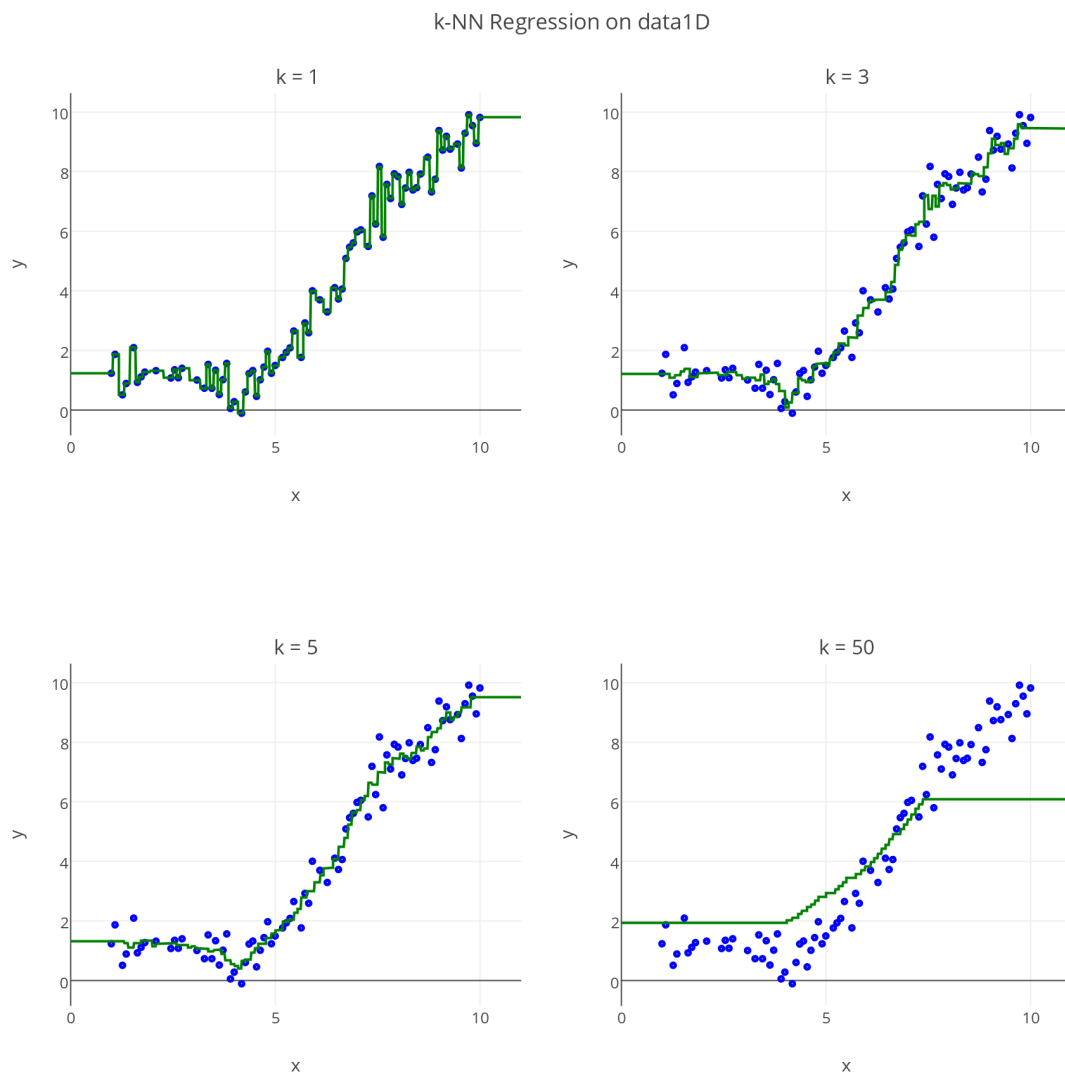


Figure 2: k-NN regression on data1D for various values of k

1.4 Soft kNN and Gaussian Processes

As shown in Table 3 , the Soft Decision performs better than the Gaussian Process Regression Model as it has a lower Test Mean Squared Error Loss. The algorithm was run on the test set.

Table 3: Loss on Test Set	
<i>Algorithm</i>	Test MSE Loss
Soft Decision	0.159
Gaussian Process Regression	0.380

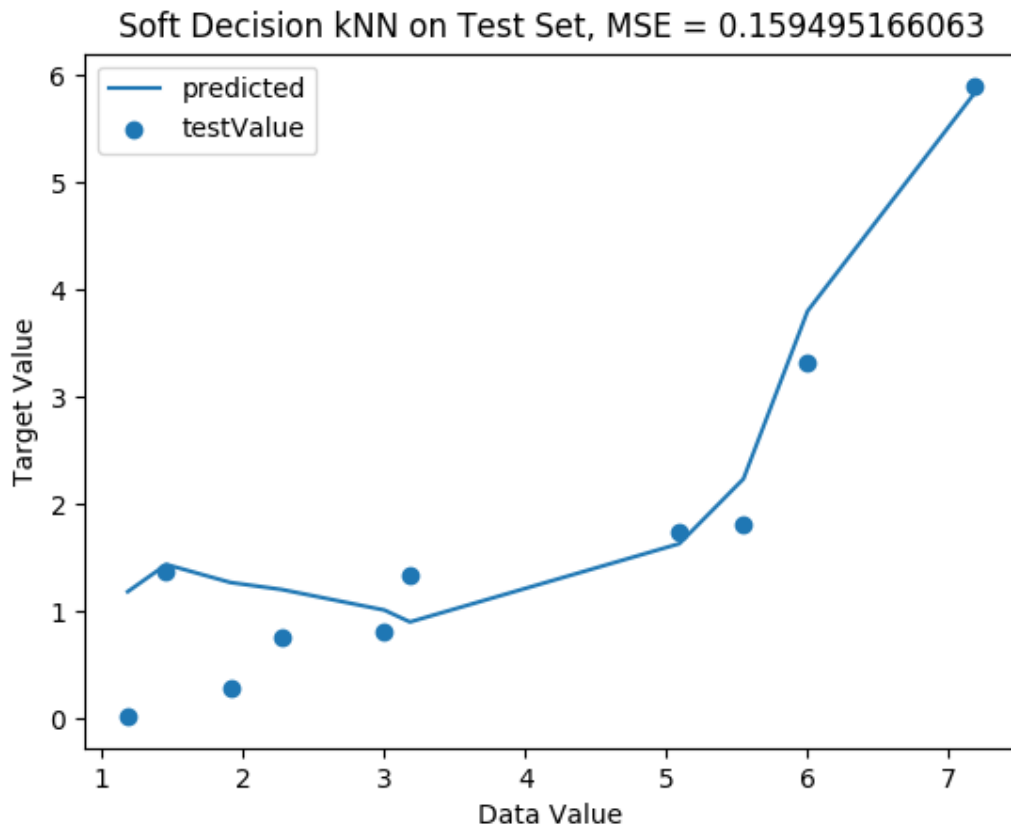


Figure 3: Soft Decision kNN on Test Set

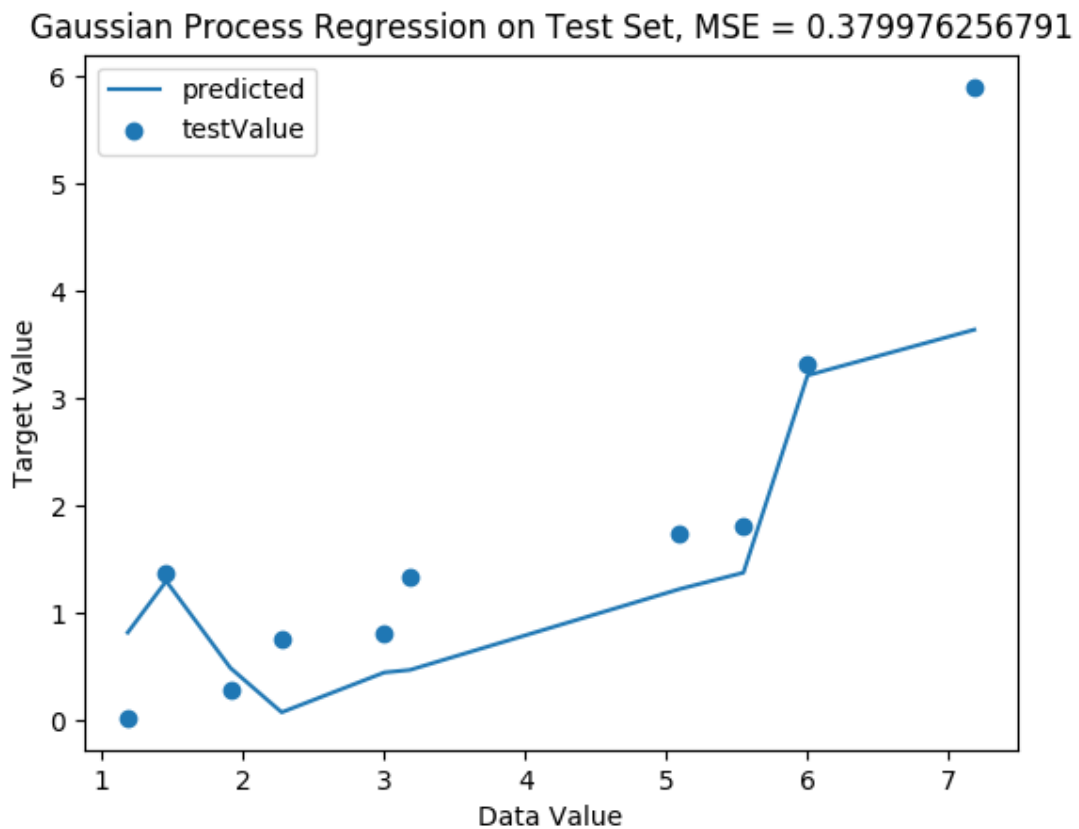


Figure 4: Gaussian Process Regression on Test Set

```

1 {python}
2 def SortData(inputVal, outputVal):
3     """
4     This sorts a given test set by the dataValue before
5     ↪ plotting it.
6     """
7     # Sort across the data values on both pairs of sets
8     # Sort in numpy itself to not lose precision.
9     p = np.argsort(inputVal, axis=0)
10    inputVal = np.array(inputVal)[p]
11    outputVal = np.array(outputVal)[p]

```

```

11     # Get rid of extra dimensions from np.argsort
12     inputVal = inputVal[:, :,0]
13     outputVal = outputVal[:, :,0]
14     return inputVal, outputVal

```

1.4.1 Soft Decisions and Gaussian Process Regression

```

1 {python}
2 # Predict values using soft decision
3 # 1.4.1.1 Soft Knn Decision
4
5 def PredictedValuesSoftDecision(x, trainData, trainTarget):
6     # use hyper parameter of 100 as given by Jimmy.
7     hyperParam = 100
8     # Compute pairwise differences.
9     D1 = PairwiseDistances(x, trainData)
10    K1 = tf.exp(-hyperParam*D1)
11    # Get the sum term used for normalization
12    sum1 = tf.reduce_sum(tf.transpose(K1), axis=0)
13    # Reshape to enable broadcasting during division.
14    N = sum1.get_shape().as_list()[0]
15    sum1 = tf.reshape(sum1, [N,1])
16    # Normalize the data using broadcast
17    # to calculate the final responsibility values
18    rStar = tf.div(K1, sum1)
19    # Calculate the predicted value
20    # using the new responsibilities, rStar
21    predictedValues = tf.matmul(rStar,trainTarget)
22    return predictedValues

```

```

1 {python}
2 # Predict values using Gaussian
3 # 1.4.1.1 Gaussian Processes
4 def PredictedValuesGaussianProcesses(x, trainData,
   ↪ trainTarget):
5     # use hyper parameter of 100 as given by Jimmy.
6     hyperParam = 100
7     D1 = PairwiseDistances(x, trainData)
8     K1 = tf.exp(-hyperParam*D1)
9     D2 = PairwiseDistances(trainData, trainData)
10    K2 = tf.matrix_inverse(tf.exp(-hyperParam*D2))
11    # Calculate the responsibilities, rStar
12    # by normalizing using the inverse of K2.
13    rStar = tf.matmul(K1, K2)
14    # Calculate the predicted value
15    # using the new responsibilities, rStar
16    predictedValues = tf.matmul(rStar, trainTarget)
17    return predictedValue

```

1.4.2 Conditional Distribution of a Gaussian

Given an $M + 1$ Gaussian random vector:

$$\mathbf{y} = \begin{bmatrix} y^* \\ y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \sim \mathcal{N}(\mathbf{0}, \Sigma) \quad (21)$$

Splitting the vector into two parts as follows and describing the resulting distribution using stacked block notation:

$$\mathbf{y} = \begin{bmatrix} y^* \\ \mathbf{y}_{train} \end{bmatrix} \sim \mathcal{N} \left(\begin{bmatrix} 0 \\ \mathbf{0} \end{bmatrix}, \begin{bmatrix} \Sigma_{y^*y^*} & \Sigma_{y^*\mathbf{y}_{train}} \\ \Sigma_{\mathbf{y}_{train}y^*} & \Sigma_{\mathbf{y}_{train}\mathbf{y}_{train}} \end{bmatrix} \right) \quad (22)$$

$$\sim \mathcal{N} \left(\begin{bmatrix} 0 \\ \mathbf{0} \end{bmatrix}, \Sigma = \Lambda^{-1} = \begin{bmatrix} \Lambda_{y^*y^*} & \Lambda_{y^*\mathbf{y}_{train}} \\ \Lambda_{\mathbf{y}_{train}y^*} & \Lambda_{\mathbf{y}_{train}\mathbf{y}_{train}} \end{bmatrix}^{-1} \right) \quad (23)$$

Given that \mathbf{y}_{train} is observed, find $P(y^*|\mathbf{y}_{train}) \sim \mathcal{N}(y^*; \mu^*, \Sigma^*)$.

By completing the squares on the quadratic term of the exponent for the multivariate Gaussian equation,

$$\begin{aligned} & -\frac{1}{2}(\mathbf{y} - \boldsymbol{\mu})^T \Sigma^{-1}(\mathbf{y} - \boldsymbol{\mu}) \\ &= -\frac{1}{2}\mathbf{y}^T \Sigma^{-1} \mathbf{y} \\ &= -\frac{1}{2} \begin{bmatrix} y^{*T} & \mathbf{y}_{train}^T \end{bmatrix} \begin{bmatrix} \Lambda_{y^*y^*} & \Lambda_{y^*\mathbf{y}_{train}} \\ \Lambda_{\mathbf{y}_{train}y^*} & \Lambda_{\mathbf{y}_{train}\mathbf{y}_{train}} \end{bmatrix} \begin{bmatrix} y^* \\ \mathbf{y}_{train} \end{bmatrix} \\ &= -\frac{1}{2} \begin{bmatrix} y^{*T} \Lambda_{y^*y^*} + \mathbf{y}_{train} \Lambda_{\mathbf{y}_{train}y^*} & y^{*T} \Lambda_{y^*\mathbf{y}_{train}} + \mathbf{y}_{train} \Lambda_{\mathbf{y}_{train}\mathbf{y}_{train}} \end{bmatrix} \begin{bmatrix} y^* \\ \mathbf{y}_{train} \end{bmatrix} \\ &= -\frac{1}{2} (y^{*T} \Lambda_{y^*y^*} y^* + \mathbf{y}_{train}^T \Lambda_{\mathbf{y}_{train}y^*} y^* + y^{*T} \Lambda_{y^*\mathbf{y}_{train}} \mathbf{y}_{train} + \mathbf{y}_{train}^T \Lambda_{\mathbf{y}_{train}\mathbf{y}_{train}} \mathbf{y}_{train}) \end{aligned} \quad (24)$$

Since $\mathbf{y}_{train}^T \Lambda_{\mathbf{y}_{train}y^*} y^*$ is a scalar and $\Lambda_{\mathbf{y}_{train}y^*}^T = \Lambda_{y^*\mathbf{y}_{train}}$,

$$(\mathbf{y}_{train}^T \Lambda_{\mathbf{y}_{train}y^*} y^*)^T = y^{*T} \Lambda_{y^*\mathbf{y}_{train}} \mathbf{y}_{train} \quad (25)$$

Hence, equation 24 simplifies to:

$$\begin{aligned} & -\frac{1}{2} (y^{*T} \Lambda_{y^*y^*} y^* + 2y^{*T} \Lambda_{y^*\mathbf{y}_{train}} \mathbf{y}_{train} + \mathbf{y}_{train}^T \Lambda_{\mathbf{y}_{train}\mathbf{y}_{train}} \mathbf{y}_{train}) \\ &= -\frac{1}{2} y^{*T} \Lambda_{y^*y^*} y^* - y^{*T} \Lambda_{y^*\mathbf{y}_{train}} \mathbf{y}_{train} - \frac{1}{2} \mathbf{y}_{train}^T \Lambda_{\mathbf{y}_{train}\mathbf{y}_{train}} \mathbf{y}_{train} \end{aligned} \quad (26)$$

Completing the square for a multivariate Gaussian quadratic term with $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ yields:

$$\begin{aligned} & -\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1}(\mathbf{x} - \boldsymbol{\mu}) \\ &= -\frac{1}{2}\mathbf{x}^T \Sigma^{-1} \mathbf{x} + \mathbf{x}^T \Sigma^{-1} \boldsymbol{\mu} - \frac{1}{2}\boldsymbol{\mu}^T \Sigma^{-1} \boldsymbol{\mu} \end{aligned} \quad (27)$$

Given that \mathbf{y}_{train} and $\Sigma_{\mathbf{y}_{train}\mathbf{y}_{train}}$ are known, the conditional Gaussian Distribution $P(y^*|\mathbf{y}_{train})$ can be inferred by performing a term-by-term comparison between the terms of equation 26 and those of equation 27.

Comparing terms that are of second order with respect to y^* ,

$$\Sigma^* = \Lambda_{y^*\mathbf{y}_{train}}^{-1} \quad (28)$$

Comparing terms that are of first order with respect to y^* ,

$$\begin{aligned} \Sigma^{*-1}\mu^* &= \Lambda_{y^*\mathbf{y}_{train}}\mathbf{y}_{train} \\ \mu^* &= \Sigma^*\Lambda_{y^*\mathbf{y}_{train}}\mathbf{y}_{train} \\ &= \Lambda_{y^*y^*}^{-1}\Lambda_{y^*\mathbf{y}_{train}}\mathbf{y}_{train} \end{aligned} \quad (29)$$

Using the matrix-inverse identity provided in Tutorial 3 (pg. 41), the terms $\Lambda_{y^*y^*}$ and $\Lambda_{y^*\mathbf{y}_{train}}$ can be expressed using terms in $\Sigma = \begin{bmatrix} \Sigma_{y^*y^*} & \Sigma_{y^*\mathbf{y}_{train}} \\ \Sigma_{\mathbf{y}_{train}y^*} & \Sigma_{\mathbf{y}_{train}\mathbf{y}_{train}} \end{bmatrix}$,

$$\begin{aligned} \Lambda_{y^*y^*} &= (\Sigma_{y^*y^*} - \Sigma_{y^*\mathbf{y}_{train}}\Sigma_{\mathbf{y}_{train}\mathbf{y}_{train}}^{-1}\Sigma_{\mathbf{y}_{train}y^*})^{-1} \\ \Lambda_{y^*\mathbf{y}_{train}} &= -(\Sigma_{\mathbf{y}_{train}\mathbf{y}_{train}} - \Sigma_{\mathbf{y}_{train}y^*}\Sigma_{y^*y^*}^{-1}\Sigma_{y^*\mathbf{y}_{train}})^{-1}\Sigma_{y^*\mathbf{y}_{train}}\Sigma_{\mathbf{y}_{train}\mathbf{y}_{train}}^{-1} \end{aligned} \quad (30)$$

The results above allow equations 28 and 29 to be expressed in terms of the original Σ block terms:

$$\mu^* = -\Sigma_{y^*\mathbf{y}_{train}}\Sigma_{\mathbf{y}_{train}\mathbf{y}_{train}}^{-1}\mathbf{y}_{train} \quad (31)$$

$$\Sigma^* = \Sigma_{y^*y^*} - \Sigma_{\mathbf{y}_{train}y^*}^T\Sigma_{\mathbf{y}_{train}\mathbf{y}_{train}}^{-1}\Sigma_{\mathbf{y}_{train}y^*} \quad (32)$$

$$\therefore P(y^*|\mathbf{y}_{train}) \sim \mathcal{N}(-\Sigma_{y^*\mathbf{y}_{train}}\Sigma_{\mathbf{y}_{train}\mathbf{y}_{train}}^{-1}\mathbf{y}_{train}, \Sigma_{y^*y^*} - \Sigma_{\mathbf{y}_{train}y^*}^T\Sigma_{\mathbf{y}_{train}\mathbf{y}_{train}}^{-1}\Sigma_{\mathbf{y}_{train}y^*}) \quad (33)$$

2 Linear and Logistic Regression

2.1 Geometry of Linear Regression

2.1.1 Convex Function

Need to show if equation 34 is a convex function of W using Jensen Inequality given by equation 35.

M is the total number of training data. N is the number of dimension for each training data.

$$\begin{aligned}\mathcal{L} &= \mathcal{L}_{\mathcal{D}} + \mathcal{L}_{\mathcal{W}} \\ &= \sum_{m=1}^M \frac{1}{2M} \|W^T x^{(m)} + b - y^{(m)}\|_2^2 + \frac{\lambda}{2} \|W\|_2^2 \\ &= \sum_{m=1}^M \frac{1}{2M} \left[\sum_{n=1}^N (W_n x_n^{(m)}) + b - y^{(m)} \right]^2 + \frac{\lambda}{2} W^T W \\ &= \sum_{m=1}^M \frac{1}{2M} [(W^T x^{(m)} + b - y^{(m)})^2] + \frac{\lambda}{2} W^T W\end{aligned}\tag{34}$$

$$f(\alpha W_1 + (1 - \alpha)W_2) \leq \alpha f(W_1) + (1 - \alpha)f(W_2)\tag{35}$$

Since the sum of two convex function is convex, we can prove each sum term, $\mathcal{L}_{\mathcal{D}}$ and $\mathcal{L}_{\mathcal{W}}$ on equation 34 separately.

You can easily prove that the sum of two convex is still convex by summing both sides of Jensen's Inequality for both convex functions and showing that Jensen's Inequality would still hold, indicating that the sum of the convex functions is still convex. This enables us to simplify the analysis by proving that each term is convex separately

As a convex function divided by a positive value is still a convex function, we can ignore the division by $2M$ since $M > 0$ means that $\frac{1}{2M} > 0$. Similarly, $\frac{\lambda}{2} \geq 0$ and it only scales $W^T W$. You can divide both sides of Jensen's inequality by a positive constant and the Jensen's inequality would still hold, implying that the function is still

convex when divided by a positive constant. This enables us to simplify the analysis by ignoring the positive constants.

This means we just have to prove that equation 36 is convex and equation 37 is convex. By doing so, we would have proven that equation 34 is convex.

$$\mathcal{L}_{\mathcal{D}} = [W_n^T x_n^{(m)} + b_n - y_n^{(m)}]^2 \quad (36)$$

$$\mathcal{L}_{\mathcal{W}} = W^T W \quad (37)$$

2.1.1.1 Proof of Convexity of $\mathcal{L}_{\mathcal{D}}$ with respect to W

Rearranging equation 35 for $\mathcal{L}_{\mathcal{D}}$,

$$\mathcal{L}_{\mathcal{D}}(\alpha W_1 + (1 - \alpha)W_2) - \alpha \mathcal{L}_{\mathcal{D}}(W_1) - (1 - \alpha) \mathcal{L}_{\mathcal{D}} \leq 0 \quad (38)$$

$$\begin{aligned} LHS &\equiv \mathcal{L}_{\mathcal{D}}[\alpha W_1 + (1 - \alpha)W_2] - \alpha \mathcal{L}_{\mathcal{D}}(W_1) - (1 - \alpha) \mathcal{L}_{\mathcal{D}}(W_2) \\ &= \{[\alpha W_1 + (1 - \alpha)W_2]^T x + (b - y)\}^T \{[\alpha W_1 + (1 - \alpha)W_2]^T x + (b - y)\} \\ &\quad - \alpha [W_1^T x + (b - y)]^T [W_1^T x + (b - y)] \\ &\quad - (1 - \alpha) [W_2^T x + (b - y)]^T [W_2^T x + (b - y)] \end{aligned} \quad (39)$$

$$\begin{aligned} &= [x^T(\alpha W_1 + (1 - \alpha)W_2) + (b - y)^T] \{[\alpha W_1 + (1 - \alpha)W_2]^T x + (b - y)\} \\ &\quad - \alpha [x^T W_1 + (b - y)^T] [W_1^T x + (b - y)] \\ &\quad - (1 - \alpha) [x^T W_2 + (b - y)^T] [W_2^T x + (b - y)] \end{aligned} \quad (40)$$

$$\begin{aligned} &= \{x^T[\alpha W_1 + (1 - \alpha)W_2][\alpha W_1^T + (1 - \alpha)W_2^T]x \\ &\quad + 2x^T[\alpha W_1 + (1 - \alpha)W_2](b - y) + (b - y)^T(b - y)\} \\ &\quad - \alpha [x^T W_1 W_1^T x + 2x^T W_1(b - y) + (b - y)^T(b - y)] \\ &\quad - (1 - \alpha) [x^T W_2 W_2^T x + 2x^T W_2(b - y) + (b - y)^T(b - y)] \end{aligned} \quad (41)$$

Rearranging similar terms together,

$$\begin{aligned}
&= \{x^T [\alpha^2 W_1 W_1^T + 2\alpha(1-\alpha)W_1^T W_2 + (1-\alpha)^2 W_2^T W_2] x \\
&\quad - \alpha(x^T W_1 W_1^T x) - (1-\alpha)(x^T W_2 W_2^T x)\} \\
&\quad + \{2\alpha x^T W_1(b-y) - 2\alpha x^T W_1(b-y)\} \\
&\quad + \{2(1-\alpha)x^T W_2(b-y) - 2(1-\alpha)x^T W_2(b-y)\} \\
&\quad + \{1-\alpha - (1-\alpha)\}(b-y)^T(b-y) \quad (42)
\end{aligned}$$

$$\begin{aligned}
&= (\alpha^2 - \alpha)x^T W_1 W_1^T x + 2\alpha(1-\alpha)x^T W_1^T W_2 x + [(1-\alpha)^2 - (1-\alpha)]x^T W_2 W_2^T x \\
&= -\alpha(1-\alpha)x^T W_1 W_1^T x + 2\alpha(1-\alpha)x^T W_1^T W_2 x - \alpha(1-\alpha)x^T W_2 W_2^T x \\
&= -\alpha(1-\alpha)[x^T W_1 W_1^T x - 2x^T W_1^T W_2 x + x^T W_2 W_2^T x] \quad (43) \\
&= -\alpha(1-\alpha)[W_1^T x - W_2^T x]^T [W_1^T x - W_2^T x] \\
&\leq 0 \equiv RHS
\end{aligned}$$

Equation 43 is less than or equal to zero as $-\alpha(1-\alpha) \leq 0$; $\forall \alpha \in [0, 1]$. Furthermore, the remaining quadratic term, $[W_1^T x - W_2^T x]^T [W_1^T x - W_2^T x] \geq 0$ since it is a square of the term $\forall (W_1^T x - W_2^T x) \in \mathbb{R}^N$.

Hence, it has been shown that $\mathcal{L}_{\mathcal{D}}$ is convex.

2.1.1.2 Proof of Convexity of $\mathcal{L}_{\mathcal{W}}$ to W

From the Left Hand Side of Equation 38,

$$\begin{aligned}
LHS &\equiv \mathcal{L}_{\mathcal{W}}[\alpha W_1 + (1 - \alpha)W_2] - \alpha \mathcal{L}_{\mathcal{W}}(W_1) - (1 - \alpha) \mathcal{L}_{\mathcal{W}}(W_2) \\
&= [\alpha W_1 + (1 - \alpha)W_2]^T [\alpha W_1 + (1 - \alpha)W_2] - \alpha W_1^T W_1 - (1 - \alpha)W_2^T W_2 \\
&= [\alpha W_1^T + (1 - \alpha)W_2^T] [\alpha W_1 + (1 - \alpha)W_2] - \alpha W_1^T W_1 - (1 - \alpha)W_2^T W_2 \\
&= [\alpha^2 W_1^T W_1 + 2\alpha(1 - \alpha)W_1^T W_2 + (1 - \alpha)^2 W_2^T W_2] - \alpha W_1^T W_1 - (1 - \alpha)W_2^T W_2 \\
&= (\alpha^2 - \alpha)x^T W_1 W_1^T x + 2\alpha(1 - \alpha)x^T W_1^T W_2 x + [(1 - \alpha)^2 - (1 - \alpha)]x^T W_2 W_2^T x \\
&= -\alpha(1 - \alpha)W_1^T W_1 + 2\alpha(1 - \alpha)W_1^T W_2 - \alpha(1 - \alpha)W_2 W_2^T \\
&= -\alpha(1 - \alpha)[W_1^T W_1 - 2W_1^T W_2 x + W_2^T W_2] \\
&= -\alpha(1 - \alpha)[W_1 - W_2]^T [W_1 - W_2] \\
&\leq 0 \equiv RHS
\end{aligned} \tag{44}$$

Using a similar argument as that for Section 2.1.1.1, the loss function $\mathcal{L}_{\mathcal{W}}$ is shown to be convex with respect to W .

Therefore, the loss function \mathcal{L} is a convex function with respect to W .

2.1.1.3 Proof of Convexity of \mathcal{L} to b

Instead of performing a similar proof for b , the bias can be thought of as the $(N + 1)$ th dimension of the weight. Thus, $b = W_{N+1}$ can be grouped together with matrix W , while the vector x will be expanded to have $x_{N+1}^m = 1 \forall m$, as shown in equation 45. Using the proof for the convexity of \mathcal{L} with respect to W , by extension \mathcal{L} is a convex function of the bias, b .

$$L = \sum_{m=1}^M \frac{1}{2M} \sum_{n=1}^{N+1} (W_n^T x_n^{(m)} - y^{(m)})^2 + \frac{\lambda}{2} W^T W \tag{45}$$

2.1.2 DeNormalization

Assuming $\lambda = 0$ from equation 34 we get equation 46.

$$\sum_{m=1}^M \frac{1}{2M} [(W^T x^{(m)} + b - y^{(m)})^2] \quad (46)$$

The original optimal weights and optimal bias are optimal to the non-transformed dataset. This means that the $\frac{\partial L}{\partial W} = 0$ and $\frac{\partial L}{\partial b} = 0$. More specifically, the partial gradient of the loss in equation 46 with respect to a specific weight W_n ,

$$\frac{\partial L}{\partial W_n} = \sum_{m=1}^M \frac{1}{M} \left(\sum_{i=1, i \neq n}^N W_i x_i^m + W_n x_n^m + b - y^m \right) x_n^m = 0 \quad (47)$$

and for the bias, b

$$\frac{\partial L}{\partial b} = \sum_{m=1}^M \frac{1}{M} \left(\sum_{i=1, i \neq n}^N W_i x_i^m + W_n x_n^m + b - y^m \right) = 0 \quad (48)$$

A single dimension of x scales by $\alpha > 1$ and shifts by $\beta > 1$. This is the same as de-normalizing the data point instead of normalizing it which normally deducts each data point by the mean and scaled by its variance. The reason for normalizing is to prevent any component from dominating the sum and to prevent the weights from learning the high bias that is not needed for prediction. As this is the opposite of normalization, it ends up training slower. However, this will not change the global minimum value as will be shown below.

Expanding equation 47 to account for the sum of M training cases, we find equation 49.

$$\frac{\partial L}{\partial W_n} = \frac{1}{M} \left(\sum_{i=1, i \neq n}^N W_i (x_i^1 + \dots + x_i^M) + W_n (x_n^1 + \dots + x_n^M) + Mb - (y^1 + \dots + y^M) \right) (x_n^1 + \dots + x_n^M) = 0 \quad (49)$$

which we can rewrite in simpler terms as equation 50

$$\frac{\partial L}{\partial W_n} = \frac{1}{M} \left(\sum_{i=1, i \neq n}^N W_i \mathbf{x}_i + W_n(\mathbf{x}_n) + Mb - \mathbf{y} \right)(\mathbf{x}_n) = 0 \quad (50)$$

Since equation 50 is equal to 0 for any value of \mathbf{x}_n , we must have that the inner term, $(\sum_{i=1, i \neq n}^N W_i \mathbf{x}_i + W_n(\mathbf{x}_n) + Mb - \mathbf{y})$ is equal 0.

This is shown clearly in equation 51.

$$\left(\sum_{i=1, i \neq n}^N W_i \mathbf{x}_i + W_n(\mathbf{x}_n) + Mb - \mathbf{y} \right) = 0 \quad (51)$$

As the model changes by scaling by a positive constant of α and shifted by a positive constant of β within the square term, it does not change the minimum value as it is these transformations happen within the square term. As a result, the minimum value remains at 0 which is the lowest value for any square term. Therefore, the new global minimum value of the transformed convex loss function will remain the same compare to the original loss function global minimum.

To illustrate, lets re-write the original loss function from equation 46 to account for the transformed Loss Function in equation 52.

$$L' = \sum_{m=1}^M \frac{1}{2M} \left[\left(\sum_{i=1, i \neq n}^N W_i^i x_i^{m'} + W'_n(x_n^m) + Mb' - y^m \right)^2 \right] \quad (52)$$

where the new variables are appended with a ' to show that their values could or has changed. This can be re-written as equation 53.

$$L' = \frac{1}{2M} \left[\left(\sum_{i=1, i \neq n}^N W'_i \mathbf{x}_i + W'_n(\mathbf{x}'_n) + Mb' - \mathbf{y} \right)^2 \right] \quad (53)$$

Taking the gradient with respect to W results in equation

54.

$$\begin{aligned}\frac{\partial L'}{\partial W_n} &= \frac{1}{M} \left[\left(\sum_{i=1, i \neq n}^N W'_i \mathbf{x}_i + W'_n(\mathbf{x}'_n) + Mb' - \mathbf{y} \right) \mathbf{x}'_n \right] \\ &= \frac{1}{M} \left[\left(\sum_{i=1, i \neq n}^N W'_i \mathbf{x}_i + W'_n(\alpha \mathbf{x}_n + \beta) + Mb' - \mathbf{y} \right) (\alpha \mathbf{x}_n + \beta) \right]\end{aligned}\quad (54)$$

To achieve the global minimum, a possible solution assignment would be to re-substitute in the original values for $W'_i = W_i \forall i \neq n$ and $W'_n = \frac{W_n}{\alpha}$ into equation 54 as shown in equation 55.

$$\begin{aligned}\frac{\partial L'}{\partial W_n} &= \frac{1}{M} \left[\left(\sum_{i=1, i \neq n}^N W_i \mathbf{x}_i + \frac{W_n}{\alpha} (\alpha \mathbf{x}_n + M\beta) + Mb' - \mathbf{y} \right) (\alpha \mathbf{x}_n + M\beta) \right] \\ &= \frac{1}{M} \left[\left(\sum_{i=1, i \neq n}^N W_i \mathbf{x}_i + W_n \mathbf{x}_n + \frac{W_n M\beta}{\alpha} + Mb' - \mathbf{y} \right) (\alpha \mathbf{x}_n + M\beta) \right] \\ &= \frac{1}{M} \left[\left(\sum_{i=1, i \neq n}^N W_i \mathbf{x}_i + W_n \mathbf{x}_n + M \left(\frac{W_n \beta}{\alpha} + b' \right) - \mathbf{y} \right) (\alpha \mathbf{x}_n + M\beta) \right]\end{aligned}\quad (55)$$

Further setting $b' = b - \frac{W_n \beta}{\alpha}$ results and using the result from equation 51, we get equation 56.

$$\begin{aligned}\frac{\partial L'}{\partial W_n} &= \frac{1}{M} \left[\left(\sum_{i=1, i \neq n}^N W_i \mathbf{x}_i + W_n \mathbf{x}_n + M \left(\frac{W_n \beta}{\alpha} + b - \frac{W_n \beta}{\alpha} \right) - \mathbf{y} \right) (\alpha \mathbf{x}_n + M\beta) \right] \\ &= \frac{1}{M} \left[\left(\sum_{i=1, i \neq n}^N W_i \mathbf{x}_i + W_n \mathbf{x}_n + Mb - \mathbf{y} \right) (\alpha \mathbf{x}_n + M\beta) \right] \\ &= \frac{1}{M} [(0)(\alpha \mathbf{x}_n + M\beta)] \\ &= 0\end{aligned}\quad (56)$$

Hence, from the final equation in 56, the partial gradient is 0, suggesting that this assignment results in an optimal assignment of W' and b' .

As this assignment results in a similar inner term of equation 51, this suggest that the new loss function will end up being equal to the old loss function. Hence, the global minimum remains the same.

The proof that this assignment works on bias is very similar and is omitted.

From this assignment of W' and b' , it shows that the weight vector W will move downwards to reach W' and similarly for b to b' . The optimal weights and bias after learning will be lower compared to the optimal weights and biased learned from the non-transformed original data.

2.1.3 Regularization

The new minimum value will increase as the weight and bias will fit to the regularize model more and less to the training set. This means the loss with respect to the training set only will increase as the regularization penalizes large values of W . However, this regularization helps the prediction to be more robust on the validation and the test set by preventing the model from over-fitting to the training set.

The large weights will reduce as they are regularized whereas the small weights will increase. The bias should remain the same as it is not affected by the regularized term.

The new minimum loss will increase as it is being compared to the training set labels, but is being optimize for both the training set and the regularized term.

2.1.4 Binary Classifiers for Multi-class Classification

Given $D > 2$ and only able to use binary classifiers. A method to solve a multi-class classification task using a number of binary classifiers would be to assign a binary classifier to each class in the D classes (see Figure 5).

Each binary classifier would predict if a given test input belong to a specific class. This assumes that each test input can belong to more than 1 class as it is not constrained to belong to more than 1 class or no classes at all from this design.

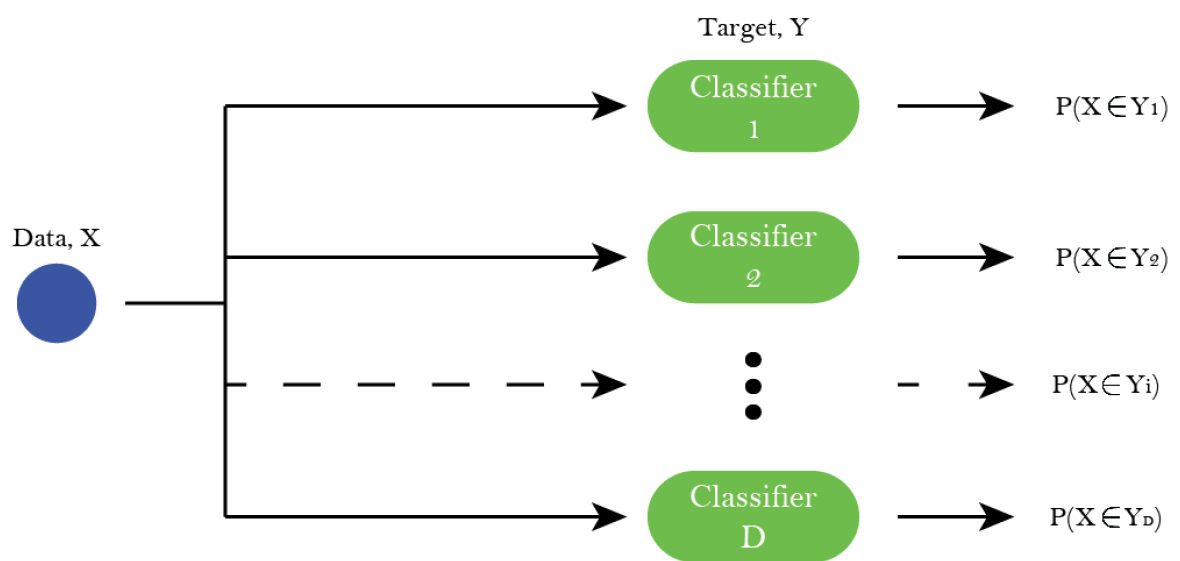


Figure 5: D binary classifiers for multi-class classification

2.2 Stochastic Gradient Descent

Before proceeding with our explanations for the subsequent sections, we would like to clarify on the convergence logic we selected for training our '3/5' digit classifier.

The model weights are randomly initialised. In our model, training convergence is implemented through early stopping. In our case, it is defined to be when the average validation MSE for the latest epoch is more than 0.99 of the average validation MSE for the previous epoch.

The logic is that when the average relative learnings between epochs is less than 1%, the training is terminated prematurely for shorter computation time. This makes sense since the model is not learning much from the features.

Training convergence also happens when the average MSE of one epoch is larger than its predecessor. This is done since an increase in the average validation MSE is a sign of over-fitting.

2.2.1 Tuning Learning Rate, η

Our programmed definition of convergence can be found in Section 2.2.

The values η was generated based on increasing orders of 1 (see Table 4). Higher values of η lead to fewer number of updates to reach convergence. However, from trial and error, it was discovered that there is an upper bound for η to ensure training convergence. When the value of η is more than 0.1 (i.e. 1, 10, etc.), the training does not converge but diverges instead as it overshoots past the minimum value.

From our experiments, The best value for η is selected to be 0.1 as it takes the lowest number of iterations to reach the lowest convergence value. The results also summarised in Figure 6.

Table 4: Number of updates until convergence for various values of η

η	Number of Updates
0.001	1863
0.01	337
0.1	57
> 0.3	N/A

Training performance for best learning rates, $\eta = 0.1$

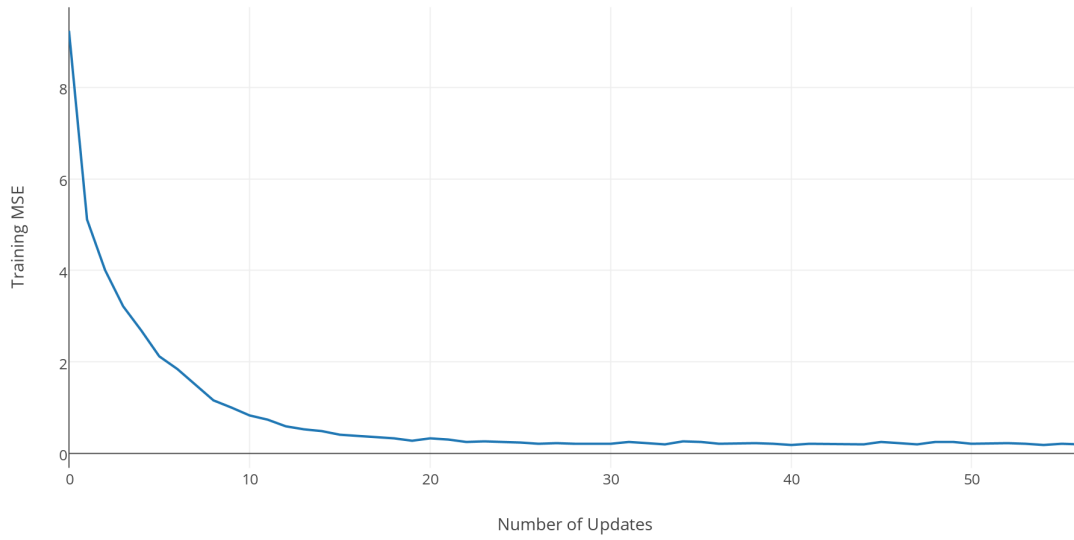


Figure 6: Graph of training mean squared error (MSE) against number of updates for the best learning rate found, $\eta = 0.1$. ($B = 50, \lambda = 1$)

2.2.2 Mini-batch Size

Our programmed definition of convergence can be found in Section 2.2.

From Table 5, the same pattern for η is observed where higher values of η lead to fewer number of updates. Meanwhile, there is an optimum value for batch size, B . The best mini-batch size is $B = 100$, which leads to the fewest number of updates, regardless of η . However, when the $B = 700$, early-stopping fails to occur in a reasonable time.

Please refer to Figures 7 and 8 for more information.

Table 5: Number of updates until training convergence for various values of η and B

B	η		
	0.001	0.01	0.1
10	2871	491	211
50	1963	337	57
100	1114	274	43
700	6001	6001	6001

Model training performance for various batch size, B , and learning rate, η

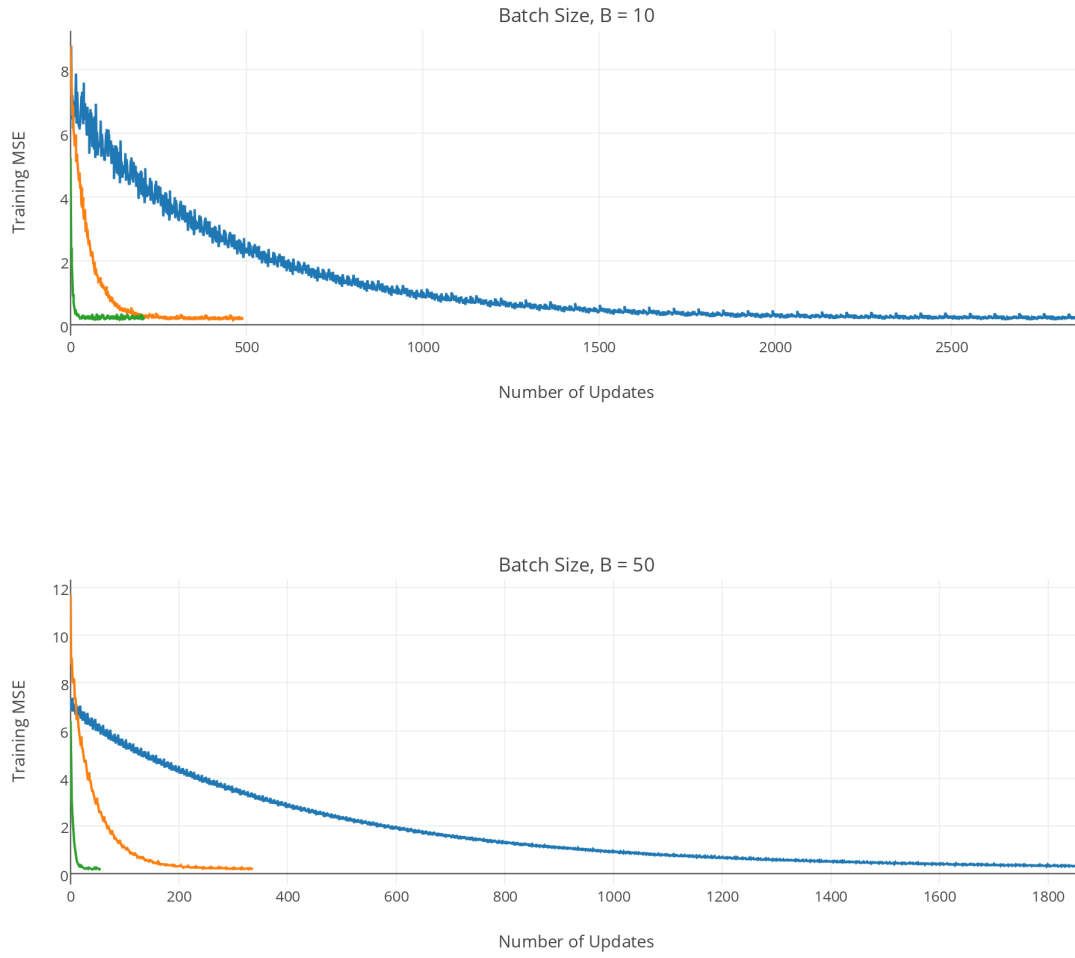


Figure 7: Subplots of training MSE against number of updates for batch sizes, $B = 10, 50$ and learning rates, $\eta = 0.001, 0.01, 0.1$. ($\lambda = 1$ for all cases)

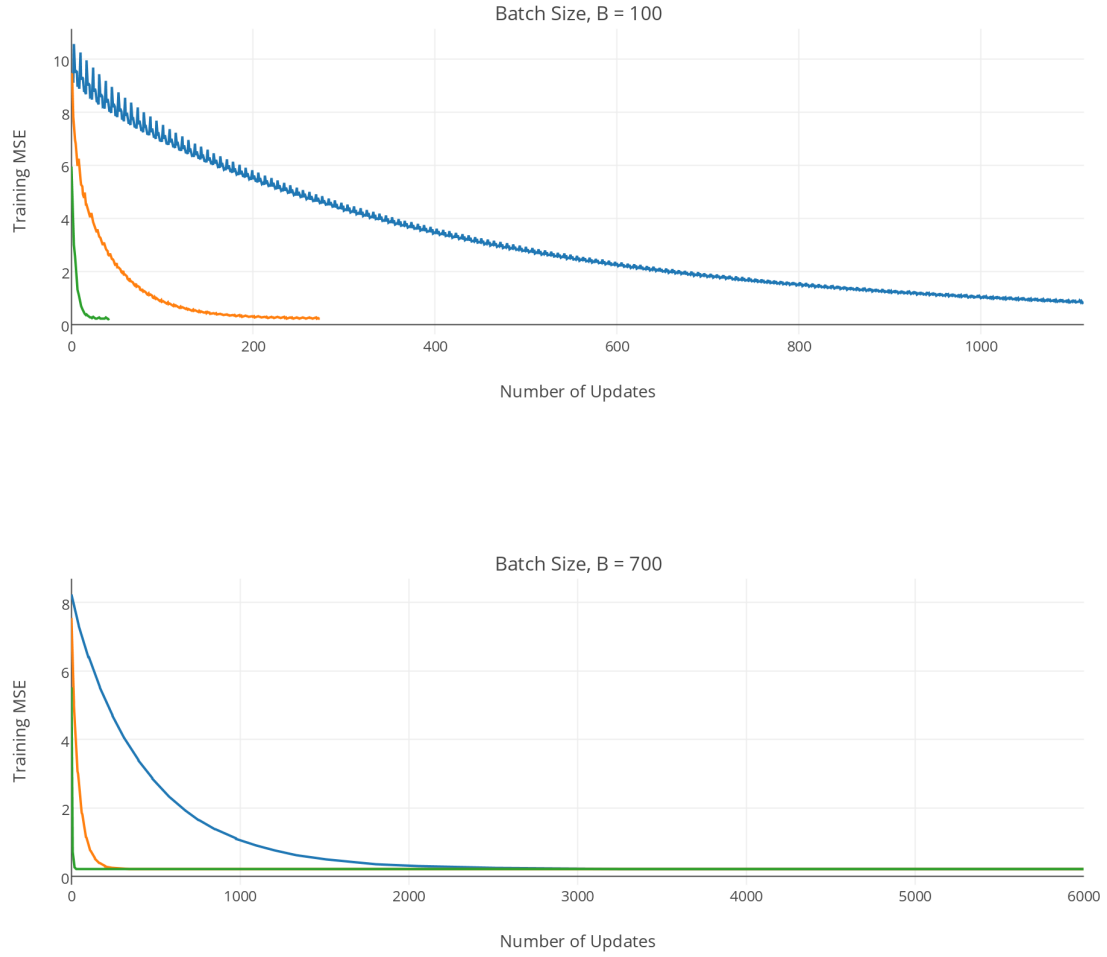


Figure 8: Subplots of training MSE against number of updates for batch sizes, $B = 100, 700$ and learning rates, $\eta = 0.001, 0.01, 0.1$. ($\lambda = 1$ for all cases)

2.2.3 Generalization

Based on [validation set](#) plot in Figure 9, the best value for λ is picked based on the highest validation accuracy obtained. As observed from Table 6, $\lambda = 0.001$, 0.01 and 0.1 all give the best validation accuracy of 93.0%. In this case, the best value of λ chosen is 0.01 , the middle ground between the three possible values.

As seen from the Figure 9, higher values of λ initially increases the test set accuracy. This is because λ prevents over-fitting of the model to the training set by penalizing large values of the weights. From a bias-variance trade-off perspective, incorporating λ effectively reduces the model variance at the cost of a slight increase in bias. This effectively leads to an overall increase in the test set accuracy. This is true up to $\lambda = 0.1$.

When $\lambda = 1$, the validation and test set accuracy drops significantly. For this high value of λ , the weights are being penalized too much and it prevents the model from effectively learning and capturing key features in the input data. From a bias-variance trade-off perspective, the high λ value has increased the model bias significantly to the point there is a net decrease in the model's performance.

λ has to be tuned to the validation set instead of the training set. If λ was tuned based on the training set, the model would further over-fitting the model to the training set. This is because λ would have been tuned to optimise the value of the training MSE to data that was not used for training. In other words, it needs to perform well on data it has not seen before that is modeled by the validation set.

Table 6: Validation and test set accuracies for various values of weight-decay regularizer, λ

λ	Validation Accuracy	Test Accuracy
0	0.930	0.900
0.0001	0.910	0.900
0.001	0.930	0.910
0.01	0.930	0.910
0.1	0.930	0.910
1	0.830	0.795

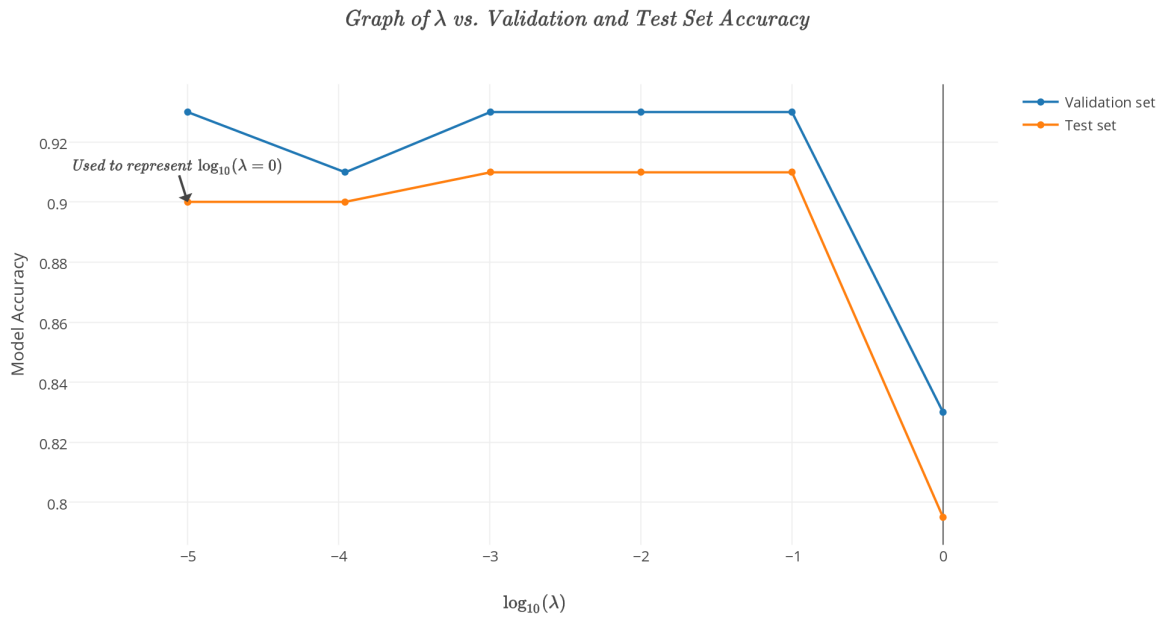


Figure 9: Graph of **validation** and **test** set accuracies against $\log_{10}(\lambda)$. ($\eta = 0.1, B = 50$)

3 Appendices

Dear Teaching Assistants, we implemented our code separately as we wanted to maximize our learning. The plots and code snippets pasted in this report come from two separate solutions which are both added as Appendices. Despite the different implementation, our results were very similar, indicating that our implementation should be correct.

For Stochastic Gradient Descent implementation, we started from Jimmy's posted code as a starter code and work from there for this assignment as suggested from Jimmy.

3.1 Entire Code 1: Chee Loong Soon's version

```
1 {python}
2
3 # Assignment 1
4 # Optimization
5 # Early Stopping
6 # Learning rate decay
7 # Momentum
8
9 import tensorflow as tf
10 import numpy as np
11 import sys
12
13 # 1.2 Euclidean Distance Function
14 # 1.2.2 Pairwise Distances
15 # Write a vectorized Tensorflow Python function that
    ↪ implements
```

```

16 # the pairwise squared Euclidean distance function for two
    ↪ input matrices.
17 # No Loops and makes use of Tensorflow broadcasting.
18 def PairwiseDistances(X, Z):
19     """
20     input:
21         X is a matrix of size (B x N)
22         Z is a matrix of size (C x N)
23     output:
24         D = matrix of size (B x C) containing the pairwise
    ↪ Euclidean distances
25     """
26     B = X.get_shape().as_list()[0]
27     N = X.get_shape().as_list()[1]
28     C = Z.get_shape().as_list()[0]
29     # Ensure the N dimensions are consistent
30     assert N == Z.get_shape().as_list()[1]
31     # Reshape to make use of broadcasting in Python
32     X = tf.reshape(X, [B, 1, N])
33     Z = tf.reshape(Z, [1, C, N])
34     # The code below automatically does broadcasting
35     D = tf.reduce_sum(tf.square(tf.sub(X, Z)), 2)
36     return D
37
38 # 1.3 Making Predictions
39 # 1.3.1 Choosing nearest neighbours
40 # Write a vectorized Tensorflow Python function that takes a
    ↪ pairwise distance matrix
41 # and returns the responsibilities of the training examples to
    ↪ a new test data point.

```

```

42 # It should not contain loops.
43 # Use tf.nn.top_k
44 def ChooseNearestNeighbours (D, K):
45     """
46     input:
47         D is a matrix of size (B x C)
48         K is the top K responsibilities for each test input
49     output:
50         topK are the value of the squared distances for the
51         ↪ topK
52         indices are the index of the location of these squared
53         ↪ distances
54     """
55     # Take topK of negative distances since it is the closest
56     ↪ data.
57     topK, indices = tf.nn.top_k(tf.neg(D), K)
58     return topK, indices
59
60 # 1.3.2 Prediction
61 # Compute the k-NN prediction with K = {1, 3, 5, 50}
62 # For each value of K, compute and report:
63     # training MSE loss
64     # validation MSE loss
65     # test MSE loss
66 # Choose best k using validation error = 50
67 def PredictKnn(trainData , testData, trainTarget, testTarget,
68     ↪ K):
69     """
70     input:
71         trainData

```

```

68         testData
69         trainTarget
70         testTarget
71     output:
72         loss
73     """
74     D = PairwiseDistances(testData, trainData)
75     topK, indices = ChooseNearestNeighbours(D, K)
76     # Select the proper outputs to be averaged from the target
77     ↪ values and average them
78     trainTargetSelectedAveraged =
79         tf.reduce_mean(tf.gather(trainTarget, indices), 1)
80     # Calculate the loss from the actual values
81     # Divide by 2.0 since it's average over 2M instead of M
82     ↪ where M = number of training data.
83     loss =
84         tf.reduce_mean(tf.square(tf.sub(trainTargetSelectedAveraged,
85                                         testTarget)))/2.0
86     return loss
87
88 # Plot the prediction function for x = [0, 11] on training
89 ↪ data.
90 def PredictedValues(x, trainData, trainTarget, K):
91     """
92     Plot the predicted values
93     input:
94         x = test target to plot and predict
95     """
96     D = PairwiseDistances(x, trainData)
97     topK, indices = ChooseNearestNeighbours(D, K)

```

```

92     predictedValues = tf.reduce_mean(tf.gather(trainTarget,
93         ↪ indices), 1)
94
95     return predictedValues
96
97 # 1.4 Soft-Knn & Gaussian Processes
98 # 1.4.1.1 Soft Decisions
99 # Write a Tensorflow python program based on the soft k-NN
100 ↪ model to compute
101 # predictions on the data1D.npy dataset.
102 # Set lambda = 100 NOT 10 as given in assignment handout
103 # and plot the test-set prediction of the model.
104
105 # Predict values using soft decision
106 def PredictedValuesSoftDecision(x, trainData, trainTarget):
107     hyperParam = 100
108     D1 = PairwiseDistances(x, trainData)
109     K1 = tf.exp(-hyperParam*D1)
110     sum1 = tf.reduce_sum(tf.transpose(K1), axis=0)
111     N = sum1.get_shape().as_list()[0]
112     sum1 = tf.reshape(sum1, [N,1])
113     rStar = tf.div(K1, sum1)
114     predictedValues = tf.matmul(rStar,trainTarget)
115     return predictedValues
116
117 # Predict values using Gaussian
118 # 1.4.1.1 Gaussian Processes
119 def PredictedValuesGaussianProcesses(x, trainData,
120     ↪ trainTarget):
121     hyperParam = 100
122     D1 = PairwiseDistances(x, trainData)

```

```

119     K1 = tf.exp(-hyperParam*D1)
120     D2 = PairwiseDistances(trainData, trainData)
121     K2 = tf.matrix_inverse(tf.exp(-hyperParam*D2))
122     rStar = tf.matmul(K1, K2)
123     predictedValues = tf.matmul(rStar,trainTarget)
124     return predictedValues
125
126     # Comment on the difference you observe between two programs
127     # Gaussian has higher loss.
128
129     # 2 Linear and Logistic Regression
130     # 2.2 Stochastic Gradient Descent
131     # Implement linear regression and stochastic gradient descent
132     ↪ algorithm
133     # with mini-batch size B = 50.
134     def buildGraph(learningRate, weightDecayCoeff):
135         # Variable creation
136         W = tf.Variable(tf.truncated_normal(shape=[64, 1],
137         ↪ stddev=0.5), name='weights')
138         b = tf.Variable(0.0, name='biases')
139         X = tf.placeholder(tf.float32, [None, 64], name='input_x')
140         y_target = tf.placeholder(tf.float32, [None,1],
141         ↪ name='target_y')
142         weightDecay =
143         ↪ tf.div(tf.constant(weightDecayCoeff),tf.constant(2.0))
144         # Graph definition
145         y_predicted = tf.matmul(X,W) + b
146         # Error definition
147         meanSquaredError =
148         ↪ tf.reduce_mean(tf.reduce_mean(tf.square(y_predicted -
149         ↪ y_target),

```


144

↪ reduction_indices=1,

145

↪ name='squared_error')

146

name='mean_squared_error')

147

weightDecayMeanSquareError =

↪ tf.reduce_mean(tf.reduce_mean(tf.square(weightDecay)))

148

weightDecayTerm = tf.multiply(weightDecay,

↪ weightDecayMeanSquareError)

149

meanSquaredError =

↪ tf.add(meanSquaredError, weightDecayTerm)

150

151

Training mechanism

152

optimizer =

↪ tf.train.GradientDescentOptimizer(learning_rate =

↪ learningRate)

153

train = optimizer.minimize(loss=meanSquaredError)

154

return W, b, X, y_target, y_predicted, meanSquaredError,

↪ train

155

156

157

def ShuffleBatches(trainData, trainTarget):

158

rngState = np.random.get_state()

159

np.random.shuffle(trainData)

160

np.random.set_state(rngState)

161

np.random.shuffle(trainTarget)

162

return trainData, trainTarget

163

164

def LinearRegression(trainData, trainTarget, validData,

↪ validTarget, testData, testTarget):

```

165     figureCount = 30
166     # 2.2.3 Generalization (done by partner)
167     # Run SGD with B = 50 and use validation performance to
168     → choose best weight decay coefficient
169     # from weightDecay = {0., 0.0001, 0.001, 0.01, 0.1, 1.}
170     # Plot weightDecay vs test set accuracy. (Done by partner)
171     weightDecayTrials= [0.0, 0.0001, 0.0001, 0.01, 0.1, 1.0]
172     # Plot total loss function vs number of updates for the
173     → best learning rate found
174     learningRateTrials = [0.1, 0.01, 0.001]
175     # 2.2.2 Effect of the mini-batch size
176     # Run with Batch Size, B = {10, 5, 100, 700} and tune the
177     → learning rate separately for each mini-batch size.
178     # Plot the total loss function vs the number of updates
179     → for each mini-batch size.
180     miniBatchSizeTrials = [10, 50, 100, 700]
181     learningRate = 0.01
182     miniBatchSize = 10
183     weightDecayCoeff = 1.0
184     # for weightDecayCoeff in weightDecayTrials:
185     for miniBatchSize in miniBatchSizeTrials:
186         for learningRate in learningRateTrials:
187             # Build computation graph
188             W, b, X, y_target, y_predicted, meanSquaredError,
189             → train = buildGraph(learningRate,
190             → weightDecayCoeff)
191             # Initialize session
192             init = tf.global_variables_initializer()
193             sess = tf.InteractiveSession()
194             sess.run(init)

```

189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213

```
initialW = sess.run(W)
initialb = sess.run(b)

# print "Initial weights: %s, initial bias: %.2f",
    ↪ initialW, initialb
# Training model
numEpoch = 200
currEpoch = 0
wList = []

xAxis = []
yTrainErr = []
yValidErr = []
yTestErr = []
numUpdate = 0
step = 0
errTrain = -1
errValid = -1
errTest = -1
while currEpoch <= numEpoch:
    # Shuffle the batches and return
    trainData, trainTarget =
        ↪ ShuffleBatches(trainData, trainTarget)
    step = 0
    # Full batch
    while step*miniBatchSize < 700:
        _, errTrain, currentW, currentb, yhat =
            ↪ sess.run([train, meanSquaredError, W,
            ↪ b, y_predicted], feed_dict={X:
            ↪ trainData[step*miniBatchSize:(step+1)*miniBatchSi
            ↪ y_target:
            ↪ trainTarget[step*miniBatchSize:(step+1)*miniBatch
```

```

214         wList.append(currentW)
215         #if not (step*miniBatchSize % 50):
216         #     print "Iter: %3d, MSE-train: %4.2f,
           ↪ weights: %s, bias: %.2f", step, err,
           ↪ currentW.T, currentb
217         step = step + 1
218         xAxis.append(numUpdate)
219         numUpdate += 1
220         yTrainErr.append(errTrain)
221         errValid = sess.run(meanSquaredError,
           ↪ feed_dict={X: validData, y_target:
           ↪ validTarget})
222         errTest = sess.run(meanSquaredError,
           ↪ feed_dict={X: testData, y_target:
           ↪ testTarget})
223         yValidErr.append(errValid)
224         yTestErr.append(errTest)
225         # Testing model
226         # TO know what is being run
227         currEpoch += 1
228         print "LearningRate: " , learningRate, " Mini
           ↪ batch Size: ", miniBatchSize
229         print "Iter: ", numUpdate
230         print "Final Train MSE: ", errTrain
231         print "Final Valid MSE: ", errValid
232         print "Final Test MSE: ", errTest
233         import matplotlib.pyplot as plt
234         plt.figure(figsize=(10, 10))
235         figureCount = figureCount + 1
236         plt.plot(np.array(xAxis), np.array(yTrainErr))

```

```

237 plt.savefig("TrainLossLearnRate" +
    ↪ str(learningRate) + "Batch" +
    ↪ str(miniBatchSize) + '.png')

238
239 plt.figure(figsize=(10, 10))
240 figureCount = figureCount + 1
241 plt.plot(np.array(xAxis), np.array(yValidErr))
242 plt.savefig("ValidLossLearnRate" +
    ↪ str(learningRate) + "Batch" +
    ↪ str(miniBatchSize) + '.png')
243 plt.figure(figsize=(10, 10))
244 figureCount = figureCount + 1
245 plt.plot(np.array(xAxis), np.array(yTestErr))
246 plt.savefig("TestLossLearnRate" +
    ↪ str(learningRate) + "Batch" +
    ↪ str(miniBatchSize) + '.png')

247 return

248
249 def SortData(inputVal, outputVal):
250     """
251     This sorts a given test set by the dataValue before
    ↪ plotting it.
252     """
253     p = np.argsort(inputVal, axis=0)
254     inputVal = np.array(inputVal)[p]
255     outputVal = np.array(outputVal)[p]
256     inputVal = inputVal[:, :, 0]
257     outputVal = outputVal[:, :, 0]
258     return inputVal, outputVal
259

```

```

260 if __name__ == "__main__":
261     print 'helloworld'
262     N = 2 # number of dimensions
263     B = 3 # number of test inputs (To get the predictions for
        ↪ all these inputs
264     C = 2 # number of training inputs (Pick closest k from
        ↪ this C)
265     X = tf.constant([1, 2, 3, 4, 5, 6], shape=[3, 2])
266     Z = tf.constant([21, 22, 31, 32], shape=[2, 2])
267     # Need to put seed so random_uniform doesn't generate new
        ↪ random values
268     # each time you evaluate when you print, so then the
        ↪ values would be
269     # inconsistent as to what you would have used or checked
270     #X = tf.random_uniform([B, N], seed=111)*30
271     #Z = tf.random_uniform([C, N], seed=112)*30
272     D = PairwiseDistances(X, Z)
273     K = 1 # number of nearest neighbours
274     # You calculate all the pairwise distances between each
        ↪ test input
275     # and existing training input
276     topK, indices = ChooseNearestNeighbours(D, K)
277     # Prediction
278     #for K in [1, 3, 5, 50]:
279     for K in [1]:
280         np.random.seed(521)
281         Data = np.linspace(1.0 , 10.0 , num =100)[:,
            ↪ np.newaxis]
282         Target = np.sin( Data ) + 0.1 * np.power( Data , 2) +
            ↪ 0.5 * np.random.randn(100 , 1)

```

```

283     randIdx = np.arange(100)
284     np.random.shuffle(randIdx)
285     # data1D.npy
286     trainData, trainTarget = Data[randIdx[:5]],
        ↪ Target[randIdx[:5]]
287     trainData, trainTarget = Data[randIdx[:80]],
        ↪ Target[randIdx[:80]]
288     validData, validTarget = Data[randIdx[80:90]],
        ↪ Target[randIdx[80:90]]
289     testData, testTarget = Data[randIdx[90:93]],
        ↪ Target[randIdx[90:93]]
290     testData, testTarget = Data[randIdx[90:100]],
        ↪ Target[randIdx[90:100]]
291
292     #trainData, trainTarget = SortData(trainData,
        ↪ trainTarget)
293     #validData, validTarget = SortData(validData,
        ↪ validTarget)
294     testData, testTarget = SortData(testData, testTarget)
295
296
297     # Convert to tensors from numpy
298     trainData = tf.pack(trainData)
299     validData = tf.pack(validData)
300     testData = tf.pack(testData)
301     trainTarget = tf.pack(trainTarget)
302     validtarget = tf.pack(validTarget)
303     testTarget = tf.pack(testTarget)
304     trainMseLoss = PredictKnn(trainData, trainData,
        ↪ trainTarget, trainTarget, K)

```

```

305 validationMseLoss = PredictKnn(trainData, validData,
    ↪ trainTarget, validTarget, K)
306 testMseLoss = PredictKnn(trainData, testData,
    ↪ trainTarget, testTarget, K)
307 init = tf.global_variables_initializer()
308 '''
309 with tf.Session() as sess:
310     sess.run(init)
311     print 'K ' + str(K)
312     print 'trainMseLoss'
313     print sess.run(trainMseLoss)
314     print 'validationMseLoss'
315     print sess.run(validationMseLoss)
316     print 'testMseLoss'
317     print sess.run(testMseLoss)
318 '''
319 # Plot the prediction for the x below
320 x = np.linspace(0.0, 11.0, num=1000)[: , np.newaxis]
321 xTensor = tf.pack(x)
322 predictedValuesKnn = PredictedValues(xTensor,
    ↪ trainData, trainTarget, K)
323 predictedValuesSoft =
    ↪ PredictedValuesSoftDecision(testData, trainData,
    ↪ trainTarget)
324 predictedValuesGaussian =
    ↪ PredictedValuesGaussianProcesses(testData,
    ↪ trainData, trainTarget)
325 lossSoft =
    ↪ tf.reduce_mean(tf.square(tf.sub(predictedValuesSoft,
    ↪ testTarget))))/2.0

```



```

326     lossGaussian =
        ↪ tf.reduce_mean(tf.square(tf.sub(predictedValuesGaussian,
        ↪ testTarget)))/2.0

327     import matplotlib.pyplot as plt
328     plt.figure(0)
329     init = tf.global_variables_initializer()
330     with tf.Session() as sess:
331         sess.run(init)
332         plt.figure(K+100)
333         plt.scatter(sess.run(trainData),
        ↪ sess.run(trainTarget))
334         plt.plot(sess.run(xTensor),
        ↪ sess.run(predictedValuesKnn))
335         fileName = str("KNN") + str(K) +
        ↪ str("trainingGraph.png")
336         plt.savefig(fileName)
337
338         # Plot for SoftDecision
339         plt.figure(K+101)
340         plt.title("Soft Decision kNN on Test Set, MSE = "
        ↪ + str(sess.run(lossSoft)))
341         plt.xlabel("Data Value")
342         plt.ylabel("Target Value")
343         plt.scatter(sess.run(testData),
        ↪ sess.run(testTarget), label= "testValue")
344         plt.plot(sess.run(testData),
        ↪ sess.run(predictedValuesSoft), label =
        ↪ "predicted")
345         plt.legend()
346         fileName = str("SoftDecision.png")

```

```

347         plt.savefig(fileName)
348         print 'SoftDecisionLoss'
349         print sess.run(lossSoft)
350
351         # Plot for Gaussian
352         plt.figure(K+102)
353         plt.title("Gaussian Process Regression on Test
354             ↪ Set, MSE = " + str(sess.run(lossGaussian)))
355         plt.xlabel("Data Value")
356         plt.ylabel("Target Value")
357         plt.scatter(sess.run(testData),
358             ↪ sess.run(testTarget), label = "testValue")
359         plt.plot(sess.run(testData),
360             ↪ sess.run(predictedValuesGaussian), label =
361             ↪ "predicted")
362         plt.legend()
363         fileName = str("ConditionalGaussian.png")
364         plt.savefig(fileName)
365         print 'ConditionalGaussianLoss'
366         print sess.run(lossGaussian)
367
368     # Part 2
369     with np.load ("tinymnist.npz") as data :
370         trainData, trainTarget = data ["x"], data["y"]
371         validData, validTarget = data ["x_valid"], data
372             ↪ ["y_valid"]
373         testData, testTarget = data ["x_test"], data
374             ↪ ["y_test"]
375         LinearRegression(trainData, trainTarget,validData,
376             ↪ validTarget, testData, testTarget)

```

3.2 Entire Code 2: FuYuan Tee's version

3.2.1 Question 1: k-Nearest Neighbour

```
1 import tensorflow as tf
2 import numpy as np
3
4 import matplotlib.pyplot as plt
5
6 import plotly.plotly as py
7 import plotly.graph_objs as go
8 from plotly import tools
9
10 import plotly.offline as pyo
11 pyo.init_notebook_mode(connected=True)
12
13 # Generate data
14 np.set_printoptions(precision=3)
15 np.random.seed(521)
16
17 # Generating data
18 Data = np.linspace(1.0, 10.0, num=100)[:, np.newaxis]
19 Target = np.sin(Data) + 0.1 * np.power(Data, 2) \
20         + 0.5 * np.random.randn(100, 1)
21
22 # Generating a random index
23 randIdx = np.arange(100)
24 np.random.shuffle(randIdx)
25
26 # Partitioning 100 datapoints into training, validation and
```

```

27 # test sets consisting of 80, 10 and 10 points respectively.
28 trainData, trainTarget = Data[randIdx[:80]],
    ↪ Target[randIdx[:80]]
29 validData, validTarget = Data[randIdx[80:90]],
    ↪ Target[randIdx[80:90]]
30 testData, testTarget = Data[randIdx[90:]],
    ↪ Target[randIdx[90:]]
31
32 # Defining TensorFlow Variables
33
34 # Calculates pairwise squared Euclidean distance between
    ↪ matrices X and Z
35 # X is the input matrix which houses data points to be
    ↪ calculated,
36 # which are calculated against reference datapoints in Z
37 def euclideanDistance(X, Z):
38     if tf.TensorShape.num_elements(X.get_shape()) == 3:
39         return tf.reduce_sum(tf.square(X - tf.transpose(Z)),
    ↪ axis=2) # Sums feature deviations for each
    ↪ variable
40     else:
41         return tf.square(X - tf.transpose(Z),
    ↪ name='Euclidean_Distance_Matrix')
42
43 # Produces a BxC responsibility vector
44 def responsibilityVector(D, k):
45     def flatten(tensor):
46         return tf.reshape(tensor, [-1])
47
48     # Obtains indices of top-K points

```

```

49     _, idx = tf.nn.top_k(tf.transpose(-D), k)
50
51     # Creates a step sequence for M values repeated k times
52     M = tf.shape(D)[1]
53     step_seq =
54         ↪ flatten(tf.transpose(tf.reshape(tf.tile(tf.range(0,
55             ↪ M), [k]), [k, M])))
56
57     # Form new index key compatible for subsequent
58     ↪ sparse_to_dense op
59     sparse_idx = tf.pack([step_seq, flatten(idx)], axis=1,
60         ↪ name='Sparse_Indices')
61
62     # Forms dense tensor
63     return tf.sparse_to_dense(tf.cast(sparse_idx, tf.int32), \
64         ↪ tf.cast(tf.shape(tf.transpose(D)),
65         ↪ tf.int32), \
66         ↪ tf.fill([tf.shape(sparse_idx)[0]],
67         ↪ tf.divide(1.0, tf.cast(k,
68         ↪ tf.float32))), \
69         validate_indices=False, \
70         name='Responsibility_Vector')
71
72 # Function that creates a TensorFlow model
73 def buildGraph(k_):
74     k = tf.constant(k_, name='k') # Hyperparameter
75     X = tf.placeholder(tf.float32, shape=[None, None],
76         ↪ name='Training_Data')

```

```

69     Y = tf.placeholder(tf.float32, shape=[None, None],
    ↪     name='Training_Target')
70     Z = tf.placeholder(tf.float32, shape=[None, None],
    ↪     name='Input_Data')
71     T = tf.placeholder(tf.float32, shape=[None, None],
    ↪     name='Input_Target')
72
73     D = euclideanDistance(X, Z)
74     R = responsibilityVector(D, k)
75
76     Y_hat = tf.matmul(R, Y, name='Y_hat')
77     MSE = tf.divide(tf.reduce_sum(tf.square(T - Y_hat)), \
78                     tf.scalar_mul(2, tf.cast(tf.shape(Z)[0],
    ↪     tf.float32))), \
79                     name='Mean_Squared_Error') # Half of
    ↪     theoretical MSE
80
81     return X, Y, Z, T, Y_hat, MSE
82
83     # Function used to adapt 'kNN' function based on type of
    ↪ dataset used for calculation
84     # Mode consists of either ['train', 'validation', 'test',
    ↪ 'full']
85     def selectDataPartition(mode):
86
87         inputData = trainData
88         inputTarget = trainTarget
89
90         if mode == 'train':
91             dataSource = trainData

```

```

92         targetSource = trainTarget
93     elif mode == 'validation':
94         dataSource = validData
95         targetSource = validTarget
96     elif mode == 'test':
97         dataSource = testData
98         targetSource = testTarget
99     elif mode == 'full':
100         dataSource = np.linspace(0.0, 11.0, num=1000)[:,
101             ↪ np.newaxis]
102         targetSource = np.sin(dataSource) + 0.1 *
103             ↪ np.power(dataSource, 2) \
104                 + 0.5 * np.random.randn(1000, 1)
105
106         inputData = trainData
107         inputTarget = trainTarget
108
109     return inputData, inputTarget, dataSource, targetSource
110
111 def kNN(k, mode):
112     X, Y, Z, T, Y_hat, MSE = buildGraph(k)
113
114     with tf.Session() as sess:
115         # print "k: %d, mode: %s" % (k, mode)
116         inputData, inputTarget, dataSource, targetSource =
117             ↪ selectDataPartition(mode)
118
119         error, y_pred = sess.run([MSE, Y_hat], \
120             feed_dict={X: inputData, Y:
121                 ↪ inputTarget, \

```

```

118                                     Z: dataSource, T:
                                     ↪ targetSource
119                                     })
120
121     return error, np.transpose(np.append(inputData,
122     ↪ inputTarget, axis=1)),
123     ↪ np.transpose(np.append(dataSource, y_pred, axis=1))
124
125 # Main Function
126 k_list = [1, 3, 5, 50]
127 kNN_mode = ['train', 'validation', 'test', 'full']
128
129 MSE_list = []
130 targetSeries = []
131 predictionSeries = []
132
133 # Performs kNN based on various calculation modes
134 for i in range(4):
135     for j in range(4):
136         MSE, target, prediction = kNN(k_list[i], kNN_mode[j])
137         MSE_list.append(MSE)
138         targetSeries.append(target)
139         predictionSeries.append(prediction)
140 MSE_list = np.reshape(MSE_list, (4, 4))
141
142 # Generate interactive Plotly graph
143 def generateVisualisation(targetSeries, predictionSeries,
144     ↪ k_list):
145     subplotTitleString = []
146     for i in range(len(k_list)):

```



```

144     subplotTitleString.append('k = %s' % str(k_list[i]))
145
146 fig = tools.make_subplots(rows=2, cols=2,
147     ↪ subplot_titles=(subplotTitleString))
148
149 for i in range(len(k_list)):
150     traceData = go.Scatter(
151         x = targetSeries[i * 4][0],
152         y = targetSeries[i * 4][1],
153         marker = {'color': 'blue',
154                 'symbol': 200},
155         mode = 'markers',
156         name = 'data_k=' + str(k_list[i])
157     )
158     tracePred = go.Scatter(
159         x = predictionSeries[4 * (i + 1) - 1][0],
160         y = predictionSeries[4 * (i + 1) - 1][1],
161         marker = {'color': 'green'},
162         mode = 'lines',
163         name = 'prediction_k=' + str(k_list[i])
164     )
165
166 fig.append_trace(traceData, i / 2 + 1, i % 2 + 1)
167 fig.append_trace(tracePred, i / 2 + 1, i % 2 + 1)
168
169 fig['layout']['xaxis'+str(i+1)].update(title='x')
170 fig['layout']['yaxis'+str(i+1)].update(title='y')
171
172 fig['layout'].update(height=900, width=950, title='k-NN
173     ↪ Regression on data1D', showlegend=False)

```

```

172     return py.ipplot(fig, filename='A1Q1_kNN_subplot2x2')
173
174 # Output summary table
175 print 'Mean Squared Error Summary:'
176 for i in range(5):
177     if i == 0:
178         print "%3s %10s %10s %6s" % ('k', 'Training',
179                                     ↪ 'Validation', 'Test')
180     else:
181         print "%3d %10.3f %10.3f %6.3f" % (k_list[i - 1],
182                                     ↪ MSE_list[i - 1][0], MSE_list[i - 1][1], MSE_list[i
183                                     ↪ - 1][2])
184
185 print "\n\n\n"
186 kNN_visuals = generateVisualisation(targetSeries,
187                                     ↪ predictionSeries, k_list)
188 kNN_visuals

```

3.2.2 Question 2.2: Stochastic Gradient Descent

```

1 {python}
2 # Import relevant packages
3 import tensorflow as tf
4 import numpy as np
5 import math
6
7 import time
8
9 # Non-interactive plotting

```

```

10 import matplotlib.pyplot as plt
11 from IPython import display
12
13 # Interactive plotting
14 from plotly import tools
15 import plotly.plotly as py
16 import plotly.graph_objs as go
17 import plotly.offline as pyo
18 from plotly.offline import download_plotlyjs
19
20 # Configure environment
21 np.set_printoptions(precision=3)
22 np.random.seed(521)
23
24 # Activate Plotly Offline for Jupyter
25 pyo.init_notebook_mode(connected=True)
26
27 # Load Tiny MNIST dataset
28 with np.load ("tinymnist.npz") as data:
29     trainData, trainTarget = data ["x"], data["y"]
30     validData, validTarget = data ["x_valid"], data
31     ↪ ["y_valid"]
32     testData, testTarget = data ["x_test"], data ["y_test"]
33
34 # Create Tensorflow Graph
35 def buildGraph(eta, lambda_):
36     # Model inputs
37     X = tf.placeholder(tf.float32, shape=[None, None],
38     ↪ name='Input')
39     Y = tf.placeholder(tf.float32, shape=[None, None],
40     ↪ name='Target')

```

```

38
39     # Model variables
40     W = tf.Variable(tf.truncated_normal(shape=[64, 1],
41         ↪ stddev=0.5), name='Weights')
42
43     # Model parameters
44     eta = tf.constant(eta, name='Learning_Rate')
45     lambda_ = tf.constant(lambda_, name='L2_Regularizer')
46
47     # Predicted target
48     Y_hat = tf.matmul(X, W)
49
50     # Mean squared error
51     MSE = tf.scalar_mul(tf.divide(1.0, tf.cast(tf.shape(X)[0],
52         ↪ tf.float32)), \
53         tf.reduce_sum(tf.square(Y_hat - Y)))
54         ↪ \
55         + tf.scalar_mul(tf.divide(tf.cast(lambda_,
56         ↪ tf.float32), 2.0), tf.matmul(tf.transpose(W),
57         ↪ W))
58
59     # Basic accuracy definition (n_correct / n_total)
60     Y_hat_thresholded = tf.cast(tf.greater_equal(Y_hat, 0.5),
61         ↪ tf.float32)
62     accuracy =
63         ↪ tf.divide(tf.reduce_sum(tf.cast(tf.equal(Y_hat_thresholded,
64         ↪ Y), tf.float64))), \
65         tf.cast(tf.shape(X)[0], tf.float64))

```

```

60     # Basic gradient descent optimizer
61     optimizer =
        ↪ tf.train.GradientDescentOptimizer(eta).minimize(MSE)
62
63     return W, b, X, Y, Y_hat, MSE, accuracy, optimizer

```

3.2.3 Question 2.2.1: Tuning the learning rate

```

1  {python}
2  # Tune learning rate
3  MAX_ITER = 2000
4  def tuneLearningRate(etaList, batchSize=50, lambda_=1):
5      # Returns the i-th batch of training data and targets
6      # Generates a new, reshuffled batch once all previous
        ↪ batches are fed
7      def getNextTrainingBatch(currentIter):
8          currentBatchNum = currentIter % (trainData.shape[0] /
        ↪ batchSize)
9          if currentBatchNum == 0:
10             np.random.shuffle(randIdx)
11             # print 'Iteration: %4d, BatchCap: %2d, BatchNum: %2d'
        ↪ % (currentIter, trainData.shape[0] / batchSize,
        ↪ currentBatchNum)
12             lowerBoundIdx = currentBatchNum * batchSize
13             upperBoundIdx = (currentBatchNum + 1) * batchSize
14             return trainData[lowerBoundIdx:upperBoundIdx],
        ↪ trainTarget[lowerBoundIdx:upperBoundIdx]
15
16     # Generate updated plots for training and validation MSE

```

```

17 def plotMSEGraph(MSEList, param):
18     label = '$\eta$ = ' + str(param)
19     label_classification = ['train.', 'valid.']
20
21     display.clear_output(wait=True)
22     plt.figure(figsize=(8,5), dpi=200)
23
24     for i, MSE in enumerate(MSEList):
25         plt.plot(range(len(MSE)), MSE, '.', markersize=3,
26                 ↪ label=label+' '+label_classification[i])
27
28     plt.axis([0, MAX_ITER, 0, np.amax(MSEList)])
29     plt.legend()
30     plt.show()
31
32     # Calculates the ratio between the n-th average epoch MSE
33     ↪ and the (n-1)-th average epoch MSE
34
35 def ratioAverageEpochMSE(currentValidMSE):
36
37     averageN =
38         ↪ np.average(currentValidMSE[-(np.arange(epochSize -
39         ↪ 1) + 1)])
40
41     averageNlessOne =
42         ↪ np.average(currentValidMSE[-(np.arange(epochSize -
43         ↪ 1) + epochSize)])
44
45     return averageN / averageNlessOne
46
47
48     # Returns True if the average epoch validation MSE is at
49     ↪ least 99% of the previous epoch average.
50
51     # i.e. Returns True if the average learnings between epoch
52     ↪ is less than +1%

```

```

39     # Otherwise, returns False
40     def shouldStopEarly(currentValidMSE):
41         if currentValidMSE.shape[0] < 2 * epochSize:
42             return False
43         return True if (ratioAverageEpochMSE(currentValidMSE)
44             ↪ >= 0.99) else False
45
46     summaryList = []
47     randIdx = np.arange(trainData.shape[0])
48     epochSize = trainData.shape[0] / batchSize
49
50     for eta in etaList:
51         W, b, X, Y, Y_hat, MSE, accuracy, optimizer =
52             ↪ buildGraph(eta, lambda_)
53
54         with tf.Session() as sess:
55             tf.global_variables_initializer().run()
56
57             # Creates blank training and validation MSE arrays
58             ↪ for the Session
59
60             currentTrainMSE = np.array([])[: , np.newaxis]
61             currentValidMSE = np.array([])[: , np.newaxis]
62
63             # Runs update
64             currentIter = 0
65             while currentIter <= MAX_ITER:
66                 inputData, inputTarget =
67                     ↪ getNextTrainingBatch(currentIter)
68
69                 _, trainError = sess.run([optimizer, MSE],
70                     ↪ feed_dict={X: inputData, Y: inputTarget})

```

```

65         validError = sess.run([MSE], feed_dict={X:
        ↪   validData, Y: validTarget})

66
67         currentTrainMSE = np.append(currentTrainMSE,
        ↪   trainError)
68         currentValidMSE = np.append(currentValidMSE,
        ↪   validError)
69
70         # Update graph of training and validation MSE
        ↪   arrays
71         if (currentIter < 3) or (currentIter % 500 ==
        ↪   0):
72             plotMSEGraph([currentTrainMSE,
        ↪   currentValidMSE], eta)
73
74         # At every epoch, check for early stopping
        ↪   possibility. If so, breaks from while loop
75         if currentIter % epochSize == 0:
76             if shouldStopEarly(currentValidMSE):
77                 break
78
79         currentIter += 1
80
81         # Save session results as dictionary and appends to
        ↪   MSEsummaryList
82         summaryList.append(
83             {
84                 'eta': eta,
85                 'B': batchSize,
86                 'lambda': lambda_,

```



```

87         'numIter': currentIter + 1,
88         'epoch': float(currentIter + 1) /
            ↳ (trainData.shape[0] / batchSize),
89         'trainMSE': currentTrainMSE,
90         'validMSE': currentValidMSE,
91     }
92 )
93
94     return summaryList
95
96
97 # Main Function
98 etaList = [0.001, 0.01, 0.1]
99 tunedEtaSummary = tuneLearningRate(etaList)
100
101 # Output summary table
102 for summary in tunedEtaSummary:
103     print 'eta: %.3f, numIter: %d, validMSE: %.3f' %
            ↳ (summary['eta'], summary['numIter'],
            ↳ summary['validMSE'][-1])
104
105 # Produce interactive graph for best learning rate
106 def etaIGraph(tunedEtaSummary):
107     # Create plot for each summary
108     traceList = []
109     for summary in tunedEtaSummary:
110         traceList.append(
111             go.Scatter(
112                 x = range(summary['numIter'] + 1),
113                 y = summary['trainMSE'],

```

```

114         name = '$\\eta = ' + str(summary['eta']) + '$'
115     )
116 )
117 data = go.Data(traceList)
118
119 # Create figure layout
120 layout = go.Layout(
121     title = '$\\textit{Training performance for various}
122     ↪ learning rates, } \\eta$',
123     xaxis = {'title': 'Number of Updates'},
124     yaxis = {'title': 'Training MSE'},
125 )
126
127 figure = go.Figure(data=data, layout=layout)
128 return py.iplot(figure, filename='A1Q2.1_bestEtaGraph')
129 fig2_1 = etaIGraph(tunedEtaSummary)
130 fig2_1

```

3.2.4 Question 2.2.2: Effect of the mini-batch size

```

1 {python}
2 # Tune the mini-batch size
3 MAX_ITER = 6000
4 def tuneBatchSize(etaList, batchSizeList, lambda_=1):
5     # Returns the i-th batch of training data and targets
6     # Generates a new, reshuffled batch once all previous
7     ↪ batches are fed
8     def getNextTrainingBatch(currentIter):
9         currentBatchNum = currentIter % (trainData.shape[0] /
10         ↪ batchSize)

```

```

9         if currentBatchNum == 0:
10             np.random.shuffle(randIdx)
11             # print currentBatchNum + 1
12             lowerBoundIdx = currentBatchNum * batchSize
13             upperBoundIdx = (currentBatchNum + 1) * batchSize
14             return trainData[lowerBoundIdx:upperBoundIdx],
               ↪ trainTarget[lowerBoundIdx:upperBoundIdx]
15
16         # Generate updated plots for training and validation MSE
17     def plotMSEGraph(MSEList, param):
18         label = '$B$ = ' + str(param[0]) + ', $\eta$: ' +
               ↪ str(param[1])
19         label_classification = ['train.', 'valid.']
20
21         display.clear_output(wait=True)
22         plt.figure(figsize=(8,5), dpi=200)
23
24         for i, MSE in enumerate(MSEList):
25             plt.plot(range(len(MSE)), MSE, '.', markersize=3,
               ↪ label=label+'\n'+label_classification[i])
26
27         plt.axis([0, MAX_ITER, 0, np.amax(MSEList)])
28         plt.legend()
29         plt.show()
30
31         # Calculates the ratio between the n-th average epoch MSE
               ↪ and the (n-1)-th average epoch MSE
32     def ratioAverageEpochMSE(currentValidMSE):
33         averageN =
               ↪ np.average(currentValidMSE[-(np.arange(epochSize -
               ↪ 1) + 1)])

```

```

34     averageNlessOne =
        ↪ np.average(currentValidMSE[-(np.arange(epochSize -
        ↪ 1) + epochSize)])
35     return averageN / averageNlessOne
36
37     # Returns True if the average epoch validation MSE is at
        ↪ least 99% of the previous epoch average.
38     # i.e. Returns True if the average learnings between epoch
        ↪ is less than +1%
39     # Otherwise, returns False
40     def shouldStopEarly(currentValidMSE):
41         if currentValidMSE.shape[0] < 2 * epochSize:
42             return False
43         return True if (ratioAverageEpochMSE(currentValidMSE)
        ↪ >= 0.99) else False
44
45     summaryList = []
46     randIdx = np.arange(trainData.shape[0])
47
48     for batchSize in batchSizeList:
49         epochSize = trainData.shape[0] / batchSize
50         batchSummary = []
51         for eta in etaList:
52             W, b, X, Y, Y_hat, MSE, accuracy, optimizer =
        ↪ buildGraph(eta, lambda_)
53
54             with tf.Session() as sess:
55                 tf.global_variables_initializer().run()
56
57                 # Creates blank training and validation MSE
        ↪ arrays for the Session

```

```

58         currentTrainMSE = np.array([])[: , np.newaxis]
59         currentValidMSE = np.array([])[: , np.newaxis]
60
61         # Runs update
62         currentIter = 0
63         while currentIter <= MAX_ITER:
64             inputData, inputTarget =
65                 ↪ getNextTrainingBatch(currentIter)
66
67             _, trainError = sess.run([optimizer, MSE],
68                 ↪ feed_dict={X: inputData, Y:
69                 ↪ inputTarget})
70             validError = sess.run([MSE], feed_dict={X:
71                 ↪ validData, Y: validTarget})
72
73             currentTrainMSE =
74                 ↪ np.append(currentTrainMSE, trainError)
75             currentValidMSE =
76                 ↪ np.append(currentValidMSE, validError)
77
78             # Update graph of training and validation
79             ↪ MSE arrays
80
81             if (currentIter < 3) or (currentIter % 500
82                 ↪ == 0):
83                 plotMSEGraph([currentTrainMSE,
84                     ↪ currentValidMSE], [batchSize,
85                     ↪ eta])
86
87             # At every epoch, check for early stopping
88             ↪ possibility. If so, breaks from while
89             ↪ loop

```

```

77         if currentIter % epochSize == 0:
78             if shouldStopEarly(currentValidMSE):
79                 break
80
81         currentIter += 1
82
83         # Save session results as dictionary and appends
84         ↪ to MSEsummaryList
85         batchSummary.append(
86             {
87                 'eta': eta,
88                 'B': batchSize,
89                 'lambda': lambda_,
90                 'numIter': currentIter + 1,
91                 'epoch': float(currentIter + 1) /
92                     ↪ (trainData.shape[0] / batchSize),
93                 'trainMSE': currentTrainMSE,
94                 'validMSE': currentValidMSE,
95             }
96         )
97         summaryList.append(batchSummary)
98
99         return summaryList
100
101     # Main function
102     etaList = [0.001, 0.01, 0.1]
103     batchSizeList = [10, 50, 100, 700]
104     tunedBatchSizeSummary = tuneBatchSize(etaList, batchSizeList)
105
106     # Output summary table:

```

```

105 for batchSummary in tunedBatchSizeSummary:
106     for summary in batchSummary:
107         print 'B: %5d, eta: %5.3f, numIter: %5d, validMSE:
            ↪ %3.3f' % \
108             (summary['B'], summary['eta'], summary['numIter'],
            ↪ summary['validMSE'][-1])
109
110 # Generate interactive Plotly plot
111 def batchSizeIGraphSubplot(tunedBatchSizeSummary):
112
113     # Define subplot title
114     subplotTitle = []
115     for batchSummary in tunedBatchSizeSummary:
116         subplotTitle.append('Batch Size, B = ' +
            ↪ str(batchSummary[0]['B']))
117
118     # Define subplot figure
119     figure = tools.make_subplots(rows=4, cols=1,
            ↪ subplot_titles=(subplotTitle))
120
121     # Define color list
122     colorList = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728',
            ↪ '#9467bd', '#8c564b']
123
124     # Create plot for each summary
125     for i, batchSummary in enumerate(tunedBatchSizeSummary):
126         traceList = []
127         for j, summary in enumerate(batchSummary):
128             trace = go.Scatter(
129                 x = range(summary['numIter'] + 1),

```

```

130         y = summary['trainMSE'],
131         marker = {'color': colorList[j]},
132         name = '$B=' + str(summary['B']) + ', \\eta='
            ↪ + str(summary['eta']) + '$'
133     )
134     figure.append_trace(trace, i + 1, 1)
135
136     ↪ figure['layout']['xaxis'+str(i+1)].update(title='Number
137     ↪ of Updates')
138
139     ↪ figure['layout']['yaxis'+str(i+1)].update(title='Training
140     ↪ MSE')
141
142     # Create figure layout
143     figure['layout'].update(
144         height = 1800,
145         title = '$\\textit{Model training performance for
            ↪ various batch size, } B' + \\
146             '\\textit{, and learning rate, } \\eta$',
147         showlegend = False
148     )
149
150     return py.iplot(figure,
            ↪ filename='A1Q2.2_batch_size_subplot2x2')
151
152 fig2_2_visual = batchSizeIGraphSubplot(tunedBatchSizeSummary)
153 fig2_2_visual

```


3.2.5 Question 2.2.3: Generalization

```
1 {python}
2 # Tune weight decay regularizer
3 MAX_ITER = 2000
4 def tuneLambda(lambdaList, eta=0.1, batchSize=50):
5     # Returns the i-th batch of training data and targets
6     # Generates a new, reshuffled batch once all previous
7     #   ↪ batches are fed
8     def getNextTrainingBatch(currentIter):
9         currentBatchNum = currentIter % (trainData.shape[0] /
10            ↪ batchSize)
11         if currentBatchNum == 0:
12             np.random.shuffle(randIdx)
13             lowerBoundIdx = currentBatchNum * batchSize
14             upperBoundIdx = (currentBatchNum + 1) * batchSize
15             return trainData[lowerBoundIdx:upperBoundIdx],
16                ↪ trainTarget[lowerBoundIdx:upperBoundIdx]
17
18     # Generate updated plots for training and validation MSE
19     def plotMSEGraph(MSEList, param):
20         label = '$\lambda$ = ' + str(param)
21         label_classification = ['train.', 'valid.']
22
23         display.clear_output(wait=True)
24         plt.figure(figsize=(8,5), dpi=200)
25
26         for i, MSE in enumerate(MSEList):
27             plt.plot(range(len(MSE)), MSE, '-', label=label+'
28                 ↪ '+label_classification[i])
```

```

25
26     plt.axis([0, MAX_ITER, 0, np.amax(MSEList)])
27     plt.legend()
28     plt.show()
29
30     # Calculates the ratio between the n-th average epoch MSE
31     ↪ and the (n-1)-th average epoch MSE
32     def ratioAverageEpochMSE(currentValidMSE):
33         averageN =
34             ↪ np.average(currentValidMSE[-(np.arange(epochSize -
35             ↪ 1) + 1)])
36         averageNlessOne =
37             ↪ np.average(currentValidMSE[-(np.arange(epochSize -
38             ↪ 1) + epochSize)])
39         return averageN / averageNlessOne
40
41     # Returns True if the average epoch validation MSE is at
42     ↪ least 99% of the previous epoch average.
43     # i.e. Returns True if the average learnings between epoch
44     ↪ is less than +1%
45     # Otherwise, returns False
46     def shouldStopEarly(currentValidMSE):
47         if currentValidMSE.shape[0] < 2 * epochSize:
48             return False
49         return True if (ratioAverageEpochMSE(currentValidMSE)
50             ↪ >= 0.99) else False
51
52     summaryList = []
53     randIdx = np.arange(trainData.shape[0])
54     epochSize = trainData.shape[0] / batchSize

```

```

47
48     for lambda_ in lambdaList:
49         W, b, X, Y, Y_hat, MSE, accuracy, optimizer =
           ↪ buildGraph(eta, lambda_)
50
51     with tf.Session() as sess:
52         tf.global_variables_initializer().run()
53
54         # Creates blank training and validation MSE arrays
           ↪ for the Session
55         currentTrainMSE = np.array([])[: , np.newaxis]
56         currentValidMSE = np.array([])[: , np.newaxis]
57
58         # Runs update
59         currentIter = 0
60         while currentIter <= MAX_ITER:
61             inputData, inputTarget =
               ↪ getNextTrainingBatch(currentIter)
62
63             _, trainError = sess.run([optimizer, MSE],
               ↪ feed_dict={X: inputData, Y: inputTarget})
64             validError = sess.run([MSE], feed_dict={X:
               ↪ validData, Y: validTarget})
65
66             currentTrainMSE = np.append(currentTrainMSE,
               ↪ trainError)
67             currentValidMSE = np.append(currentValidMSE,
               ↪ validError)
68
69             # Update graph of training and validation MSE
               ↪ arrays

```

```

70         if (currentIter < 3) or (currentIter % 500 ==
    ↪ 0):
71             plotMSEGraph([currentTrainMSE,
    ↪ currentValidMSE], lambda_)
72
73             # At every epoch, check for early stopping
    ↪ possibilty. If so, breaks from while loop
74             if currentIter % epochSize == 0:
75                 if shouldStopEarly(currentValidMSE):
76                     break
77
78             currentIter += 1
79
80             # Compute validation and test accuracy
81             validAccuracy = sess.run(accuracy, feed_dict={X:
    ↪ validData, Y: validTarget})
82             testAccuracy = sess.run(accuracy, feed_dict={X:
    ↪ testData, Y: testTarget})
83
84             # Save session results as dictionary and appends to
    ↪ MSEsummaryList
85             summaryList.append(
86                 {
87                     'eta': eta,
88                     'B': batchSize,
89                     'lambda': lambda_,
90                     'numIter': currentIter + 1,
91                     'epoch': float(currentIter + 1) /
    ↪ (trainData.shape[0] / batchSize),
92                     'trainMSE': currentTrainMSE,

```

```

93         'validMSE': currentValidMSE,
94         'validAccuracy': validAccuracy,
95         'testAccuracy': testAccuracy
96     }
97 )
98
99     return summaryList
100
101 # Main Function
102 lambdaList = [0.0, 0.0001, 0.001, 0.01, 0.1, 1.0]
103 tunedLambdaSummary = tuneLambda(lambdaList)
104
105 # Output summary table
106 for summary in tunedLambdaSummary:
107     print 'lambda: %5.4f, numIter: %5d, validMSE: %5.3f,
108           ↪ validAcc: %3.3f, testAcc: %3.3f' % \
109           (summary['lambda'], summary['numIter'],
110            ↪ summary['validMSE'][-1], summary['validAccuracy'],
111            ↪ summary['testAccuracy'])
112
113 # Produce interactive Plotly graph
114 def lambdaIGraph(tunedLambdaSummary):
115     # Create plot for each summary
116     trace1 = go.Scatter(
117         x = [np.log10(summary['lambda'] + 1e-5) for summary in
118              ↪ tunedLambdaSummary],
119         y = [summary['validAccuracy'] for summary in
120              ↪ tunedLambdaSummary],
121         name = 'Validation set accuracy'
122     )

```

```

118
119     trace2 = go.Scatter(
120         x = [np.log10(summary['lambda'] + 1e-5) for summary in
121             ↳ tunedLambdaSummary],
122         y = [summary['testAccuracy'] for summary in
123             ↳ tunedLambdaSummary],
124         name = 'Test set accuracy'
125     )
126
127     data = go.Data([trace1, trace2])
128
129     # Create figure layout
130     layout = go.Layout(
131         title = '$\\textit{Validation and Test set accuracy}
132             ↳ vs. } \\lambda$',
133         xaxis = {'title': '$\\log_{10}(\\lambda)$'},
134         yaxis = {'title': 'Model Accuracy'},
135         annotations = [
136             dict(
137                 text = '$\\textit{Used to represent }
138                     ↳ \\log_{10}(\\lambda=0)$',
139                 x = -5,
140                 y = 0.90,
141             )
142         ]
143     )
144
145     figure = go.Figure(data=data, layout=layout)
146     return py.iplot(figure,
147         ↳ filename='A1Q2.3_accuracyVsLambda')

```

```
143 fig2_3 = lambdaIGraph(tunedLambdaSummary)
144 fig2_3
```