ECE521 Winter 2017: Assignment 1

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

February 8th, 2017

Contents

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

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

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.

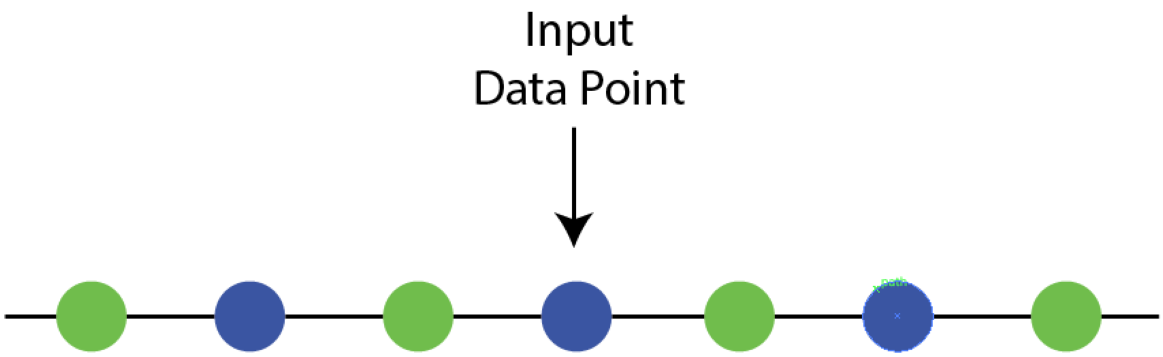

Figure 1: 1-D Dataset Illustration with classification accuracy that is periodic to k.

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,

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

We can utilise equation 2 from Probability Theory,

$$\mathbf{var}[x] = \mathbf{E}[x^2] - \mathbf{E}[x]^2 \tag{2}$$

Below are the given equations,

$$x \in \mathbb{R}^n \tag{3}$$

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

where n represents the n^{th} dimension.

N represents the number of training data.

$$d_n = x_n^i - x_n^j \tag{5}$$

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

j represents the j^{th} training data.

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

$$\mathbf{E}[d_n^2 d_m^2] = \mathbf{E}[d_n^2]\mathbf{E}[d_m^2] \tag{7}$$

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

From equations 4 and 2, it is implied that

$$\mathbf{E}[x_n] = 0 \tag{9}$$

$$\mathbf{var}[x_n] = \sigma^2 = \mathbf{E}[x_n^2] \tag{10}$$

From equations 6 and 2, it is implied

$$\mathbf{E}[d_n] = 0 \tag{11}$$

$$\mathbf{var}[d_n] = 2\sigma^2 = \mathbf{E}[d_n^2] \tag{12}$$

From equations 7 and 12,

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

From equation 1 and 5,

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

Substituting equations 1, 14 into 2,

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

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

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

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 $N \to \infty$,

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

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. $||x^{(1)}||_2^2 = ... = ||x^{(M)}||_2^2$

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:

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

```
    input:
        X is a matrix of size (B x N)
        Z is a matrix of size (C x N)
    output:
        D = matrix of size (B x C) containing
        the pairwise Euclidean distances
    """
    B = X.get_shape().as_list()[0]
    N = X.get_shape().as_list()[1]
    C = Z.get_shape().as_list()[0]
    # Ensure the N dimensions are consistent
    assert  N == Z.get_shape().as_list()[1]
    # Reshape to make use of broadcasting in Python
    X = tf.reshape(X, [B, 1, N])
    Z = tf.reshape(Z, [1, C, N])
    # The code below automatically does broadcasting.
    # Calculates the
    # pairwise squared Euclidean distance function
    D = tf.reduce_sum(tf.square(tf.sub(X, Z)), 2)
    return D
```

1.3 Making Predictions

1.3.1 Choosing Nearest Neighbour

Code snippets are included below:

```
{python}
# 1.3 Making Predictions
# 1.3.1 Choosing nearest neighbours
```

```python
# Write a vectorized Tensorflow Python function that
# takes a pairwise distance matrix
# and returns the responsibilities of the
# training examples to a new test data point.
# It should not contain loops.
# Use tf.nn.top_k
def ChooseNearestNeighbours(D, K):
    """
    input:
        D is a matrix of size (B x C)
        K is the top K responsibilities for each test input
    output:
        topK are the value of the squared distances
        for the top K values
        indices are the index of the location of
        these top K squared distances.
    """
    # Take topK of negative distances since
    # closer data ranks higher.
    topK, indices = tf.nn.top_k(tf.neg(D), K)
    return topK, indices
```

1.3.2 Prediction

Code snippets are included below:

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

```
    # training MSE loss
    # validation MSE loss
    # test MSE loss
# Choose best k using validation error = 50
def PredictKnn(trainData , testData, trainTarget,  testTarget,
 ↪  K):
    """
    input:
        trainData: Data for training KNN
        testData: Data used in testing
        trainTarget: Targets used to create prediction.
        testTarget: Targets used to calculate loss.
    output:
        loss: The mean squared loss of the prediction.
    """
    D = PairwiseDistances(testData, trainData)
    topK, indices = ChooseNearestNeighbours(D, K)
    # Select the proper outputs to be averaged
    # from the target values and average them
    trainTargetSelectedAveraged = tf.reduce_mean( \
            tf.gather(trainTarget, indices), 1)
    # Calculate the loss from the actual values
    loss = tf.reduce_mean(tf.square(tf.sub( \
            trainTargetSelectedAveraged, testTarget)))
    return loss

# Plot the prediction function for x = [0, 11]
def PredictedValues(x, trainData, trainTarget, K):
    """
    Plot the predicted values
```

```
    input:
        x = test target to plot and predict
    """
    D = PairwiseDistances(x, trainData)
    topK, indices = ChooseNearestNeighbours(D, K)
    predictedValues = tf.reduce_mean( \
            tf.gather(trainTarget, indices), 1)
    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.

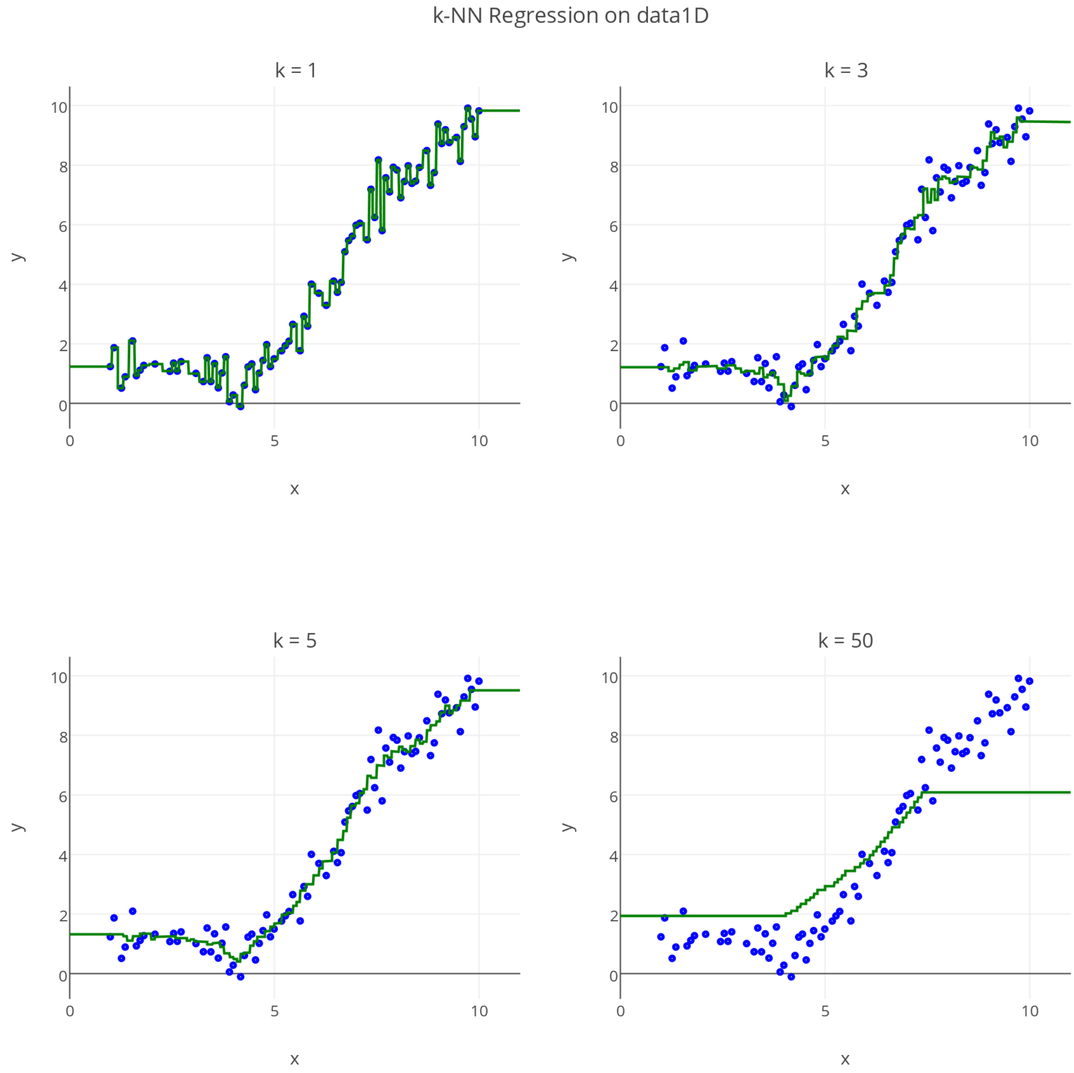

Figure 2: k-NN regression on data1D for various values of k

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

Algorithm	Test MSE Loss
Soft Decision	0.159
Gaussian Process Regression	0.380

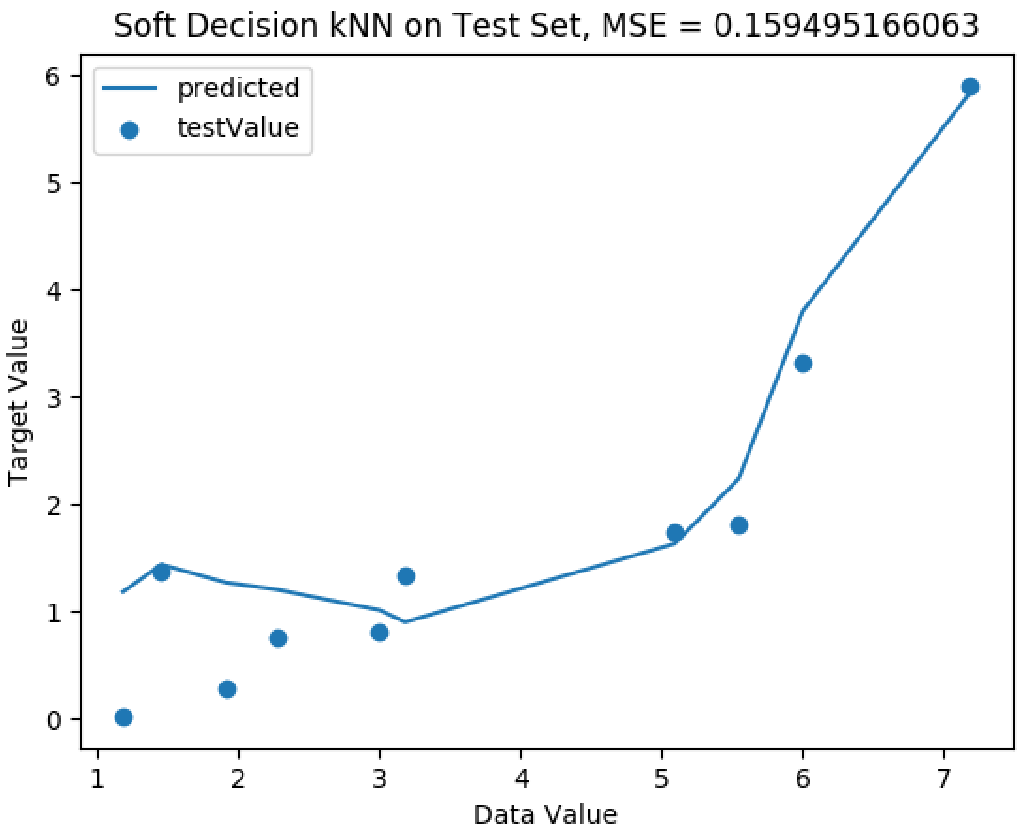

Figure 3: Soft Decision kNN on Test Set

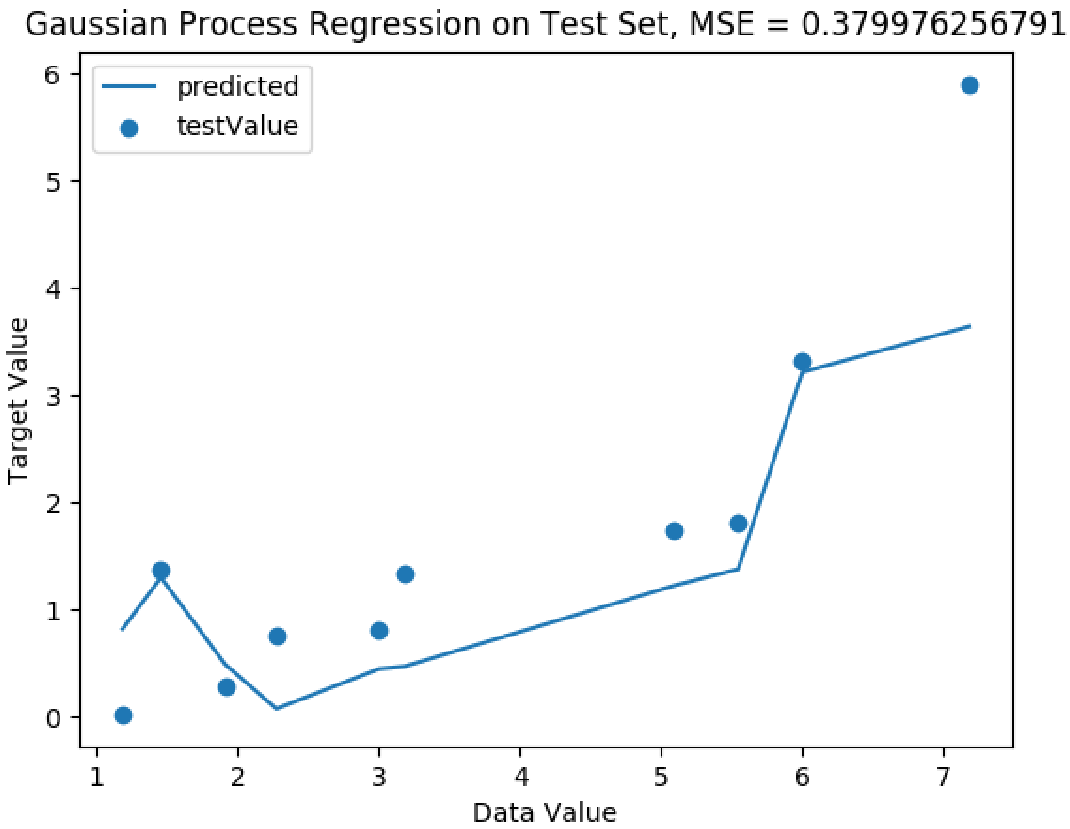

Figure 4: Gaussian Process Regression on Test Set

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

```
    # Get rid of extra dimensions from np.argsort
    inputVal = inputVal[:, :,0]
    outputVal = outputVal[:, :,0]
    return inputVal, outputVal
```

1.4.1 Soft Decisions and Gaussian Process Regression

```
{python}
# Predict values using soft decision
# 1.4.1.1 Soft Knn Decision

def PredictedValuesSoftDecision(x, trainData, trainTarget):
    # use hyper parameter of 100 as given by Jimmy.
    hyperParam = 100
    # Compute pairwise differences.
    D1 = PairwiseDistances(x, trainData)
    K1 =  tf.exp(-hyperParam*D1)
    # Get the sum term used for normalization
    sum1 = tf.reduce_sum(tf.transpose(K1), axis=0)
    # Reshape to enable broadcasting during division.
    N = sum1.get_shape().as_list()[0]
    sum1 = tf.reshape(sum1, [N,1])
    # Normalize the data using broadcast
    # to calculate the final responsibility values
    rStar = tf.div(K1, sum1)
    # Calculate the predicted value
    # using the new responsibilities, rStar
    predictedValues = tf.matmul(rStar,trainTarget)
    return predictedValues
```

```python
{python}
# Predict values using Gaussian
# 1.4.1.1 Gaussian Processes
def PredictedValuesGaussianProcesses(x, trainData,
 ↪  trainTarget):
    # use hyper parameter of 100 as given by Jimmy.
    hyperParam = 100
    D1 = PairwiseDistances(x, trainData)
    K1 =  tf.exp(-hyperParam*D1)
    D2 = PairwiseDistances(trainData, trainData)
    K2 =  tf.matrix_inverse(tf.exp(-hyperParam*D2))
    # Calculate the responsibilites, rStar
    # by normalizing using the inverse ofK2.
    rStar = tf.matmul(K1, K2)
    # Calculate the predicted value
    # using the new responsibilities, rStar
    predictedValues = tf.matmul(rStar,trainTarget)
    return predictedValue
```

1.4.2 Conditional Distribution of a Gaussian

Given an $M+1$ Gaussian random vector:

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

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

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

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

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

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

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

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

$$\left(\boldsymbol{y}_{train}^T\Lambda_{\boldsymbol{y}_{train}y^*}y^*\right)^T = y^{*T}\Lambda_{y^*\boldsymbol{y}_{train}}\boldsymbol{y}_{train} \tag{25}$$

Hence, equation 24 simplifies to:

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

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

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

Given that $\boldsymbol{y}_{train}$ and $\Sigma_{\boldsymbol{y}_{train}\boldsymbol{y}_{train}}$ are known, the conditional Gaussian Distribution $P(y^*|\boldsymbol{y}_{train})$ 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^*,

$$\Sigma^* = \Lambda^{-1}_{y^*\boldsymbol{y}_{train}} \tag{28}$$

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

$$\begin{aligned} \Sigma^{*-1}\mu^* &= \Lambda_{y^*\boldsymbol{y}_{train}}\boldsymbol{y}_{train} \\ \mu* &= \Sigma^*\Lambda_{y^*\boldsymbol{y}_{train}}\boldsymbol{y}_{train} \\ &=\Lambda^{-1}_{y^*y^*}\Lambda_{y^*\boldsymbol{y}_{train}}\boldsymbol{y}_{train} \end{aligned} \tag{29}$$

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

$$\begin{aligned} \Lambda_{y^*y^*} &= \left(\Sigma_{y^*y^*} - \Sigma_{y^*\boldsymbol{y}_{train}}\Sigma^{-1}_{\boldsymbol{y}_{train}\boldsymbol{y}_{train}}\Sigma_{\boldsymbol{y}_{train}y^*}\right)^{-1} \\ \Lambda_{y^*\boldsymbol{y}_{train}} &= -\left(\Sigma_{\boldsymbol{y}_{train}\boldsymbol{y}_{train}} - \Sigma_{\boldsymbol{y}_{train}y^*}\Sigma^{-1}_{y^*y^*}\Sigma_{y^*\boldsymbol{y}_{train}}\right)^{-1}\Sigma_{y^*\boldsymbol{y}_{train}}\Sigma^{-1}_{\boldsymbol{y}_{train}\boldsymbol{y}_{train}} \end{aligned} \tag{30}$$

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

$$\mu^* = -\Sigma_{y^*\boldsymbol{y}_{train}}\Sigma^{-1}_{\boldsymbol{y}_{train}\boldsymbol{y}_{train}}\boldsymbol{y}_{train} \tag{31}$$

$$\Sigma^* = \Sigma_{y^*y^*} - \Sigma^T_{\boldsymbol{y}_{train}y^*}\Sigma^{-1}_{\boldsymbol{y}_{train}\boldsymbol{y}_{train}}\Sigma_{\boldsymbol{y}_{train}y^*} \tag{32}$$

$$\therefore P(y^*|\boldsymbol{y}_{train}) \sim \mathcal{N}\left(-\Sigma_{y^*\boldsymbol{y}_{train}}\Sigma^{-1}_{\boldsymbol{y}_{train}\boldsymbol{y}_{train}}\boldsymbol{y}_{train}, \Sigma_{y^*y^*} - \Sigma^T_{\boldsymbol{y}_{train}y^*}\Sigma^{-1}_{\boldsymbol{y}_{train}\boldsymbol{y}_{train}}\Sigma_{\boldsymbol{y}_{train}y^*}\right) \tag{33}$$

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.

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

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

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.

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

$$\mathcal{L}_{\mathcal{W}} = W^T W \tag{37}$$

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

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

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

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

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

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

Rearranging similar terms together,

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

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

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 $\forall\ \left(W_1^T x - W_2^T x\right) \in \mathbb{R}^N$.

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 $x^m_{N+1} = 1 \; \forall m$, 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.

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

2.1.2 DeNormalization

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

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

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,

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

and for the bias, b

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

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.

Note: This note is added after submission. Realize after 1 hour after submitting that the below 3 equations is flawed as sum of products is not product of sums. Should have not brought in the sum. With a similar proof, each individual gradient would be 0. So the sum of all the gradients would be 0. Hence, proving this theorem. So shouldn't have brought in the training case. Or wait, the original equation's gradient of the sum is 0, but doesn't show that each individual gradient itself is 0. So although the individual original gradient sums to 0, each individual gradient may not necessarily be 0. Therefore, this proof, doesn't work. Sigh. Update: This equation should be true since all the terms inside is 0, which means we can factor out the 0's and group the x term on the right :) The reason why all term inside is 0 is because it is optimal with respect to the weights. What this means is that it passes by every point. What this means is that it doesn't matter if I calculate the gradient with respect to 1, 2, ..., M training points, the gradient must be 0 since it passes by all of them. Take batch size = 1, this means that each gradient must be = 0, therefore, we are able to group the X up. Proven! :) End of note after submission =D

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

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

which we can rewrite in simpler terms as equation 50

$$\frac{\partial L}{\partial W_n} = \frac{1}{M}(\sum_{i=1, i\neq n}^{N} W_i\mathbf{x_i} + W_n(\mathbf{x_n}) + Mb - \mathbf{y})(\mathbf{x_n}) = 0 \tag{50}$$

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.

$$(\sum_{i=1, i\neq n}^{N} W_i\mathbf{x_i} + W_n(\mathbf{x_n}) + Mb - \mathbf{y}) = 0 \tag{51}$$

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.

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

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.

$$L^{'} = \frac{1}{2M} \left[\left(\sum_{i=1, i \neq n}^{N} W_i^{'} \mathbf{x_i} + W_n^{'}(\mathbf{x_n^{'}}) + Mb^{'} - \mathbf{y} \right)^2 \right] \tag{53}$$

Taking the gradient with respect to W results in equation

54.

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

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.

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

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

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

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 after learning will be lower compared to the optimal weights. The biased will be lower or higher or same depending on the sign of W_n to converge into the 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 with D classes

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.

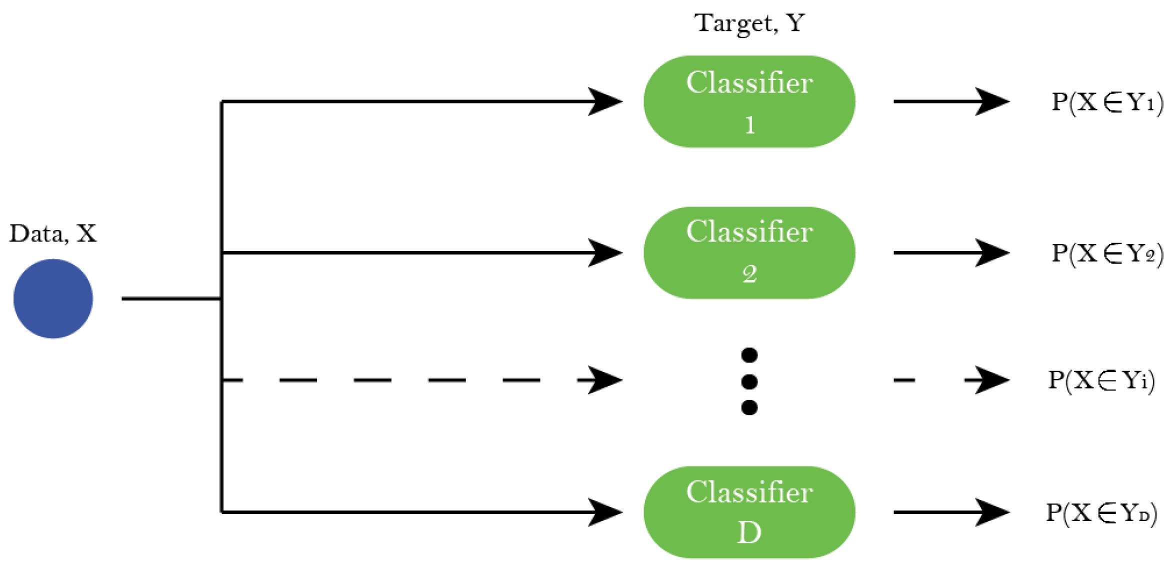

Figure 5: D binary classifiers for multi-class classification

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*

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

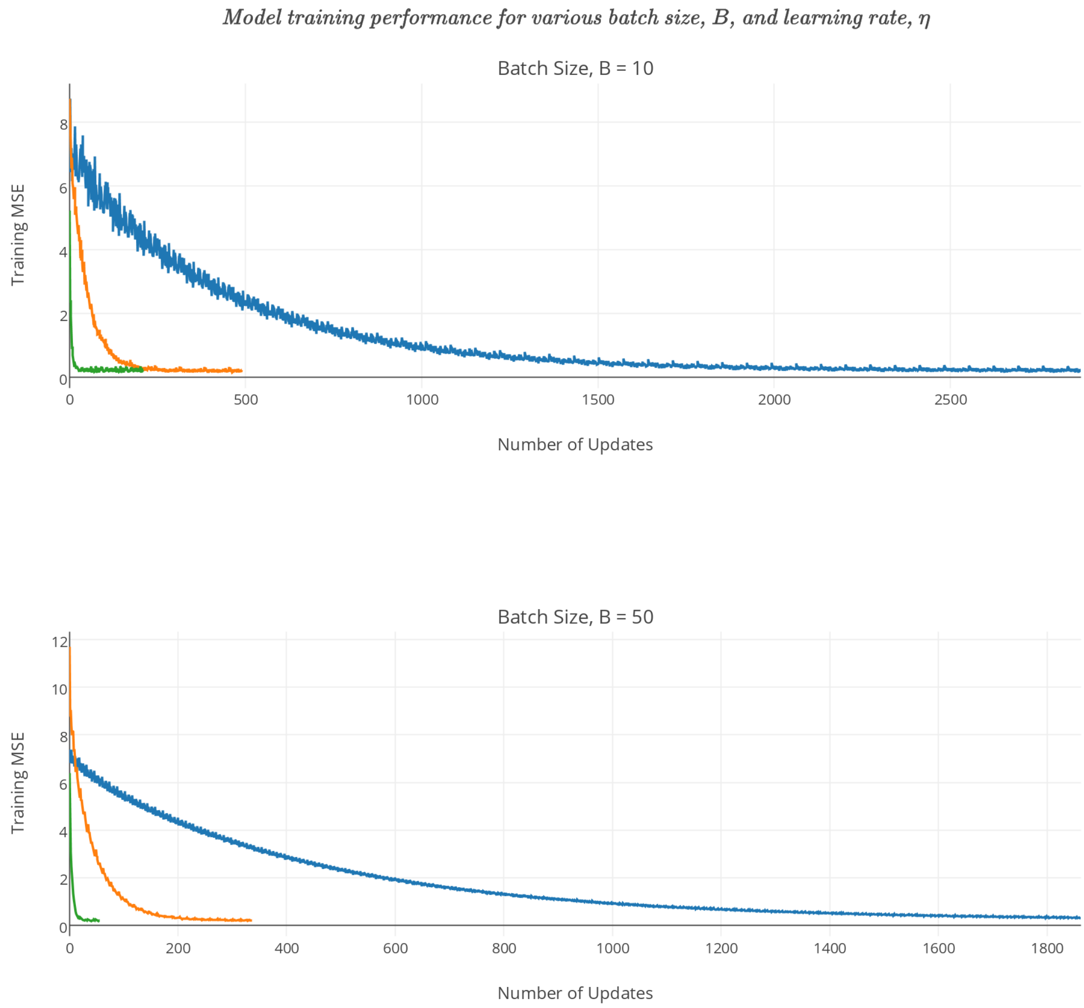

Figure 7: Subplots of training MSE against number of updates for batch sizes, $B = 10, 50$ and learning rates, η = **0.001**, **0.01**, **0.1**. ($\lambda = 1$ for all cases)

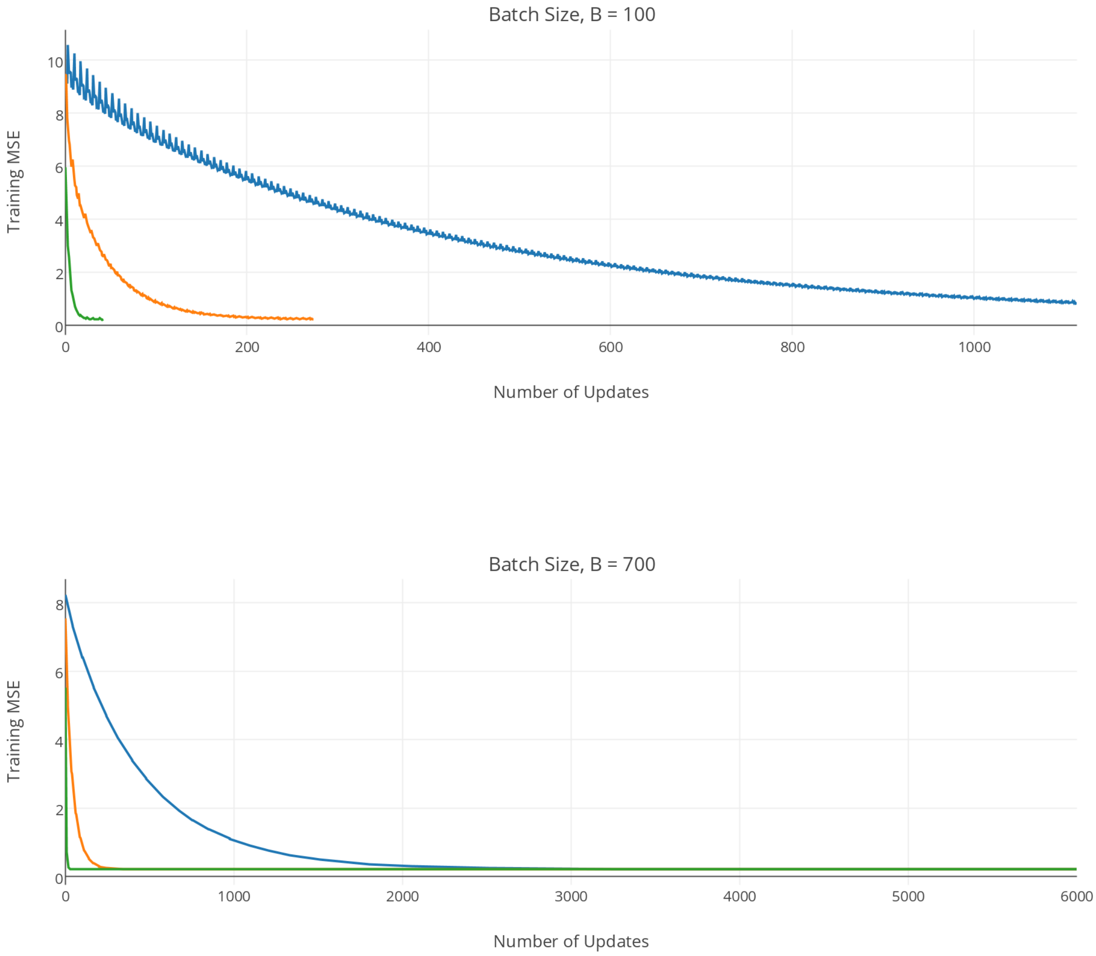

Figure 8: Subplots of training MSE against number of updates for batch sizes, $B = 100, 700$ and learning rates, η = **0.001**, **0.01**, **0.1**. ($\lambda = 1$ for all cases)

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, λ = 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 form 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

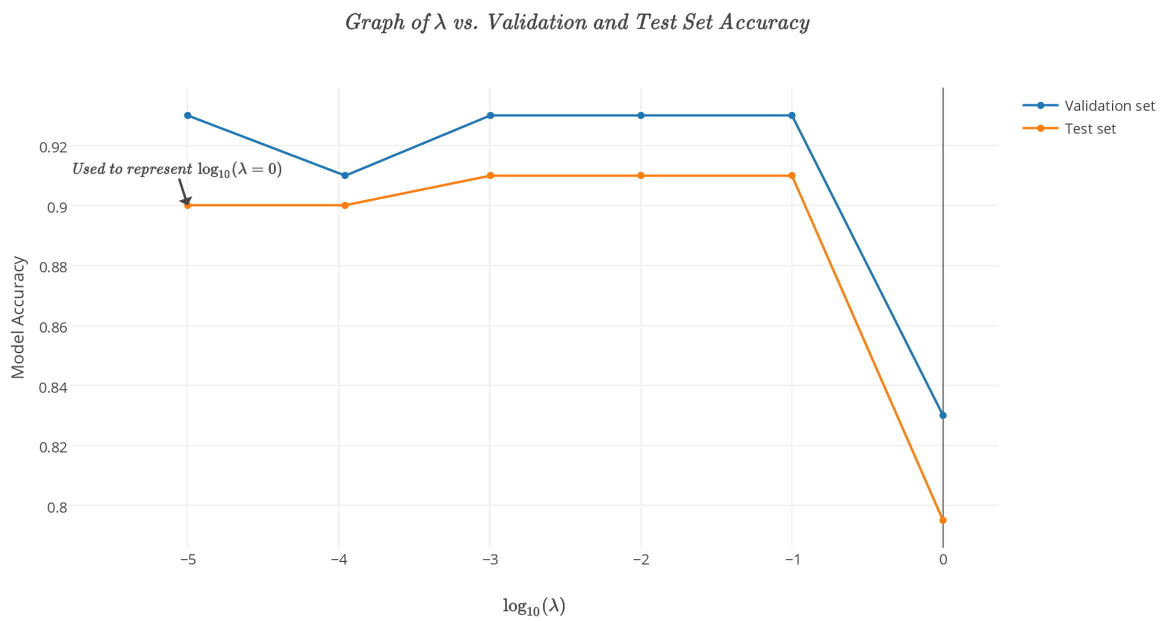

Figure 9: Graph of validation and test set accuracies against $\log_{10}(\lambda)$. ($\eta = 0.1, B = 50$)

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

```
{python}

# Assignment 1
# Optimization
# Early Stopping
# Learning rate decay
# Momentum

import tensorflow as tf
import numpy as np
import sys

# 1.2 Euclidean Distance Function
# 1.2.2 Pairwise Distances
# Write a vectorized Tensorflow Python function that
  ↪ implements
```

```
# the pairwise squared Euclidean distance function for two input matrices.
# No Loops and makes use of Tensorflow broadcasting.
def PairwiseDistances(X, Z):
    """
    input:
        X is a matrix of size (B x N)
        Z is a matrix of size (C x N)
    output:
        D = matrix of size (B x C) containing the pairwise Euclidean distances
    """
    B = X.get_shape().as_list()[0]
    N = X.get_shape().as_list()[1]
    C = Z.get_shape().as_list()[0]
    # Ensure the N dimensions are consistent
    assert  N == Z.get_shape().as_list()[1]
    # Reshape to make use of broadcasting in Python
    X = tf.reshape(X, [B, 1, N])
    Z = tf.reshape(Z, [1, C, N])
    # The code below automatically does broadcasting
    D = tf.reduce_sum(tf.square(tf.sub(X, Z)), 2)
    return D

# 1.3 Making Predictions
# 1.3.1 Choosing nearest neighbours
# Write a vectorized Tensorflow Python function that takes a pairwise distance matrix
# and returns the responsibilities of the training examples to a new test data point.
```

```python
# It should not contain loops.
# Use tf.nn.top_k
def ChooseNearestNeighbours(D, K):
    """
    input:
        D is a matrix of size (B x C)
        K is the top K responsibilities for each test input
    output:
        topK are the value of the squared distances for the
    ↪ topK
        indices are the index of the location of these squared
    ↪ distances
    """
    # Take topK of negative distances since it is the closest
     ↪ data.
    topK, indices = tf.nn.top_k(tf.neg(D), K)
    return topK, indices

# 1.3.2 Prediction
# Compute the k-NN prediction with K = {1, 3, 5, 50}
# For each value of K, compute and report:
    # training MSE loss
    # validation MSE loss
    # test MSE loss
# Choose best k using validation error = 50
def PredictKnn(trainData , testData, trainTarget,  testTarget,
    ↪ K):
    """
    input:
        trainData
```

```
        testData
        trainTarget
        testTarget
    output:
        loss
    """
    D = PairwiseDistances(testData, trainData)
    topK, indices = ChooseNearestNeighbours(D, K)
    # Select the proper outputs to be averaged from the target values and average them
    trainTargetSelectedAveraged = tf.reduce_mean(tf.gather(trainTarget, indices), 1)
    # Calculate the loss from the actual values
    # Divide by 2.0 since it's average over 2M instead of M where M = number of training data.
    loss = tf.reduce_mean(tf.square(tf.sub(trainTargetSelectedAveraged, testTarget)))/2.0
    return loss

# Plot the prediction function for x = [0, 11] on training data.
def PredictedValues(x, trainData, trainTarget, K):
    """
    Plot the predicted values
    input:
        x = test target to plot and predict
    """
    D = PairwiseDistances(x, trainData)
    topK, indices = ChooseNearestNeighbours(D, K)
```

```
    predictedValues = tf.reduce_mean(tf.gather(trainTarget,
     ↪  indices), 1)
    return predictedValues

# 1.4 Soft-Knn & Gaussian Processes
# 1.4.1.1 Soft Decisions
# Write a Tensorflow python program based on the soft k-NN
 ↪  model to compute
# predictions on the data1D.npy dataset.
# Set lambda = 100 NOT 10 as given in assignment handout
# and plot the test-set prediction of the model.

# Predict values using soft decision
def PredictedValuesSoftDecision(x, trainData, trainTarget):
    hyperParam = 100
    D1 = PairwiseDistances(x, trainData)
    K1 =  tf.exp(-hyperParam*D1)
    sum1 = tf.reduce_sum(tf.transpose(K1), axis=0)
    N = sum1.get_shape().as_list()[0]
    sum1 = tf.reshape(sum1, [N,1])
    rStar = tf.div(K1, sum1)
    predictedValues = tf.matmul(rStar,trainTarget)
    return predictedValues

# Predict values using Gaussian
# 1.4.1.1 Gaussian Processes
def PredictedValuesGaussianProcesses(x, trainData,
 ↪  trainTarget):
    hyperParam = 100
    D1 = PairwiseDistances(x, trainData)
```

```python
    K1 =  tf.exp(-hyperParam*D1)
    D2 = PairwiseDistances(trainData, trainData)
    K2 =  tf.matrix_inverse(tf.exp(-hyperParam*D2))
    rStar = tf.matmul(K1, K2)
    predictedValues = tf.matmul(rStar,trainTarget)
    return predictedValues

# Comment on the difference you observe between two programs
# Gaussian has higher loss.

# 2 Linear and Logistic Regression
# 2.2 Stochastic Gradient Descent
# Implement linear regression and stochastic gradient descent
 ↪ algorithm
# with mini-batch size B = 50.
def buildGraph(learningRate, weightDecayCoeff):
    # Variable creation
    W = tf.Variable(tf.truncated_normal(shape=[64, 1],
     ↪ stddev=0.5), name='weights')
    b = tf.Variable(0.0, name='biases')
    X = tf.placeholder(tf.float32, [None, 64], name='input_x')
    y_target = tf.placeholder(tf.float32, [None,1],
     ↪ name='target_y')
    weightDecay =
     ↪ tf.div(tf.constant(weightDecayCoeff),tf.constant(2.0))
    # Graph definition
    y_predicted = tf.matmul(X,W) + b
    # Error definition
    meanSquaredError =
     ↪ tf.reduce_mean(tf.reduce_mean(tf.square(y_predicted -
     ↪ y_target),
```

```
                                                                   reduction_indices=1,
                                                                   name='squared_error')
                                              name='mean_squared_error')
    weightDecayMeanSquareError = tf.reduce_mean(tf.reduce_mean(tf.square(weightDecay)))
    weightDecayTerm = tf.multiply(weightDecay, weightDecayMeanSquareError)
    meanSquaredError = tf.add(meanSquaredError,weightDecayTerm)

    # Training mechanism
    optimizer = tf.train.GradientDescentOptimizer(learning_rate = learningRate)
    train = optimizer.minimize(loss=meanSquaredError)
    return W, b, X, y_target, y_predicted, meanSquaredError, train


def ShuffleBatches(trainData, trainTarget):
    rngState = np.random.get_state()
    np.random.shuffle(trainData)
    np.random.set_state(rngState)
    np.random.shuffle(trainTarget)
    return trainData, trainTarget

def LinearRegression(trainData, trainTarget, validData, validTarget, testData, testTarget):
```

```
    figureCount = 30
    # 2.2.3 Generalization (done by partner)
    # Run SGD with B = 50 and use validation performance to
     choose best weight decay coefficient
    # from weightDecay = {0., 0.0001, 0.001, 0.01, 0.1, 1.}
    # Plot weightDecay vs test set accuracy. (Done by partner)
    weightDecayTrials= [0.0, 0.0001, 0.0001, 0.01, 0.1, 1.0]
    # Plot total loss function vs number of updates for the
     best learning rate found
    learningRateTrials = [0.1, 0.01, 0.001]
    # 2.2.2 Effect of the mini-batch size
    # Run with Batch Size, B = {10, 5, 100, 700} and tune the
     learning rate separately for each mini-batch size.
    # Plot  the total loss function vs the number of updates
     for each mini-batch size.
    miniBatchSizeTrials = [10, 50, 100, 700]
    learningRate = 0.01
    miniBatchSize = 10
    weightDecayCoeff = 1.0
    # for weightDecayCoeff in weightDecayTrials:
    for miniBatchSize in miniBatchSizeTrials:
        for learningRate in learningRateTrials:
            # Build computation graph
            W, b, X, y_target, y_predicted, meanSquaredError,
             train = buildGraph(learningRate,
             weightDecayCoeff)
            # Initialize session
            init = tf.global_variables_initializer()
            sess = tf.InteractiveSession()
            sess.run(init)
```

```
        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
```

```
                wList.append(currentW)
                #if not (step*miniBatchSize % 50):
                #    print "Iter: %3d, MSE-train: %4.2f, weights: %s, bias: %.2f", step, err, currentW.T, currentb
                step = step + 1
                xAxis.append(numUpdate)
                numUpdate += 1
                yTrainErr.append(errTrain)
                errValid = sess.run(meanSquaredError, feed_dict={X: validData, y_target: validTarget})
                errTest = sess.run(meanSquaredError, feed_dict={X: testData, y_target: testTarget})
                yValidErr.append(errValid)
                yTestErr.append(errTest)
            # Testing model
            # TO know what is being run
            currEpoch += 1
        print "LearningRate: " , learningRate, " Mini batch Size: ", miniBatchSize
        print "Iter: ", numUpdate
        print "Final Train MSE: ", errTrain
        print "Final Valid MSE: ", errValid
        print "Final Test MSE: ", errTest
        import matplotlib.pyplot as plt
        plt.figure(figureCount)
        figureCount = figureCount + 1
        plt.plot(np.array(xAxis), np.array(yTrainErr))
```

```python
            plt.savefig("TrainLossLearnRate" + str(learningRate) + "Batch" + str(miniBatchSize) + '.png')

            plt.figure(figureCount)
            figureCount = figureCount + 1
            plt.plot(np.array(xAxis), np.array(yValidErr))
            plt.savefig("ValidLossLearnRate" + str(learningRate) + "Batch" + str(miniBatchSize) + '.png')
            plt.figure(figureCount)
            figureCount = figureCount + 1
            plt.plot(np.array(xAxis), np.array(yTestErr))
            plt.savefig("TestLossLearnRate" + str(learningRate) + "Batch" + str(miniBatchSize) + '.png')
    return

def SortData(inputVal, outputVal):
    """
    This sorts a given test set by the dataValue before plotting it.
    """
    p = np.argsort(inputVal, axis=0)
    inputVal = np.array(inputVal)[p]
    outputVal = np.array(outputVal)[p]
    inputVal = inputVal[:, :,0]
    outputVal = outputVal[:, :,0]
    return inputVal, outputVal

```

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

```
        randIdx = np.arange(100)
        np.random.shuffle(randIdx)
        # data1D.npy
        trainData, trainTarget  = Data[randIdx[:5]], Target[randIdx[:5]]
        trainData, trainTarget  = Data[randIdx[:80]], Target[randIdx[:80]]
        validData, validTarget = Data[randIdx[80:90]], Target[randIdx[80:90]]
        testData, testTarget = Data[randIdx[90:93]], Target[randIdx[90:93]]
        testData, testTarget = Data[randIdx[90:100]], Target[randIdx[90:100]]

        #trainData, trainTarget = SortData(trainData, trainTarget)
        #validData, validTarget = SortData(validData, validTarget)
        testData, testTarget = SortData(testData, testTarget)


        # Convert to tensors from numpy
        trainData = tf.pack(trainData)
        validData = tf.pack(validData)
        testData = tf.pack(testData)
        trainTarget = tf.pack(trainTarget)
        validtarget = tf.pack(validTarget)
        testTarget = tf.pack(testTarget)
        trainMseLoss = PredictKnn(trainData, trainData, trainTarget, trainTarget, K)
```

```
        validationMseLoss = PredictKnn(trainData, validData, trainTarget, validTarget, K)
        testMseLoss = PredictKnn(trainData, testData, trainTarget, testTarget, K)
        init = tf.global_variables_initializer()
        '''
        with tf.Session() as sess:
            sess.run(init)
            print 'K ' + str(K)
            print 'trainMseLoss'
            print sess.run(trainMseLoss)
            print 'validationMseLoss'
            print sess.run(validationMseLoss)
            print 'testMseLoss'
            print sess.run(testMseLoss)
        '''
        # Plot the prediction for the x below
        x = np.linspace(0.0, 11.0, num=1000)[:, np.newaxis]
        xTensor = tf.pack(x)
        predictedValuesKnn = PredictedValues(xTensor, trainData, trainTarget, K)
        predictedValuesSoft = PredictedValuesSoftDecision(testData, trainData, trainTarget)
        predictedValuesGaussian = PredictedValuesGaussianProcesses(testData, trainData, trainTarget)
        lossSoft = tf.reduce_mean(tf.square(tf.sub(predictedValuesSoft, testTarget)))/2.0
```

```
        lossGaussian = tf.reduce_mean(tf.square(tf.sub(predictedValuesGaussian, testTarget)))/2.0
        import matplotlib.pyplot as plt
        plt.figure(0)
        init = tf.global_variables_initializer()
        with tf.Session() as sess:
            sess.run(init)
            plt.figure(K+100)
            plt.scatter(sess.run(trainData), sess.run(trainTarget))
            plt.plot(sess.run(xTensor), sess.run(predictedValuesKnn))
            fileName = str("KNN") + str(K) + str("trainingGraph.png")
            plt.savefig(fileName)

            # Plot for SoftDecision
            plt.figure(K+101)
            plt.title("Soft Decision kNN on Test Set, MSE = " + str(sess.run(lossSoft)))
            plt.xlabel("Data Value")
            plt.ylabel("Target Value")
            plt.scatter(sess.run(testData), sess.run(testTarget), label= "testValue")
            plt.plot(sess.run(testData), sess.run(predictedValuesSoft), label = "predicted")
            plt.legend()
            fileName = str("SoftDecision.png")
```

```
            plt.savefig(fileName)
            print 'SoftDecisionLoss'
            print sess.run(lossSoft)

            # Plot for Gaussian
            plt.figure(K+102)
            plt.title("Gaussian Process Regression on Test Set, MSE = " + str(sess.run(lossGaussian)))
            plt.xlabel("Data Value")
            plt.ylabel("Target Value")
            plt.scatter(sess.run(testData), sess.run(testTarget), label = "testValue")
            plt.plot(sess.run(testData), sess.run(predictedValuesGaussian), label = "predicted")
            plt.legend()
            fileName = str("ConditionalGaussian.png")
            plt.savefig(fileName)
            print 'ConditionalGaussianLoss'
            print sess.run(lossGaussian)
    # Part 2
    with np.load ("tinymnist.npz") as data :
        trainData, trainTarget = data ["x"], data["y"]
        validData, validTarget = data ["x_valid"], data ["y_valid"]
        testData, testTarget = data ["x_test"], data ["y_test"]
        LinearRegression(trainData, trainTarget,validData, validTarget, testData, testTarget)
```

3.2 Entire Code 2: FuYuan Tee's version

3.2.1 Question 1: k-Nearest Neighbour

```
import tensorflow as tf
import numpy as np

import matplotlib.pyplot as plt

import plotly.plotly as py
import plotly.graph_objs as go
from plotly import tools

import plotly.offline as pyo
pyo.init_notebook_mode(connected=True)

# Generate data
np.set_printoptions(precision=3)
np.random.seed(521)

# Generating data
Data = np.linspace(1.0, 10.0, num=100) [:, np.newaxis]
Target = np.sin(Data) + 0.1 * np.power(Data, 2) \
     + 0.5 * np.random.randn(100, 1)

# Generating a random index
randIdx = np.arange(100)
np.random.shuffle(randIdx)

# Partitioning 100 datapoints into training, validation and
```

```
# test sets consisting of 80, 10 and 10 points respectively.
trainData, trainTarget = Data[randIdx[:80]], Target[randIdx[:80]]
validData, validTarget = Data[randIdx[80:90]], Target[randIdx[80:90]]
testData, testTarget = Data[randIdx[90:]], Target[randIdx[90:]]

# Defining TensorFlow Variables

# Calculates pairwise squared Euclidean distance between matrices X and Z
# X is the input matrix which houses data points to be calculated,
# which are calculated against reference datapoints in Z
def euclideanDistance(X, Z):
    if tf.TensorShape.num_elements(X.get_shape()) == 3:
        return tf.reduce_sum(tf.square(X - tf.transpose(Z)), axis=2) # Sums feature deviations for each variable
    else:
        return tf.square(X - tf.transpose(Z), name='Euclidean_Distance_Matrix')

# Produces a BxC responsibility vector
def responsibilityVector(D, k):
    def flatten(tensor):
        return tf.reshape(tensor, [-1])

    # Obtains indices of top-K points
```

```
    _, idx = tf.nn.top_k(tf.transpose(-D), k)

    # Creates a step sequence for M values repeated k times
    M = tf.shape(D)[1]
    step_seq =
     ↪ flatten(tf.transpose(tf.reshape(tf.tile(tf.range(0,
     ↪ M), [k]), [k, M])))

    # Form new index key compatible for subsequent
     ↪ sparse_to_dense op
    sparse_idx = tf.pack([step_seq, flatten(idx)], axis=1,
     ↪ name='Sparse_Indices')

    # Forms dense tensor
    return tf.sparse_to_dense(tf.cast(sparse_idx, tf.int32), \

                                      ↪ tf.cast(tf.shape(tf.transpose(D)),
                                      ↪ tf.int32), \

                                      ↪ tf.fill([tf.shape(sparse_idx)[0]],
                                      ↪ tf.divide(1.0, tf.cast(k,
                                      ↪ tf.float32))), \
                                    validate_indices=False, \
                                    name='Responsibility_Vector')

# Function that creates a TensorFlow model
def buildGraph(k_):
    k = tf.constant(k_, name='k') # Hyperparameter
    X = tf.placeholder(tf.float32, shape=[None, None],
     ↪ name='Training_Data')
```

```
    Y = tf.placeholder(tf.float32, shape=[None, None], name='Training_Target')
    Z = tf.placeholder(tf.float32, shape=[None, None], name='Input_Data')
    T = tf.placeholder(tf.float32, shape=[None, None], name='Input_Target')

    D = euclideanDistance(X, Z)
    R = responsibilityVector(D, k)

    Y_hat = tf.matmul(R, Y, name='Y_hat')
    MSE = tf.divide(tf.reduce_sum(tf.square(T - Y_hat)), \
                    tf.scalar_mul(2, tf.cast(tf.shape(Z)[0], tf.float32)), \
                    name='Mean_Squared_Error') # Half of theorectical MSE

    return X, Y, Z, T, Y_hat, MSE

# Function used to adapt 'kNN' function based on type of dataset used for calculation
# Mode consists of either ['train', 'validation', 'test', 'full']
def selectDataPartition(mode):

    inputData = trainData
    inputTarget = trainTarget

    if mode == 'train':
        dataSource = trainData
```

```
        targetSource = trainTarget
    elif mode == 'validation':
        dataSource = validData
        targetSource = validTarget
    elif mode == 'test':
        dataSource = testData
        targetSource = testTarget
    elif mode == 'full':
        dataSource = np.linspace(0.0, 11.0, num=1000) [:, np.newaxis]
        targetSource = np.sin(dataSource) + 0.1 * np.power(dataSource, 2) \
                       + 0.5 * np.random.randn(1000, 1)

        inputData = trainData
        inputTarget = trainTarget

    return inputData, inputTarget, dataSource, targetSource

def kNN(k, mode):
    X, Y, Z, T, Y_hat, MSE = buildGraph(k)

    with tf.Session() as sess:
#           print "k: %d, mode: %s" % (k, mode)
        inputData, inputTarget, dataSource, targetSource = selectDataPartition(mode)

        error, y_pred = sess.run([MSE, Y_hat], \
                                    feed_dict={X: inputData, Y: inputTarget, \
```

```
                                                  Z: dataSource, T: targetSource
                                                 })

    return error, np.transpose(np.append(inputData, inputTarget, axis=1)), np.transpose(np.append(dataSource, y_pred, axis=1))

# Main Function
k_list = [1, 3, 5, 50]
kNN_mode = ['train', 'validation', 'test', 'full']

MSE_list = []
targetSeries = []
predictionSeries = []

# Performs kNN based on various calculation modes
for i in range(4):
    for j in range(4):
        MSE, target, prediction = kNN(k_list[i], kNN_mode[j])
        MSE_list.append(MSE)
        targetSeries.append(target)
        predictionSeries.append(prediction)
MSE_list = np.reshape(MSE_list, (4, 4))

# Generate interactive Plotly graph
def generateVisualisation(targetSeries, predictionSeries, k_list):
    subplotTitleString = []
    for i in range(len(k_list)):
```

```
        subplotTitleString.append('k = %s' % str(k_list[i]))

    fig = tools.make_subplots(rows=2, cols=2,
     ↪  subplot_titles=(subplotTitleString))

    for i in range(len(k_list)):
        traceData = go.Scatter(
            x = targetSeries[i * 4][0],
            y = targetSeries[i * 4][1],
            marker = {'color': 'blue',
                      'symbol': 200},
            mode = 'markers',
            name = 'data_k=' + str(k_list[i])
        )
        tracePred = go.Scatter(
            x = predictionSeries[4 * (i + 1) - 1][0],
            y = predictionSeries[4 * (i + 1) - 1][1],
            marker = {'color': 'green'},
            mode = 'lines',
            name = 'prediction_k=' + str(k_list[i])
        )

        fig.append_trace(traceData, i / 2 + 1, i % 2 + 1)
        fig.append_trace(tracePred, i / 2 + 1, i % 2 + 1)

        fig['layout']['xaxis'+str(i+1)].update(title='x')
        fig['layout']['yaxis'+str(i+1)].update(title='y')

    fig['layout'].update(height=900, width=950, title='k-NN
     ↪  Regression on data1D', showlegend=False)
```

```
    return py.iplot(fig, filename='A1Q1_kNN_subplot2x2')

# Output summary table
print 'Mean Squared Error Summary:'
for i in range(5):
    if i == 0:
        print "%3s %10s %10s %6s" % ('k', 'Training', 'Validation', 'Test')
    else:
        print "%3d %10.3f %10.3f %6.3f" % (k_list[i - 1], MSE_list[i - 1][0], MSE_list[i - 1][1], MSE_list[i - 1][2])

print "\n\n\n"
kNN_visuals = generateVisualisation(targetSeries, predictionSeries, k_list)
kNN_visuals
```

3.2.2 Question 2.2: Stochastic Gradient Descent

```
{python}
# Import relevant packages
import tensorflow as tf
import numpy as np
import math

import time

# Non-interactive plotting
```

```
import matplotlib.pyplot as plt
from IPython import display

# Interactive plotting
from plotly import tools
import plotly.plotly as py
import plotly.graph_objs as go
import plotly.offline as pyo
from plotly.offline import download_plotlyjs

# Configure environment
np.set_printoptions(precision=3)
np.random.seed(521)

# Activate Plotly Offline for Jupyter
pyo.init_notebook_mode(connected=True)

# Load Tiny MNIST dataset
with np.load ("tinymnist.npz") as data:
    trainData, trainTarget = data ["x"], data["y"]
    validData, validTarget = data ["x_valid"], data
     ↪ ["y_valid"]
    testData, testTarget = data ["x_test"], data ["y_test"]

# Create Tensorflow Graph
def buildGraph(eta, lambda_):
    # Model inputs
    X = tf.placeholder(tf.float32, shape=[None, None],
     ↪ name='Input')
    Y = tf.placeholder(tf.float32, shape=[None, None],
     ↪ name='Target')
```

```

    # Model variables
    W = tf.Variable(tf.truncated_normal(shape=[64, 1], stddev=0.5), name='Weights')
    b = tf.Variable(0.0, name='Biases')

    # Model parameters
    eta = tf.constant(eta, name='Learning_Rate')
    lambda_ = tf.constant(lambda_, name='L2_Regularizer')

    # Predicted target
    Y_hat = tf.matmul(X, W)

    # Mean squared error
    MSE = tf.scalar_mul(tf.divide(1.0, tf.cast(tf.shape(X)[0], tf.float32)), \
                                  tf.reduce_sum(tf.square(Y_hat - Y))) \
            + tf.scalar_mul(tf.divide(tf.cast(lambda_, tf.float32), 2.0), tf.matmul(tf.transpose(W), W))

    # Basic accuracy definition (n_correct / n_total)
    Y_hat_thresholded = tf.cast(tf.greater_equal(Y_hat, 0.5), tf.float32)
    accuracy = tf.divide(tf.reduce_sum(tf.cast(tf.equal(Y_hat_thresholded, Y), tf.float64)), \
                                 tf.cast(tf.shape(X)[0], tf.float64))

```

```
    # Basic gradient descent optimizer
    optimizer = tf.train.GradientDescentOptimizer(eta).minimize(MSE)

    return W, b, X, Y, Y_hat, MSE, accuracy, optimizer
```

3.2.3 Question 2.2.1: Tuning the learning rate

```
{python}
# Tune learning rate
MAX_ITER = 2000
def tuneLearningRate(etaList, batchSize=50, lambda_=1):
    # Returns the i-th batch of training data and targets
    # Generates a new, reshuffled batch once all previous batches are fed
    def getNextTrainingBatch(currentIter):
        currentBatchNum = currentIter % (trainData.shape[0] / batchSize)
        if currentBatchNum == 0:
            np.random.shuffle(randIdx)
        # print 'Iteration: %4d, BatchCap: %2d, BatchNum: %2d' % (currentIter, trainData.shape[0] / batchSize, currentBatchNum)
        lowerBoundIdx = currentBatchNum * batchSize
        upperBoundIdx = (currentBatchNum + 1) * batchSize
        return trainData[lowerBoundIdx:upperBoundIdx], trainTarget[lowerBoundIdx:upperBoundIdx]

    # Generate updated plots for training and validation MSE
```

```python
    def plotMSEGraph(MSEList, param):
        label = '$\eta$ = ' + str(param)
        label_classification = ['train.', 'valid.']

        display.clear_output(wait=True)
        plt.figure(figsize=(8,5), dpi=200)

        for i, MSE in enumerate(MSEList):
            plt.plot(range(len(MSE)), MSE, '.', markersize=3, label=label+' '+label_classification[i])

        plt.axis([0, MAX_ITER, 0, np.amax(MSEList)])
        plt.legend()
        plt.show()

    # Calculates the ratio between the n-th average epoch MSE and the (n-1)-th average epoch MSE
    def ratioAverageEpochMSE(currentValidMSE):
        averageN = np.average(currentValidMSE[-(np.arange(epochSize - 1) + 1)])
        averageNlessOne = np.average(currentValidMSE[-(np.arange(epochSize - 1) + epochSize)])
        return averageN / averageNlessOne

    # Returns True if the average epoch validation MSE is at least 99% of the previous epoch average.
    # i.e. Returns True if the average learnings between epoch is less than +1%
```

```
    # Otherwise, returns False
    def shouldStopEarly(currentValidMSE):
        if currentValidMSE.shape[0] < 2 * epochSize:
            return False
        return True if (ratioAverageEpochMSE(currentValidMSE) >= 0.99) else False

    summaryList = []
    randIdx = np.arange(trainData.shape[0])
    epochSize = trainData.shape[0] / batchSize

    for eta in etaList:
        W, b, X, Y, Y_hat, MSE, accuracy, optimizer = buildGraph(eta, lambda_)

        with tf.Session() as sess:
            tf.global_variables_initializer().run()

            # Creates blank training and validation MSE arrays for the Session
            currentTrainMSE = np.array([])[:, np.newaxis]
            currentValidMSE = np.array([])[:, np.newaxis]

            # Runs update
            currentIter = 0
            while currentIter <= MAX_ITER:
                inputData, inputTarget = getNextTrainingBatch(currentIter)

                _, trainError = sess.run([optimizer, MSE], feed_dict={X: inputData, Y: inputTarget})
```

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

                currentTrainMSE = np.append(currentTrainMSE, trainError)
                currentValidMSE = np.append(currentValidMSE, validError)

                # Update graph of training and validation MSE arrays
                if (currentIter < 3) or (currentIter % 500 == 0):
                    plotMSEGraph([currentTrainMSE, currentValidMSE], eta)

                # At every epoch, check for early stopping possibilty. If so, breaks from while loop
                if currentIter % epochSize == 0:
                    if shouldStopEarly(currentValidMSE):
                        break

                currentIter += 1

        # Save session results as dictionary and appends to MSEsummaryList
        summaryList.append(
            {
                'eta': eta,
                'B': batchSize,
                'lambda': lambda_,
```

```
                'numIter': currentIter + 1,
                'epoch': float(currentIter + 1) /
                 ↪  (trainData.shape[0] / batchSize),
                'trainMSE': currentTrainMSE,
                'validMSE': currentValidMSE,
            }
        )

    return summaryList


# Main Function
etaList = [0.001, 0.01, 0.1]
tunedEtaSummary = tuneLearningRate(etaList)

# Output summary table
for summary in tunedEtaSummary:
    print 'eta: %.3f, numIter: %d, validMSE: %.3f' %
     ↪  (summary['eta'], summary['numIter'],
     ↪  summary['validMSE'][-1])

# Produce interactive graph for best learning rate
def etaIGraph(tunedEtaSummary):
    # Create plot for each summary
    traceList = []
    for summary in tunedEtaSummary:
        traceList.append(
            go.Scatter(
                x = range(summary['numIter'] + 1),
                y = summary['trainMSE'],
```

```
                name = '$\\eta = ' + str(summary['eta']) + '$'
            )
        )
    data = go.Data(traceList)

    # Create figure layout
    layout = go.Layout(
        title = '$\\textit{Training performance for various learning rates, } \\eta$',
        xaxis = {'title': 'Number of Updates'},
        yaxis = {'title': 'Training MSE'},
    )

    figure = go.Figure(data=data, layout=layout)
    return py.iplot(figure, filename='A1Q2.1_bestEtaGraph')
fig2_1 = etaIGraph(tunedEtaSummary)
fig2_1
```

3.2.4 Question 2.2.2: Effect of the mini-batch size

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

```
        if currentBatchNum == 0:
            np.random.shuffle(randIdx)
        # print currentBatchNum + 1
        lowerBoundIdx = currentBatchNum * batchSize
        upperBoundIdx = (currentBatchNum + 1) * batchSize
        return trainData[lowerBoundIdx:upperBoundIdx], trainTarget[lowerBoundIdx:upperBoundIdx]

    # Generate updated plots for training and validation MSE
    def plotMSEGraph(MSEList, param):
        label = '$B$ = ' + str(param[0]) + ', $\eta$: ' + str(param[1])
        label_classification = ['train.', 'valid.']

        display.clear_output(wait=True)
        plt.figure(figsize=(8,5), dpi=200)

        for i, MSE in enumerate(MSEList):
            plt.plot(range(len(MSE)), MSE, '.', markersize=3, label=label+'\n'+label_classification[i])

        plt.axis([0, MAX_ITER, 0, np.amax(MSEList)])
        plt.legend()
        plt.show()

    # Calculates the ratio between the n-th average epoch MSE and the (n-1)-th average epoch MSE
    def ratioAverageEpochMSE(currentValidMSE):
        averageN = np.average(currentValidMSE[-(np.arange(epochSize - 1) + 1)])
```

```python
        averageNlessOne = np.average(currentValidMSE[-(np.arange(epochSize - 1) + epochSize)])
        return averageN / averageNlessOne

    # Returns True if the average epoch validation MSE is at least 99% of the previous epoch average.
    # i.e. Returns True if the average learnings between epoch is less than +1%
    # Otherwise, returns False
    def shouldStopEarly(currentValidMSE):
        if currentValidMSE.shape[0] < 2 * epochSize:
            return False
        return True if (ratioAverageEpochMSE(currentValidMSE) >= 0.99) else False

    summaryList = []
    randIdx = np.arange(trainData.shape[0])

    for batchSize in batchSizeList:
        epochSize = trainData.shape[0] / batchSize
        batchSummary = []
        for eta in etaList:
            W, b, X, Y, Y_hat, MSE, accuracy, optimizer = buildGraph(eta, lambda_)

            with tf.Session() as sess:
                tf.global_variables_initializer().run()

                # Creates blank training and validation MSE arrays for the Session
```

```
                currentTrainMSE = np.array([])[:, np.newaxis]
                currentValidMSE = np.array([])[:, np.newaxis]

                # Runs update
                currentIter = 0
                while currentIter <= MAX_ITER:
                    inputData, inputTarget = getNextTrainingBatch(currentIter)

                    _, trainError = sess.run([optimizer, MSE], feed_dict={X: inputData, Y: inputTarget})
                    validError = sess.run([MSE], feed_dict={X: validData, Y: validTarget})

                    currentTrainMSE = np.append(currentTrainMSE, trainError)
                    currentValidMSE = np.append(currentValidMSE, validError)

                    # Update graph of training and validation MSE arrays
                    if (currentIter < 3) or (currentIter % 500 == 0):
                        plotMSEGraph([currentTrainMSE, currentValidMSE], [batchSize, eta])

                    # At every epoch, check for early stopping possibilty. If so, breaks from while loop
```

```
                    if currentIter % epochSize == 0:
                        if shouldStopEarly(currentValidMSE):
                            break

                    currentIter += 1

            # Save session results as dictionary and appends to MSEsummaryList
            batchSummary.append(
                {
                    'eta': eta,
                    'B': batchSize,
                    'lambda': lambda_,
                    'numIter': currentIter + 1,
                    'epoch': float(currentIter + 1) / (trainData.shape[0] / batchSize),
                    'trainMSE': currentTrainMSE,
                    'validMSE': currentValidMSE,
                }
            )
        summaryList.append(batchSummary)

    return summaryList

# Main function
etaList = [0.001, 0.01, 0.1]
batchSizeList = [10, 50, 100, 700]
tunedBatchSizeSummary = tuneBatchSize(etaList, batchSizeList)

# Output summary table:
```

```
for batchSummary in tunedBatchSizeSummary:
    for summary in batchSummary:
        print 'B: %5d, eta: %5.3f, numIter: %5d, validMSE: %3.3f' % \
            (summary['B'], summary['eta'], summary['numIter'], summary['validMSE'][-1])

# Generate interactive Plotly plot
def batchSizeIGraphSubplot(tunedBatchSizeSummary):

    # Define subplot title
    subplotTitle = []
    for batchSummary in tunedBatchSizeSummary:
        subplotTitle.append('Batch Size, B  = ' + str(batchSummary[0]['B']))

    # Define subplot figure
    figure = tools.make_subplots(rows=4, cols=1, subplot_titles=(subplotTitle))

    # Define color list
    colorList = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b']

    # Create plot for each summary
    for i, batchSummary in enumerate(tunedBatchSizeSummary):
        traceList = []
        for j, summary in enumerate(batchSummary):
            trace = go.Scatter(
                x = range(summary['numIter'] + 1),
```

```
                y = summary['trainMSE'],
                marker = {'color': colorList[j]},
                name = '$B=' + str(summary['B']) + ', \\eta=' + str(summary['eta']) + '$'
            )
            figure.append_trace(trace, i + 1, 1)
            figure['layout']['xaxis'+str(i+1)].update(title='Number of Updates')
            figure['layout']['yaxis'+str(i+1)].update(title='Training MSE')

    # Create figure layout
    figure['layout'].update(
        height = 1800,
        title = '$\\textit{Model training performance for various batch size, } B' + \
                '\\textit{, and learning rate, } \\eta$',
        showlegend = False
    )

    return py.iplot(figure, filename='A1Q2.2_batch_size_subplot2x2')
fig2_2_visual = batchSizeIGraphSubplot(tunedBatchSizeSummary)
fig2_2_visual
```

3.2.5 Question 2.2.3: Generalization

```
{python}
# Tune weight decay regularizer
MAX_ITER = 2000
def tuneLambda(lambdaList, eta=0.1, batchSize=50):
    # Returns the i-th batch of training data and targets
    # Generates a new, reshuffled batch once all previous
     ↪ batches are fed
    def getNextTrainingBatch(currentIter):
        currentBatchNum = currentIter % (trainData.shape[0] /
         ↪ batchSize)
        if currentBatchNum == 0:
            np.random.shuffle(randIdx)
        lowerBoundIdx = currentBatchNum * batchSize
        upperBoundIdx = (currentBatchNum + 1) * batchSize
        return trainData[lowerBoundIdx:upperBoundIdx],
         ↪ trainTarget[lowerBoundIdx:upperBoundIdx]

    # Generate updated plots for training and validation MSE
    def plotMSEGraph(MSEList, param):
        label = '$\lambda$ = ' + str(param)
        label_classification = ['train.', 'valid.']

        display.clear_output(wait=True)
        plt.figure(figsize=(8,5), dpi=200)

        for i, MSE in enumerate(MSEList):
            plt.plot(range(len(MSE)), MSE, '-', label=label+'
             ↪ '+label_classification[i])
```

```
        plt.axis([0, MAX_ITER, 0, np.amax(MSEList)])
        plt.legend()
        plt.show()

    # Calculates the ratio between the n-th average epoch MSE and the (n-1)-th average epoch MSE
    def ratioAverageEpochMSE(currentValidMSE):
        averageN = np.average(currentValidMSE[-(np.arange(epochSize - 1) + 1)])
        averageNlessOne = np.average(currentValidMSE[-(np.arange(epochSize - 1) + epochSize)])
        return averageN / averageNlessOne

    # Returns True if the average epoch validation MSE is at least 99% of the previous epoch average.
    # i.e. Returns True if the average learnings between epoch is less than +1%
    # Otherwise, returns False
    def shouldStopEarly(currentValidMSE):
        if currentValidMSE.shape[0] < 2 * epochSize:
            return False
        return True if (ratioAverageEpochMSE(currentValidMSE) >= 0.99) else False

    summaryList = []
    randIdx = np.arange(trainData.shape[0])
    epochSize = trainData.shape[0] / batchSize
```

```
    for lambda_ in lambdaList:
        W, b, X, Y, Y_hat, MSE, accuracy, optimizer = buildGraph(eta, lambda_)

        with tf.Session() as sess:
            tf.global_variables_initializer().run()

            # Creates blank training and validation MSE arrays for the Session
            currentTrainMSE = np.array([])[:, np.newaxis]
            currentValidMSE = np.array([])[:, np.newaxis]

            # Runs update
            currentIter = 0
            while currentIter <= MAX_ITER:
                inputData, inputTarget = getNextTrainingBatch(currentIter)

                _, trainError = sess.run([optimizer, MSE], feed_dict={X: inputData, Y: inputTarget})
                validError = sess.run([MSE], feed_dict={X: validData, Y: validTarget})

                currentTrainMSE = np.append(currentTrainMSE, trainError)
                currentValidMSE = np.append(currentValidMSE, validError)

                # Update graph of training and validation MSE arrays
```

```
            if (currentIter < 3) or (currentIter % 500 == 0):
                plotMSEGraph([currentTrainMSE, currentValidMSE], lambda_)

            # At every epoch, check for early stopping possibilty. If so, breaks from while loop
            if currentIter % epochSize == 0:
                if shouldStopEarly(currentValidMSE):
                    break

            currentIter += 1

        # Compute validation and test accuracy
        validAccuracy = sess.run(accuracy, feed_dict={X: validData, Y: validTarget})
        testAccuracy = sess.run(accuracy, feed_dict={X: testData, Y: testTarget})

    # Save session results as dictionary and appends to MSEsummaryList
    summaryList.append(
        {
            'eta': eta,
            'B': batchSize,
            'lambda': lambda_,
            'numIter': currentIter + 1,
            'epoch': float(currentIter + 1) / (trainData.shape[0] / batchSize),
            'trainMSE': currentTrainMSE,
```

```
                    'validMSE': currentValidMSE,
                    'validAccuracy': validAccuracy,
                    'testAccuracy': testAccuracy
                }
        )

    return summaryList

# Main Function
lambdaList = [0.0, 0.0001, 0.001, 0.01, 0.1, 1.0]
tunedLambdaSummary = tuneLambda(lambdaList)

# Output summary table
for summary in tunedLambdaSummary:
    print 'lambda: %5.4f, numIter: %5d, validMSE: %5.3f,
     ↪  validAcc: %3.3f, testAcc: %3.3f' % \
        (summary['lambda'], summary['numIter'],
         ↪  summary['validMSE'][-1], summary['validAccuracy'],
         ↪  summary['testAccuracy'])

# Produce interactive Plotly graph
def lambdaIGraph(tunedLambdaSummary):
    # Create plot for each summary
    trace1 = go.Scatter(
        x = [np.log10(summary['lambda'] + 1e-5) for summary in
         ↪  tunedLambdaSummary],
        y = [summary['validAccuracy'] for summary in
         ↪  tunedLambdaSummary],
        name = 'Validation set accuracy'
    )
```

```
    trace2 = go.Scatter(
        x = [np.log10(summary['lambda'] + 1e-5) for summary in
         ↪ tunedLambdaSummary],
        y = [summary['testAccuracy'] for summary in
         ↪ tunedLambdaSummary],
        name = 'Test set accuracy'
    )

    data = go.Data([trace1, trace2])

    # Create figure layout
    layout = go.Layout(
        title = '$\\textit{Validation and Test set accuracy
         ↪ vs. } \\lambda$',
        xaxis = {'title': '$\\log_{10}(\\lambda)$'},
        yaxis = {'title': 'Model Accuracy'},
        annotations = [
            dict(
                text = '$\\textit{Used to represent }
                 ↪ \\log_{10}(\\lambda=0)$',
                x = -5,
                y = 0.90,
            )
        ]
    )

    figure = go.Figure(data=data, layout=layout)
    return py.iplot(figure,
     ↪ filename='A1Q2.3_accuracyVsLambda')
```

```
fig2_3 = lambdaIGraph(tunedLambdaSummary)
fig2_3
```