ECE 521 Inference Algorithms and Machine Learning

Assignment #1

Student Name: Shengze Gao(ID: 1002935942)
Student Name: Jingxiong Luo(ID: 1002772332)

Feb. 7th, 2017

Table of Content

Percentage of Contribution:

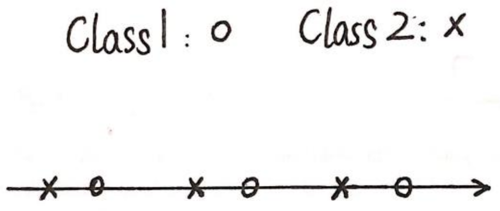
JingXiong Luo: 50% Shengze Gao: 50%

1. k-Nearest Neighbor

1.1 Geometry of k-NN

1.1.1

Suppose there is a 1-D dataset of two classes shown as the following figure:



If we apply the train data as test points. The relationship of hyper-parameter k and classification accuracy of the k-NN classifier is shown as follows:

k	classification accuracy
1	100%
2	50%
3	0%
4	50%
5	100%
6	50%

It is clearly shown that the classification accuracy of the k-NN classifier on the training set is a periodic function of the hyper-parameter k.

1.1.2

The detailed process of verification is shown in the following capture:

$$Var\left(\frac{\|x^{(i)}-x^{(j)}\|_2^2}{E[\|x^{(i)}-x^{(j)}\|_2^2]}\right) = Var\left(\frac{\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2}{E(\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2)}\right) = E\left(\frac{[\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2]^2}{E^2(\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2)}\right) - E^2\left(\frac{\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2}{E(\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2)}\right)$$

$$= E\left(\frac{(\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2)^2}{E^2(\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2)}\right) - E^2\left(\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2\right)\cdot\frac{1}{E^2(\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2)}$$

$$= E\left(\frac{(\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2)^2}{E^2(\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2)}\right) - 1$$

$$= \frac{E\left([\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2]^2\right)}{E^2(\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2)} - 1$$

$$= \frac{E\left([\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2]^2\right)}{(N\cdot 2\sigma^2)^2} - 1 = \frac{Var(\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2) + E^2(\sum_{n=1}^{N}(x_n^{(i)}-x_n^{(j)})^2)}{4N^2\sigma^4} - 1$$

$$= \frac{\sum_{n=1}^{N} Var[(x_n^{(i)}-x_n^{(j)})^2] + 4N^2\sigma^4}{4N^2\sigma^4} - 1$$

$$= \frac{\sum_{n=1}^{N}(E(d_n^4)-E^2(d_n^2)) + 4N^2\sigma^4}{4N^2\sigma^4} - 1 = \frac{8N\sigma^4 + 4N^2\sigma^4}{4N^2\sigma^4} - 1$$

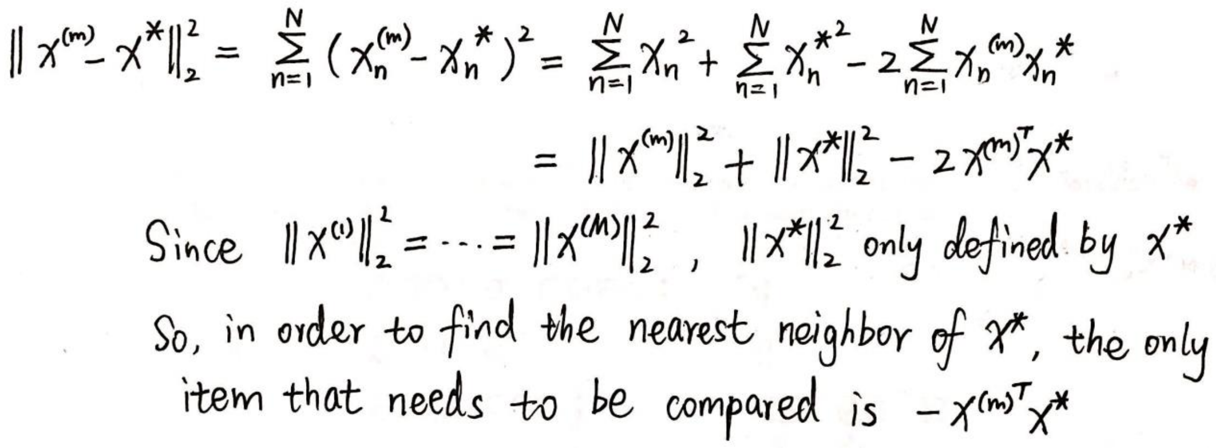
$$= \frac{2+N}{N} - 1$$

(2+N) / N – 1 = 2 / N
It is clearly to see that when N approaches to infinity, Var vanishes (Var = 0).

1.2 Euclidean distance function

1.2.1

The detailed process of verification is shown in the following capture:



$$\|x^{(m)}-x^*\|_2^2 = \sum_{n=1}^{N} (x_n^{(m)}-x_n^*)^2 = \sum_{n=1}^{N} x_n^2 + \sum_{n=1}^{N} x_n^{*2} - 2\sum_{n=1}^{N} x_n^{(m)} x_n^*$$

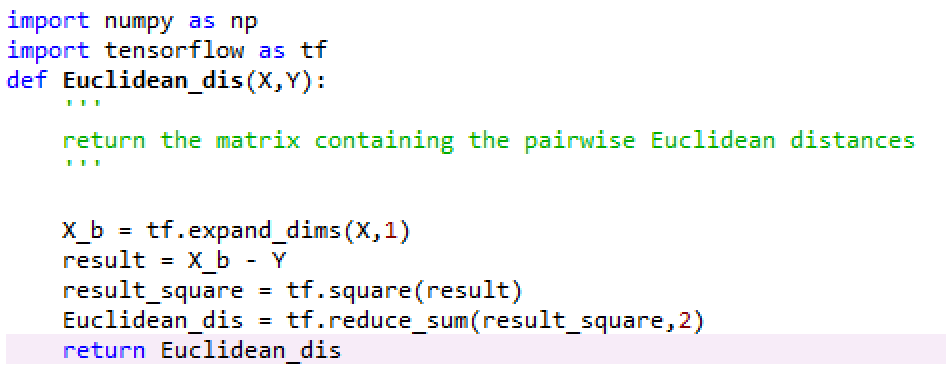
$$= \|x^{(m)}\|_2^2 + \|x^*\|_2^2 - 2x^{(m)T}x^*$$

Since $\|x^{(1)}\|_2^2 = \cdots = \|x^{(M)}\|_2^2$, $\|x^*\|_2^2$ only defined by x^*

So, in order to find the nearest neighbor of x^*, the only item that needs to be compared is $-x^{(m)T}x^*$

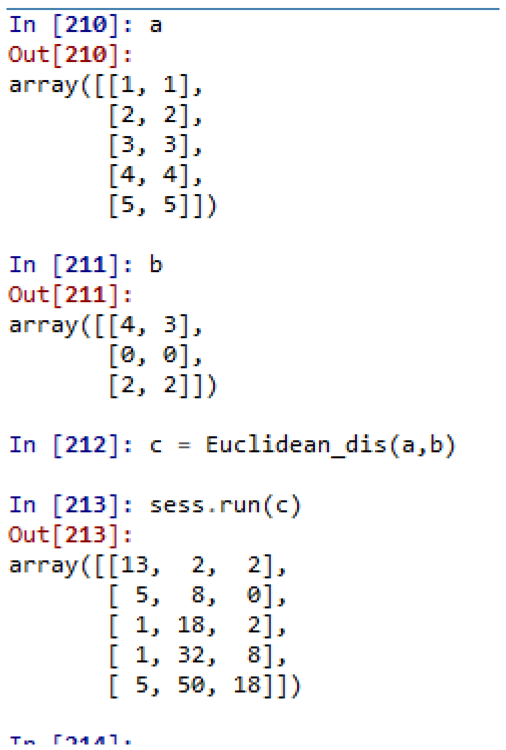
1.2.2

The following capture displaying the snippet of the Python code and an example of how this specific function works.



```python
import numpy as np
import tensorflow as tf
def Euclidean_dis(X,Y):
    '''
    return the matrix containing the pairwise Euclidean distances
    '''

    X_b = tf.expand_dims(X,1)
    result = X_b - Y
    result_square = tf.square(result)
    Euclidean_dis = tf.reduce_sum(result_square,2)
    return Euclidean_dis
```

Snippets of the Python code



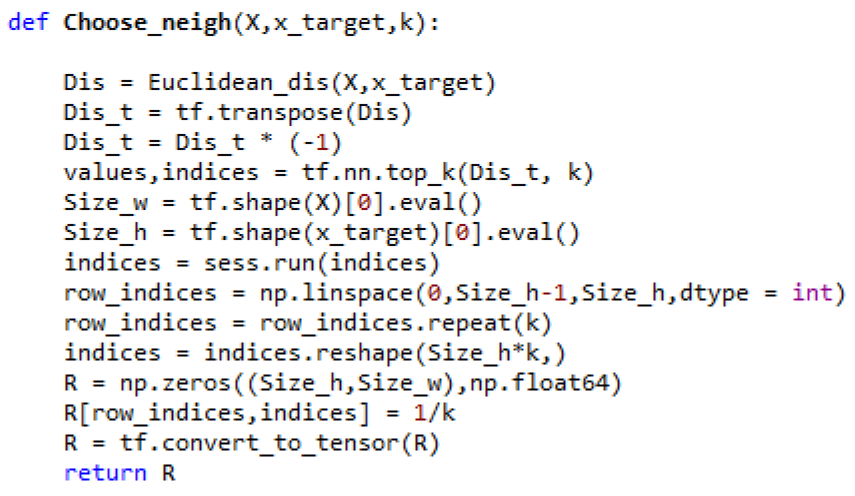
```
In [210]: a
Out[210]:
array([[1, 1],
       [2, 2],
       [3, 3],
       [4, 4],
       [5, 5]])

In [211]: b
Out[211]:
array([[4, 3],
       [0, 0],
       [2, 2]])

In [212]: c = Euclidean_dis(a,b)

In [213]: sess.run(c)
Out[213]:
array([[13,  2,  2],
       [ 5,  8,  0],
       [ 1, 18,  2],
       [ 1, 32,  8],
       [ 5, 50, 18]])

In [214]:
```

In this example, a, b are inputs of the function, c returns the Euclidean distance matrix

1.3 Making predictions

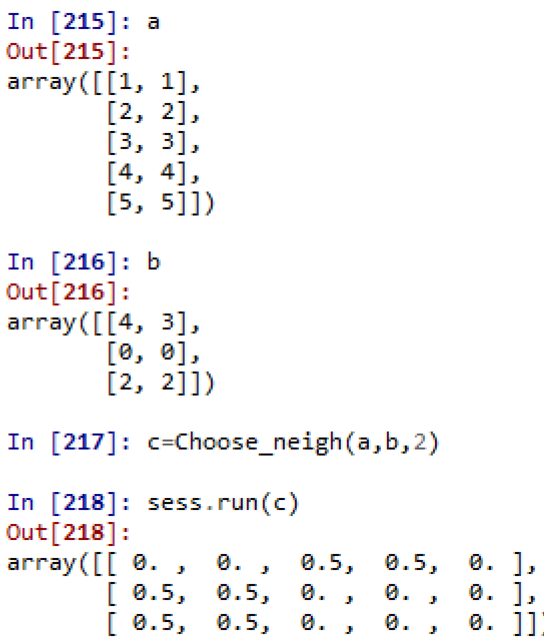
1.3.1

The snippet of choosing the nearest neighbors is shown as following. It also displays an example to prove the function of this code. Note that it returns the responsibility vector r* from this program.



```
def Choose_neigh(X,x_target,k):

    Dis = Euclidean_dis(X,x_target)
    Dis_t = tf.transpose(Dis)
    Dis_t = Dis_t * (-1)
    values,indices = tf.nn.top_k(Dis_t, k)
    Size_w = tf.shape(X)[0].eval()
    Size_h = tf.shape(x_target)[0].eval()
    indices = sess.run(indices)
    row_indices = np.linspace(0,Size_h-1,Size_h,dtype = int)
    row_indices = row_indices.repeat(k)
    indices = indices.reshape(Size_h*k,)
    R = np.zeros((Size_h,Size_w),np.float64)
    R[row_indices,indices] = 1/k
    R = tf.convert_to_tensor(R)
    return R
```

Snippets of the Python code



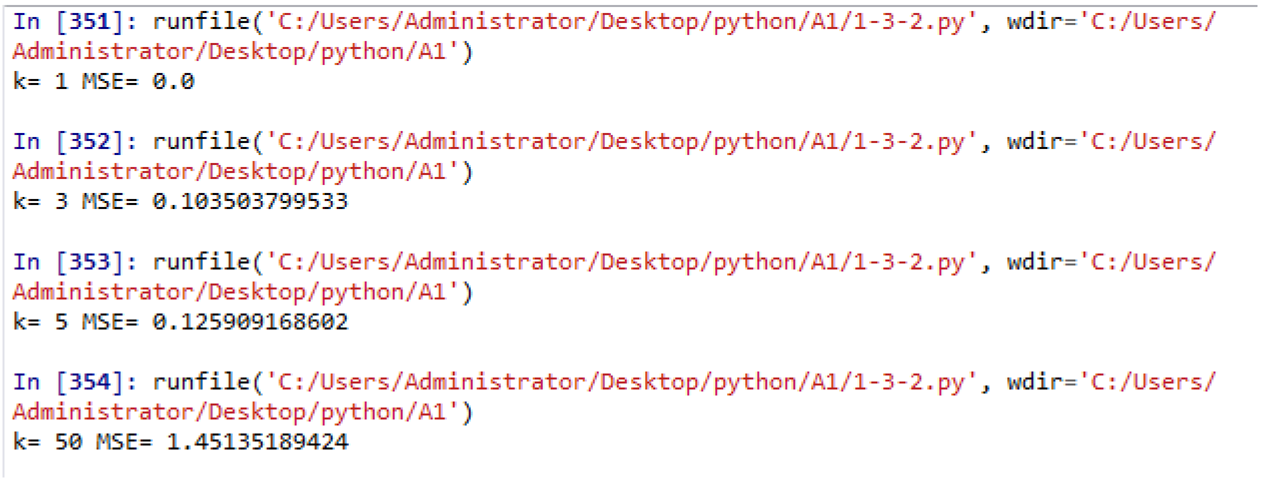
```
In [215]: a
Out[215]:
array([[1, 1],
       [2, 2],
       [3, 3],
       [4, 4],
       [5, 5]])

In [216]: b
Out[216]:
array([[4, 3],
       [0, 0],
       [2, 2]])

In [217]: c=Choose_neigh(a,b,2)

In [218]: sess.run(c)
Out[218]:
array([[ 0. ,  0. ,  0.5,  0.5,  0. ],
       [ 0.5,  0.5,  0. ,  0. ,  0. ],
       [ 0.5,  0.5,  0. ,  0. ,  0. ]])
```

In this example, a, b are inputs of the function, c returns the responsibilities matrix

1.3.2

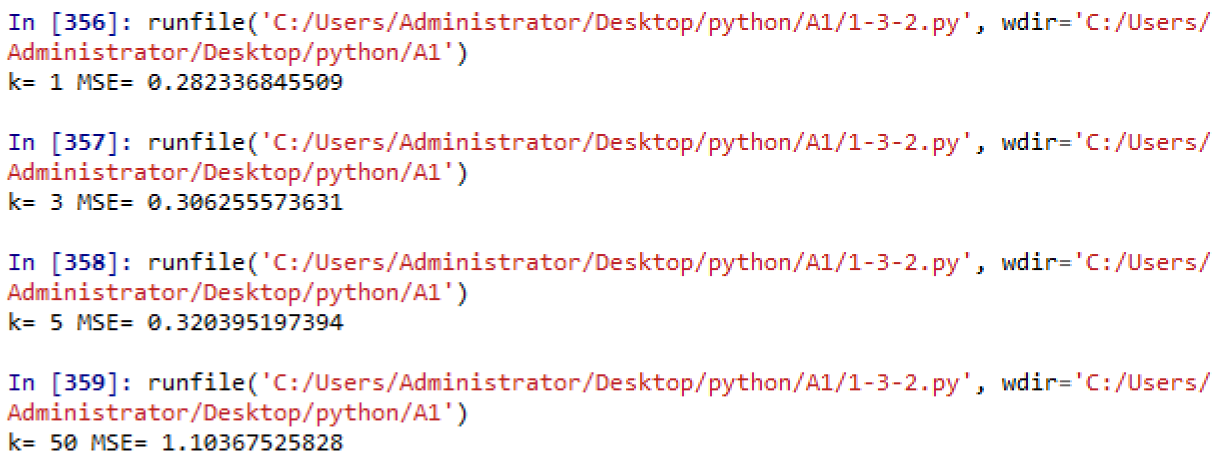
Note that the Python code in this section is attached in Appendix.

By setting the value of k to {1,3,5,50} separately, the corresponding training MSE loss, validation MSE loss and test MSE loss are shown in the following captures.



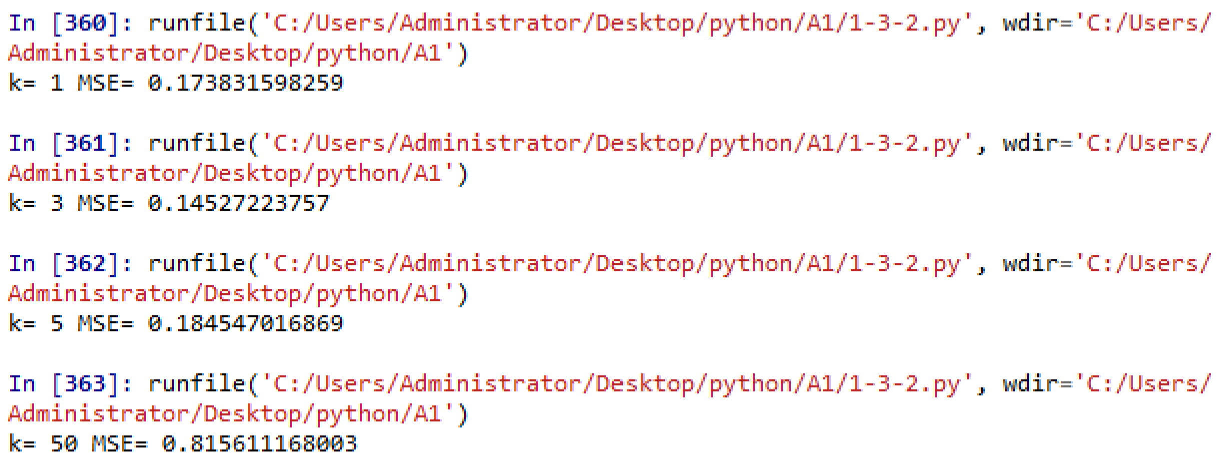
```
In [351]: runfile('C:/Users/Administrator/Desktop/python/A1/1-3-2.py', wdir='C:/Users/
Administrator/Desktop/python/A1')
k= 1 MSE= 0.0

In [352]: runfile('C:/Users/Administrator/Desktop/python/A1/1-3-2.py', wdir='C:/Users/
Administrator/Desktop/python/A1')
k= 3 MSE= 0.103503799533

In [353]: runfile('C:/Users/Administrator/Desktop/python/A1/1-3-2.py', wdir='C:/Users/
Administrator/Desktop/python/A1')
k= 5 MSE= 0.125909168602

In [354]: runfile('C:/Users/Administrator/Desktop/python/A1/1-3-2.py', wdir='C:/Users/
Administrator/Desktop/python/A1')
k= 50 MSE= 1.45135189424
```

Training MSE loss



```
In [356]: runfile('C:/Users/Administrator/Desktop/python/A1/1-3-2.py', wdir='C:/Users/
Administrator/Desktop/python/A1')
k= 1 MSE= 0.282336845509

In [357]: runfile('C:/Users/Administrator/Desktop/python/A1/1-3-2.py', wdir='C:/Users/
Administrator/Desktop/python/A1')
k= 3 MSE= 0.306255573631

In [358]: runfile('C:/Users/Administrator/Desktop/python/A1/1-3-2.py', wdir='C:/Users/
Administrator/Desktop/python/A1')
k= 5 MSE= 0.320395197394

In [359]: runfile('C:/Users/Administrator/Desktop/python/A1/1-3-2.py', wdir='C:/Users/
Administrator/Desktop/python/A1')
k= 50 MSE= 1.10367525828
```

Validation MSE loss

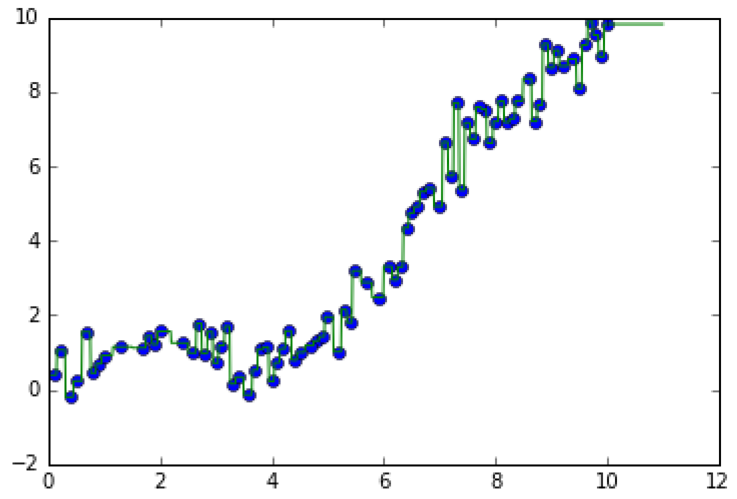


```
In [360]: runfile('C:/Users/Administrator/Desktop/python/A1/1-3-2.py', wdir='C:/Users/
Administrator/Desktop/python/A1')
k= 1 MSE= 0.173831598259

In [361]: runfile('C:/Users/Administrator/Desktop/python/A1/1-3-2.py', wdir='C:/Users/
Administrator/Desktop/python/A1')
k= 3 MSE= 0.14527223757

In [362]: runfile('C:/Users/Administrator/Desktop/python/A1/1-3-2.py', wdir='C:/Users/
Administrator/Desktop/python/A1')
k= 5 MSE= 0.184547016869

In [363]: runfile('C:/Users/Administrator/Desktop/python/A1/1-3-2.py', wdir='C:/Users/
Administrator/Desktop/python/A1')
k= 50 MSE= 0.815611168003
```

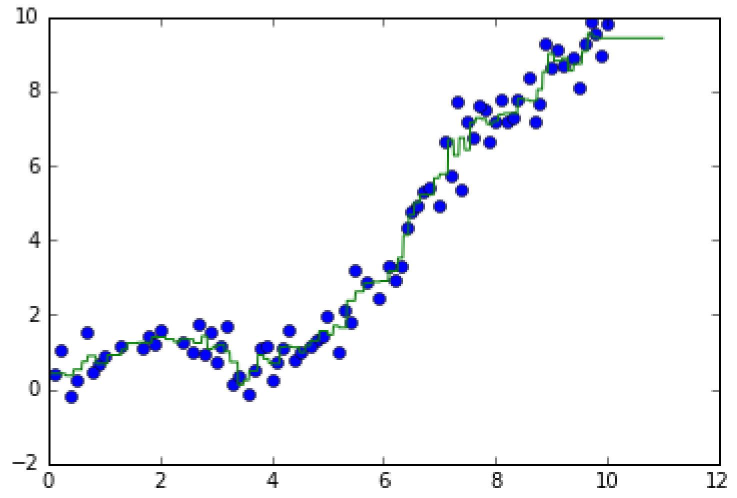
Testing MSE loss

According to the validation error, the best k can be obtained when k = 1 since it generates the smallest MSE loss.

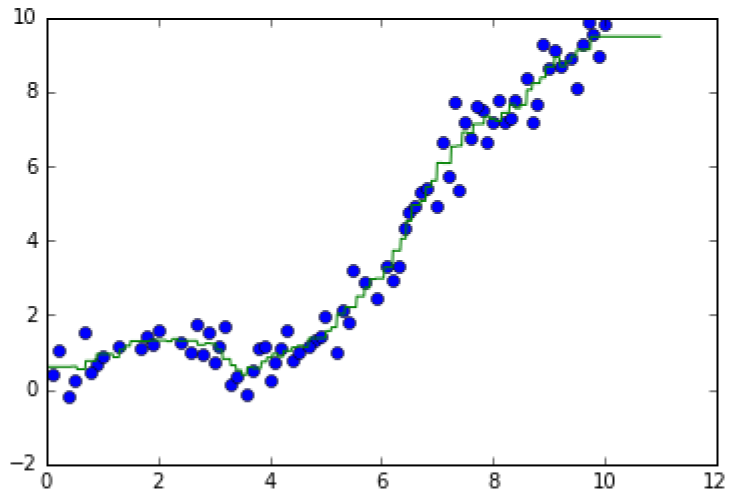
The prediction functions for x from 0 to 11 in different k values are shown as follows. Note that the green line in each graph is the prediction function for each case while the blue dots is the target value of the training data set.



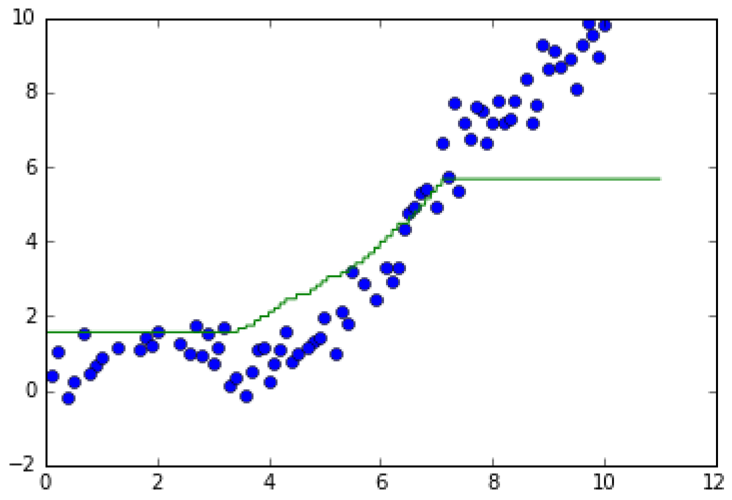
Prediction function when k = 1



Prediction function when k = 3



Prediction function when k = 5



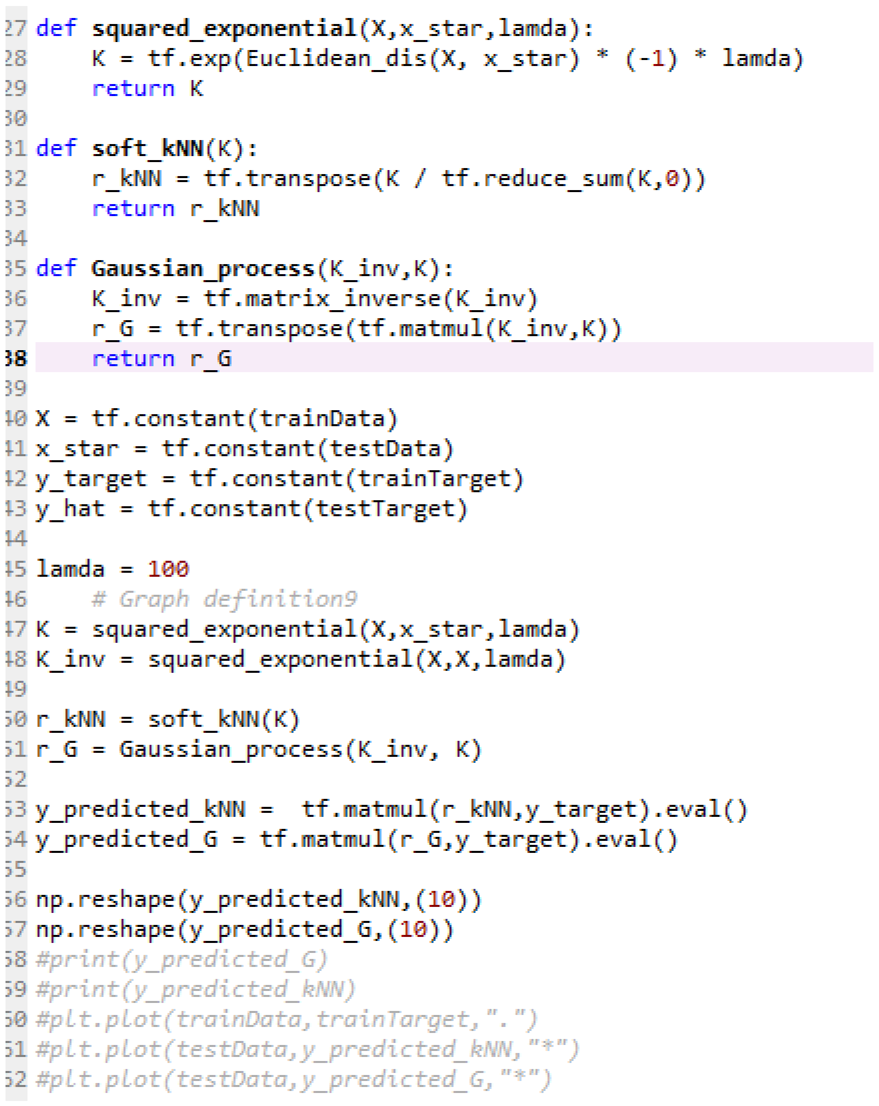
Prediction function when k = 50

According to the results shown above, the prediction function will overfit the data set when k is too small (k = 1). However, the prediction function is not accurate when k is too large (k = 50). Overall, we will obtain the best prediction function when k = 5.

1.4 Soft k-NN and Gaussian process

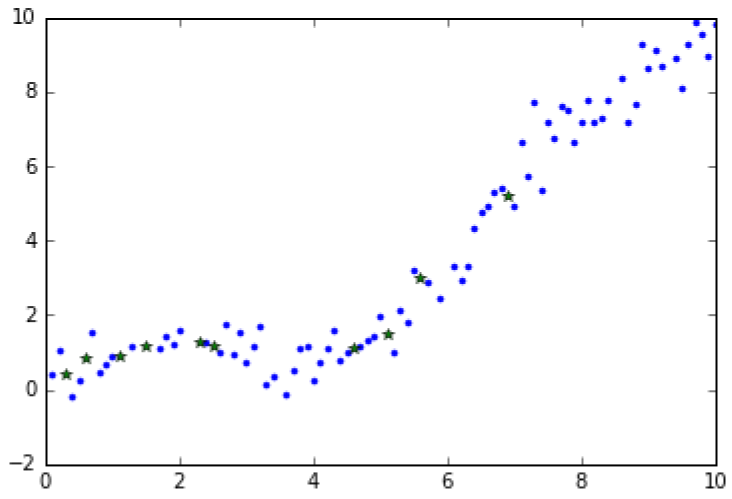
1.4.1

By setting lambda = 100 as indicated, the plotted test-set prediction of two different models and the snippets of Python code are shown as follows.

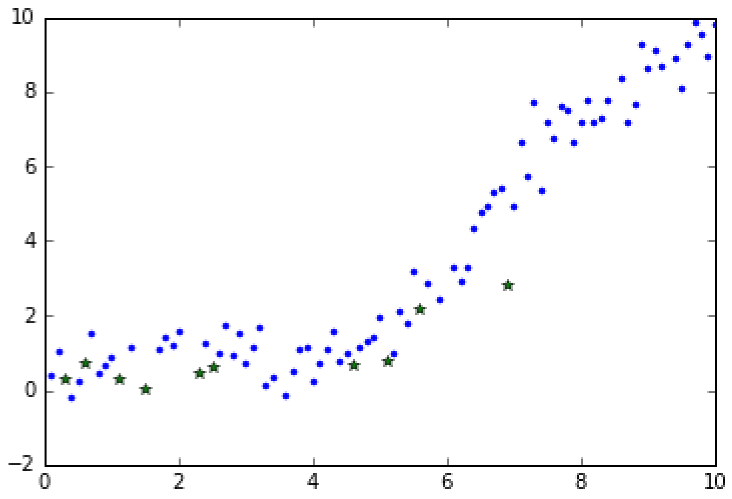


```python
def squared_exponential(X,x_star,lamda):
    K = tf.exp(Euclidean_dis(X, x_star) * (-1) * lamda)
    return K

def soft_kNN(K):
    r_kNN = tf.transpose(K / tf.reduce_sum(K,0))
    return r_kNN

def Gaussian_process(K_inv,K):
    K_inv = tf.matrix_inverse(K_inv)
    r_G = tf.transpose(tf.matmul(K_inv,K))
    return r_G

X = tf.constant(trainData)
x_star = tf.constant(testData)
y_target = tf.constant(trainTarget)
y_hat = tf.constant(testTarget)

lamda = 100
    # Graph definition9
K = squared_exponential(X,x_star,lamda)
K_inv = squared_exponential(X,X,lamda)

r_kNN = soft_kNN(K)
r_G = Gaussian_process(K_inv, K)

y_predicted_kNN =  tf.matmul(r_kNN,y_target).eval()
y_predicted_G = tf.matmul(r_G,y_target).eval()

np.reshape(y_predicted_kNN,(10))
np.reshape(y_predicted_G,(10))
#print(y_predicted_G)
#print(y_predicted_kNN)
#plt.plot(trainData,trainTarget,".")
#plt.plot(testData,y_predicted_kNN,"*")
#plt.plot(testData,y_predicted_G,"*")
```

Snippets of Python code



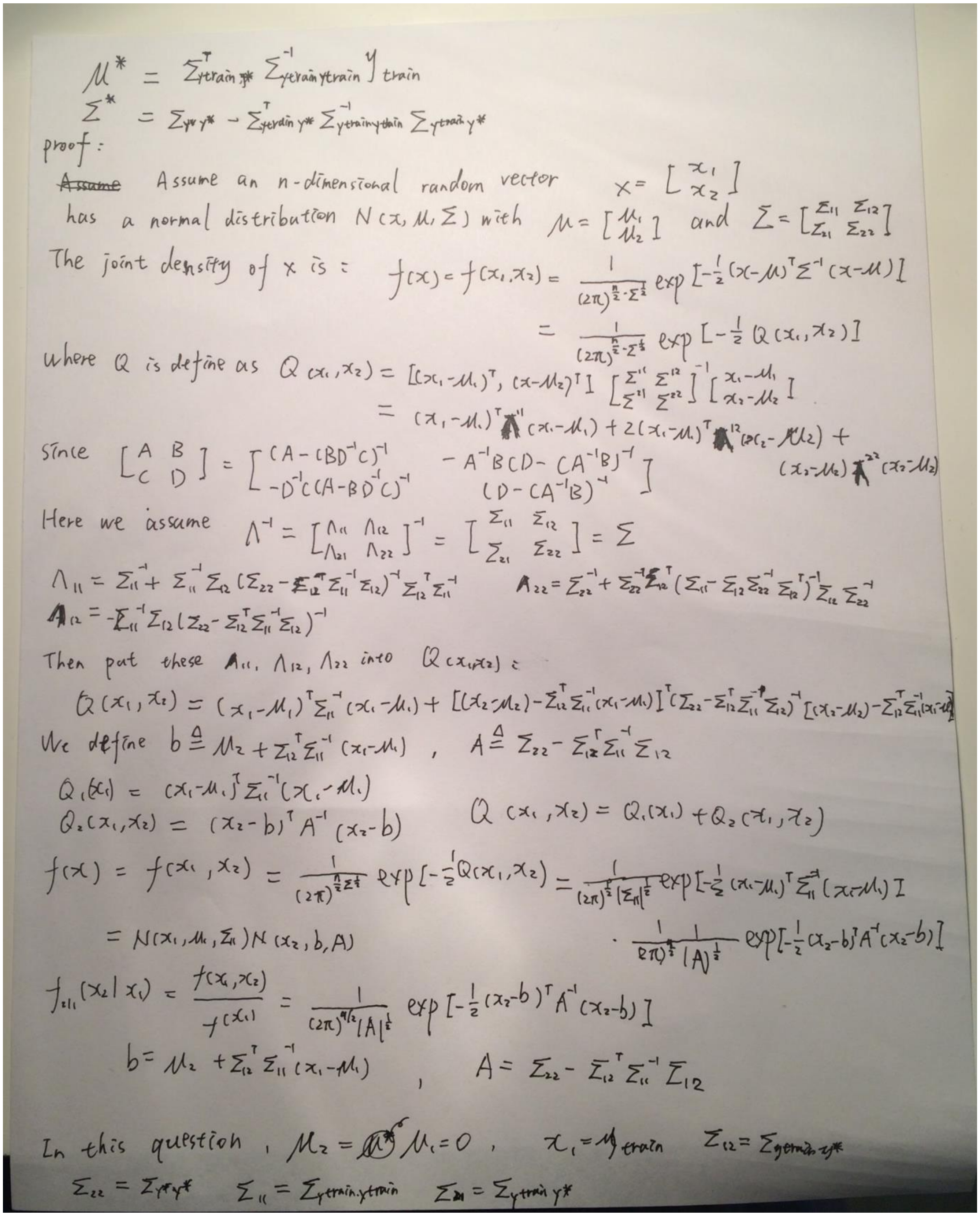
Plotted test-set prediction of soft k-NN model



Plotted test-set prediction of Gaussian process regression model

In the figure above, the blue dots are the training data and the green stars are the prediction results. By observing the predictions of the two different models, soft k-NN model has a better description to the training data set. The predictions of Gaussian process regression does not fit the training data as perfectly as the prediction of soft k-NN model does.

1.4.2



$$\mu^* = \Sigma_{ytrain\, y*}^{T}\, \Sigma_{ytrain\, ytrain}^{-1}\, y_{train}$$

$$\Sigma^* = \Sigma_{y* y*} - \Sigma_{ytrain\, y*}^{T}\, \Sigma_{ytrain\, ytrain}^{-1}\, \Sigma_{ytrain\, y*}$$

proof:

~~Assume~~ Assume an n-dimensional random vector $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ has a normal distribution $N(x, \mu, \Sigma)$ with $\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}$ and $\Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$

The joint density of x is: $f(x) = f(x_1, x_2) = \frac{1}{(2\pi)^{\frac{n}{2}} \cdot \Sigma^{\frac{1}{2}}} \exp[-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)]$

$$= \frac{1}{(2\pi)^{\frac{n}{2}} \cdot \Sigma^{\frac{1}{2}}} \exp[-\frac{1}{2} Q(x_1, x_2)]$$

where Q is define as $Q(x_1, x_2) = [(x_1-\mu_1)^T, (x-\mu_2)^T] \begin{bmatrix} \Sigma^{11} & \Sigma^{12} \\ \Sigma^{21} & \Sigma^{22} \end{bmatrix}^{-1} \begin{bmatrix} x_1-\mu_1 \\ x_2-\mu_2 \end{bmatrix}$

$$= (x_1-\mu_1)^T \Lambda^{11} (x_1-\mu_1) + 2(x_1-\mu_1)^T \Lambda^{12} (x_2-\mu_2) + (x_2-\mu_2) \Lambda^{22} (x_2-\mu_2)$$

since $\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} (A-(BD^{-1}C)^{-1} & -A^{-1}B(D-(A^{-1}B)^{-1} \\ -D^{-1}C(A-BD^{-1}C)^{-1} & (D-CA^{-1}B)^{-1} \end{bmatrix}$

Here we assume $\Lambda^{-1} = \begin{bmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{21} & \Lambda_{22} \end{bmatrix}^{-1} = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} = \Sigma$

$\Lambda_{11} = \Sigma_{11}^{-1} + \Sigma_{11}^{-1} \Sigma_{12} (\Sigma_{22} - \Sigma_{12}^{T} \Sigma_{11}^{-1} \Sigma_{12})^{-1} \Sigma_{12}^{T} \Sigma_{11}^{-1}$ $\quad \Lambda_{22} = \Sigma_{22}^{-1} + \Sigma_{22}^{-1} \Sigma_{12}^{T} (\Sigma_{11} - \Sigma_{12} \Sigma_{22}^{-1} \Sigma_{12}^{T})^{-1} \Sigma_{12} \Sigma_{22}^{-1}$

$\Lambda_{12} = -\Sigma_{11}^{-1} \Sigma_{12} (\Sigma_{22} - \Sigma_{12}^{T} \Sigma_{11}^{-1} \Sigma_{12})^{-1}$

Then put these $\Lambda_{11}, \Lambda_{12}, \Lambda_{22}$ into $Q(x_1, x_2)$:

$$Q(x_1, x_2) = (x_1-\mu_1)^T \Sigma_{11}^{-1} (x_1-\mu_1) + [(x_2-\mu_2) - \Sigma_{12}^{T} \Sigma_{11}^{-1} (x_1-\mu_1)]^T (\Sigma_{22} - \Sigma_{12}^{T} \Sigma_{11}^{-1} \Sigma_{12})^{-1} [(x_2-\mu_2) - \Sigma_{12}^{T} \Sigma_{11}^{-1} (x_1-\mu_1)]$$

We define $b \triangleq \mu_2 + \Sigma_{12}^{T} \Sigma_{11}^{-1} (x_1-\mu_1)$, $\quad A \triangleq \Sigma_{22} - \Sigma_{12}^{T} \Sigma_{11}^{-1} \Sigma_{12}$

$Q_1(x_1) = (x_1-\mu_1)^T \Sigma_{11}^{-1} (x_1-\mu_1)$

$Q_2(x_1, x_2) = (x_2-b)^T A^{-1} (x_2-b)$ $\quad Q(x_1, x_2) = Q_1(x_1) + Q_2(x_1, x_2)$

$$f(x) = f(x_1, x_2) = \frac{1}{(2\pi)^{\frac{n}{2}} \Sigma^{\frac{1}{2}}} \exp[-\frac{1}{2} Q(x_1, x_2)) = \frac{1}{(2\pi)^{\frac{1}{2}} |\Sigma_{11}|^{\frac{1}{2}}} \exp[-\frac{1}{2}(x_1-\mu_1)^T \Sigma_{11}^{-1} (x_1-\mu_1)] \cdot \frac{1}{(2\pi)^{\frac{1}{2}}} \frac{1}{|A|^{\frac{1}{2}}} \exp[-\frac{1}{2}(x_2-b)^T A^{-1} (x_2-b)]$$

$$= N(x_1, \mu_1, \Sigma_{11}) N(x_2, b, A)$$

$$f_{2|1}(x_2 | x_1) = \frac{f(x_1, x_2)}{f(x_1)} = \frac{1}{(2\pi)^{q/2} |A|^{\frac{1}{2}}} \exp[-\frac{1}{2}(x_2-b)^T A^{-1} (x_2-b)]$$

$b = \mu_2 + \Sigma_{12}^{T} \Sigma_{11}^{-1} (x_1-\mu_1)$, $\quad A = \Sigma_{22} - \Sigma_{12}^{T} \Sigma_{11}^{-1} \Sigma_{12}$

In this question, $\mu_2 = \cancel{\mu^*}$ $\mu_1 = 0$, $\quad x_1 = y_{train}$ $\quad \Sigma_{12} = \Sigma_{ytrain\, y*}$

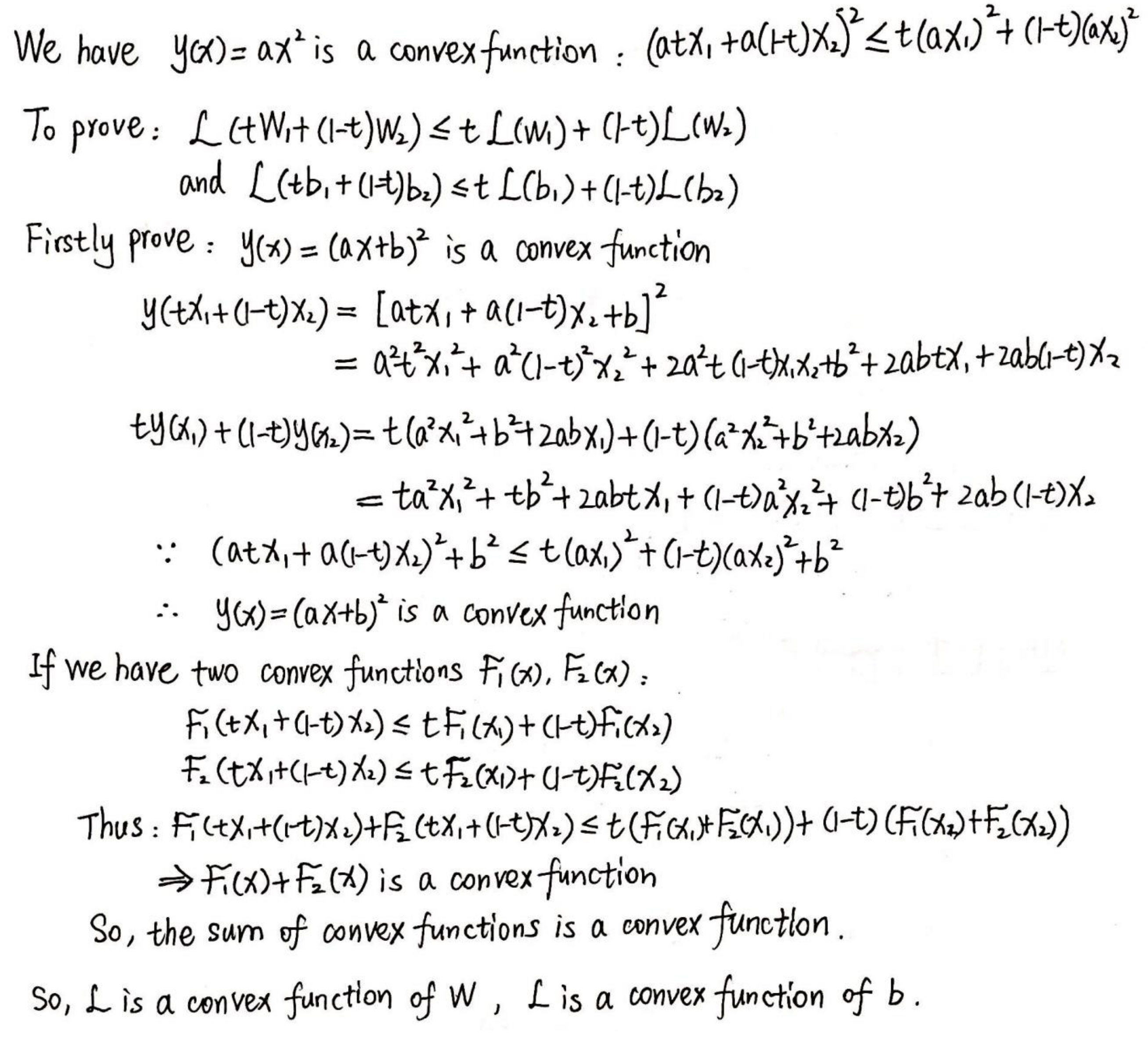
$\Sigma_{22} = \Sigma_{y* y*}$ $\quad \Sigma_{11} = \Sigma_{ytrain\, ytrain}$ $\quad \Sigma_{21} = \Sigma_{ytrain\, y*}$

2. Linear and logistic regression

2.1 Geometry of linear regression

2.1.1

The l2 penalized mean squared error loss function is a convex function for either W or b. The following captures shows the detailed process of verification.



We have $y(x) = ax^2$ is a convex function : $(atx_1 + a(1-t)x_2)^2 \le t(ax_1)^2 + (1-t)(ax_2)^2$

To prove : $\mathcal{L}(tW_1 + (1-t)W_2) \le t\mathcal{L}(W_1) + (1-t)\mathcal{L}(W_2)$

and $\mathcal{L}(tb_1 + (1-t)b_2) \le t\mathcal{L}(b_1) + (1-t)\mathcal{L}(b_2)$

Firstly prove : $y(x) = (ax+b)^2$ is a convex function

$$y(tx_1+(1-t)x_2) = [atx_1 + a(1-t)x_2 + b]^2$$
$$= a^2t^2x_1^2 + a^2(1-t)^2x_2^2 + 2a^2t(1-t)x_1x_2 + b^2 + 2abtx_1 + 2ab(1-t)x_2$$

$$ty(x_1) + (1-t)y(x_2) = t(a^2x_1^2 + b^2 + 2abx_1) + (1-t)(a^2x_2^2 + b^2 + 2abx_2)$$
$$= ta^2x_1^2 + tb^2 + 2abtx_1 + (1-t)a^2x_2^2 + (1-t)b^2 + 2ab(1-t)x_2$$

$\because$ $(atx_1 + a(1-t)x_2)^2 + b^2 \le t(ax_1)^2 + (1-t)(ax_2)^2 + b^2$

$\therefore$ $y(x) = (ax+b)^2$ is a convex function

If we have two convex functions $F_1(x)$, $F_2(x)$:

$$F_1(tx_1 + (1-t)x_2) \le tF_1(x_1) + (1-t)F_1(x_2)$$
$$F_2(tx_1 + (1-t)x_2) \le tF_2(x_1) + (1-t)F_2(x_2)$$

Thus : $F_1(tx_1+(1-t)x_2) + F_2(tx_1+(1-t)x_2) \le t(F_1(x_1)+F_2(x_1)) + (1-t)(F_1(x_2)+F_2(x_2))$

$\Rightarrow F_1(x) + F_2(x)$ is a convex function

So, the sum of convex functions is a convex function.

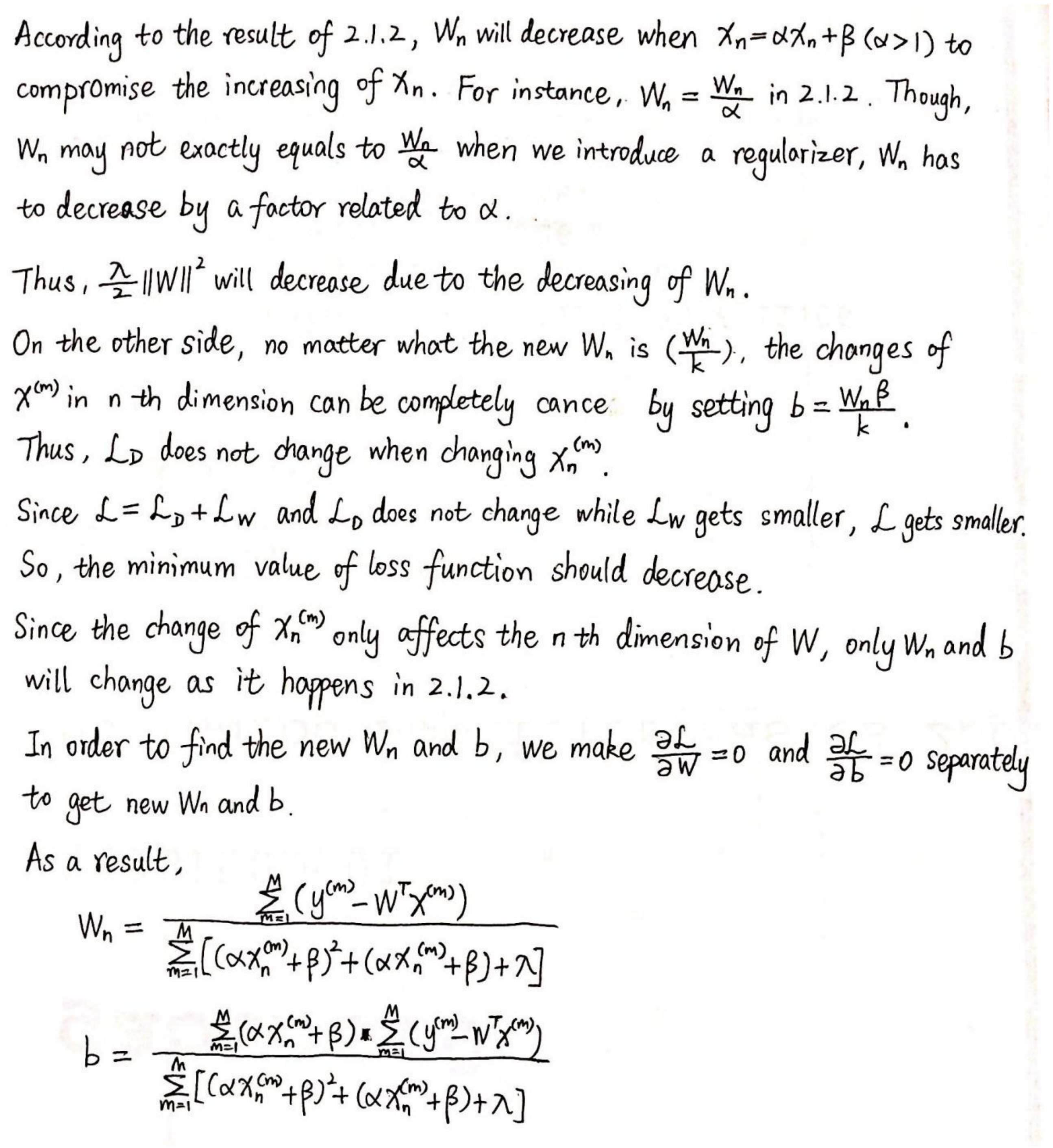
So, $\mathcal{L}$ is a convex function of W , $\mathcal{L}$ is a convex function of b.

2.1.2

Firstly, considering the case nothing has been changed. Learning a linear regression model will return a W vector with N dimensions for each of the M elements in W. It will also return bias b with different values in N dimensions. If the n th input dimensions, $x_n^{(m)}$, is scaled by a constant factor $\alpha > 1$ and then shifted by a constant $\beta > 1$. Thus, $x_n^{(m)} = \alpha\ x_n + \beta$ for each of the M vectors in the dataset.

By applying the same learning model, the corresponding n th dimension of W will change to Wn/**α** while keeping other dimensions of W unchanged. Similarly, bias b for the n th dimension will change to b - β * (Wn/**α**) while the bias b in other dimensions will not change. The goal of doing that is to make sure the loss function approaches the same minimum value as it does before the changes. Thus, the new minimum value of the loss function will not change.

2.1.3



According to the result of 2.1.2, W_n will decrease when $x_n = \alpha x_n + \beta$ $(\alpha > 1)$ to compromise the increasing of x_n. For instance, $W_n = \frac{W_n}{\alpha}$ in 2.1.2. Though, W_n may not exactly equals to $\frac{W_n}{\alpha}$ when we introduce a regularizer, W_n has to decrease by a factor related to α.

Thus, $\frac{\lambda}{2}\|W\|^2$ will decrease due to the decreasing of W_n.

On the other side, no matter what the new W_n is $(\frac{W_n}{k})$, the changes of $x^{(m)}$ in n-th dimension can be completely cance by setting $b = \frac{W_n \beta}{k}$.

Thus, $\mathcal{L}_D$ does not change when changing $x_n^{(m)}$.

Since $\mathcal{L} = \mathcal{L}_D + \mathcal{L}_W$ and $\mathcal{L}_D$ does not change while $\mathcal{L}_W$ gets smaller, $\mathcal{L}$ gets smaller. So, the minimum value of loss function should decrease.

Since the change of $x_n^{(m)}$ only affects the n-th dimension of W, only W_n and b will change as it happens in 2.1.2.

In order to find the new W_n and b, we make $\frac{\partial \mathcal{L}}{\partial W} = 0$ and $\frac{\partial \mathcal{L}}{\partial b} = 0$ separately to get new W_n and b.

As a result,

$$W_n = \frac{\sum_{m=1}^{M}(y^{(m)} - W^T x^{(m)})}{\sum_{m=1}^{M}[(\alpha x_n^{(m)} + \beta)^2 + (\alpha x_n^{(m)} + \beta) + \lambda]}$$

$$b = \frac{\sum_{m=1}^{M}(\alpha x_n^{(m)} + \beta) * \sum_{m=1}^{M}(y^{(m)} - W^T x^{(m)})}{\sum_{m=1}^{M}[(\alpha x_n^{(m)} + \beta)^2 + (\alpha x_n^{(m)} + \beta) + \lambda]}$$

2.1.4

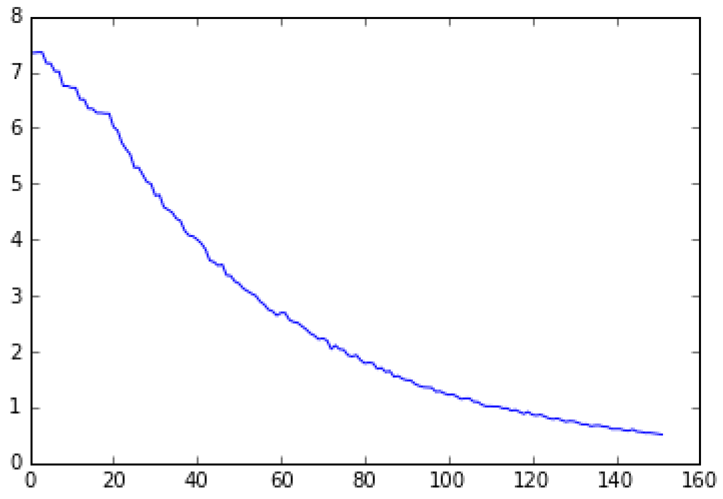
To solve a multi-class task with binary classifier, first, we can divide the D classes into two part, one part contain one class (class A), and the other part contain the rest D-1 classes. We then can use the two part to train a binary classifier. So in the first classifier, we can classify A and the rest classes. After that, we divide the rest class into two parts: class B and the other classes. Then we train the classifier with these two parts. So in this way, after D-1 iteration, we can solve a multi-class task with binary classifier.

2.2 Stochastic gradient descent

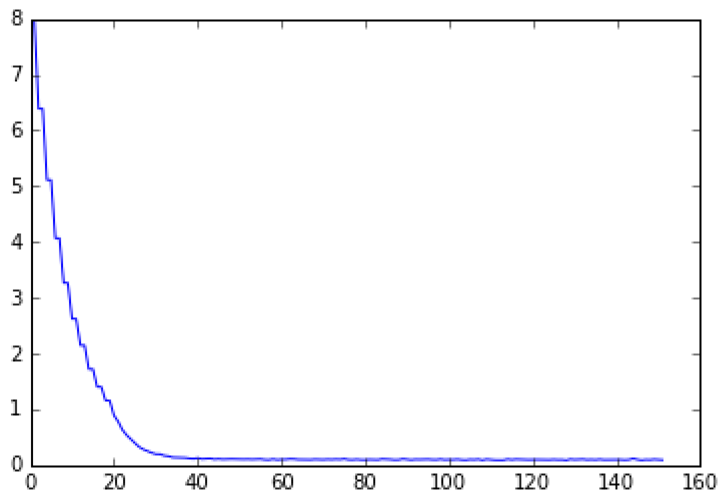
Note that the Python code in this section is attached in Appendix.

2.2.1

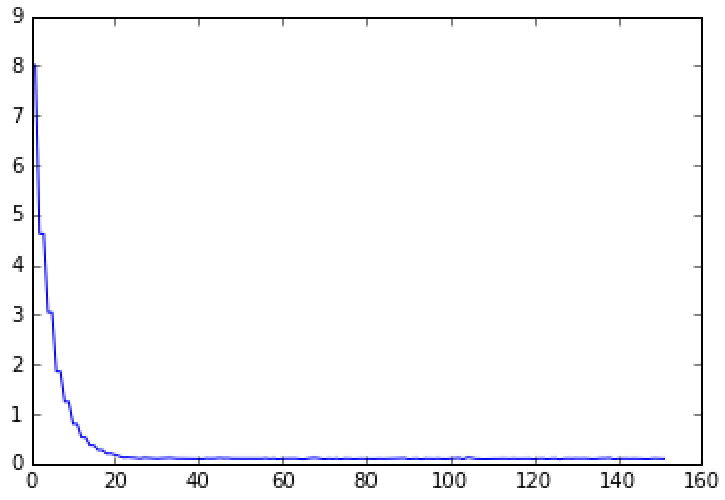
The following plots display the plots of total loss function vs. the number updates by tuning the learning rate into different values to obtain the best overall convergence speed of the algorithm. Note that in each case, the mini-batch size = 50 and lambda = 1.



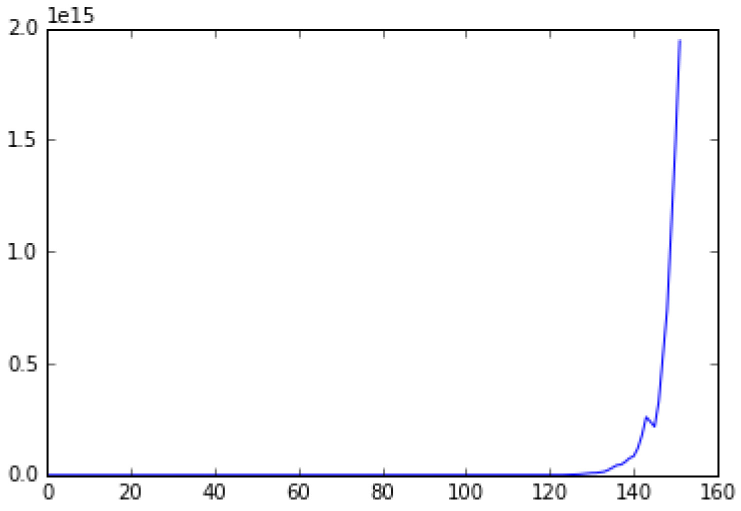
The plot of total loss function vs. the number updates when learning rate = 0.01



The plot of total loss function vs. the number updates when learning rate = 0.1



The plot of total loss function vs. the number updates when learning rate = 0.2(best learning rate)

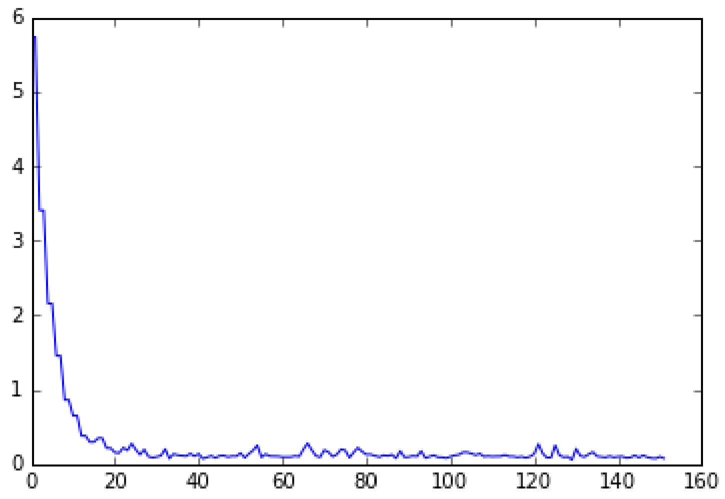


The plot of total loss function vs. the number updates when learning rate = 0.3

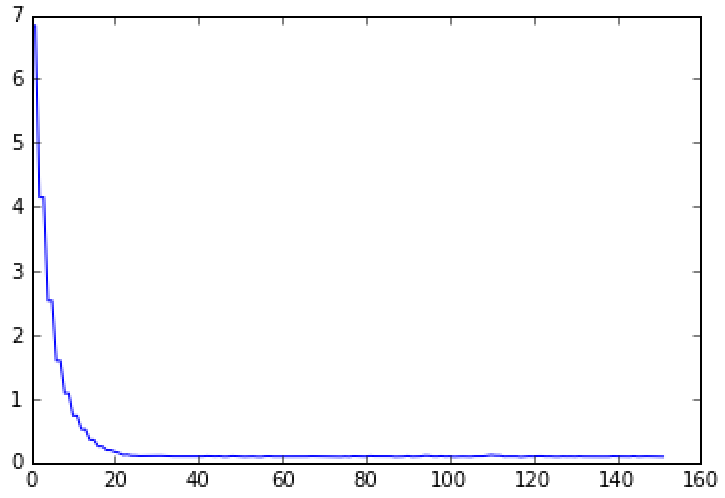
As the figures shown above, when learning rate = 0.2 we obtain the best overall convergence speed. We can conclude that before reach a certain value (in the question, 0.3), the convergence speed increase as the learning rate increase. But when the learning rate is too big, the algorithm will fail to converge.

2.2.2

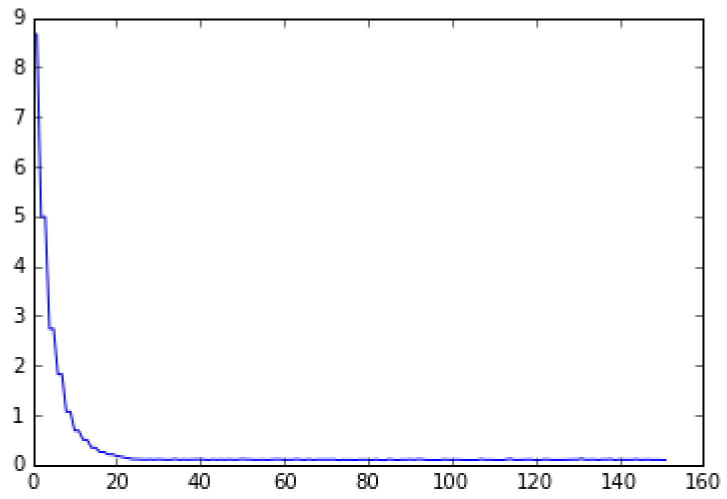
The following graphs are the plot of total loss function vs. the number of updates when mini-batch size = {10, 50, 100, 700} separately.



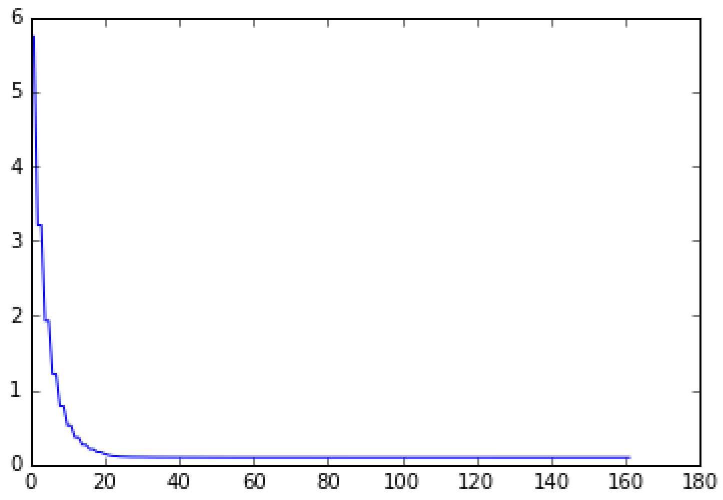
The plot of total loss function vs. the number updates when mini-batch size = 10(best learning rate = 0.2)



The plot of total loss function vs. the number updates when mini-batch size = 50(best learning rate = 0.2)



The plot of total loss function vs. the number updates when mini-batch size = 100(best learning rate = 0.2)

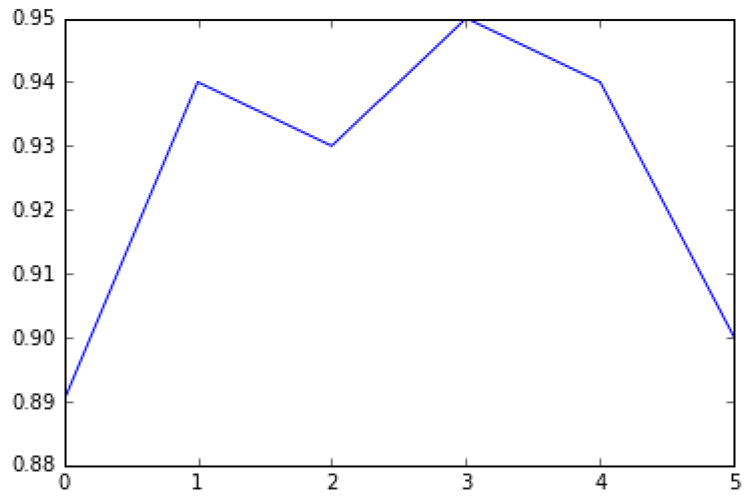


The plot of total loss function vs. the number updates when mini-batch size = 700(best learning rate = 0.2)

As the figures shown, for different mini-batch, the loss functions all converge after the 20th iteration. But for mini-batch size = 10, it has the smallest training data in every training iteration. So in terms of training time, the mini-batch size =10 is the best mini-batch size.(However, when mini-batch size = 10, the training process is very noisy, we may also choose mini-batch size = 50 as our best size)

2.2.3

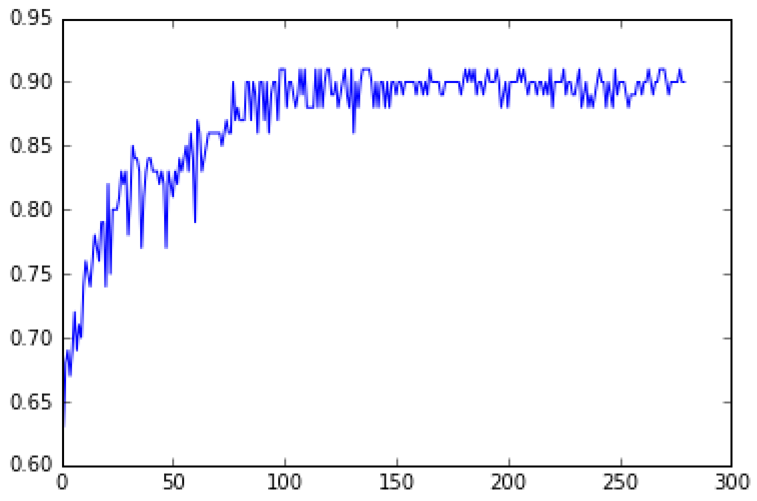
The following figure displays the plot of lambda vs. test set accuracy while lambda = {0., 0.0001, 0.001, 0.01, 0.1, 1}. The plot of test set accuracy vs. the number of updates also is plotted for each case of lambda equals to {0., 0.0001, 0.001, 0.01, 0.1, 1} separately.



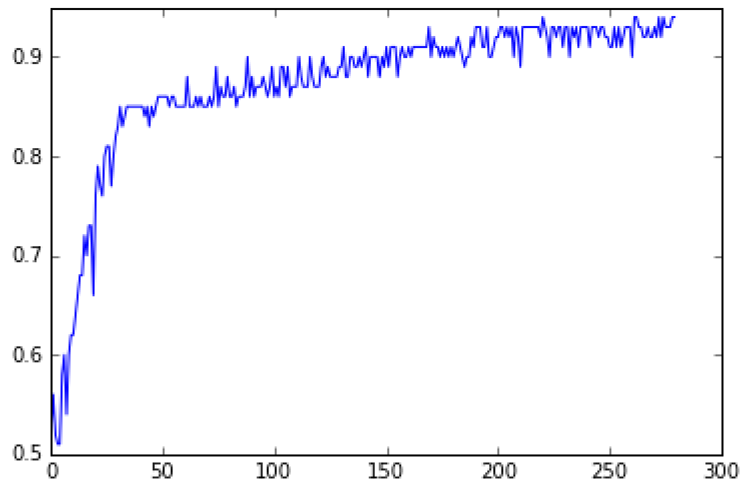
Plot of lambda vs. test set accuracy for lambda = {0., 0.0001, 0.001, 0.01, 0.1, 1}

As the figure shown above, the best weight decay coefficient = 0.01. And either the weight decay coefficient is too big or too small (lambda = 0), we may get a relatively low classification accuracy.

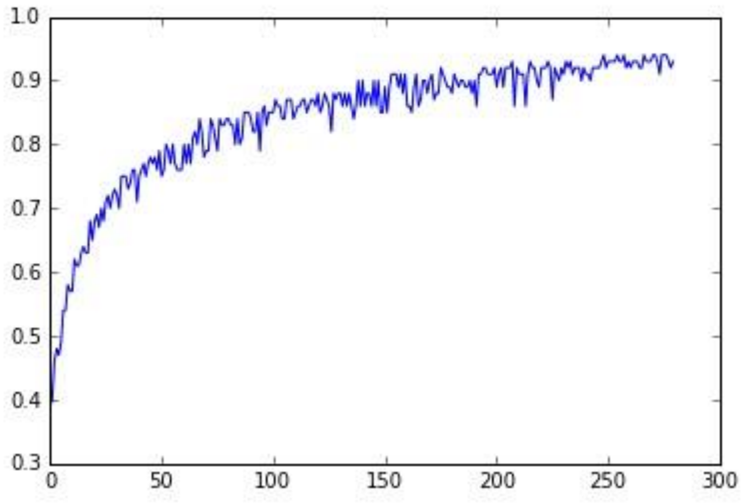
Machine Learning algorithms should predict the behavior of future unseen data, this is usually called the generalization ability of the algorithm/model. By creating a validation set, you are mimicking a situation where you haven't seen some data, and trying to see how your model works on that data. Generally, by using validation set to choose the parameter, we can avoid overfitting.



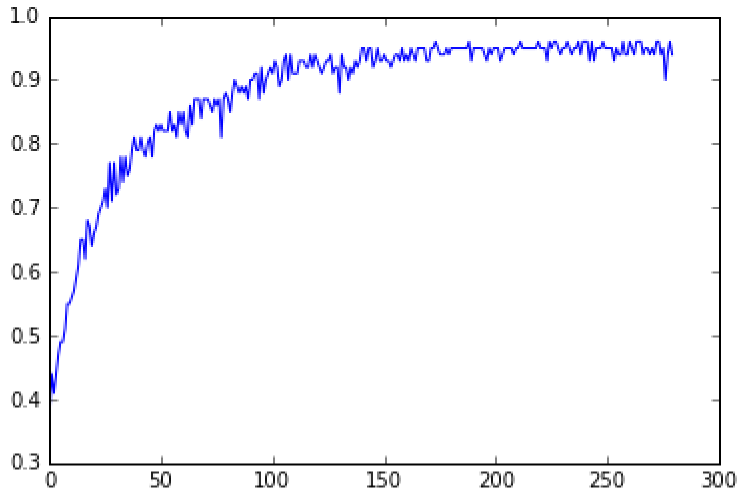
Plot test set accuracy vs. number of updates when lambda = 0



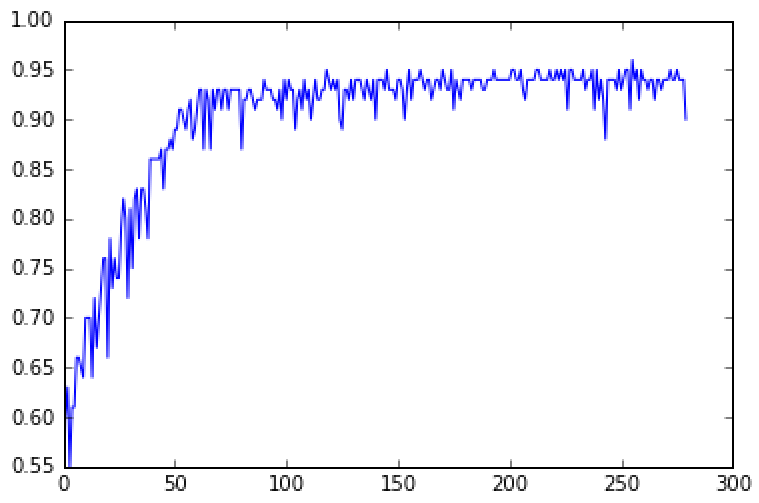
Plot test set accuracy vs. number of updates when lambda = 0.0001



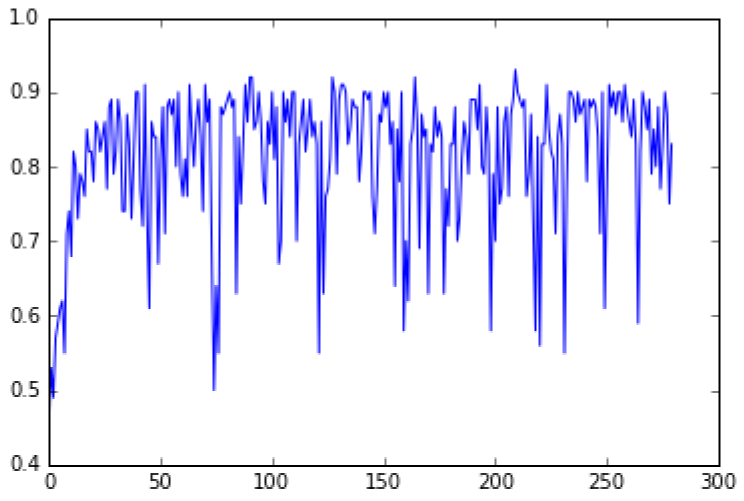
Plot test set accuracy vs. number of updates when lambda = 0.001



Plot test set accuracy vs. number of updates when lambda = 0.01



Plot test set accuracy vs. number of updates when lambda = 0.1



Plot test set accuracy vs. number of updates when lambda = 1

Appendix

Python code of section 1.3.2:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(521)
Data = np.linspace(0.1,10.0,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)
trainData, trainTarget = Data[randIdx[:80]],Target[randIdx[:80]]
validData, validTarget = Data[randIdx[80:90]],Target[randIdx[80:90]]
testData, testTarget = Data[randIdx[90:100]],Target[randIdx[90:100]]
#plt.plot(Data[randIdx[:80]],Target[randIdx[:80]],".")

sess=tf.InteractiveSession()

import tensorflow as tf
import numpy as np

sess=tf.InteractiveSession()
def Euclidean_dis(X,Y):
    '''
    return the matrix containing the pairwise Euclidean distances
    '''

    X_b = tf.expand_dims(X,1)
    result = X_b - Y
    result_square = tf.square(result)
    Euclidean_dis = tf.reduce_sum(result_square,2)

    return Euclidean_dis


def Choose_neigh(X,x_target,k):

    Dis = Euclidean_dis(X,x_target)
    Dis_t = tf.transpose(Dis)
    Dis_t = Dis_t * (-1)
    values,indices = tf.nn.top_k(Dis_t, k)
    Size_w = tf.shape(X)[0].eval()
    Size_h = tf.shape(x_target)[0].eval()
    indices = sess.run(indices)
    row_indices = np.linspace(0,Size_h-1,Size_h,dtype = int)
    row_indices = row_indices.repeat(k)
    indices = indices.reshape(Size_h*k,)
    R = np.zeros((Size_h,Size_w),np.float64)
    R[row_indices,indices] = 1/k
    R = tf.convert_to_tensor(R)
    return R

def buildGraph(k):
    # Variable creation
    X = tf.constant(trainData)
    X_star = tf.constant(Z)
    y_target = tf.constant(trainTarget)
    y_hat = tf.constant(testTarget)

    # Graph definition9

    r = Choose_neigh(X,X_star,k)

    y_predicted =  tf.matmul(r,y_target)



   # Error definition
    meanSquaredError = tf.reduce_mean(tf.reduce_mean(tf.square(y_predicted - y_hat),
reduction_indices=1, name='squared_error'), name='mean_squared_error')/2
    return meanSquaredError,X,y_target,y_predicted


for k in [1,3,5,50]:

    Z=np.linspace(0.0,11.0,num=1000)[:,np.newaxis]
    meanSquaredError,X,y_target,y_predicted = buildGraph(k)
#print("k=",k,"MSE=",sess.run(meanSquaredError))


    plt.plot(Z,sess.run(y_predicted),"-")
```

Python code of section 2.2:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf


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"]

sess = tf.InteractiveSession()

y = np.zeros(np.shape(validData)[0])
accuracy = []
wList = []
bList = []
batch_size = 50
N = 0.2
z=[]
def buildGraph(lamda):
    # 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')

    # Graph definition
    y_predicted = tf.matmul(X,W) + b

    # Error definition0
    meanSquaredError = 1/2*tf.reduce_mean(tf.reduce_mean(tf.square(y_predicted - y_target), 
reduction_indices=1, name='squared_error') + lamda * tf.reduce_mean(tf.matmul(tf.transpose(W),W)), 
name='mean_squared_error')

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


def runMult(lamda):
    #Build computation graph
    W, b, X, y_target, y_predicted, meanSquaredError, train = buildGraph(lamda)

    #Initialize session
    init = tf.initialize_all_variables()


    sess.run(init)

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

    #print("Initial weights: %s, initial bias: %.2f"%(initialW, initialb))
    # Training model
    coordinate_x = np.array([0,1,2,3,4,5,6,7,8,9,10,20])
    coordinate_y=np.array([])

    for step in range(0,20):

        idx = np.arange(0,700)
        np.random.shuffle(idx)
        for index in range(0,14):


            batch1 = trainData[idx]
            batch2 = trainTarget[idx]
            batch1 = batch1[index*batch_size:(index+1)*batch_size]
            batch2 = batch2[index*batch_size:(index+1)*batch_size]
            _, err, currentW, currentb, yhat = sess.run([train, meanSquaredError, W, b, y_predicted],
feed_dict={X: batch1, y_target: batch2})
#           W1 = np.float64(currentW)
#           b1 = np.float64(currentb)
#           y_predicted_v = np.matmul(validData,W1) + b1
#           for i in range(np.shape(y_predicted_v)[0]):
#               if (y_predicted_v[i] > 0.5):
#                   y[i] = 1
#               else:
#                   y[i] = 0
#               y[i] = abs(validTarget[i]-y[i])
#           accuracy.append(1 - (np.sum(y)/100))
#
#
            #           z.append(err)
#           if not ((int(700/batch_size)*step+index) % 50) or int(700/batch_size)*step+index < 10 :
#               print("Iter: %3d, MSE-train: %4.2f"%(int(700/batch_size)*step+index, err))
#
    # Testing model
    #errTest = sess.run(meanSquaredError, feed_dict= {X: testData, y_target: testTarget})
    #print("Final testing MSE: %.2f      lamda: %4.2f    batch_size: %3d"%(errTest, lamda, batch_size))
    #plt.plot(coordinate_x,coordinate_y)
    return currentW,currentb

np.set_printoptions(precision=4)


#
zl = [0]


for lamda in [0,0.0001,0.001,0.01,0.1,1]:
    W,b=runMult(lamda)
    W = np.float64(W)
    b = np.float64(b)
    y_predicted_v = np.matmul(validData,W) + b
    for i in range(np.shape(y_predicted_v)[0]):
        if (y_predicted_v[i] > 0.5):
            y[i] = 1
        else:
            y[i] = 0
        y[i] = abs(validTarget[i]-y[i])
    accuracy.append(1 - (np.sum(y)/100))

plt.plot(accuracy)
```