ECE521 A1

Baiwu Zhang 1001127540
Daiqing Li 1000795357

Janurary 2017

1

1.1

1.1.1

Construct a data set as follow, alternatively place two nodes from different class one after the other. A graphic illustration of the data set is shown in Figure 1. The data should be large enough to avoid corner cases at the edges. Accuracy with different K values is in Table 1.

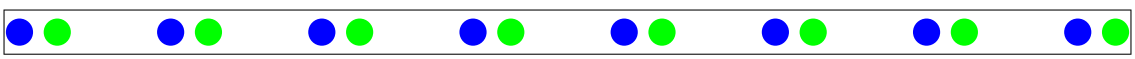


Figure 1: The data set for KNN with periodical accuracy.

Table 1: K vs Accuracy

K	1	2	3	4	5	6	7	8	9	...
Acc	100	50	0	50	100	50	0	50	100	(period = 4)

Analysis:
K = 1: Every data point chooses itself. Accuracy is 100%.
K = 2: Every data point chooses itself and its opposite color neighbour. A tie is broke by randomly pick one. Accuracy is 50%.
k = 3: Other than the two points that already been picked, each data point picks the third nearest node, which is in the opposite color. Accuracy is 0%.
k = 4: Other than the above three points, two closest points to the data point have the same distance and same color. Pick either one, a tie is created and it will have 50% accuracy.
k = 5: Pick the one not been picked in last step, which has the same color as the data point. Accuracy is 100%. The pattern starts to repeat.

1.1.2

$$
\begin{aligned}
var\left(\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|_2^2}{E[\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|_2^2]}\right) &= var\left(\frac{\|\boldsymbol{d}\|_2^2}{E[\|\boldsymbol{d}\|_2^2]}\right) \\
&= \frac{var(\|\boldsymbol{d}\|_2^2)}{E[\|\boldsymbol{d}\|_2^2]^2} \\
&= \frac{E[\|\boldsymbol{d}\|_2^4]-E[\|\boldsymbol{d}\|_2^2]^2}{E[\|\boldsymbol{d}\|_2^2]^2} \\
&= \frac{E[\|\boldsymbol{d}\|_2^4]}{E[\|\boldsymbol{d}\|_2^2]^2}-1 \\
&= \frac{E[(\sum_{i=1}^N d_i^2)^2]}{E[\sum_{i=1}^N d_i^2]^2}-1 \\
&= \frac{E[\sum_{i=1}^N\sum_{j=1}^N(d_i^2d_j^2)]}{\sum_{i=1}^N\sum_{j=1}^N E[d_i^2]E[d_j^2]}-1 \\
&= \frac{\sum_{i=1}^N\sum_{j=1}^N E[d_i^2d_j^2]}{\sum_{i=1}^N\sum_{j=1}^N E[d_i^2]E[d_j^2]}-1 \\
&= \frac{\sum_{i=1}^N E[d_i^4]+\sum_{i=1}^N\sum_{j=1,j\neq i}^N(E[d_i^2]E[d_j^2])}{\sum_{i=1}^N E[d_i^2]^2+\sum_{i=1}^N\sum_{j=1,j\neq i}^N(E[d_i^2]E[d_j^2])}-1 \\
&= \frac{N*12\sigma^4+\sum_{i=1}^N\sum_{j=1,j\neq i}^N 4\sigma^4}{N*4\sigma^4+\sum_{i=1}^N\sum_{j=1,j\neq i}^N 4\sigma^4}-1 \\
&= \frac{N*12\sigma^4+(N^2-N)4\sigma^4}{N*4\sigma^4+(N^2-N)4\sigma^4}-1 \\
&= \frac{3N+N^2-N}{N+N^2-N}-1 \\
&= \frac{N+2}{N}-1
\end{aligned} \tag{1}
$$

1.2

1.2.1

$$
\begin{aligned}
\left\|\boldsymbol{x^{(m)}}-\boldsymbol{x^*}\right\|_2^2 &= \sum_{n=1}^N\left(x_n^{(m)}-x_n^*\right)^2 \\
&= \sum_{n=1}^N({x_n^{(m)}}^2+{x_n^*}^2)-2\sum_{n=1}^N x_n^{(m)}x_n^* \\
&= 2C-2{\boldsymbol{x}^{(m)}}^T\boldsymbol{x}^* \\
&= constant+constant*-{\boldsymbol{x}^{(m)}}^T\boldsymbol{x}^*
\end{aligned} \tag{2}
$$

From equation (2), we can conclude that as long as we rank $-{\boldsymbol{x}^{(m)}}^T\boldsymbol{x}^*$ for every input, we can find the nearest neighbour for $\boldsymbol{x^*}$.

1.2.2

Please see the code section for our implementation.

1.3

1.3.1

Please see code section for our implementation.

1.3.2

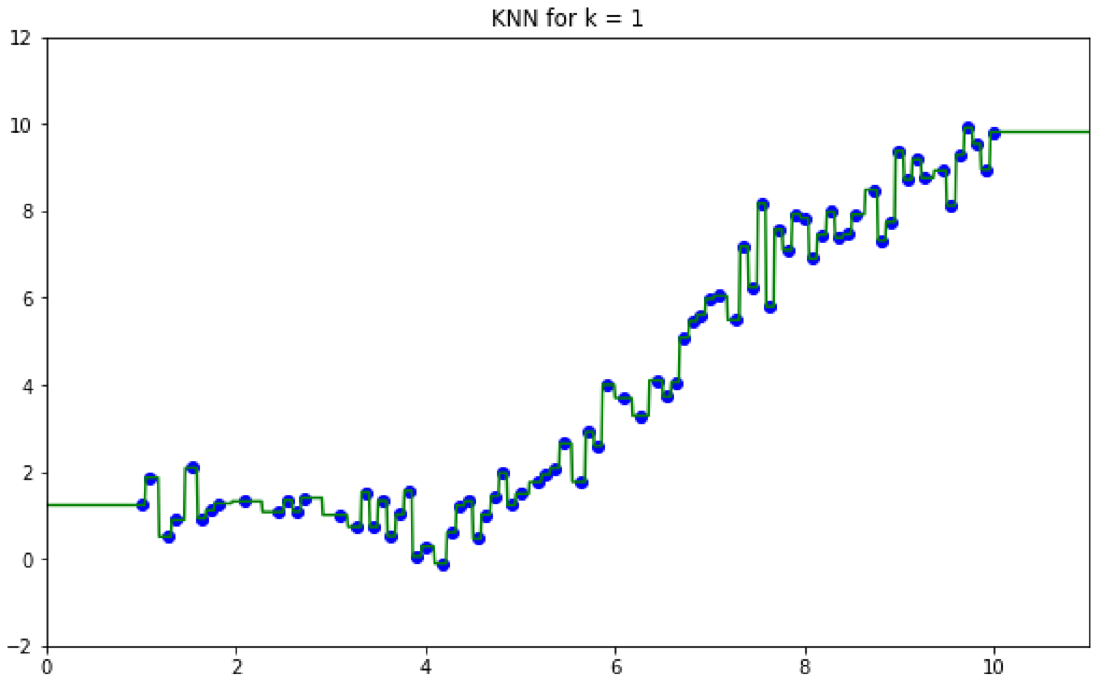


Figure 2: k: 1, trainMSE: 0.00, validMSE: 5.43, testMSE: 6.22

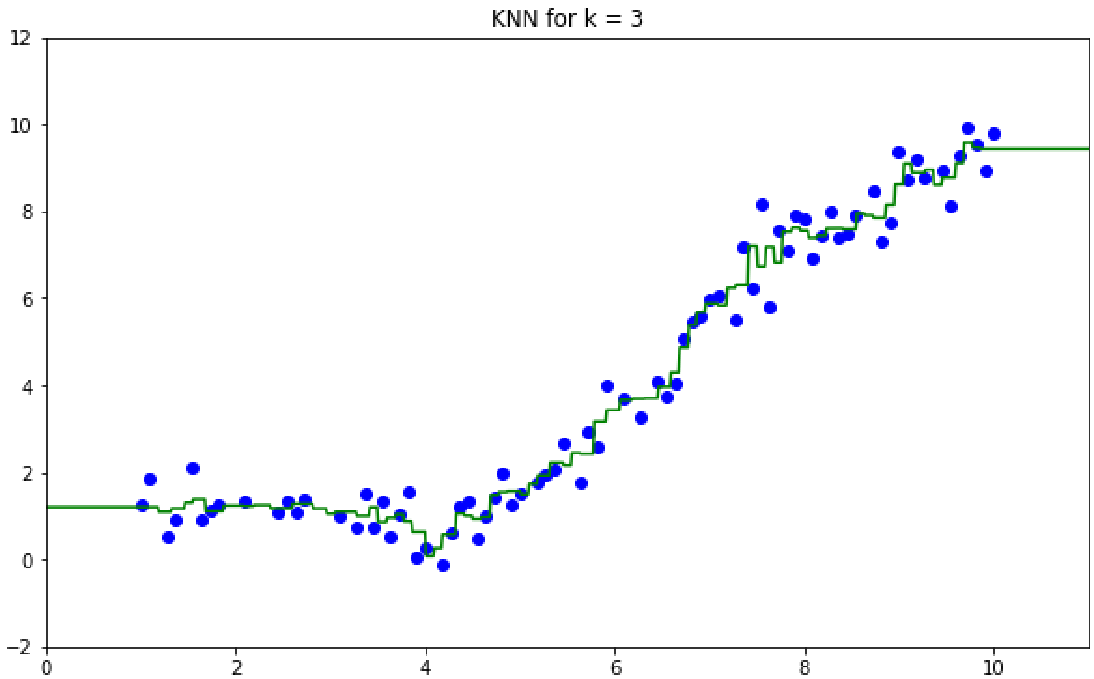


Figure 3: k: 3, trainMSE: 16.84, validMSE: 6.53, testMSE: 2.90

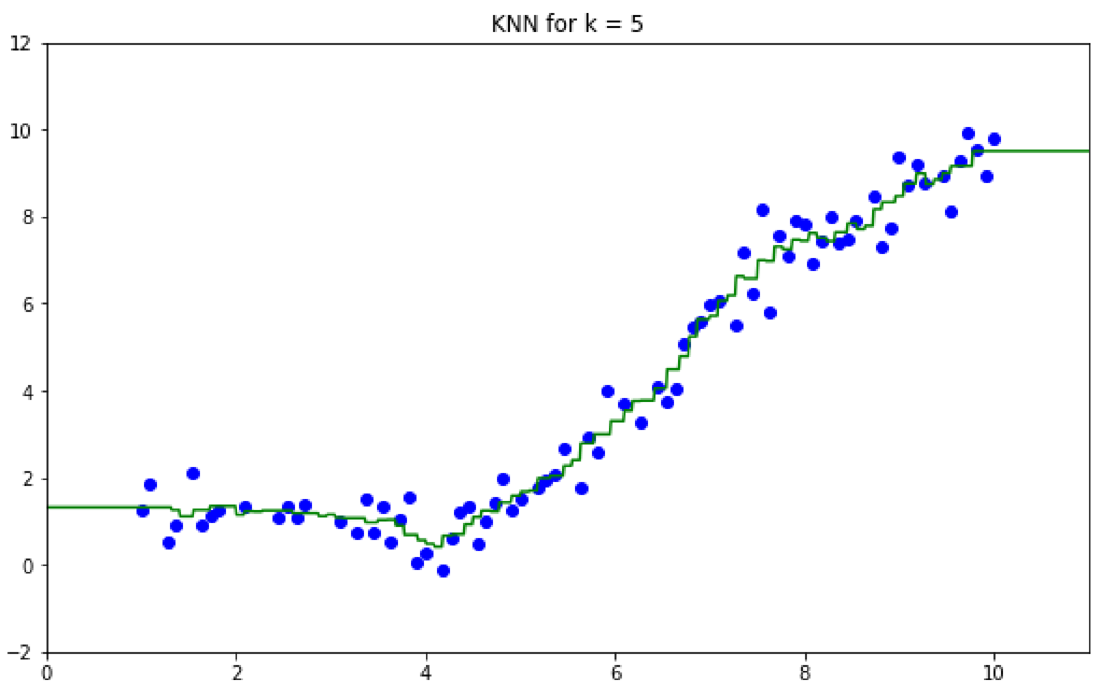


Figure 4: k: 5, trainMSE: 18.97, validMSE: 6.21, testMSE: 3.57

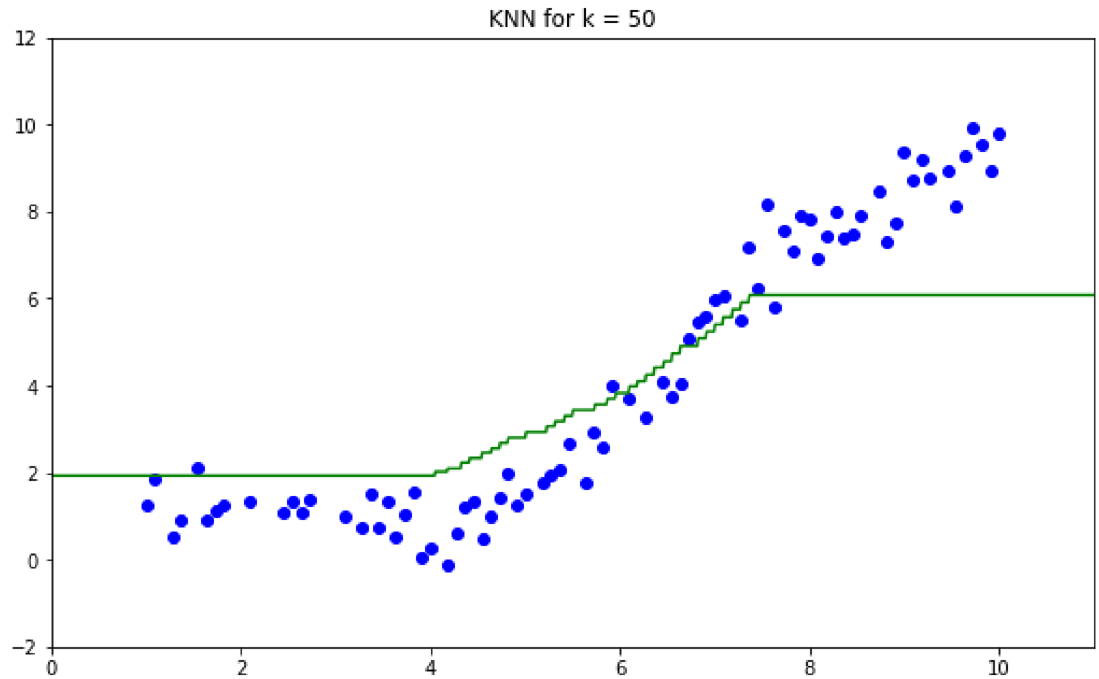


Figure 5: k: 50, trainMSE: 199.68, validMSE: 24.57, testMSE: 14.14

From the calculation, we find the best k value based on validation error is k = 1. From the plotting, k = 1 fits every training points perfectly, but it is overfitting. For k = 3, it is more smooth, but still a little bit overfitting. For k = 50, the prediction has a great offset comparing to the training points. We think k = 5 is a better choice because it is more general and smooth.

1.4

1.4.1

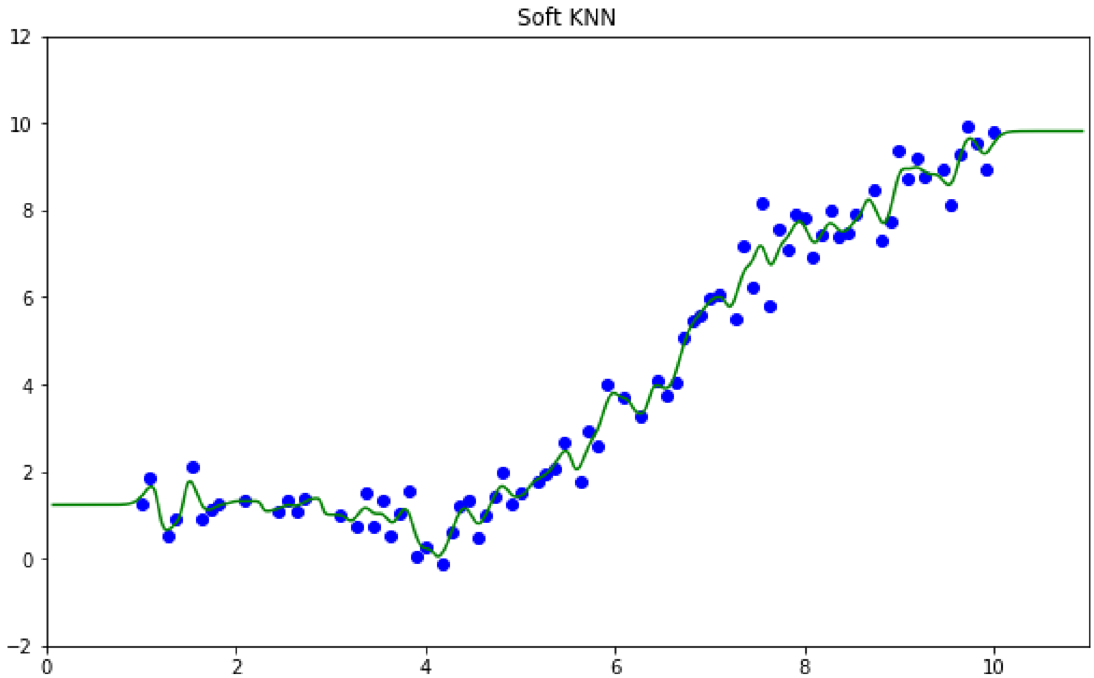


Figure 6: Soft KNN

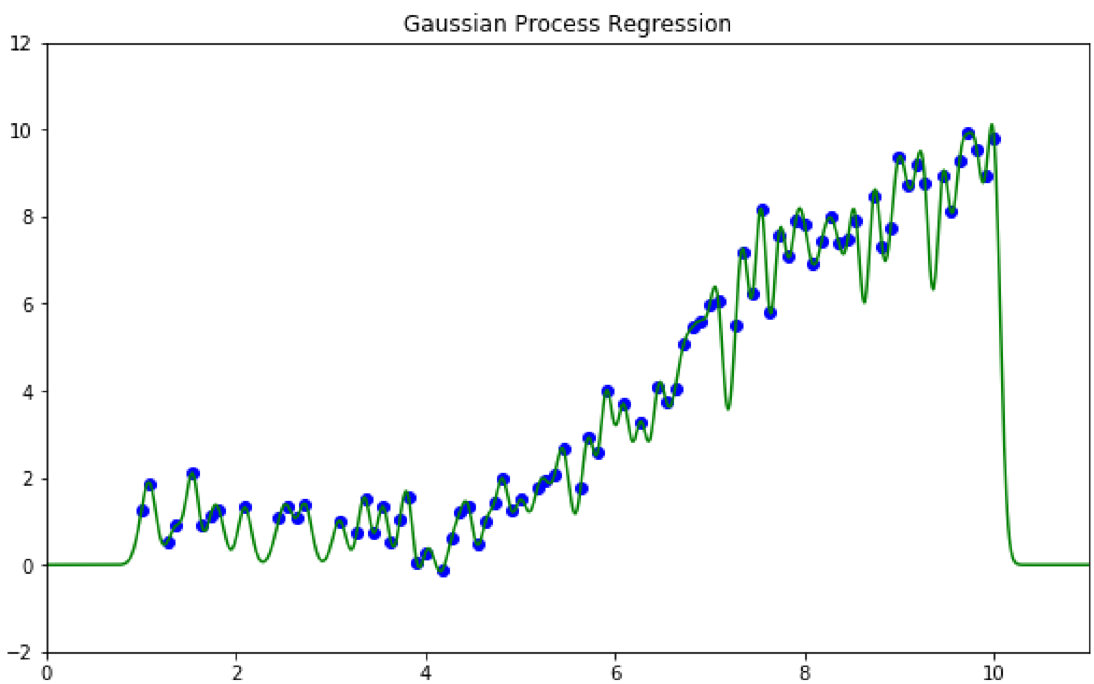


Figure 7: Gaussian Process Regression

The soft KNN doesn't overfit the data, and it gives a smooth curve compared to regular KNN. The Gaussian process regression overfits the data. But it gives confident predictions when the testing point is close to any of the training points. In the region where no data point exists, it gives predictions close to its prior, which is zero in this case.

1.4.2

Let $z = y^* + Ay_{train}$ where $A = -\Sigma_{y^* y_{train}} \Sigma^{-1}_{y_{train} y_{train}}$.
Now we can write

$$\begin{aligned} cov(z, y_{train}) &= cov(y^*, y_{train}) + cov(Ay_{train}, y_{train}) \\ &= \Sigma_{y^* y_{train}} + Avar(y_{train}) \\ &= \Sigma_{y^* y_{train}} - \Sigma_{y^* y_{train}} \Sigma^{-1}_{y_{train} y_{train}} \Sigma_{y_{train} y_{train}} \\ &= 0 \end{aligned} \tag{3}$$

Therefore z and y_{train} and uncorrelated and, since they are jointly normal, they are independent. We know $E(z) = \mu_{y*} + A\mu_{y_{train}} = 0$. Therefore,

$$\begin{aligned} \mu^* = E(y^*|y_{train}) &= E(z - Ay_{train}|y_{train}) \\ &= E(z|y_{train}) - E(Ay_{train}|y_{train}) \\ &= E(z) - Ay_{train} \\ &= -Ay_{train} \\ &= \Sigma^T_{y_{train} y^*} \Sigma^{-1}_{y_{train} y_{train}} y_{train} \end{aligned} \tag{4}$$

Variance:

$$\begin{aligned} \Sigma^* = var(y^*|y_{train}) &= var(z - Ay_{train}|y_{train}) \\ &= var(z|y_{train}) + var(Ay_{train}|y_{train}) - Acov(z, -y_{train}) - cov(z, -y_{train})A' \\ &= \Sigma_{y^* y^*} + \Sigma_{y^* y_{train}} \Sigma^{-1}_{y_{train} y_{train}} \Sigma_{y_{train} y_{train}} \Sigma^{-1}_{y_{train} y_{train}} \Sigma_{y_{train} y^*} - 2\Sigma_{y^* y_{train}} \Sigma^{-1}_{y_{train} y_{train}} \Sigma_{y_{train} y^*} \\ &= \Sigma_{y^* y^*} + \Sigma_{y^* y_{train}} \Sigma^{-1}_{y_{train} y_{train}} \Sigma_{y_{train} y^*} - 2\Sigma_{y^* y_{train}} \Sigma^{-1}_{y_{train} y_{train}} \Sigma_{y_{train} y^*} \\ &= \Sigma_{y^* y^*} - \Sigma^T_{y_{train} y^*} \Sigma^{-1}_{y_{train} y_{train}} \Sigma_{y_{train} y^*} \end{aligned} \tag{5}$$

2

2.1

2.1.1

Let $L(\boldsymbol{W}) = \sum_{m=1}^{M} \frac{1}{2M} \left\| \boldsymbol{W}^T \boldsymbol{x}^m + b - y^m \right\|_2^2 + \frac{\lambda}{2} \left\| \boldsymbol{W} \right\|_2^2$.
For $\boldsymbol{W}$:

$$
\begin{aligned}
L(t\boldsymbol{W_1} + (1-t)\boldsymbol{W_2}) &= \sum_{m=1}^{M} \frac{1}{2M} \left\| (t\boldsymbol{W_1} + (1-t)\boldsymbol{W_2})^T \boldsymbol{x}^m + b - y^m \right\|_2^2 + \frac{\lambda}{2} \left\| t\boldsymbol{W_1} + (1-t)\boldsymbol{W_2} \right\|_2^2 \\
&= \sum_{m=1}^{M} \frac{1}{2M} \left\| t\boldsymbol{W_1}^T \boldsymbol{x}^m + (1-t)\boldsymbol{W_2}^T \boldsymbol{x}^m + b - y^m \right\|_2^2 + \frac{\lambda}{2} \left\| t\boldsymbol{W_1} + (1-t)\boldsymbol{W_2} \right\|_2^2 \\
&= \sum_{m=1}^{M} \frac{1}{2M} \left\| t(\boldsymbol{W_1}^T \boldsymbol{x}^m + b - y^m) + (1-t)(\boldsymbol{W_2}^T \boldsymbol{x}^m + b - y^m) \right\|_2^2 + \frac{\lambda}{2} \left\| t\boldsymbol{W_1} + (1-t)\boldsymbol{W_2} \right\|_2^2 \\
&<= \sum_{m=1}^{M} \frac{1}{2M} \left\| t(\boldsymbol{W_1}^T \boldsymbol{x}^m + b - y^m) \right\|_2^2 + \frac{\lambda}{2} \left\| t\boldsymbol{W_1} \right\|_2^2 + \\
&\qquad \sum_{m=1}^{M} \frac{1}{2M} \left\| (1-t)(\boldsymbol{W_2}^T \boldsymbol{x}^m + b - y^m) \right\|_2^2 + \frac{\lambda}{2} \left\| (1-t)\boldsymbol{W_2} \right\|_2^2 \\
&<= t\sum_{m=1}^{M} \frac{1}{2M} \left\| \boldsymbol{W_1}^T \boldsymbol{x}^m + b - y^m \right\|_2^2 + t\frac{\lambda}{2} \left\| \boldsymbol{W_1} \right\|_2^2 + \\
&\qquad (1-t)\sum_{m=1}^{M} \frac{1}{2M} \left\| \boldsymbol{W_2}^T \boldsymbol{x}^m + b - y^m \right\|_2^2 + (1-t)\frac{\lambda}{2} \left\| (1-t)\boldsymbol{W_2} \right\|_2^2 \\
&<= tL(\boldsymbol{W_1}) + (1-t)L(\boldsymbol{W_2})
\end{aligned}
\tag{6}
$$

For b:

$$
\begin{aligned}
L(tb_1 + (1-t)b_2) &= \sum_{m=1}^{M} \frac{1}{2M} \left\| \boldsymbol{W}^T \boldsymbol{x}^m + (tb_1 + (1-t)b_2) - y^m \right\|_2^2 + \frac{\lambda}{2} \left\| \boldsymbol{W} \right\|_2^2 \\
&= \sum_{m=1}^{M} \frac{1}{2M} \left\| t(\boldsymbol{W}^T \boldsymbol{x}^m + b_1 - y^m) + (1-t)(\boldsymbol{W}^T \boldsymbol{x}^m + b_2 - y^m) \right\|_2^2 + \frac{\lambda}{2} \left\| \boldsymbol{W} \right\|_2^2 \\
&<= \sum_{m=1}^{M} \frac{1}{2M} \left\| t(\boldsymbol{W}^T \boldsymbol{x}^m + b_1 - y^m) \right\|_2^2 + \sum_{m=1}^{M} \frac{1}{2M} \left\| (1-t)(\boldsymbol{W}^T \boldsymbol{x}^m + b_2 - y^m) \right\|_2^2 + \frac{\lambda}{2} \left\| \boldsymbol{W} \right\|_2^2 \\
&<= t\sum_{m=1}^{M} \frac{1}{2M} \left\| \boldsymbol{W}^T \boldsymbol{x}^m + b_1 - y^m \right\|_2^2 + t\frac{\lambda}{2} \left\| \boldsymbol{W} \right\|_2^2 \\
&\qquad + (1-t)\sum_{m=1}^{M} \frac{1}{2M} \left\| \boldsymbol{W}^T \boldsymbol{x}^m + b_2 - y^m \right\|_2^2 + (1-t)\frac{\lambda}{2} \left\| \boldsymbol{W} \right\|_2^2 \\
&<= tL(b_1) + (1-t)L(b_2)
\end{aligned}
\tag{7}
$$

2.1.2

Nth dimension of W, W_n, will be scaled by $\frac{1}{\alpha}$ while other dimensions will not change. It is obvious that change in one dimension is orthogonal to all other dimensions, thus no effect on other dimensions of the weights. Scaling one dimension is essentially spread out all the data points on one dimension (if $\alpha > 1$), and the weight is subsequently scaled by $\frac{1}{\alpha}$.

If we assume shifting happens after scaling, bias will decrease by $\beta * W_n$ where W_n is the new one. This offset happens because shifting a line in one dimension is equivalent to shift the line in another dimension with the offset scaled by the gradient.

Plug it into the inner part of loss equation:

$$\begin{aligned}
\left\| W^{*T} X^{*(m)} + b^* - y^{(m)} \right\| &= \left\| \sum_{i=1, i\neq n}^{N} W_i X_i^{(m)} + W_n^*(\alpha * X_n^{(m)} + \beta) + (b - \beta * W_n^*) - y^{(m)} \right\| \\
&= \left\| \sum_{i=1, i\neq n}^{N} W_i X_i^{(m)} + W_n * X_n^{(m)} + W_n^* * \beta + b - \beta * W_n^* - y^{(m)} \right\| \\
&= \left\| \sum_{i=1}^{N} W_i X_i^{(m)} + b - y^{(m)} \right\| \\
&= \left\| W^{*T} X^{*(m)} + b^* - y^{(m)} \right\|
\end{aligned} \tag{8}$$

We can see the loss stays the same. We can reconfirm our choice above since in order for the loss to be minimum, the derivative of the loss need to equal to 0, which is equivalent for the inner term of the loss function to be zero.

2.1.3

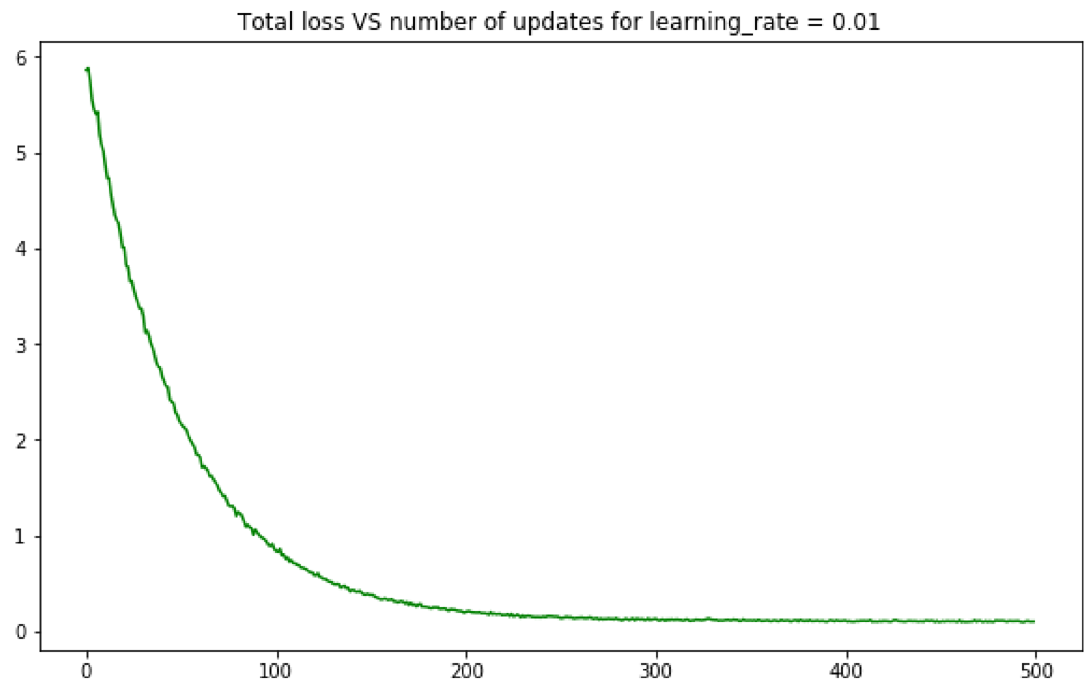
Since we want to find the minimum loss, which does not depend on what method we use to find it, whether regularized or not regularized. Also because linear regression has a exact solution that achieves minimum loss, W and b should be the same as the ones in 2.1.2.
However, the loss will be smaller since one dimension of W is scaled by $\frac{1}{\alpha}$.

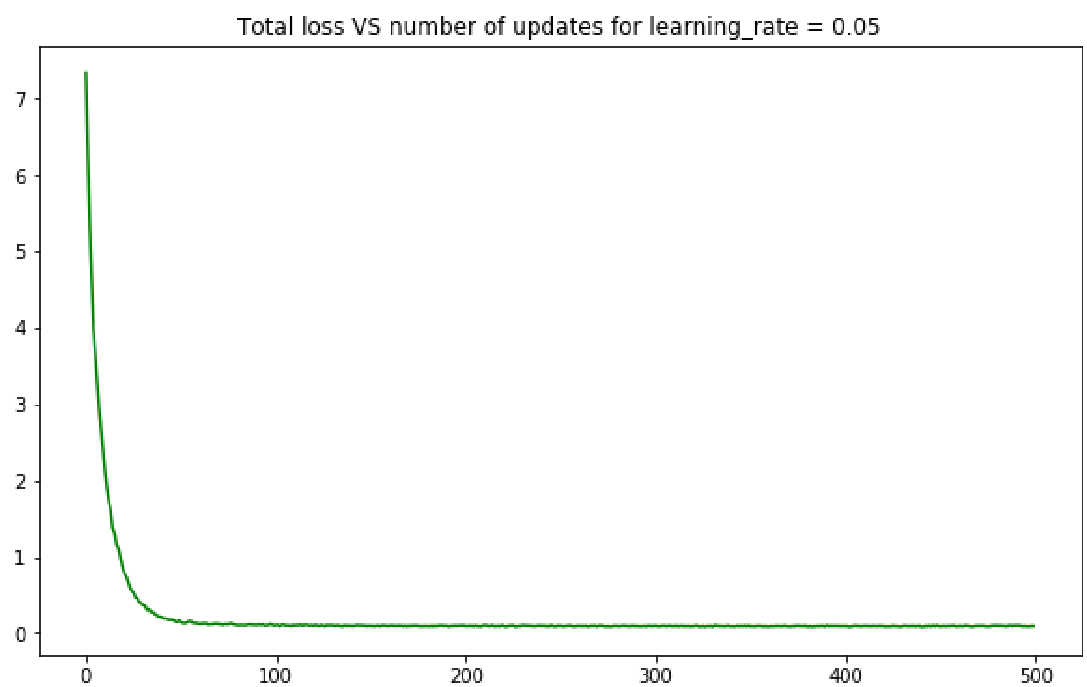
2.1.4

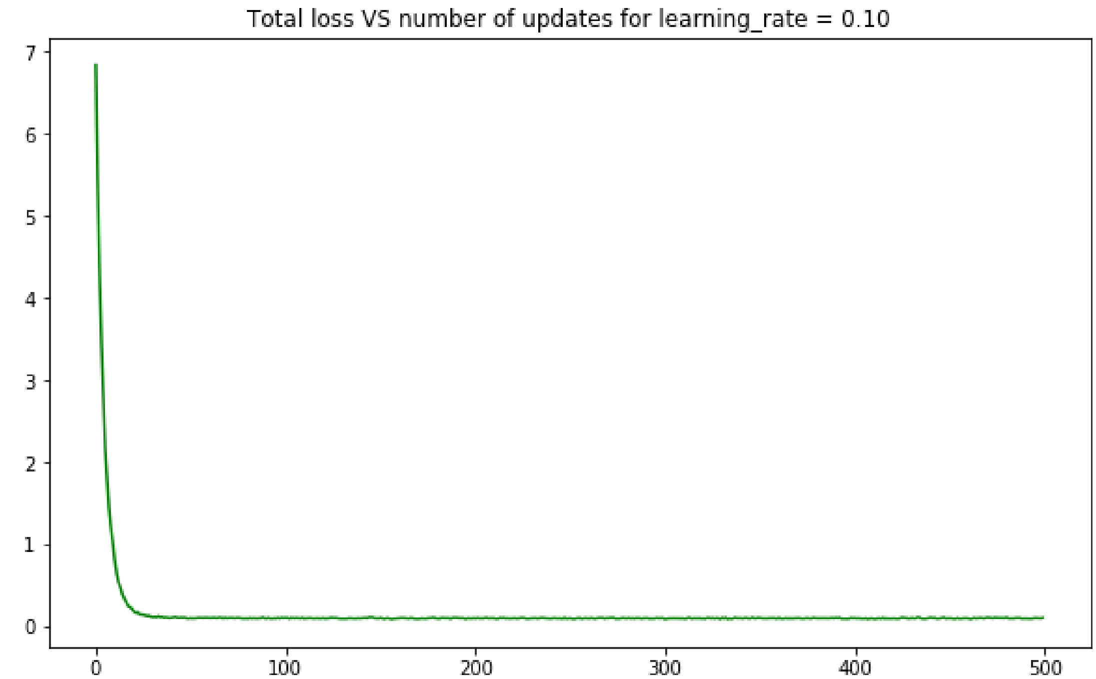
We can use on-hot encoding for every class, and obtain D classifiers for D classes in training set. To classify a new data point, we will feed it into all classifiers and pick the maximum score as the result.

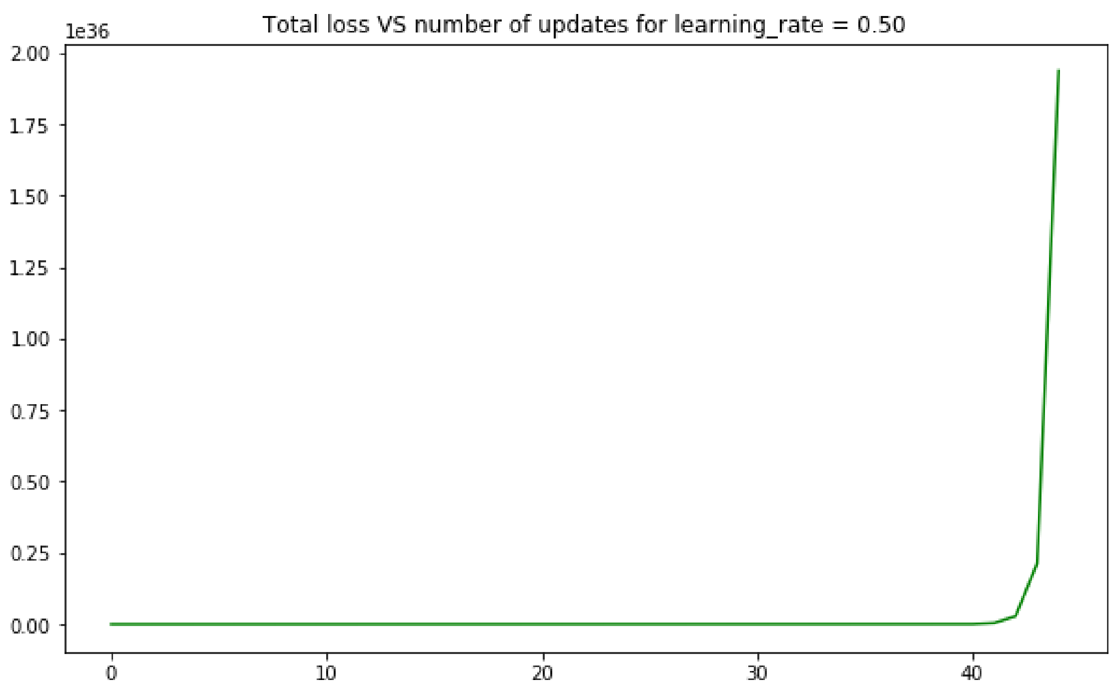
2.2

2.2.1


Total loss VS number of updates for learning_rate = 0.01
6
5
4
3
2
1
0
0
100
200
300
400
500

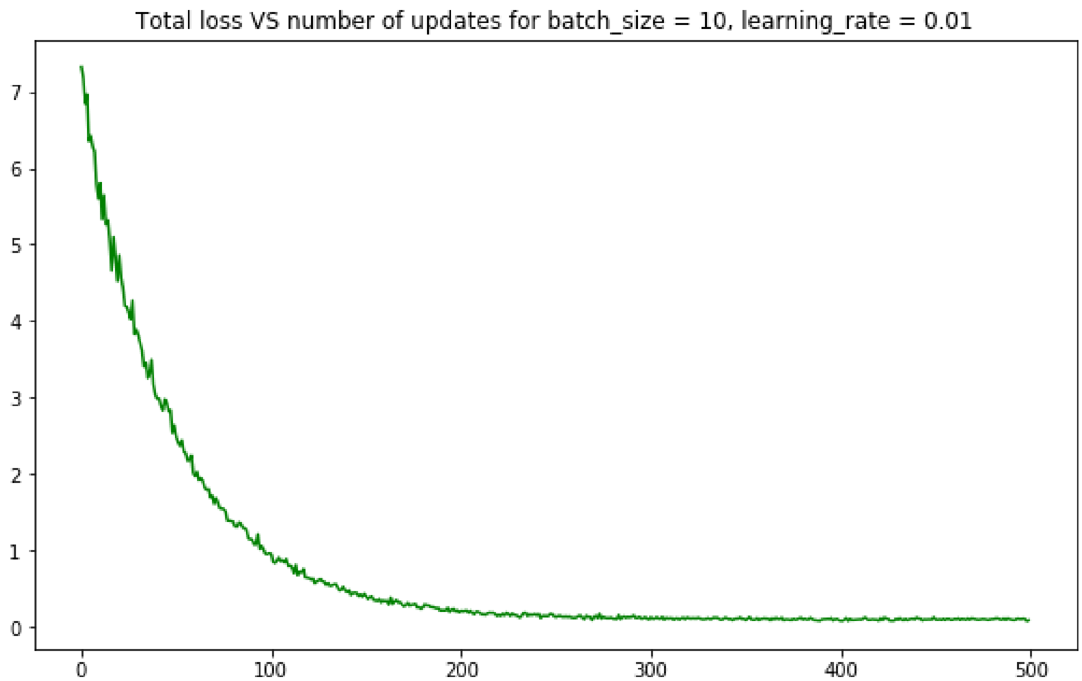

Total loss VS number of updates for learning_rate = 0.05
7
6
5
4
3
2
1
0
0
100
200
300
400
500



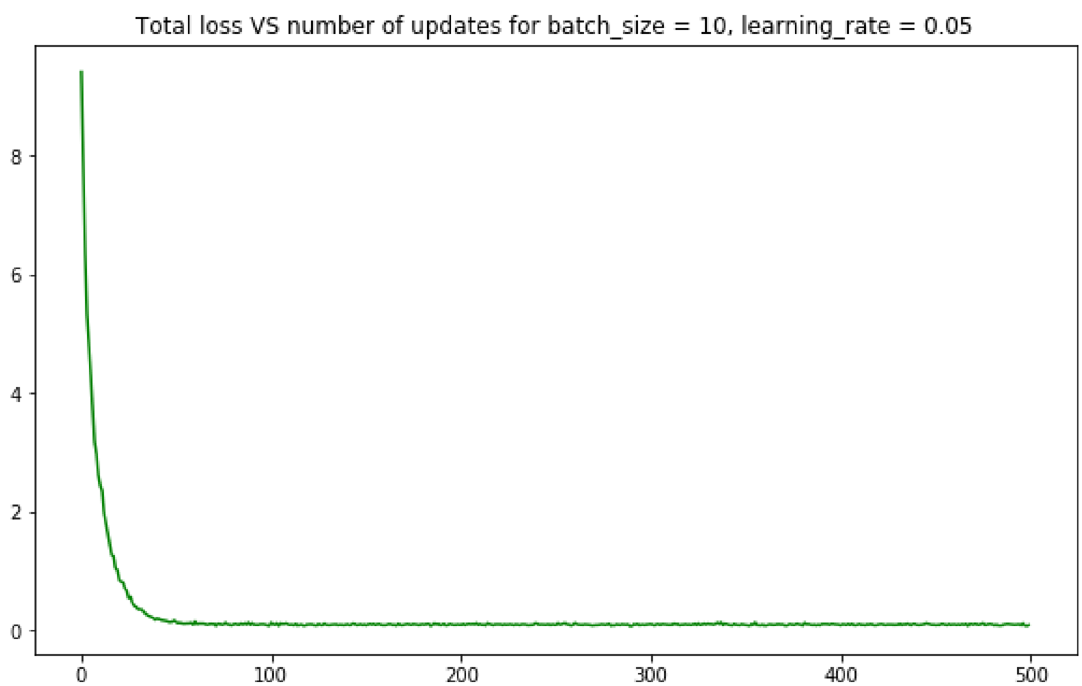


In this problem, we fix batch size to be 50 and decay rate to be 1 and try different learning rate 0.01, 0.05, 0.1, 0.5. From the plot we can see increase learning rate will speed up convergence rate, and we obtain the best learning rate 0.1. However, as learning rate increases, it might diverge like the case of learning rate 0.5.

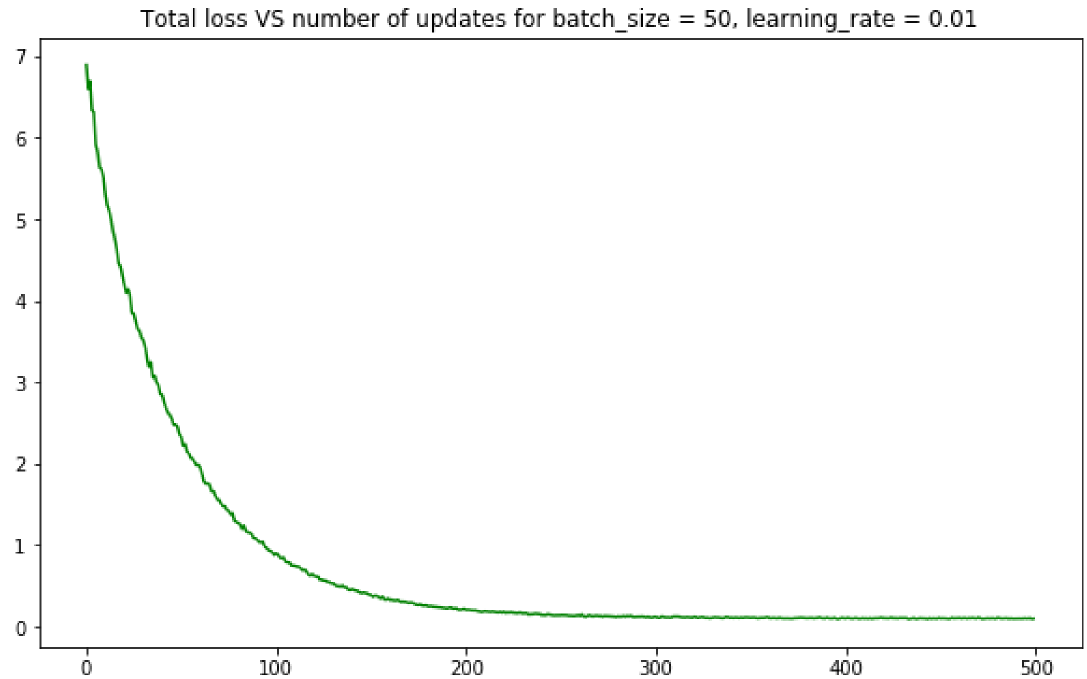
2.2.2

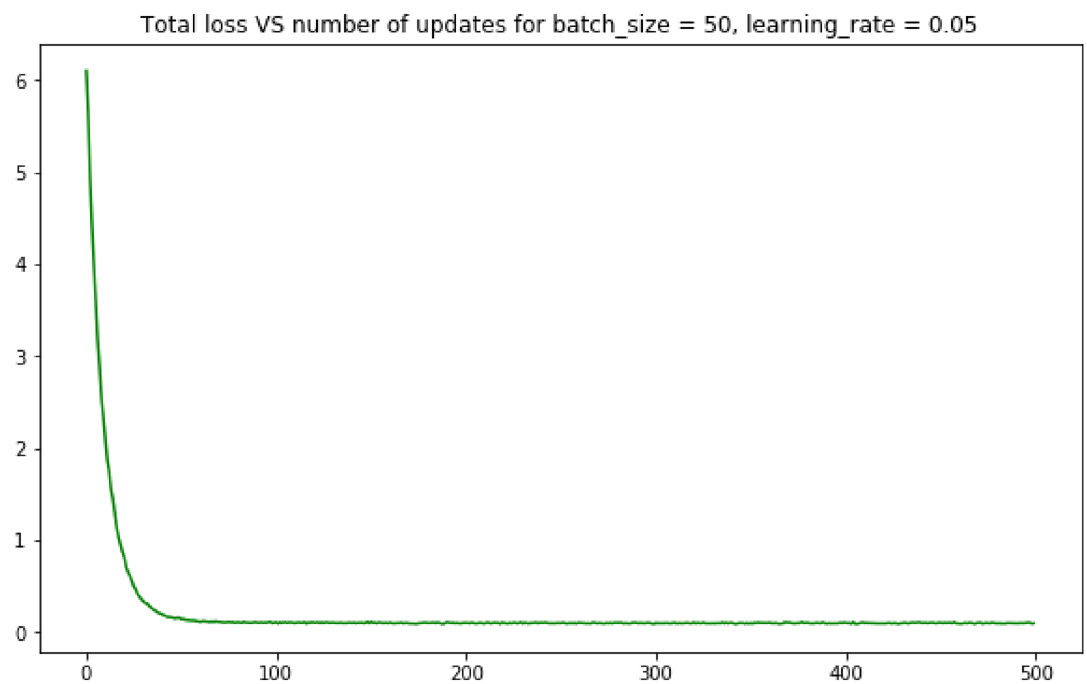


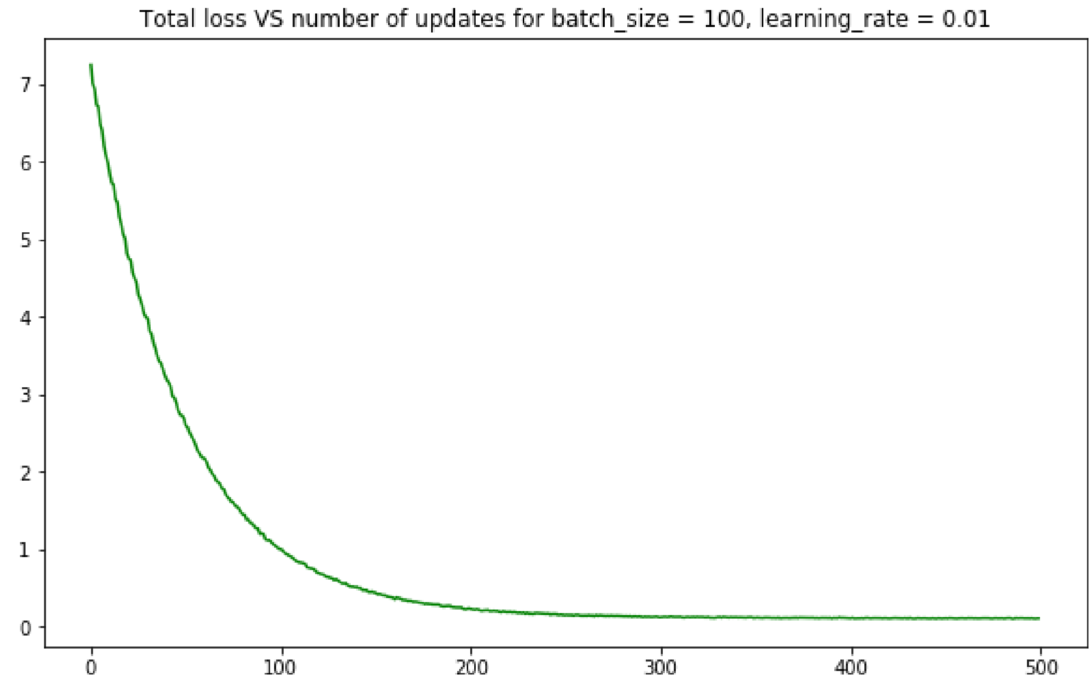
Total loss VS number of updates for batch_size = 10, learning_rate = 0.01
7
6
5
4
3
2
1
0
0
100
200
300
400
500

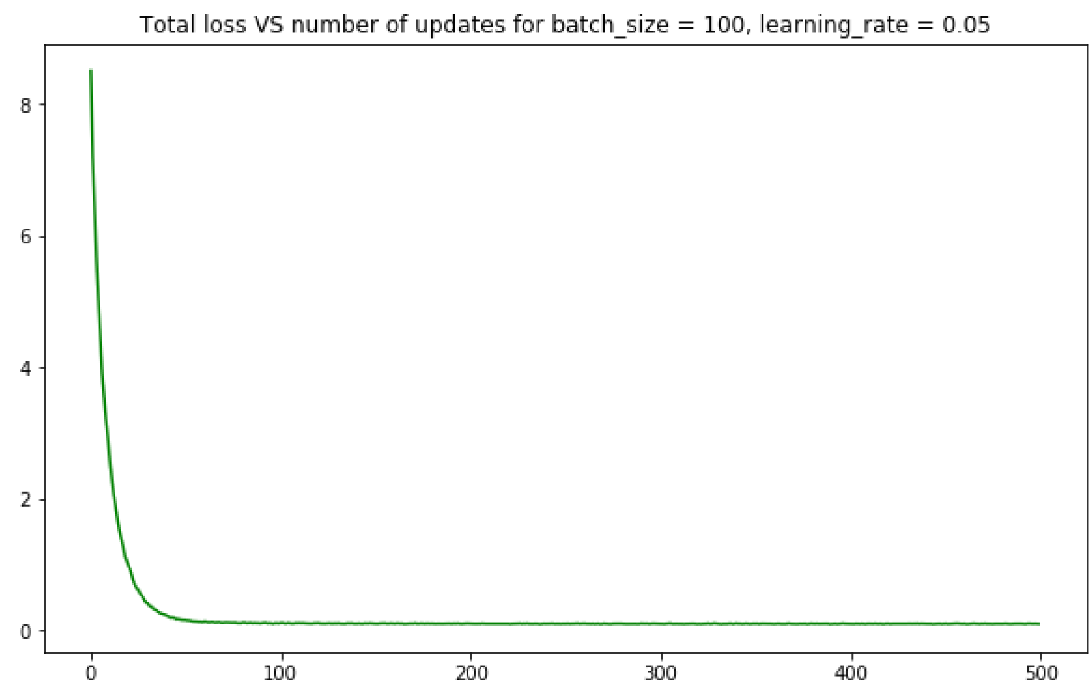


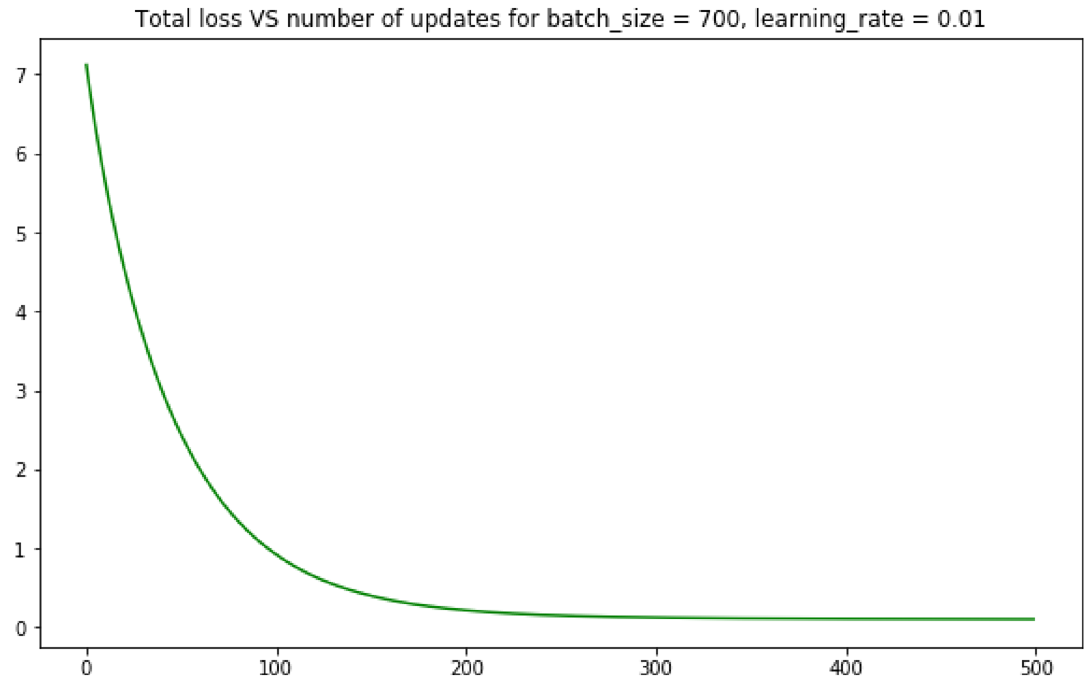
Total loss VS number of updates for batch_size = 10, learning_rate = 0.05
8
6
4
2
0
0
100
200
300
400
500

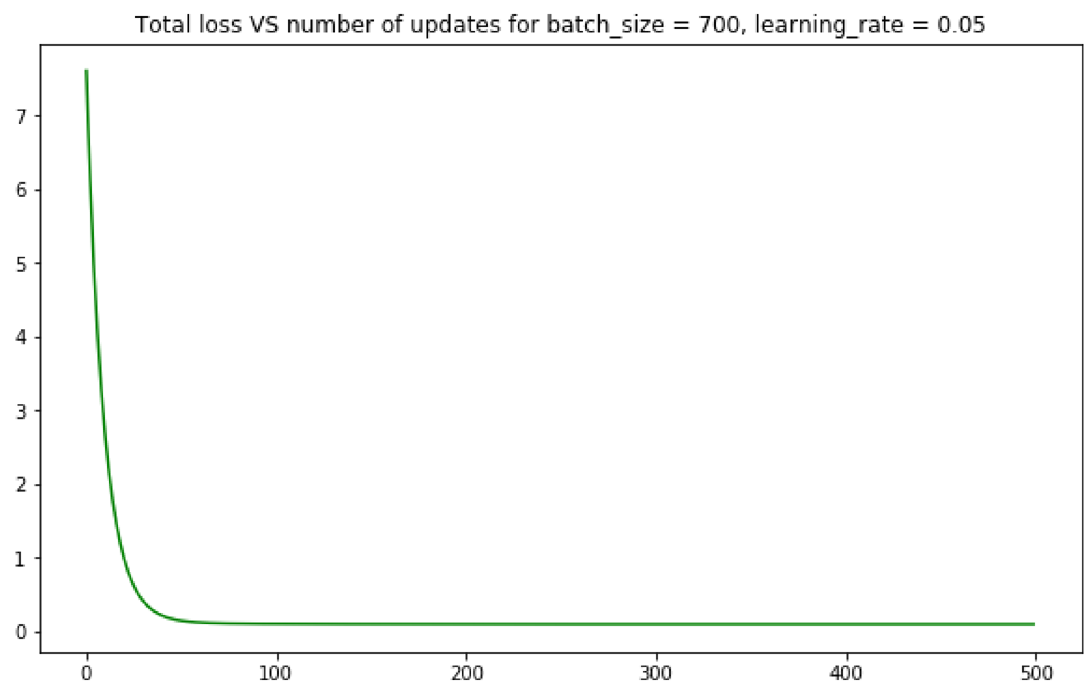

Total loss VS number of updates for batch_size = 50, learning_rate = 0.01
7
6
5
4
3
2
1
0
0
100
200
300
400
500


Total loss VS number of updates for batch_size = 50, learning_rate = 0.05
6
5
4
3
2
1
0
0
100
200
300
400
500


Total loss VS number of updates for batch_size = 100, learning_rate = 0.01
7
6
5
4
3
2
1
0
0
100
200
300
400
500

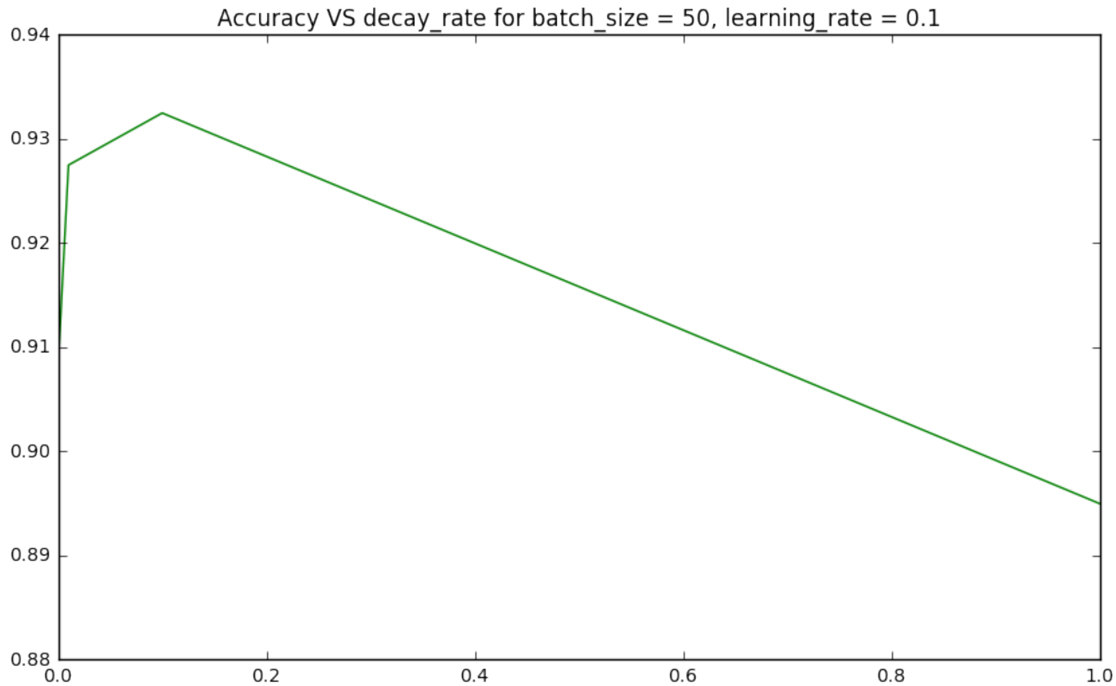

Total loss VS number of updates for batch_size = 100, learning_rate = 0.05
8
6
4
2
0
0
100
200
300
400
500





In this problem we try different batch size and tune the learning rate respectively. We observe that setting learning rate 0.1 for all different batch size obtains the best convergence time overall. We also notice the smaller the batch size is, the more jagged the loss curve will be. And we found that batch size has linear relation with training time. Considering the trade off between training time and randomization, we think batch size 50 is a good choice.

2.2.3



From the experiment, we found that the best accuracy in valid set is 0.94 when the decay rate is 0. Although decay rate 0 obtains good accuracy in valid set, it performs bad in test set with accuracy less than 90%. It encounters overfitting problem when we do not assign any regularization. However, when decay rate is set to 1, we can see its accuracy in test set also decreases dramatically. It is a problem of under fitting. The best decay rate is about 0.1 which obtains best accuracy in test set about 93%. The reason why we use validation set to tune hyper-parameter is to make sure we do not overfit the model. If we use training set to tune hyper-parameter, the weight will be adjusted to minimize the error. However, we cannot test if the model is general enough for some data set that is not used for training. Validation set is such a data set for testing if the training is overfit. If the accuracy in training set is increasing, but the accuracy in validation set remains the same or even decreasing, we know it might be overfiting.

3 Code

```
import tensorflow as tf

def pairwise_dist(x, z):
    z = tf.transpose(z)
    return tf.squared_difference(x, z)

#testing pairwise_dist function
x = tf.constant([[1],[2],[3],[4]])
z = tf.constant([[1],[2],[3]])
init = tf.global_variables_initializer()
sess = tf.InteractiveSession()
sess.run(init)
print(sess.run(pairwise_dist(x,z)))
```

Listing 1: 1.2.2

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

def pairwise_dist(x, z):
    z = tf.transpose(z)
    return tf.squared_difference(x, z)

def get_respon_mat(m, k):
    # get sorted index
    values, indices = tf.nn.top_k(-m, k, sorted=True)

    # build up [[1,2], [1,3]] index form
    indices_pair = tf.tile(tf.reshape(tf.range(0, tf.shape(m)[0]), [-1,1]), [1,k])

    concated = tf.concat(2, [tf.reshape(indices_pair, [-1,k,1]),
        tf.reshape(indices, [-1,k,1])])

    concated = tf.reshape(concated, [-1,2])

    # return dense matrix
    value = 1.0 / k
    res = tf.sparse_to_dense(sparse_indices=concated, output_shape=[tf.shape(m)[0],
        tf.shape(m)[1]], sparse_values=value, validate_indices=False)
    return res

# testing for get_respon_mat function
train = tf.placeholder(tf.float32)
X = tf.placeholder(tf.float32)
target = tf.placeholder(tf.float32)
test_input = np.linspace(0.0, 11.0, num = 1000)
test_input = np.expand_dims(test_input, axis=0)
test_input = np.transpose(test_input)
r = get_respon_mat(pairwise_dist(X, train), 2)
y_hat = tf.matmul(tf.transpose(target), tf.transpose(r))
sess.run(init)
predict = sess.run(y_hat, feed_dict={X: test_input, train: trainData, target:
    trainTarget})

plt.plot(trainData, trainTarget, 'bo', test_input, np.transpose(predict), 'g-')
plt.axis([0,11, -2, 12])
plt.title("KNN for k = 2")
plt.show()
```

Listing 2: 1.3.1

```python
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
kList = [1,3,5,50]
train = tf.placeholder(tf.float32)
X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)
target = tf.placeholder(tf.float32)
test_input = np.linspace(0.0, 11.0, num = 1000)
test_input = np.expand_dims(test_input, axis=0)
test_input = np.transpose(test_input)
plot_input = tf.constant(test_input)
best_k = 0
min_err = float("inf")
sess.run(init)
for k in kList:
    # Error definition
    r = get_respon_mat(pairwise_dist(X, train), k)
    y_hat = tf.matmul(tf.transpose(target), tf.transpose(r))
    meanSquaredError = tf.reduce_mean(tf.reduce_sum(tf.square(y_hat -
        tf.transpose(Y)), reduction_indices=[1]))
```

```
    trainMSE = sess.run(meanSquaredError, feed_dict={X: trainData, Y: trainTarget,
        train: trainData, target: trainTarget})
    validMSE = sess.run(meanSquaredError, feed_dict={X: validData, Y: validTarget,
        train: trainData, target: trainTarget})
    testMSE = sess.run(meanSquaredError, feed_dict={X: testData, Y: testTarget,
        train: trainData, target: trainTarget})
    plot_predict = sess.run(y_hat, feed_dict={X: test_input, train: trainData,
        target: trainTarget})
    if validMSE < min_err:
        min_err = validMSE
        best_k = k
    print("k: %2d, trainMSE: %0.2f, validMSE: %0.2f, testMSE: %0.2f"%(k, trainMSE,
    validMSE, testMSE))
    plt.plot(trainData, trainTarget, 'bo', test_input, np.transpose(plot_predict),
        'g-')
    plt.axis([0,11, -2, 12])
    plt.title("KNN for k = %d"%(k))
    plt.show()
print("Best k value based on validation err is: %d"%(best_k))
```

Listing 3: 1.3.2

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

def get_soft_knn_res_mat(x, z, lamb):
    z = tf.transpose(z)
    Kxx = tf.exp(tf.cast(tf.squared_difference(x, z), tf.float32) * -lamb)
    K_norm = tf.expand_dims(tf.reduce_sum(Kxx, reduction_indices=1), 1)
    return tf.div(Kxx, K_norm)

def get_Gaussian_knn_res_mat(x, z, lamb):
    z = tf.transpose(z)
    KxX = tf.exp(tf.cast(tf.squared_difference(x, z), tf.float32) * -lamb)
    KXX = tf.exp(tf.cast(tf.squared_difference(x, tf.transpose(x)), tf.float32)
        * -lamb)
    KXX_inverse = tf.matrix_inverse(KXX)
    return tf.matmul(KXX_inverse, KxX)


myLambda = 100
train = tf.placeholder(tf.float32)
X = tf.placeholder(tf.float32)
target = tf.placeholder(tf.float32)
test_input = np.linspace(0.0, 11.0, num = 1000)
test_input = np.expand_dims(test_input, axis=0)
test_input = np.transpose(test_input)

# r = get_Gaussian_knn_res_mat(train, X, myLambda)
r = get_soft_knn_res_mat(X, train, myLambda)

y_hat = tf.matmul(tf.transpose(target), tf.transpose(r))
sess.run(init)
predict = sess.run(y_hat, feed_dict={X: test_input, train: trainData, target:
    trainTarget})

plt.plot(trainData, trainTarget, 'bo', test_input, np.transpose(predict), 'g-')
plt.axis([0,11, -2, 12])
plt.title("Soft KNN")
plt.show()
```

Listing 4: 1.4.1

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

```

```
with np.load ("../data/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"]

def buildGraph(decay_rate, learning_rate):
    # 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 definition
    error =  0.5 *
        tf.reduce_mean(tf.cast(tf.reduce_sum(tf.cast(tf.square(y_predicted -
        y_target), tf.float32), reduction_indices=1,
        name='squared_error'),
        tf.float32), name='mean_squared_error')
        + 0.5 * decay_rate * tf.reduce_sum(tf.cast(tf.square(W), tf.float32))

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

def getRandomBatch(trainData, trainTarget, size):
    idx = np.random.choice(trainData.shape[0], size, replace=False)
    return trainData[idx,:], trainTarget[idx,:]
```

Listing 5: 2.2

```
#2.1 fix decay_rate = 1 and tune learning_rate
decay_rate = 1
batch_size = 50
learning_rate_list = [0.01, 0.05, 0.1, 0.5]
for learning_rate in learning_rate_list:
    W, b, X, y_target, y_predicted, error, train = buildGraph(decay_rate,
    learning_rate)
    init = tf.global_variables_initializer()
    sess = tf.InteractiveSession()
    sess.run(init)
    loss_recorder = np.array([])
    for itr in range(500):
        batch_xs, batch_ys = getRandomBatch(trainData, trainTarget, batch_size)
        loss, _ = sess.run([error, train], feed_dict={X: batch_xs, y_target:
    batch_ys})
        loss_recorder = np.append(loss_recorder, loss)
    plt.plot(np.arange(500), loss_recorder, 'g')
    #plt.axis([0,2000, 0, 2])
    plt.title("Total loss VS number of updates for learning_rate = %0.2f"%(
    learning_rate))
    plt.show()
```

Listing 6: 2.2.1

```
#2.2 fix decay_rate = 1 and tune learning_rate and batch_size
decay_rate = 1
batch_size_list = [10,50,100,700]
learning_rate_list = [0.01, 0.05, 0.1, 0.5]
for batch_size in batch_size_list:
    for learning_rate in learning_rate_list:
        W, b, X, y_target, y_predicted, error, train = buildGraph(decay_rate,
    learning_rate)
        init = tf.global_variables_initializer()
        sess = tf.InteractiveSession()
```

```python
        sess.run(init)
        loss_recorder = np.array([])
        for itr in range(500):
            batch_xs, batch_ys = getRandomBatch(trainData, trainTarget, batch_size)
            loss, _ = sess.run([error, train], feed_dict={X: batch_xs, y_target: batch_ys})
            loss_recorder = np.append(loss_recorder, loss)
        plt.plot(np.arange(500), loss_recorder, 'g')
        #plt.axis([0,2000, 0, 2])
        plt.title("Total loss VS number of updates for batch_size = %d, learning_rate = %0.2f"%(batch_size, learning_rate))
        plt.show()
```

Listing 7: 2.2.2

```python
#2.3 fix batch_size = 50 and learning_rate = 0.1 and tune decay_rate
decay_rate_list = [0., 0.0001, 0.001, 0.01, 0.1, 1.]
batch_size = 50
learning_rate = 0.1
best_decay_rate = 0
best_acc = 0
test_acc_recorder = np.array([])
for decay_rate in decay_rate_list:
    W, b, X, y_target, y_predicted, error, train = buildGraph(decay_rate, learning_rate)
    correct_prediction = tf.equal(y_target, (tf.sign(y_predicted - 0.5) + 1)/2)
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    init = tf.global_variables_initializer()
    sess = tf.InteractiveSession()
    sess.run(init)
    for itr in range(500):
        batch_xs, batch_ys = getRandomBatch(trainData, trainTarget, batch_size)
        sess.run(train, feed_dict={X: batch_xs, y_target: batch_ys})
    # use valid data set accuracy to tune decay rate
    cur_acc = sess.run(accuracy, feed_dict={X: validData, y_target: validTarget})
    if cur_acc > best_acc:
        best_acc = cur_acc
        best_decay_rate = decay_rate
    # use test set to test accuracy
    test_acc_recorder = np.append(test_acc_recorder, sess.run(accuracy, \
                                                              feed_dict={X: testData, y_target: testTarget}))
print("Best accuracy in valid set is: %0.2f, decay_rate is: %0.2f"%(best_acc, best_decay_rate))
plt.plot(decay_rate_list, test_acc_recorder, 'g')
#plt.axis([0,2000, 0, 2])
plt.title("Accuracy VS decay_rate for batch_size = 50, learning_rate = 0.1")
plt.show()
```

Listing 8: 2.2.3