CSE - 512 ML (HW 6)

Name: Venkata Subba Narasa Bharath, Meadam

SBU ID: 112672986

Q1) (a) $f(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\sum_{k=1}^{K} y_{ik} x_i^T \theta_k - \log \sum_{k=1}^{K} \exp(x_i^T \theta_k) \right)$

For gradient, $f'(\theta) = 0$

$$f'(\theta) = \frac{1}{m} \frac{\partial}{\partial \theta} \left(\sum_{i=1}^{m} \left(\sum_{k=1}^{K} y_{ik} x_i^T \theta_k - \log \sum_{k=1}^{K} \exp(x_i^T \theta_k) \right) \right)$$

$$= \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial \theta} \left(\left(\sum_{k=1}^{K} y_{ik} x_i^T \theta_k - \log \sum_{k=1}^{K} \exp(x_i^T \theta_k) \right) \right)$$

$$= \frac{1}{m} \left(\sum_{k=1}^{K} y_{ik} x_i^T \right) - \frac{1}{m} \left(\frac{1}{\sum_{k=1}^{K} \exp(x_i^T \theta_k)} \frac{\partial}{\partial \theta} \left(\exp(x_i^T \theta_k) \right) \right)$$

$$= \frac{1}{m} \left(\sum_{k=1}^{K} y_{ik} x_i^T \right) - \frac{1}{m} \left(\frac{\exp(x_i^T \theta_k)}{\sum_{k=1}^{K} \exp(x_i^T \theta_k)} * x_i^T \right)$$

$$\boxed{f'(\theta) = \frac{1}{m} \left(\sum_{k=1}^{m} y_{ik} x_i^T - \frac{\exp(x_i^T \theta_k) * x_i^T}{\sum_{k=1}^{K} \exp(x_i^T \theta_k} \right)}$$

(C) $f(\theta) = \log\left(\sum_{i=1}^{m} \exp(\theta_i - D)\right)$

For $f(\theta) \leq 1$

$$\Rightarrow \log\left(\sum_{i=1}^{m} \exp(\theta_i - D)\right) \leq 1$$

$$\Rightarrow \sum_{i=1}^{m} e^{(\theta_i - D)} \leq e$$

$$\Rightarrow \sum_{i=1}^{m} \frac{e^{\theta_i}}{e^{D}} \leq e$$

$$\Rightarrow \frac{1}{e^{D}} \sum_{i=1}^{m} e^{\theta_i} \leq e$$

$$\sum_{i=1}^{m} e^{\theta_i} \leq e \cdot e^{D}$$

$$\sum_{i=1}^{m} e^{\theta_i} \leq e^{D+1}$$

$$\log\left(\sum_{i=1}^{m} e^{\theta_i}\right) \leq D+1$$

$$\boxed{D \geq \log\left(\sum_{i=1}^{m} e^{\theta_i}\right) - 1}$$ → Prevent underflow

For $f(\theta) \geq 1$

$$\Rightarrow \log\left(\sum_{i=1}^{m} \exp(\theta_i - D)\right) \geq$$

$$\Rightarrow \sum_{i=1}^{m} e^{(\theta_i - D)} \geq e$$

$$\Rightarrow \sum_{i=1}^{m} \frac{e^{\theta_i}}{e^{D}} \geq e$$

$$\Rightarrow \frac{1}{e^{D}} \sum_{i=1}^{m} e^{\theta_i} \geq e$$

$$\Rightarrow \sum_{i=1}^{m} e^{\theta_i} \geq e \cdot e^{D}$$

$$\Rightarrow \sum_{i=1}^{m} e^{\theta_i} \geq e^{D+1}$$

$$\Rightarrow \log\left(\sum_{i=1}^{m} e^{\theta_i}\right) \geq D+1$$

$$\boxed{D \leq \log\left(\sum_{i=1}^{m} e^{\theta_i}\right) - 1} \rightarrow \text{Prevent overflow}$$

HW6 Q1(d)

In [1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.io as sio

from collections import Counter
import random

from tabulate import tabulate
```

In [2]:
```python
from sklearn import preprocessing
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
```

In [3]:
```python
data = sio.loadmat('mnist.mat')
```

Converting data from uint8 to float

In [4]:
```python
print("Type Before type-casting: "+str(type(data['trainX'][0][19])))

XTrain = data['trainX'].astype(float)
yTrain = data['trainY'][0].astype(float)

XTest = data['testX'].astype(float)
yTest = data['testY'][0].astype(float)

print("Type After  type-casting: "+str(type(XTrain[0][19])))
```

```
Type Before type-casting: <class 'numpy.uint8'>
Type After  type-casting: <class 'numpy.float64'>
```

In [5]:
```python
scaler = StandardScaler()
XTrain = scaler.fit_transform(XTrain)
XTest = scaler.fit_transform(XTest)
```

In [6]:
```python
print("Shape of XTrain: "+str(np.shape(XTrain)))
print("Shape of yTrain: "+str(np.shape(yTrain)))

print("Shape of XTest: "+str(np.shape(XTest)))
print("Shape of yTest: "+str(np.shape(yTest)))
```

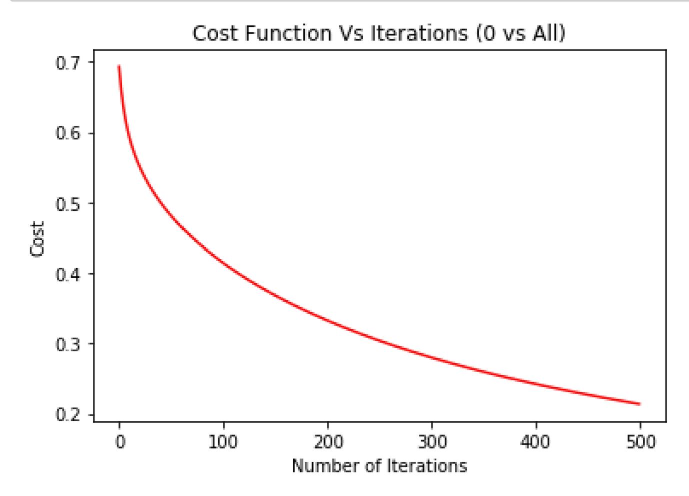
```
Shape of XTrain: (60000, 784)
Shape of yTrain: (60000,)
Shape of XTest: (10000, 784)
Shape of yTest: (10000,)
```

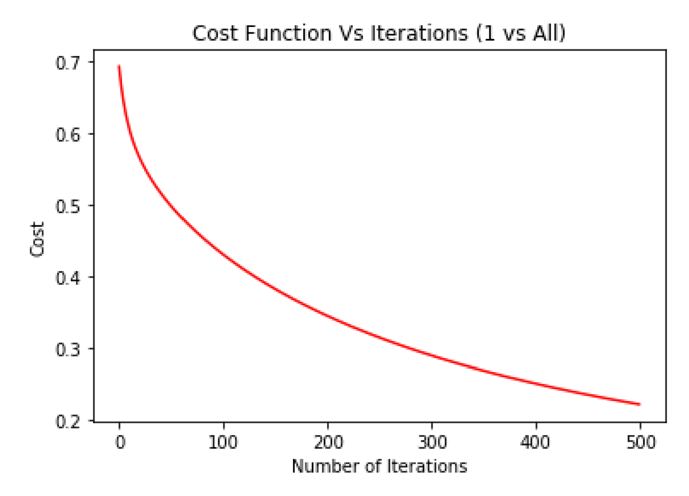
In [7]:
```python
def sigmoid(x):
    return (1 / (1 + np.exp(-x)))

def costFunction(h,theta, y):
    m = len(y)
    cost = (1 / m) * (np.sum(-y.T.dot(np.log(h)) - (1 - y).T.dot(np.log(1 - h))))
    return cost

def gradientDescent(X,h,theta,y,m,alpha=0.01): # This function calculates the theta value by gradient d
escent
    gradient_value = np.dot(X.T, (h - y)) / m
    theta -= alpha * gradient_value
    return theta

def predict(X, theta):
    X = np.insert(X, 0, 1, axis=1)
    X_predicted = [max((sigmoid(i.dot(thetaTemp)), c) for thetaTemp, c in theta)[1] for i in X ]
    return X_predicted


def getMisClassificationRate(y,yPred):
    total = 0
    for i in range(len(y)):
        if y[i] == yPred[i]:
            total+=1
    return 1 - total/len(y)

def plotCost(cost):
    df = pd.DataFrame(data=cost)
    for i in range(df.shape[1]):
        plt.plot(df[i],'r')
        plt.title("Cost Function Vs Iterations " + '(' + str(i) +" vs All)")
        plt.xlabel("Number of Iterations")
        plt.ylabel("Cost")
        plt.show()

def plotMisClassificationRate(misClassificationRate, datasetType):
    plt.figure(figsize=(12,8))
    plt.plot(misClassificationRate)
    plt.title("misClassificationRate Vs Iterations ("+str(datasetType) + ')',fontsize=18)
    plt.xlabel("Number of Iterations",fontsize=12)
    plt.ylabel("misClassificationRate",fontsize=12)
    plt.show()
```

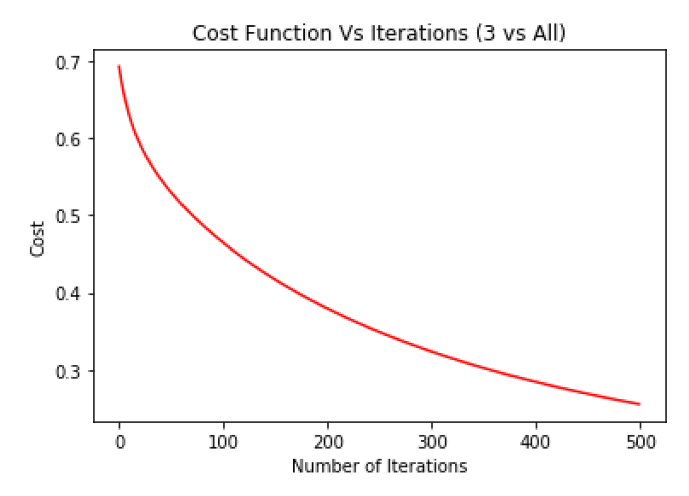
In [8]:
```python
def fitLogisticRegression(X, y, XTest, yTest,iterations):

    theta = [[]] * 10
    cost = np.zeros((iterations,10))
    # The bias component
    XT = X.copy()
    X = np.insert(X, 0, 1, axis=1)
    m = len(y)

    misClassificationRateTest = []
    misClassificationRateTrain = []
    # Building a one vs all model
    for iteration in range(iterations):
        for i in np.unique(y):
            # Uniqie values will be [0,1,2,3,4,5,6,7,8,9]
            y_onevsall = np.where(y == i, 1, 0)
            # number of features (28 * 28 = 784)
            if iteration == 0:
                thetaTemp = np.zeros(X.shape[1])
            else:
                thetaTemp = theta[int(i)][0]

            z = X.dot(thetaTemp)
            h = sigmoid(z)
            thetaTemp = gradientDescent(X,h,thetaTemp,y_onevsall,m)
            costTemp = costFunction(h,thetaTemp,y_onevsall)
            theta[int(i)] = [thetaTemp,i]
            cost[iteration][int(i)] = costTemp
        predition1 = predict(XTest,theta)
        score1 = getMisClassificationRate(predition1,yTest)
        misClassificationRateTest.append(score1)

        predition2 = predict(XT,theta)
        score2 = getMisClassificationRate(predition2,y)
        misClassificationRateTrain.append(score2)

    return theta,cost,misClassificationRateTest, misClassificationRateTrain
```

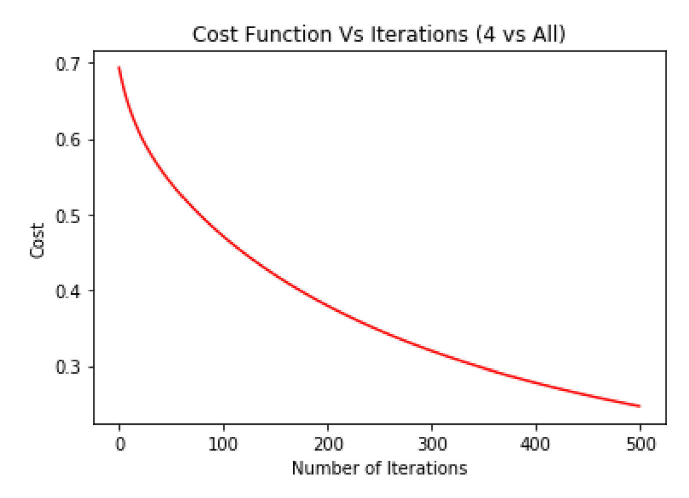
In [9]:
```python
theta,cost,misClassificationRateTest, misClassificationRateTrain = fitLogisticRegression(XTrain, yTrain
,
                                                                                          XTest,yTest,
                                                                                          500)
```

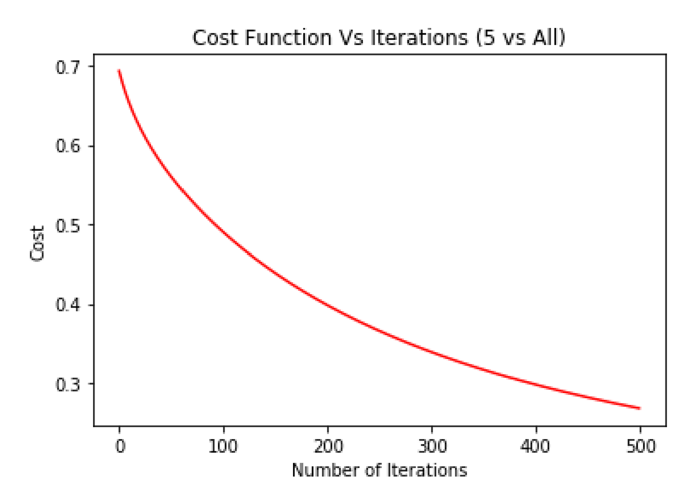
In [10]:
```python
plotCost(cost)
```

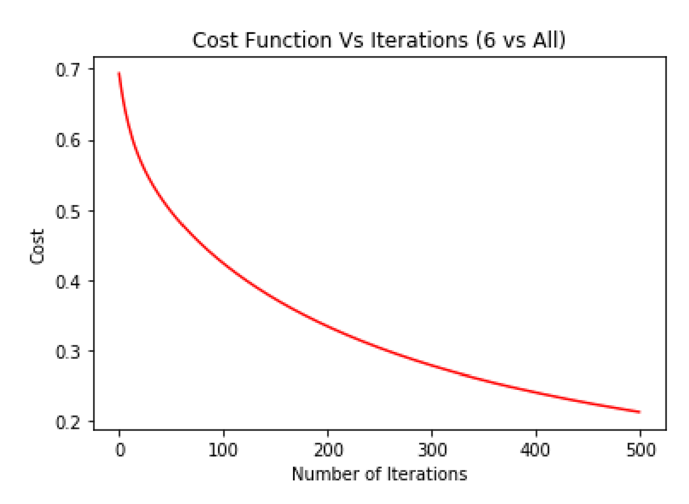


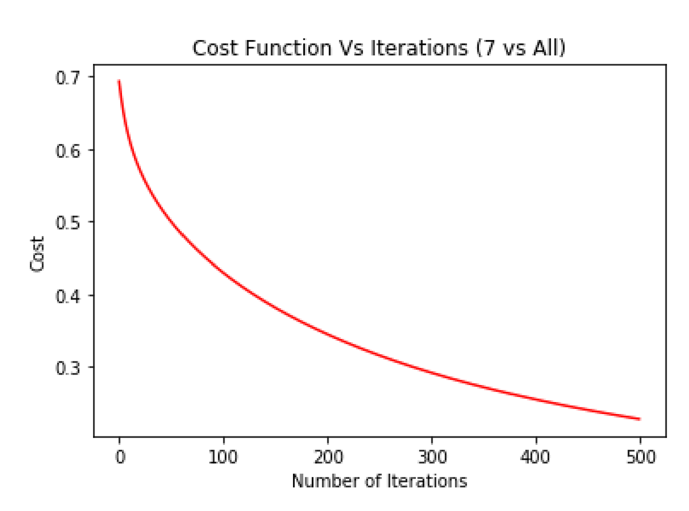


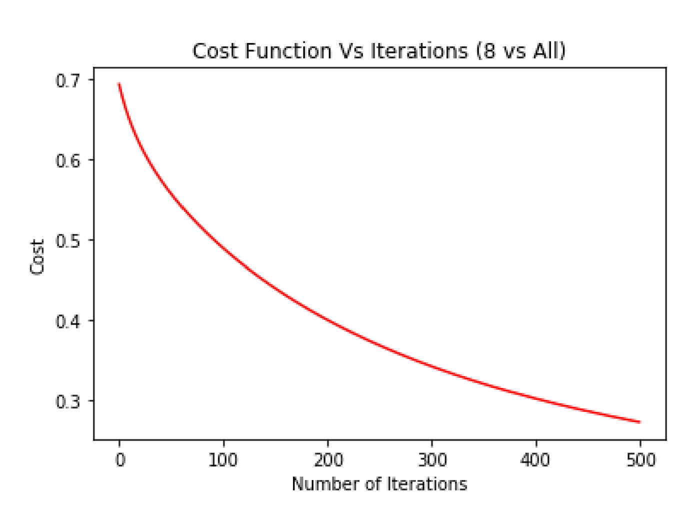


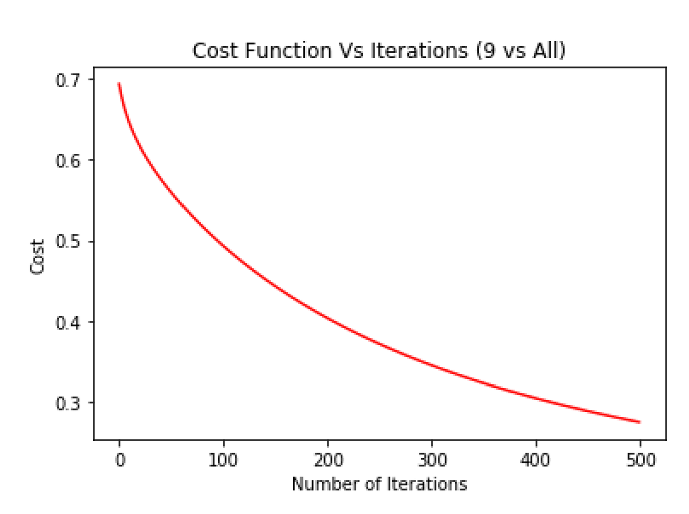




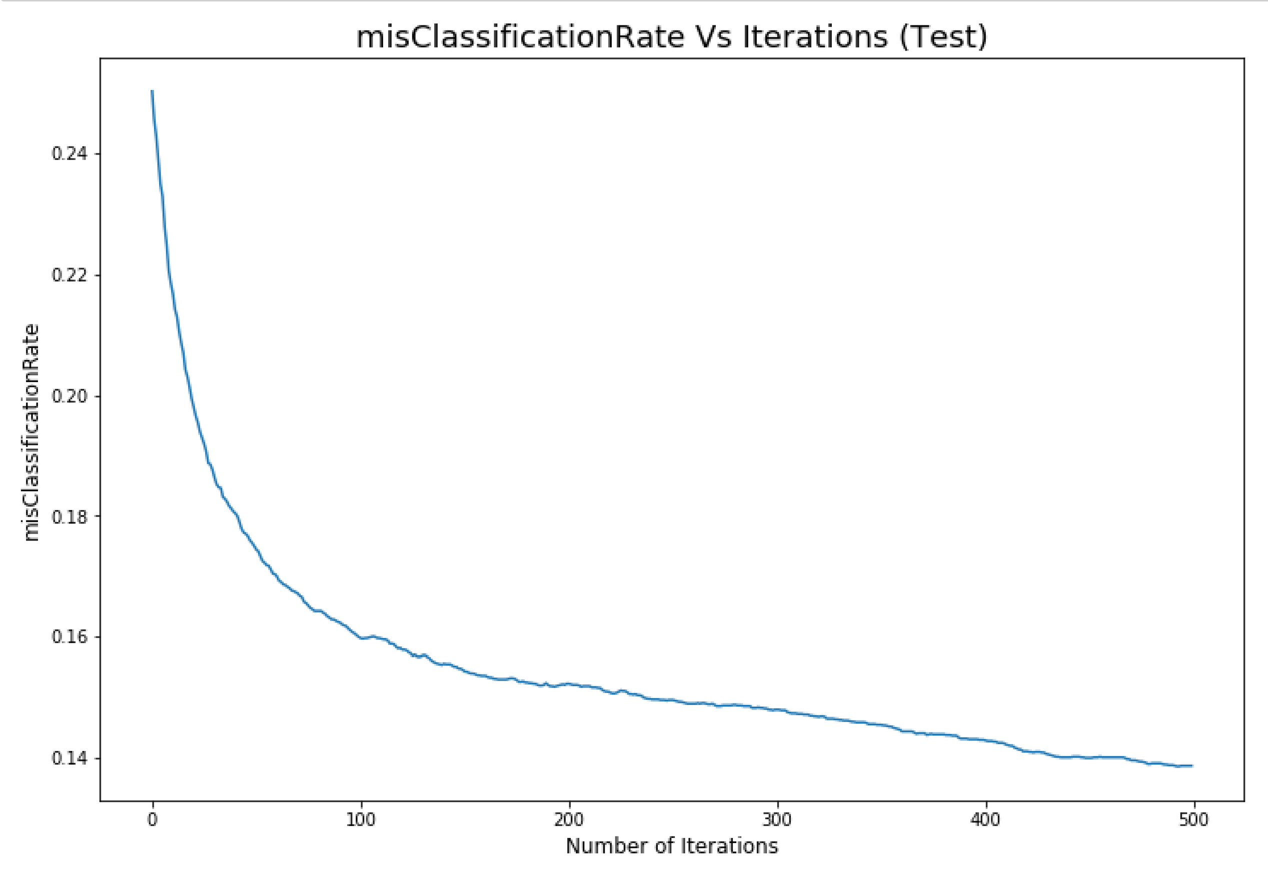




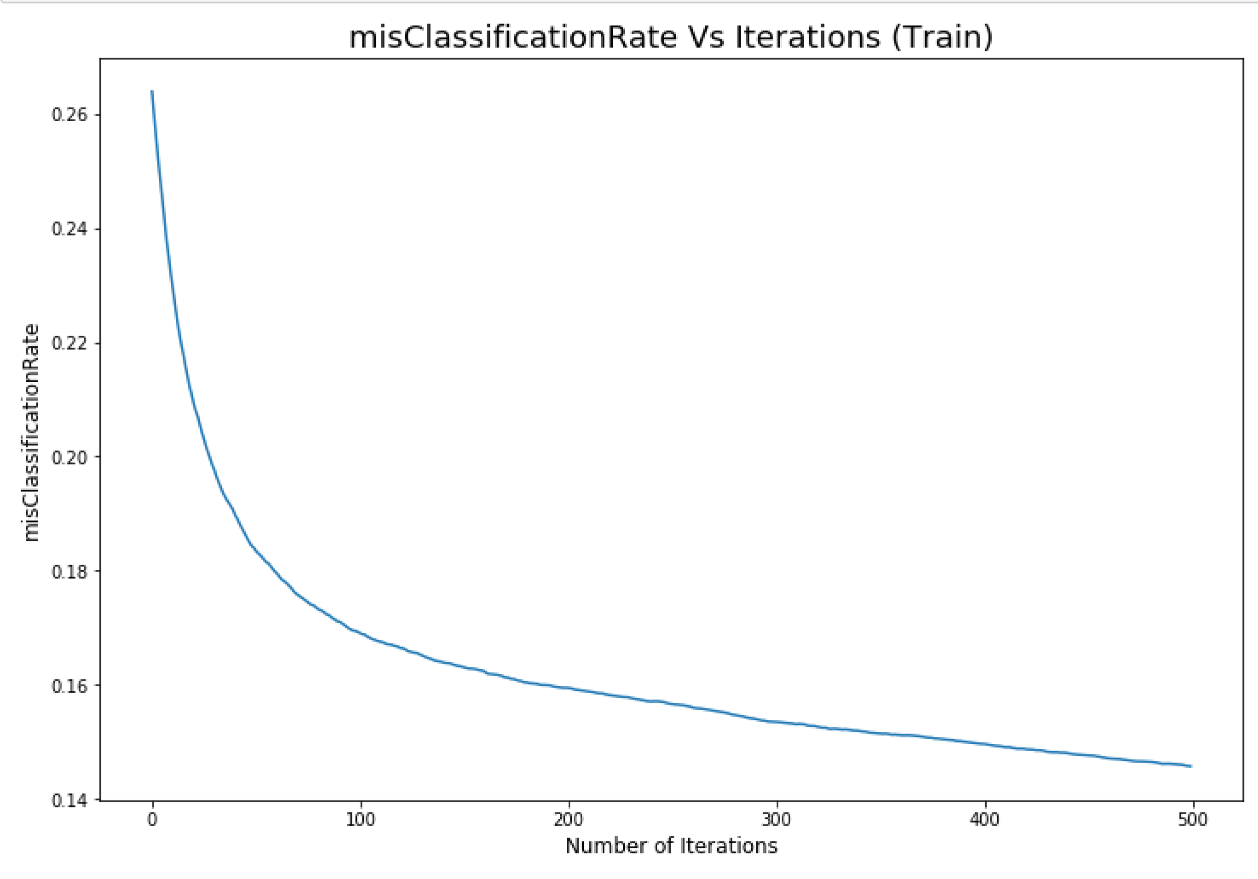




In [11]:
```python
plotMisClassificationRate(misClassificationRateTest, "Test")
```



In [12]:
```python
plotMisClassificationRate(misClassificationRateTrain, "Train")
```



In [15]:
```python
finalTestMisClassificationRate = round(misClassificationRateTest[-1]*100,3)
finalTrainMisClassificationRate = round(misClassificationRateTrain[-1]*100,3)
```

The Final Train misclassification rate: 14.57%

The Final Test misclassification rate: 13.86%

Resources:

1) https://gluon.mxnet.io/chapter02_supervised-learning/softmax-regression-scratch.html

2)https://www.pugetsystems.com/labs/hpc/Machine-Learning-and-Data-Science-Multinomial-Multiclass-Logistic-Regression-1007/

3) https://www.codeproject.com/Articles/821347/MultiClass-Logistic-Classifier-in-Python

Q2) (a)

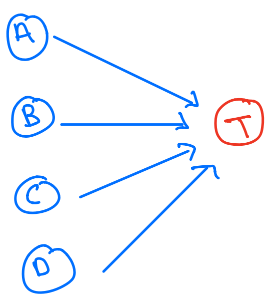
A → It is raining

B → want to walk outside

C → feel sick

D → Day of the week

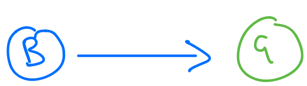
T → wear Top



(b)

B → want to walk outside

G → wear green Hoodie



$P(B|C) = 0.1$, $P(B|\neg C) = 0.6$

$P(C|A) = 0.7$, $P(C|\neg A) = 0.15$

$P(\neg C|A) = 0.3$

$$P(G) = P(G|B)\left[P(B|C) * P(C|A) + P(B|\neg C) * P(\neg C|A)\right]$$

$$= 1 * [(0.1 * 0.7) + (0.6 \times 0.3)]$$

$$= 1 * [0.07 + 0.18] = 0.25$$

$\boxed{P(G) = 0.25}$ → Probability of wearing a green hoodie

(c) TT → Tank Top

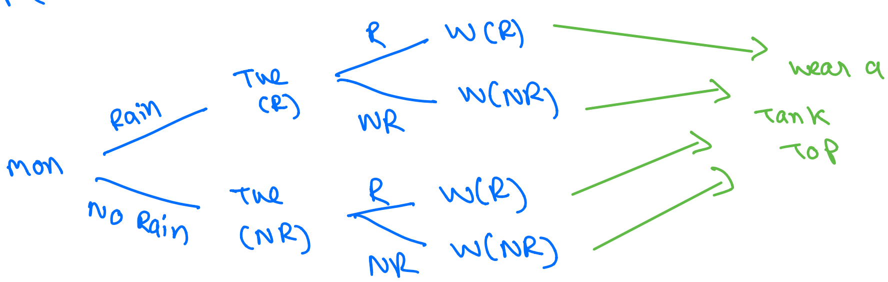
$P(TT|A) = 0.75$, $P(TT|\neg A) = 0.25$

D → monday

today previous day

$P(A|A) = 0.7$, $P(\neg A|A) = 0.3$

$P(A|\neg A) = 0.1$ $P(\neg A|\neg A) = 0.9$



$= (0.7 \times 0.7 * 0.75) +$

$(0.7 \times 0.3 \times 0.25) +$

$(0.3 \times 0.1 \times 0.75) +$

$(0.3 \times 0.9 \times 0.25)$

$= 0.3675 + 0.0525 + 0.0225 + 0.0675$

$= 0.51$

$$\boxed{P\left(\text{wearing Tank Top on wednesday} \,\middle|\, \text{monday was Raining}\right) = 0.51}$$

HW6 Q3

In [1]:

```
import numpy as np
import pickle
import matplotlib.pyplot as plt
import copy

from tabulate import tabulate
import pandas as pd
```

In [2]:

```
with open('alice_spelling.pkl','rb') as f:
    u = pickle._Unpickler(f)
    u.encoding = 'latin1'
    data = u.load()

#Take a look at how the data looks, and let's make some helper functions.
# data = pickle.load(open('alice_spelling.pkl','rb'))
vocab = np.unique(data['corpus'])
V = len(vocab)

## CORRECT VS INCORRECT CORPUS
##For now, we will hold onto both the correct and incorrect corpuses. Later, you will only process the
 incorrect corpus, and the correct corpus is only used as a reference to check for recovery accuracy.
def recovery_rate(new_corpus, correct_corpus):
    wrong = 0
    for k in range(len(new_corpus)):
        if new_corpus[k] != correct_corpus[k]:
            wrong += 1
    return 1.- wrong/(len(new_corpus)+0.)
print('current recovery rate', recovery_rate(data['corpus'],data['corrupted_corpus'] ))

## Probability of a word mispelling
## We will use the following function to predict whether a misspelled word was actually another word.
# To avoid numerical issues, we make sure that the probablity is always something nonzero.
def prob_correct(word1,word2):
    SMALLNUM = 0.000001
    if len(word1) != len(word2): return SMALLNUM
    num_wrong = np.sum(np.array([word1[k] == word2[k] for k in range(len(word1))]))
    return np.maximum(num_wrong / (len(word1)),SMALLNUM)

# print('prob not misspelling alice vs alace', prob_correct('alice','alice'))
print('prob not misspelling alice vs alace', prob_correct('alice','alace'))
print('prob not misspelling alice vs earth', prob_correct('alice','earth'))
print('prob not misspelling machinelearning vs machinedreaming', prob_correct('machinelearning','machin
edreaming'))
print('prob not misspelling machinelearning vs artificalintell', prob_correct('machinelearning','artifi
calintell'))


##HASHING
#all of our objects should be vectors of length V or matrices which are V x V.
#the kth word in the vocab list is represented by the kth element of the vector, and the relationship b
etween the i,jth words is represented in the i,jth element in the matrix.
# to easily go between the word indices and words themselves, we need to make a hash table.
vocab_hash = {}
for k in range(len(vocab)):
    vocab_hash[vocab[k]] = k

#now, to access the $k$th word, we do vocab[k]. To access the index of a word, we call vocab_hash[wor
d].
```

```
current recovery rate 0.7716434266712013
prob not misspelling alice vs alace 0.8
prob not misspelling alice vs earth 1e-06
prob not misspelling machinelearning vs machinedreaming 0.6666666666666666
prob not misspelling machinelearning vs artificalintell 1e-06
```

In [3]:

```
## FILL ME IN ##

#WORD FREQUENCY
#create an array of length V where V[k] returns the normalized frequency of word k in the entire data c
orpus.
# Do so by filling in this function.
def get_word_prob(corpus):
    wordList,countArray = np.unique(corpus, return_counts=True)
    totalWords = sum(countArray)
    word_prob = np.zeros(len(wordList))
    for i in range(len(wordList)):
        word_prob[i] = countArray[i] / totalWords

    return word_prob

word_prob =  get_word_prob(data['corpus'])

#report the answer of the following:
print('prob. of "alice"', word_prob[vocab_hash['alice']])
print('prob. of "queen"', word_prob[vocab_hash['queen']])
print('prob. of "chapter"', word_prob[vocab_hash['chapter']])

def getPrevWordAndCurrentWordDict():
    prevWordAndCurrentWordDict = {}
    prevWord = data['corpus'][0]
    for i in range(1,len(data['corpus'])):
        word = data['corpus'][i]
        if prevWord not in prevWordAndCurrentWordDict:
            prevWordAndCurrentWordDict[prevWord] = {}
            prevWordAndCurrentWordDict[prevWord][word] = 1
        elif word not in  prevWordAndCurrentWordDict[prevWord]:
            prevWordAndCurrentWordDict[prevWord][word] = 1
        elif word in  prevWordAndCurrentWordDict[prevWord]:
            prevWordAndCurrentWordDict[prevWord][word] += 1
        else:
            print("Shouldn't happen")
        prevWord = word
    return prevWordAndCurrentWordDict

## FILL ME IN ##

# Pr(word | prev word)
# Using the uncorrupted corpus, accumulate the conditional transition probabilities. Do so via this for
mula:
# pr(word | prev) = max(# times 'prev' preceded 'word' , 1) / # times prev appears
# where again, we ensure that this number is never 0 with some small smoothing.
def get_transition_matrix(corpus):

    transition_matrix = np.zeros((len(vocab),len(vocab)))
    wordList,countArray = np.unique(corpus, return_counts=True)
    prevWordAndCurrentWordDict = getPrevWordAndCurrentWordDict()
    for word in range(len(vocab)):
        wordString = wordList[word]
        for prevWord in range(len(vocab)):
            prevWordString = wordList[prevWord]
            prevWordPreceded = 0
            if wordString in prevWordAndCurrentWordDict[prevWordString]:
                prevWordPreceded = prevWordAndCurrentWordDict[prevWordString][wordString]
            occurences = max(prevWordPreceded, 1)
            transition_matrix[word][prevWord] = occurences / countArray[prevWord]
    return transition_matrix
transition_matrix = get_transition_matrix(data['corpus'])

print('prob. of "the alice"', transition_matrix[vocab_hash['alice'],vocab_hash['the']])
print('prob. of "the queen"', transition_matrix[vocab_hash['queen'],vocab_hash['the']])
print('prob. of "the chapter"', transition_matrix[vocab_hash['hatter'],vocab_hash['the']])
```

```
prob. of "alice" 0.014548615047424706
prob. of "queen" 0.002569625514869818
prob. of "chapter" 0.0009069266523069947
prob. of "the alice" 0.0006105006105006105
prob. of "the queen" 0.03968253968253968
prob. of "the chapter" 0.031135531135531136
```

In [4]:

```
#The prior probabilities are just the word frequencies
prior = word_prob

#write a function that returns the emission probability of a potentially misspelled word, by comparing
 its probabilities against every word in the correct vocabulary
def get_emission(mword):
    emission_prob = np.zeros(len(vocab))
    for index, word in enumerate(vocab):
        emission_prob[index] = prob_correct(mword,word)
    return emission_prob

#find the 10 closest words to 'abice' and report them
idx = np.argsort(get_emission('abice'))[::-1]
print([vocab[j] for j in idx[:10]])
```

```
['abide', 'alice', 'above', 'voice', 'alive', 'twice', 'thick', 'dance', 'stick', 'prize']
```

In [5]:

```
#now we reduce our attention to a small segment of the corrupted corpus
corrupt_corpus =   data['corrupted_corpus'][:1000]
correct_corpus =   data['corpus'][:1000]
```

In [6]:

```
def normalize(vector):
    return vector/np.sum(vector)
```

In [7]:

```
# encode the HMM spelling corrector.
# To debug, you can see the first hundred words of both the corrupted and proposed corpus,
# to see if spelling words got corrupted.
# report the recovery rate of the proposed (corrected) corpus.

totalStates = len(transition_matrix)
node_values_fwd = np.zeros((len(corrupt_corpus), totalStates))
for i, sequence_val in enumerate(correct_corpus):
    if (i == 0):
        word_prob =  get_word_prob(data['corpus'])
        start_probs =  word_prob[vocab_hash[sequence_val]]
        emission = get_emission(sequence_val)
        firstStateBeforeNormalisation = start_probs * emission
        node_values_fwd[i, :] = normalize(firstStateBeforeNormalisation)
    else:
        emission = get_emission(sequence_val)
        nextStateBeforeNormalization = np.multiply(emission,np.dot(transition_matrix ,node_values_fwd[i
-1, :]))
        node_values_fwd[i, :] = normalize(nextStateBeforeNormalization)

totalStates = len(transition_matrix)
node_values_bwd = np.zeros((len(corrupt_corpus), totalStates))
for i, e in reversed(list(enumerate(corrupt_corpus))):
    if (i == len(corrupt_corpus)-1):
        word_prob =  get_word_prob(data['corpus'])
        start_probs =  word_prob[vocab_hash[sequence_val]]
        emission = get_emission(sequence_val)
        firstStateBeforeNormalisation = start_probs * emission
        node_values_bwd[i, :] = normalize(firstStateBeforeNormalisation)
    else:
        emission = get_emission(sequence_val)
        nextStateBeforeNormalization = np.multiply(emission,np.dot(transition_matrix ,node_values_bwd[i
+1, :]))
        node_values_bwd[i, :] = normalize(nextStateBeforeNormalization)
```

In [8]:

```
forward_backward = np.multiply(node_values_fwd, node_values_bwd)
row_index = np.argmax(forward_backward, axis=1)
proposed_corpus = []
for index in row_index:
    proposed_corpus.append(vocab[index])

results = []
results.append(('correct_corpus','corrupt_corpus',recovery_rate(corrupt_corpus, correct_corpus)))
results.append(('correct_corpus','proposed_corpus',recovery_rate(proposed_corpus, correct_corpus)))
columns = ['corpus_A','corpus_B', 'recovery_rate']
df = pd.DataFrame(results, columns=columns)
print(tabulate(df, headers='keys', tablefmt='psql'))
```

```
+----+----------------+-----------------+-----------------+
|    | corpus_A       | corpus_B        |   recovery_rate |
|----+----------------+-----------------+-----------------|
|  0 | correct_corpus | corrupt_corpus  |           0.759 |
|  1 | correct_corpus | proposed_corpus |           0.804 |
+----+----------------+-----------------+-----------------+
```