STA314 HW3

1 Q1

1.1 a b

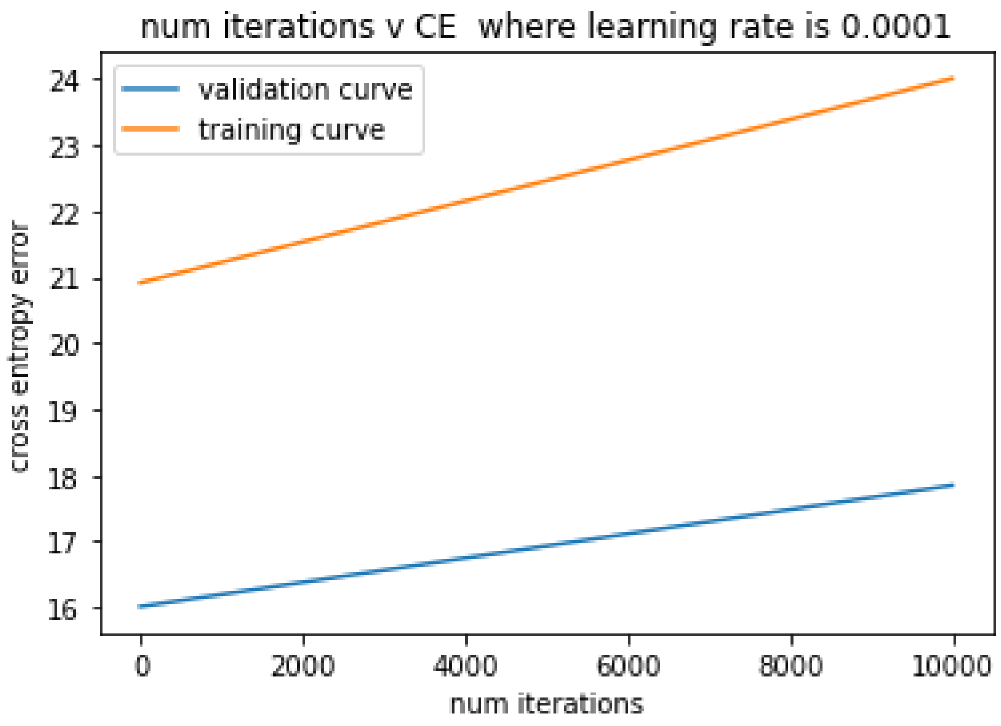
```
"""STA314 Homework 3.

Copyright and Usage Information
===============================

This file is provided solely for the personal and private use of students
taking STA314 at the University of Toronto St. George campus. All forms of
distribution of this code, whether as given or with any changes, are
expressly prohibited.
"""


from utils import *

import matplotlib.pyplot as plt
import numpy as np
# for the random starting matrix
np.random.seed(1)

def logistic_predict(weights, data):
    """ Compute the probabilities predicted by the logistic classifier.

    Note: N is the number of examples
          D is the number of features per example

    :param weights: A vector of weights with dimension (D + 1) x 1, where
    the last element corresponds to the bias (intercept).
    :param data: A matrix with dimension N x D, where each row corresponds to
    one data point.
    :return: A vector of probabilities with dimension N x 1, which is the output
    to the classifier.
    """


    n,d = np.shape(data)

    ones = np.expand_dims(np.ones(n), axis = 1)
    expanded_data = np.append(data, ones, axis = 1)
    z = expanded_data @ weights
    y = 1/(1 + np.exp(z))

    return y


def evaluate(targets, y):
    """ Compute evaluation metrics.

    Note: N is the number of examples
          D is the number of features per example

    :param targets: A vector of targets with dimension N x 1.
    :param y: A vector of probabilities with dimension N x 1.
    :return: A tuple (ce, frac_correct)
        WHERE
        ce: (float) Averaged cross entropy
        frac_correct: (float) Fraction of inputs classified correctly
    """
    t = targets
    y_non_zero = np.where(y == 0, 0.0000000000001, y)
    # this helps with weird log 0 errors
    first_term = -t.T @ np.log(y_non_zero)
    second_term = (1-t).T @ np.log(1-y_non_zero)
    ce = (first_term -second_term)/ np.size(y)


    pred = np.where(y > .5,  1,  0)
    correct = np.where(pred == t, 1 , 0)

    frac_correct = sum(correct)/np.size(correct)

    return ce, frac_correct


def logistic(weights, data, targets, hyperparameters):
    """ Calculate the cost of penalized logistic regression and its derivatives
    with respect to weights. Also return the predictions.

    Note: N is the number of examples
          D is the number of features per example

    :param weights: A vector of weights with dimension (D + 1) x 1, where
    the last element corresponds to the bias (intercept).
    :param data: A matrix with dimension N x D, where each row corresponds to
    one data point.
    :param targets: A vector of targets with dimension N x 1.
    :param hyperparameters: The hyperparameter dictionary.
    :returns: A tuple (f, df, y)
        WHERE
        f: The average of the loss over all data points, plus a penalty term.
           This is the objective that we want to minimize.
        df: (D+1) x 1 vector of derivative of f w.r.t. weights.
        y: N x 1 vector of probabilities.
    """
    s = np.shape(weights)[0]
    t = targets
    y = logistic_predict(weights, data)
    w = weights.T
    x = data
    n, d = np.shape(data)
    lambd = hyperparameters["weight_regularization"]
    ce = evaluate(targets, y)[0]
    regularizer = lambd *((np.square(np.linalg.norm(w[0 ,0:s - 1]))))/2
    f = ce + regularizer
    first_df = (y - t)/n
    ones = np.expand_dims(np.ones(n), axis = 1)
    expanded_data = np.append(data, ones, axis = 1)
    whole_first_df = expanded_data.T@first_df
    reg_df =  (np.expand_dims(lambd* w[0  , 0:s - 1], axis = 1))
    reg_df_exp = np.append(reg_df, np.array([[w[0, -1]]]), axis = 0)
    df = whole_first_df + reg_df_exp



    return f, df, y


def run_logistic_regression():


    n, d = x_train.shape


    weights = np.random.rand(d + 1,1)
# =============================================================================
    val_test_vector = []

    for _ in range(hyperparameters["num_iterations"]):
        f,df, y = logistic(
            weights, x_train, y_train, hyperparameters)
        update = ( hyperparameters["learning_rate"]/n) * df
        weights = np.subtract(weights, update)

    #val_test = evaluate(y_valid, logistic_predict(weights, x_valid))

    #val_test = evaluate(y_valid, logistic_predict(weights, x_valid))


    #test_value = evaluate(y_test, logistic_predict(weights, x_test))
    #train_value = evaluate(y_train, logistic_predict(weights, x_train))
    val_value = evaluate(y_valid, logistic_predict(weights, x_valid))


    return(val_value[0], val_value[1], weights, lr, num_iters)

if __name__ == "__main__":

    # Load all necessary datasets:
    x_train, y_train = load_train()
    # If you would like to use digits_train_small, please uncomment this line:
    #x_train, y_train = load_train_small()
    x_valid, y_valid = load_valid()
    x_test, y_test = load_test()

    validations = []
    for num_iters in [50, 100, 500,1000]:
        for lr in [.001, .01, .1, .5, 1]:
            hyperparameters = {
            "learning_rate": lr,
            "weight_regularization": 0.,
            "num_iterations": num_iters
            }
            x =  run_logistic_regression()
            validations.append(x)
    for i in validations:
        print(i[0], i[1])
    model_choice =  min(validations, key= lambda item:item[0])
    index = validations.index(model_choice)
    weights = validations[index][-3]
    train_value = evaluate(y_train, logistic_predict(weights, x_train))
    test_value = evaluate(y_test, logistic_predict(weights, x_test))
    val_value = [validations[index][0], validations[index][1]]
    learning_rate_choice = validations[index][-2]
    num_iters_choice = validations[index][-1]
    print(test_value, "test")
    print(train_value, "train")
    print(val_value, "validation")
    print(learning_rate_choice)
    print(num_iters_choice)
# ==========================================================================
#
# ==========================================================================
```

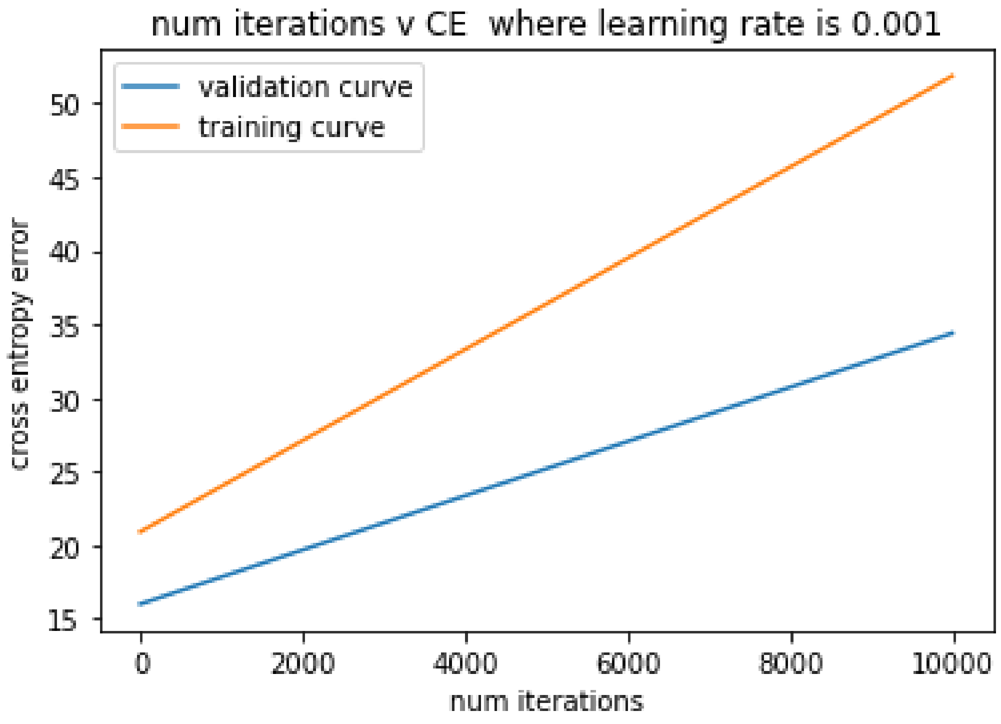
code output

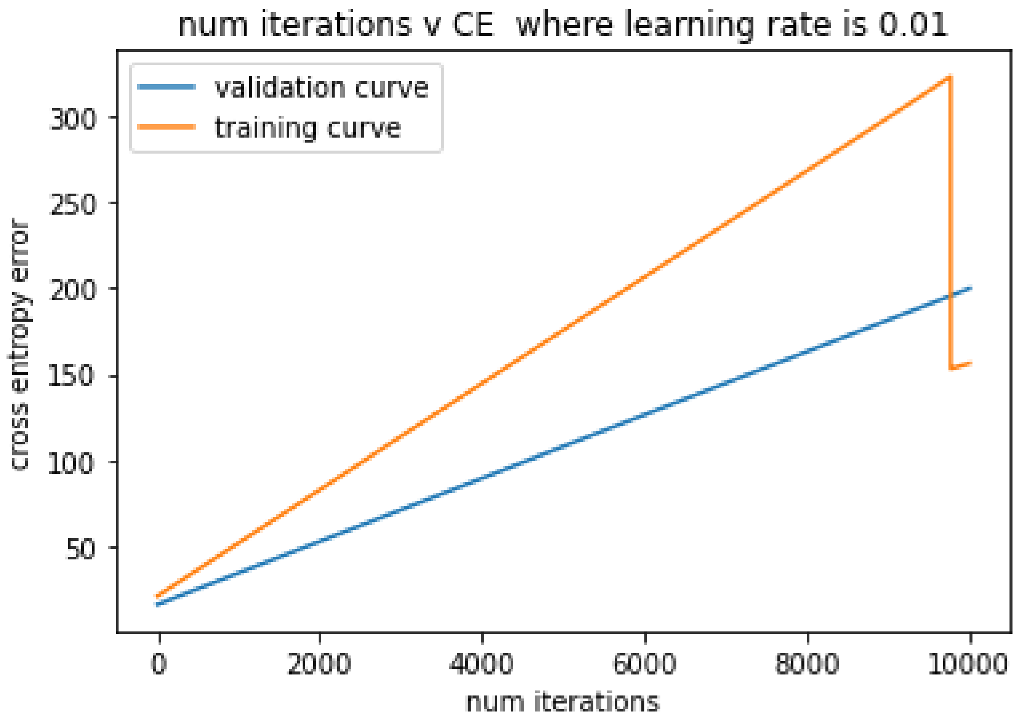
```
(array([[15.87179322]]), array([0.5])) test
(array([[15.65922308]]), array([0.5])) train
(array([[15.1610923]]), array([0.5])] validation)
0.01, learning rate
1000, number of iterations.
```

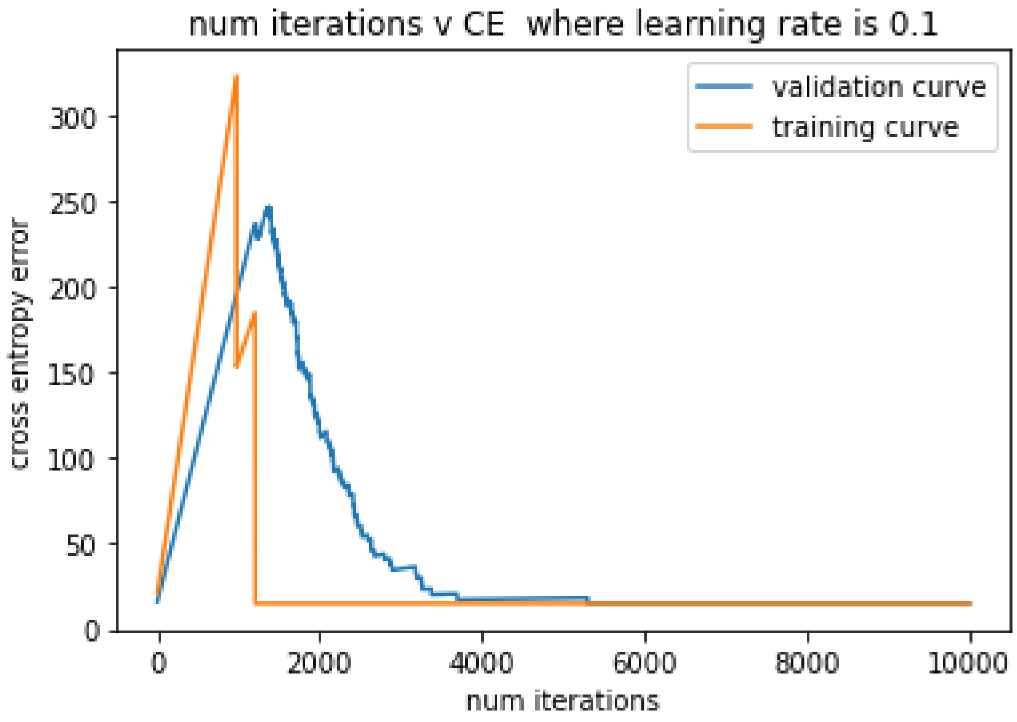
Therefore, we found that the learning rate of 0.01 with 1000 iterations gives us the ideal model.

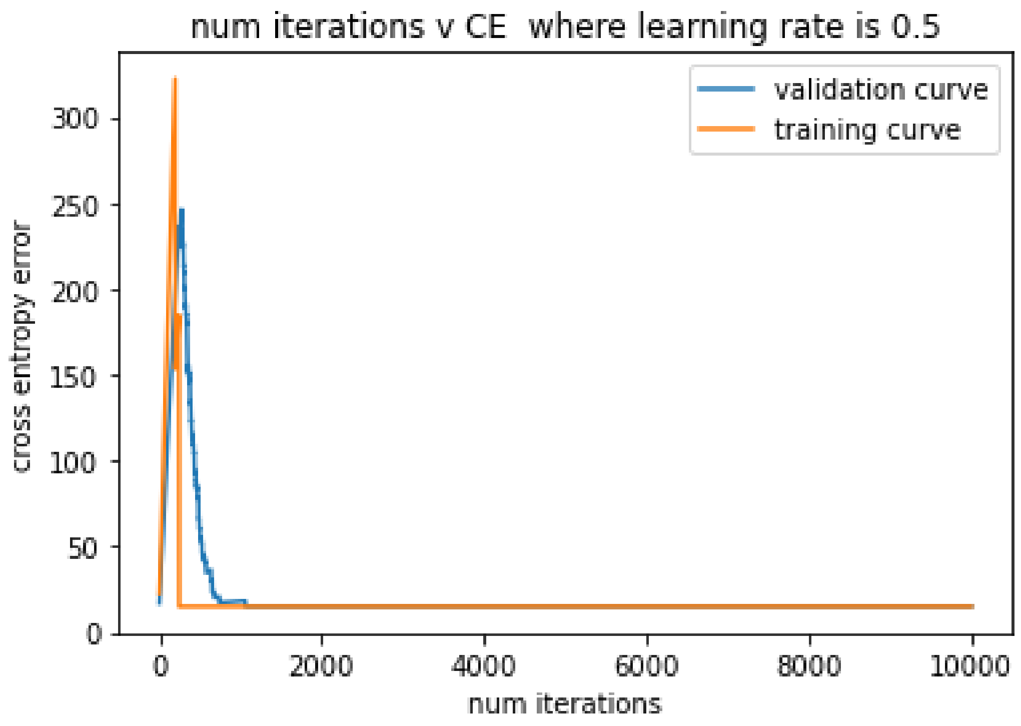
1.2 c

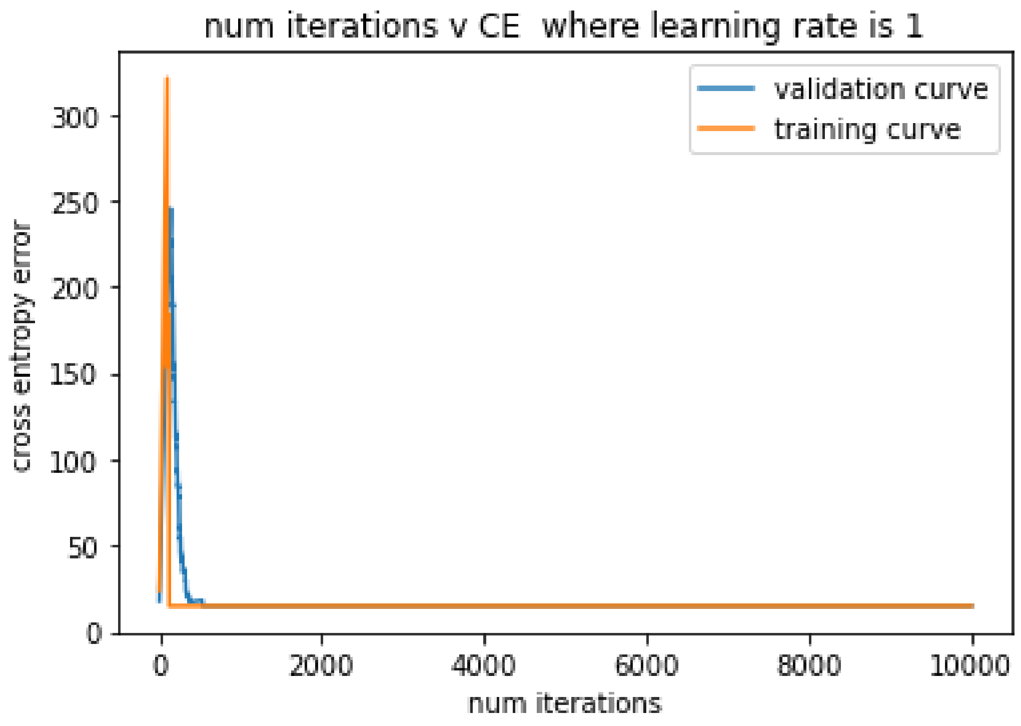

num iterations v CE where learning rate is 0.0001
validation curve
training curve
cross entropy error
num iterations
0
2000
4000
6000
8000
10000
16
17
18
19
20
21
22
23
24


num iterations v CE where learning rate is 0.001
validation curve
training curve
cross entropy error
num iterations
0
2000
4000
6000
8000
10000
15
20
25
30
35
40
45
50


num iterations v CE where learning rate is 0.01
validation curve
training curve
cross entropy error
50
100
150
200
250
300
0
2000
4000
6000
8000
10000
num iterations


num iterations v CE where learning rate is 0.1
validation curve
training curve
cross entropy error
0
50
100
150
200
250
300
0
2000
4000
6000
8000
10000
num iterations

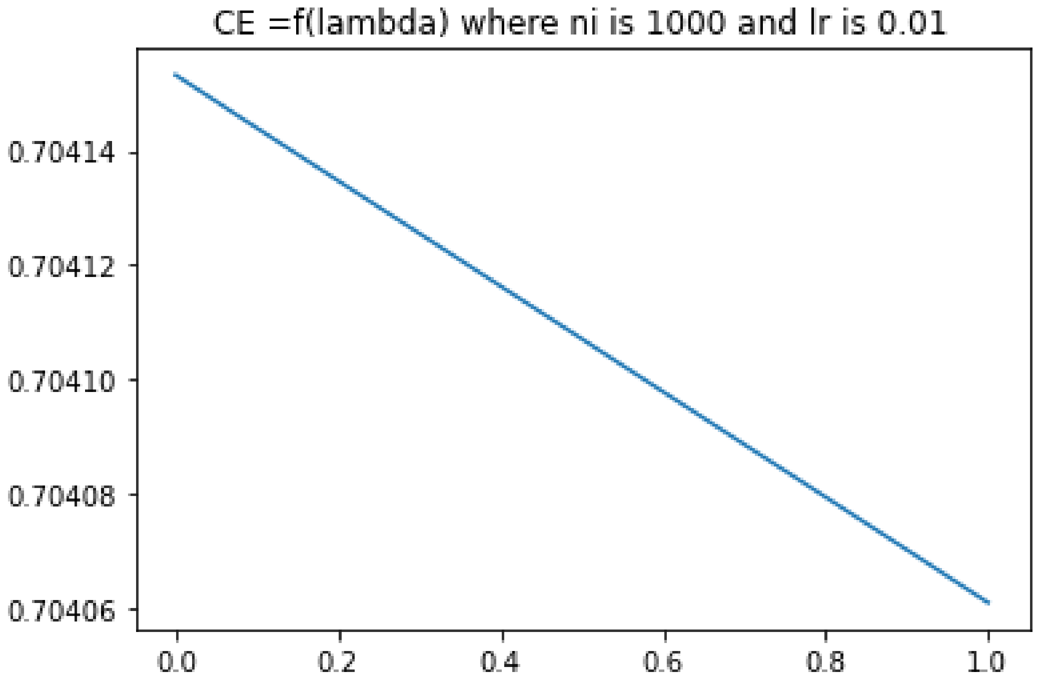


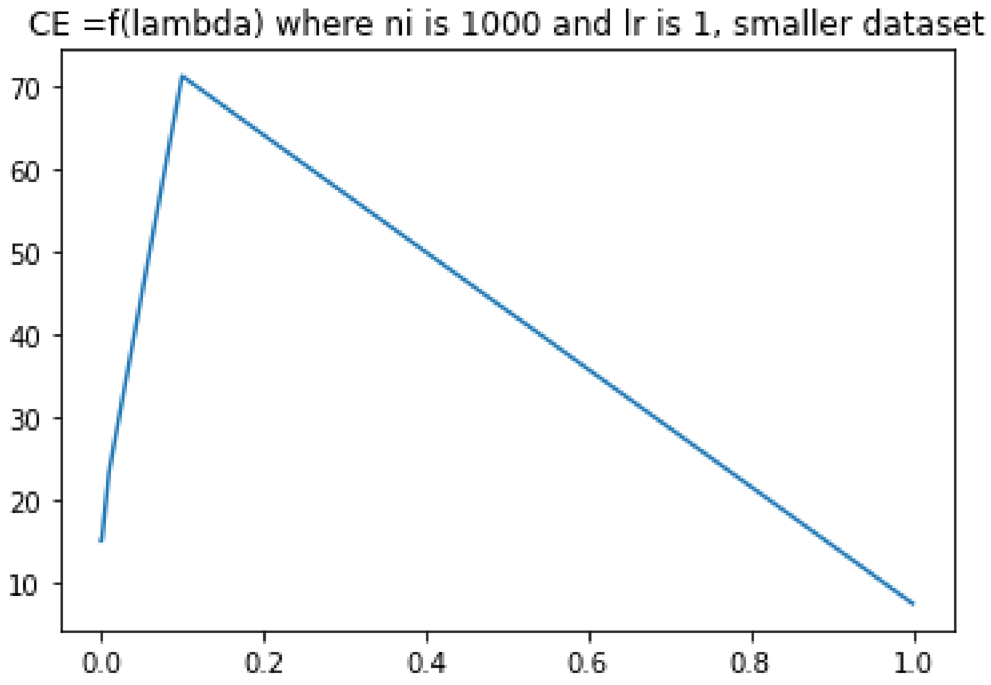


from the graphics provided I would take either 0.5 or 1 as my ideal learning rate as the cross entropy erro for the validation curve converges most quickly compared to the other values.

1.3 d

regular ds





1.4 e

From my understanding, I would expectde the error to got down and then go back up for increasing values of lambda, this would be because of under fitting and over fitting issues. However, this is not true of our graphs. One possibility is that this trend would be true for increasingly larger values of λ, and we are just not considering those values in our simulation. Alternatively, we have a mistake in our modeling, which could become apparent in this step. Further, we can notice that for the smaller data set, the trend is opposite of that which is expected. I would think that occurs, because our training dataset is quite small, implying that the training plane will be quite different from the underlying data, due to high variance from the small model. Therefore, this unexpected behaviour could be explained from the high variance from the small dataset, which results in strange behaviour.
In the end our test cross entropy error and our validation accuracy would be [[7.5523743]] [0.5] respectively

2 Q2

2.1 a

For our optimal center, since we are only dealing with a single center, we would want to place the center at the mean of the 3 vectors

$$(0,1) \tag{1}$$
$$(0,-1) \tag{2}$$
$$(4,0) \tag{3}$$

Which are the vectors $x^{(i)}$ where $i = 1, 2, 3$. Since all the vectors must be assigned to a center, we can find the global min of

$$(\sqrt{(0-x_1)^2 + (1-x_2)^2 + (0-x_1)^2 + (-1-x_2)^2(4-x_1)^2 + (0-x_2)^2})$$

. Now by plugging into wolfram alpha, we get a global min of about 2.19 and $m = (x_1, x_2) = (0.0576, -.82)$ per W.A. Now since, they all get the same center Therefore, for all $x^{(i)}$ $r^{(i)} = \begin{bmatrix}1\end{bmatrix}$.
final answer

x$^{(i)}$	m$^{(i)}$
1	1
2	1
3	1

2.2 b

note for simplicity in type scripting we are using the notation $t^i = t^{(i)}$ Let us begin by noticing that since we have 3 points and two centers, we essentially have 2 different options: the first option would be to assign two points to one center, and the third point to the second center in which the second center is equal to the third point. The second option would be to assign all three of these point to the same center and have an empty center, i.e which has no points assigned to it. Now let us notice that under the first option, we could assign the two point as either x^1, x^2 to center one, x^1, x^3 to center one, or x^2, x^3 to center one. Now let us notice that the minimum distance between x^1 or x^2 to x^3 would be formed by a center located along the line segment between these points, and we can notice that this would result in an error of $\sqrt{\frac{17}{2}}$ which was obtained by minimzing, the function, $\sqrt{(0-x)^2 + (1-y)^2 + (4-x)^2 + (0-y)^2)}$, per Wolfram alpha , which would mean that the center would be placed between these points. Further, since we are dealing with the k-means algorithim it would be placed along the mean of these lines, ande hence would be at

$$\begin{bmatrix} 2 \\ \pm .5 \end{bmatrix}$$

if x^3 was assigned to the same center as one of these point. Now let us notice that if we take the other option and assign x^1, x^2 to the same center, it would be the center a center located between on the line segement [-1, -1] . This would result in an error of $\sqrt{2}$. Let us notice that the reason for this is because no matter where I place the center along this line segment the sum of the distance from x^1 to the center + x^2 to the center would be 2. However, again we are dealing with the k-means algorithim, and hecnce we would be interested in only the origing. Now let us consider the other option of assigning all 3 of the point to the same center, and keeping one center empty, we can already notice, that if x^1 and x^2 are assigned to this cetner, which would be the same center, m_1 from the previous question, we would notice that the error for just these two terms, is greater than 2, and therefore greater than the error that we would have if we did not have a non empty center. Therefore, our optimal centers would be the following $m_1 = (0, 0)$, where $m_2 = (4, 0)$ and $r^1 = (1, 0)$, $r^2 = (1, 0)$ and $r^3 = (0, 1)$.
final answer

x$^{(i)}$	m$^{(i)}$
1	1
2	1
3	1

2.3 c

note we will continue using the notation outlined in the previous part Let us begin by considering the sum which we are trying to minimize. Let $m_1 = (y_1^1, y_2^1)$ and $m_2 = (y_2^1, y_2^2)$. We are trying to minimize, which we shall denote as $f(m_1, m_2)$

$$\sqrt{\frac{0-y_1^i}{4}^2 + (1-y_2^i)^2}$$
$$+\sqrt{\frac{0-y_1^i}{4}^2 + (1+y_2^i)^2} \text{ by factoring out the negative}$$
$$+\sqrt{\frac{4-y_1^i}{4}^2 + (0+y_2^i)^2}$$

Where y_k^i is the respective element in one of the centers. Also important to note that we abuse the notation, and y^i in different radicands are not necessarily referring to the same center. Now let us notice that similarly to the previous question, we mainly have two different options, either we could assign all 3 to the same center, with one of the centers being empty, or two to one of the centers, and the third to the second center. Now let us notice the following, the first component of the vectors, as a smaller weight then the second component, implying that when considering our three points, we would want to be closer to the two points x^1, x^2, as the second component has a much higher weight. Now Let us consider the first case in which all three are assigned to the same center, this would just be the same as finding the global minimum of $f(y_1^1, y_1^2,)$, which implies that all x^i belong to center m_1 and hence all y_i are the same. Therefore, we can notice by wolfram alpha that the global minimum of $f(y_1^1, y_2^1)$. From wolfram alpha, we are told that a global min does not exist for with these parameters, however, we are also told that there is no solution of $\sqrt(2) \geq f(y_1^1, y_2^1)$, which implies that this can not be the optimal assigment, per W.A. Now let us notice that we could consider the following possible assignments, $r^1 = (1,0) = r^2$, $r^1 = (1,0) = r^3$, , $r^2 = (1,0) = r^3$. Let us note that even under this new metric x^1, x^2 are equidistant to x^3 as they both have the same x_1 values, and hence we would only need to calculate either $r^1 = r^3$ or $r^2 = r^3$. Now by wolfram alpha, for $r^1 = r^3$, for this we get the minimum value of approximately 1.41421 per Wolfram alpha. Now if we assign $r^1 = r^2$, we end up with a minimum value of 2 as per Wolfram alpha. Now we can notice that assigning $r^1 = r^3$ or $r^2 = r^3$ we end up with the a smaller value. Note, the observation which is not part of the equality is has it's key equal to the observation, and therefore in the minimization calculations the radicand with the observation has a value of zero. Therefore, as we have determined that $r^1 = r^3$ and $r^2 = r^3$ is the ideal assigment, we can now determine the centers which allign with these assigment which give us the minimum. Now again per wolfram alpha, we get this minimum at $r^1 = r^3$ $m_1 = (y_1, y_2)$ where $y_1 = 4 - 4y_2$ per W.A as that is the solution to the minimization of

$$\sqrt{((x/4)^2 + (1-y)^2)} + \sqrt{(((4-x)/4)^2 + y^2))}$$

. Now if for $r^2 = r^3$, we again get the same minimum, but along a different line, along the line of $y_1 = 4 + 4_{y2}$ as per W.A. Therefore, our possible assigment would be as follows. $m_1 = (4 - 4y_2, y_2)$, $m_2 = x^2$, and $r^1 = r^3 = (1,0)$, $r^2 = (0,1)$. we can also have $m_1 = (4+4y_2, y_2)$, $m_2 = x^1$, where $r^1 = (0,1)$, $r^2 = r^3 = (1,0)$ final answer

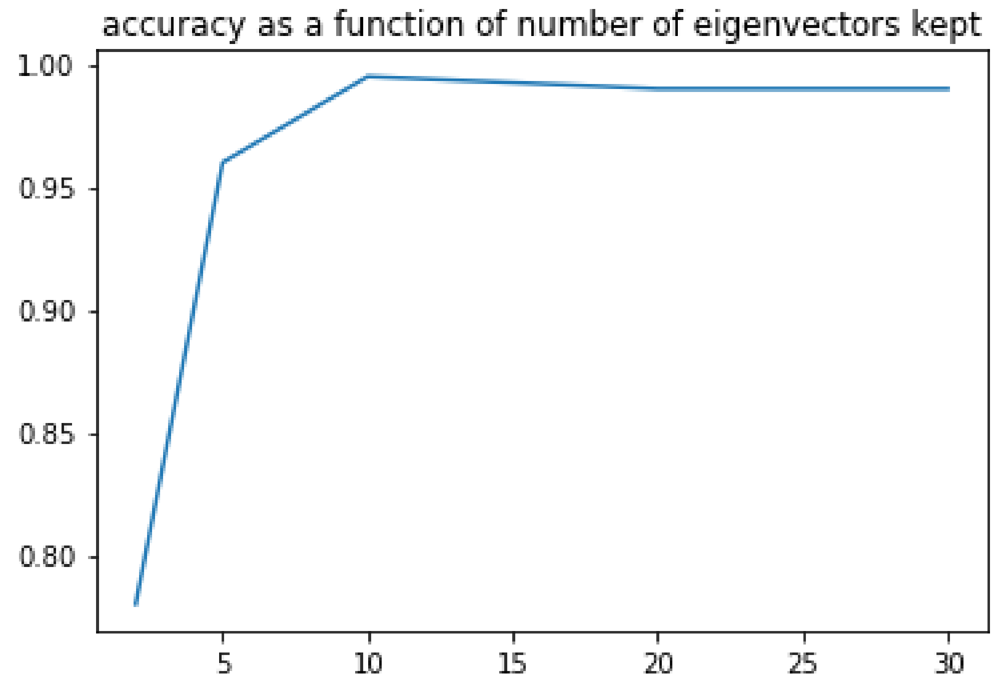
x$^{(i)}$	m$^{(i)}$
1	1
2	2
3	1
formation 2	
1	2
2	1
3	1

3 Q3

3.1 a b

```
"""STA314 Homework 3.

Copyright and Usage Information
===============================

This file is provided solely for the personal and private use of students
taking STA314 at the University of Toronto St. George campus. All forms of
distribution of this code, whether as given or with any changes, are
expressly prohibited.
"""


from utils import *

import matplotlib.pyplot as plt
import scipy.linalg as lin
import numpy as np


def pca(x, k):
    """ PCA algorithm. Given the data matrix x and k,
    return the eigenvectors, mean of x, and the projected data (code vectors).

    Hint: You may use NumPy or SciPy to compute the eigenvectors/eigenvalues.

    :param x: A matrix with dimension N x D, where each row corresponds to
    one data point.
    :param k: int
        Number of dimension to reduce to.
    :return: Tuple of (Numpy array, Numpy array, Numpy array)
        WHERE
        v: A matrix of dimension D x k that stores top k eigenvectors
        mean: A vector of dimension D x 1 that represents the mean of x.
        proj_x: A matrix of dimension k x N where x is projected down to k dimension.
    """
    n, d = x.shape

# ==========================================================================
#       mean
# ==========================================================================
    #xt = x.T

    mean = np.mean(x, axis = 0)
    centered_x = x - mean
    ########################################################################
    # TODO:                                                                #
    ########################################################################
# ==========================================================================
#       v
# ==========================================================================
   #v = None
    covmat =( (x - np.ones(d)@mean.T).T@(x-np.ones(d)@mean.T))/n
    eigvec = np.linalg.eig(covmat)[1]
    v = eigvec[:, 0:k]

# ==========================================================================
#      projection
# ==========================================================================
   # mean = None
    proj_x = v.T@(x - np.ones(d)@mean.T).T


    ##########################################################################
    #                           END OF YOUR CODE                             #
    ##########################################################################
    return v, mean, proj_x


def show_eigenvectors(v):
    """ Display the eigenvectors as images.
    :param v: NumPy array
        The eigenvectors
    :return: None
    """
    plt.figure(1)
    plt.clf()
    for i in range(v.shape[1]):
        plt.subplot(1, v.shape[1], i + 1)
        plt.imshow(v[:, v.shape[1] - i - 1].reshape(16, 16).T, cmap=plt.cm.gray)
    plt.show()


def pca_classify():
    # Load all necessary datasets:
    x_train, y_train = load_train()
    x_valid, y_valid = load_valid()
    x_test, y_test = load_test()

    # Make sure the PCA algorithm is correctly implemented.
    v, mean, proj_x = pca(x_train, 5)
    # The below code visualize the eigenvectors.
    show_eigenvectors(v)

    ##########################################################################
    # TODO:                                                                  #
    ##########################################################################
    k_lst = [2, 5, 10, 20, 30]
    n, d = np.shape(x_train)
    val_acc = np.zeros(len(k_lst))
    for j, k in enumerate(k_lst):
        v,mean,proj_x = pca(x_train, k)
        show_eigenvectors(v)
        x_tilde = (v @ proj_x + np.ones(d) @ mean.T)
```

```
        values = []
        options = []
        for i in range(x_valid.shape[0]):
            x_valid_i = np.expand_dims(x_valid[i], axis = 1)
            to_min = np.linalg.norm(x_tilde.T - np.expand_dims(
                np.ones(n), axis = 1)@x_valid_i.T, axis = 1)
            opt = np.argmin(to_min)
            options.append(opt)
            y_value = y_train[opt]
            if y_value == y_valid[i]:
                values.append(1)
            else:
                values.append(0)
            #to_append = np.linalg.norm(x_tilde.T[opt] - np.ones(d)*y_valid[i])


            # For each validation sample, perform 1-NN classifier on
            # the training code vector.
        #    pass
        val_acc[j] = sum(values)/len(values)
    ##############################################################################
    #                              END OF YOUR CODE                              #
    ##############################################################################
    plt.plot(k_lst, val_acc)
    plt.title("accuracy as a function of number of eigenvectors kept")
    plt.show()
    error = []
    v,mean,proj_x = pca(x_train, 5)
    x_tilde = (v @ proj_x + np.ones(d) @ mean.T)
    for r in range(x_test.shape[0]):
         x_test_r = np.expand_dims(x_test[r], axis = 1)
         x_test_2 = np.expand_dims(np.ones(n), axis = 1)@x_test_r.T
         to_min = np.linalg.norm(x_tilde.T - x_test_2, axis = 1)
         opt = np.argmin(to_min)
         y_value = y_train[opt]
         if y_value == y_test[r]:
             error.append(1)
         else:
             error.append(0)
         #options.append(opt)
         #to_append = np.linalg.norm(x_tilde.T[opt] - np.ones(d)*y_test[i])

    print(sum(error)/ len(error))


if __name__ == "__main__":
    pca_classify()
```



From this graph we can see that the accruacy for the validation set is maximized at k = 10.

3.2 c

From the graph above, one would assume that they should ideally take k = 10, because this would likely be the most accurate, however, k= 5, also has very high accuracy, and is half the dimensions of k = 10. Therefore, in a situation, where one must run this algorithms many times,i.e, i.e they have a very large dataset, choosing k = 5 might be more valuable. As a result, I would say k =5 is the ideal choice for a trade off between accuracy and speed.

3.3 d

our test accuracy when k = 10, is 0.9825, and as a side note, when k =5, the test accuracy is .94, This seems to perform much better than logistic regression, however, I am skeptical of my results from logistic regression out of fear that they are incorrect. Further, I am also concerned about my results from pca, as the accuracy is quite high which makes me hesitant to think that it is correct.