Connor Roberts - 805626088, Michael Kleinman - 105035249, Johannes Lee - 904733616

Section #1: Centralized Algorithms

Section #1.1: LINEAR REGRESSION

Please follow our instructions in the same order to solve the linear regresssion problem.

Please print out the entire results and codes when completed.

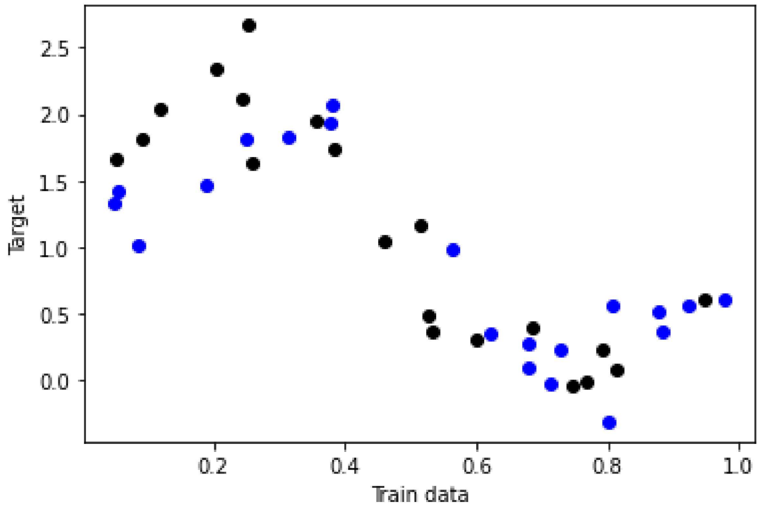
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
import csv
from data_load import load
import scipy.io as io
# Load matplotlib images inline
%matplotlib inline
# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
%reload_ext autoreload
```

In [88]:
```
def get_data():
    """
    Load the dataset from disk and perform preprocessing to prepare it for the linear r
    """
    X_train, y_train = load('regression_train.csv')
    X_val, y_val = load('regression_val.csv')
    X_test, y_test = load('regression_test.csv')
    return X_train, y_train, X_val, y_val, X_test, y_test

X_train, y_train, X_val, y_val, X_test, y_test= get_data()


print('Train data shape: ', X_train.shape)
print('Train target shape: ', y_train.shape)
print('Validation data shape: ',X_val.shape)
print('Validation target shape: ',y_val.shape)
print('Test data shape: ',X_test.shape)
print('Test target shape: ',y_test.shape)
```

```
Train data shape:  (20, 1)
Train target shape:  (20,)
Validation data shape:  (20, 1)
Validation target shape:  (20,)
Test data shape:  (20, 1)
Test target shape:  (20,)
```

In [3]:
```
## Plot the training and test data ##

plt.plot(X_train, y_train,'o', color='black')
plt.plot(X_test, y_test,'o', color='blue')
```

```
plt.xlabel('Train data')
plt.ylabel('Target')
plt.show()
```



a) The visualized data shown above appears to have a sinusoidal pattern which would make linear regression not very effective at seperating the data.

Training Linear Regression

In the following cells, you will build a linear regression. You will implement its loss function, then subsequently train it with gradient descent. You will choose the learning rate of gradient descent to optimize its classification performance. Finally, you will get the opimal solution using closed form expression.

In [4]:

```
from Regression import Regression
```

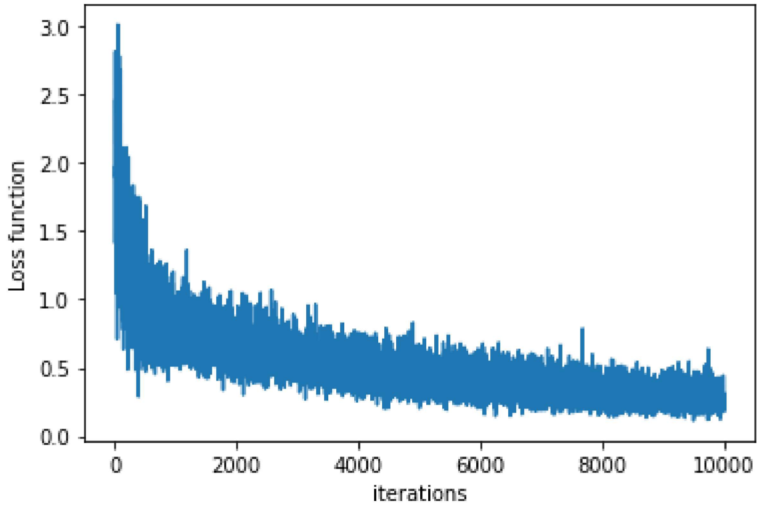
In [83]:

```
## Complete loss_and_grad function in Regression.py file and test your results.
regression = Regression(m=1, reg_param=0)
loss, grad = regression.loss_and_grad(X_train,y_train)
print('Loss for m = 1:', loss)
print('Gradient for m = 1:',grad)
##
```

```
Loss for m = 1: [[2.01169237]]
Gradient for m = 1: [-2.2602119  -0.67366233]
```

In [84]:

```
## Complete train_LR function in Regression.py file
loss_history, w = regression.train_LR(X_train,y_train, eta=1e-3,batch_size=20, num_iter
plt.plot(loss_history)
plt.xlabel('iterations')
plt.ylabel('Loss function')
plt.show()
print('Weight:', w)
print('Loss function final value:',loss_history[9999])
```




```
Weight: [[ 1.91478942]
 [-1.74618367]]
Loss function final value: [0.22708448]
```

e) The best values after testing appear to be eta=1e-2,batch_size=10, num_iters=10000 which consistently produced loss values of approximately 0.10

In [21]:

```
## Complete closed_form function in Regression.py file
loss_2, w_2 = regression.closed_form(X_train, y_train)
print('Loss for closed form:', loss_2)
print('Gradient for closed form:', w_2)
```

```
Loss: 0.1956288202895732
Gradient: [ 2.44640709 -2.81635359]
```

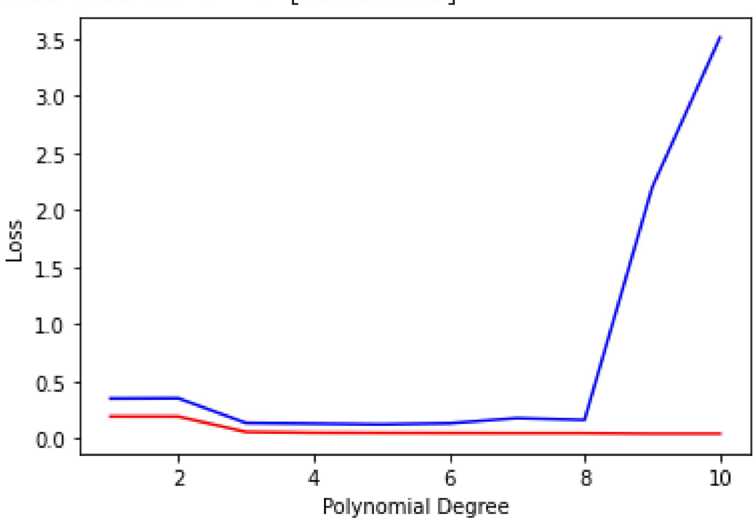
f) We can compare the closed form loss to the gradient descent loss and see that the closed form performs better than the gradient

In [96]:

```
train_loss=np.zeros((10,1))
test_loss=np.zeros((10,1))
# ================================================================ #
# YOUR CODE HERE:
# complete the following code to plot both the training and test loss in the same plot
# for m range from 1 to 10
# ================================================================ #


#N,d = X.shape


#print(loss_2)
#print(w_2)
#print(y_train.shape)






for m in range(0,10):
    regression = Regression(m=m+1, reg_param=0)
    #print(m)
    if m+1 == 1:
        X = X_train
```

```
        X_t = X_test
    else:
        X = regression.gen_poly_features(X_train)
        X_t = regression.gen_poly_features(X_test)
    trn_loss, w = regression.closed_form(X, y_train)
    train_loss[m] = trn_loss
    #print(w)
    #print(X)
    y_pred = regression.predict(X_t)
    tst_loss = 0.0
    temp = []
    for i in range(0,len(y_pred)):
        #print(y_pred[i])
        #print(y_train[i])
        temp = (y_pred[i] - y_test[i])*(y_pred[i] - y_test[i])
        #print(temp)
        tst_loss = tst_loss + temp
    test_loss[m] = tst_loss/(len(y_pred))
#print(test_loss)
#print(train_loss)

## Plot the training and test loss ##

print('Test loss for m = 3', test_loss[2])
plt.plot(np.arange(1,11), train_loss, color='red')
plt.plot(np.arange(1,11), test_loss, color='blue')
plt.xlabel('Polynomial Degree')
plt.ylabel('Loss')
plt.show()
best_m_ind = int(np.where(test_loss == np.amin(test_loss))[0])
print('Best m:', best_m_ind+1)
print('Test loss for best m', test_loss[best_m_ind])
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

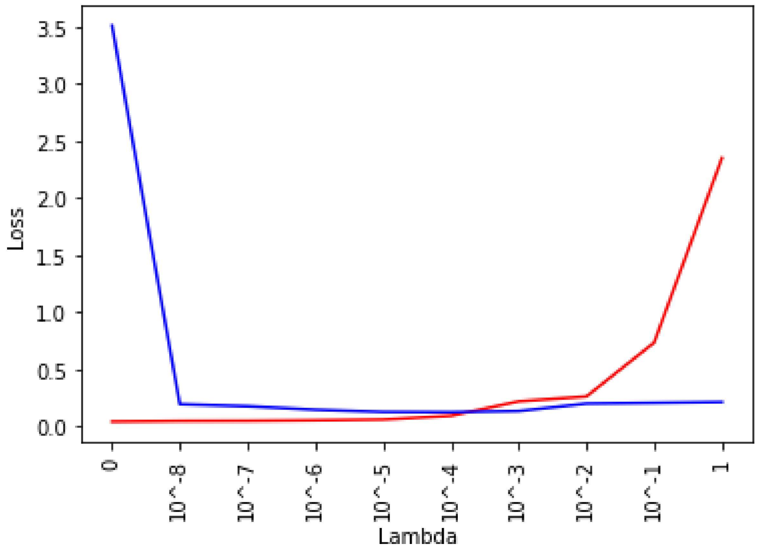
```
Test loss for m = 3 [0.13834158]
```




```
Best m: 5
Test loss for best m [0.12612282]
```

We can see in the plot above that as m approaches 8 - 10, we start to see large test loss which signifies we are overfitting the training data. Based on analysis of the lowest test loss, we see that m = 5 gives us the best performance.

In [32]:
```
train_loss=np.zeros((10,1))
test_loss=np.zeros((10,1))
# ================================================================ #
# YOUR CODE HERE:
# complete the following code to plot both the training and test loss in the same plot
# for lambda from set of values given.
# ================================================================ #
reg_terms = [0, (1/10) ** 8, (1/10) ** 7, (1/10) ** 6, (1/10) ** 5, (1/10) ** 4, (1/10)
j = 0
for r in reg_terms:

    regression = Regression(m=10, reg_param=r)
    X = regression.gen_poly_features(X_train)
    X_t = regression.gen_poly_features(X_test)
    trn_loss, w = regression.closed_form(X, y_train)
    train_loss[j] = trn_loss
    #print(w)
    #print(X)
    y_pred = regression.predict(X_t)
    tst_loss = 0.0
    temp = []
    for i in range(0,len(y_pred)):
        #print(y_pred[i])
        #print(y_train[i])
        temp = (y_pred[i] - y_test[i])*(y_pred[i] - y_test[i])
        #print(temp)
        tst_loss = tst_loss + temp
    test_loss[j] = tst_loss/(len(y_pred))
    j = j + 1
reg_term_labels = ['0', '10^-8', '10^-7', '10^-6', '10^-5', '10^-4', '10^-3', '10^-2',
line1 = plt.plot(np.arange(1,11), train_loss, color='red', )
line2 = plt.plot(np.arange(1,11), test_loss, color='blue')

plt.xticks(np.arange(1,11),reg_term_labels, rotation='vertical')
plt.xlabel('Lambda')
plt.ylabel('Loss')

plt.show()
best_lambda_ind = int(np.where(test_loss == np.amin(test_loss))[0])
print('Best Lambda:', reg_term_labels[best_lambda_ind])
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```




```
Best Lambda: 10^-4
```

Notebook_Binary_Classification

February 20, 2021

1 Section #1: Centralized Algorithms

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
import csv
from data_load import load
import scipy.io as io
# Load matplotlib images inline
%matplotlib inline
# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

1.1 Section #1.2: Binary Classification (Johannes Lee)

Please follow our instructions in the same order to solve the binary classification problem.

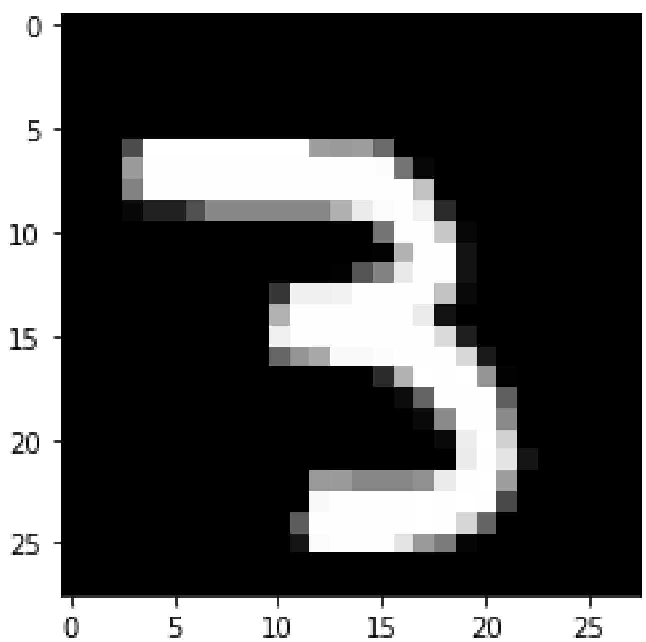
Please print out the entire results and codes when completed.

[2]:
```
"""
Load the dataset from disk and perform preprocessing to prepare it for the␣
 ↪linear regression problem.
"""
train = io.loadmat('mnist_train')
test = io.loadmat('mnist_test')
X_train = train['X_train']
y_train = np.transpose(train['y_train'])
X_test = test['X_test']
y_test = np.transpose(test['y_test'])

print('Train data shape: ', X_train.shape)
print('Train target shape: ', y_train.shape)
print('Test data shape: ',X_test.shape)
print('Test target shape: ',y_test.shape)
```

```
Train data shape:  (5000, 784)
Train target shape:  (5000, 1)
Test data shape:  (500, 784)
Test target shape:  (500, 1)
```

[3]:

```
# To Visualize a point in the dataset
index = 4000
X = np.array(X_train[index], dtype='uint8')
X = X.reshape((28, 28))
fig = plt.figure()
plt.imshow(X, cmap='gray')
plt.show()
fig.savefig('Sample.pdf')
if y_train[index] == 1:
    label = 3
else:
    label = 2
print('label is', label)
```




```
label is 3
```

(a) The dimensions of X_train and X_test are 5000 x 784 and 500 x 784, respectively.

1.2 Train Perceptron

In the following cells, you will build Perceptron Algorithm.

[19]:

```
N = X_train.shape[0] # Number of data point
d = X_train.shape[1] # Number of features
loss_hist = []
W = np.zeros((d,1))
# ================================================================ #
# YOUR CODE HERE:
# Implement the perceptron Algorithm and compute the number of misclassified␣
 ↪points at each training step
# ================================================================ #
max_iter = N
num_updates = 0
for ii in range(max_iter):
    loss = 0
    for idx in range(N):
        a = np.dot(W[:, 0], X_train[idx, :])
        if np.sign(a) != y_train[idx]:
            W[:, 0] += y_train[idx, 0]*X_train[idx]
            loss += 1
            num_updates += 1
    loss_hist.append(loss)
    if loss == 0:
        print('Terminated after {} iterations.'.format(ii))
        break
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

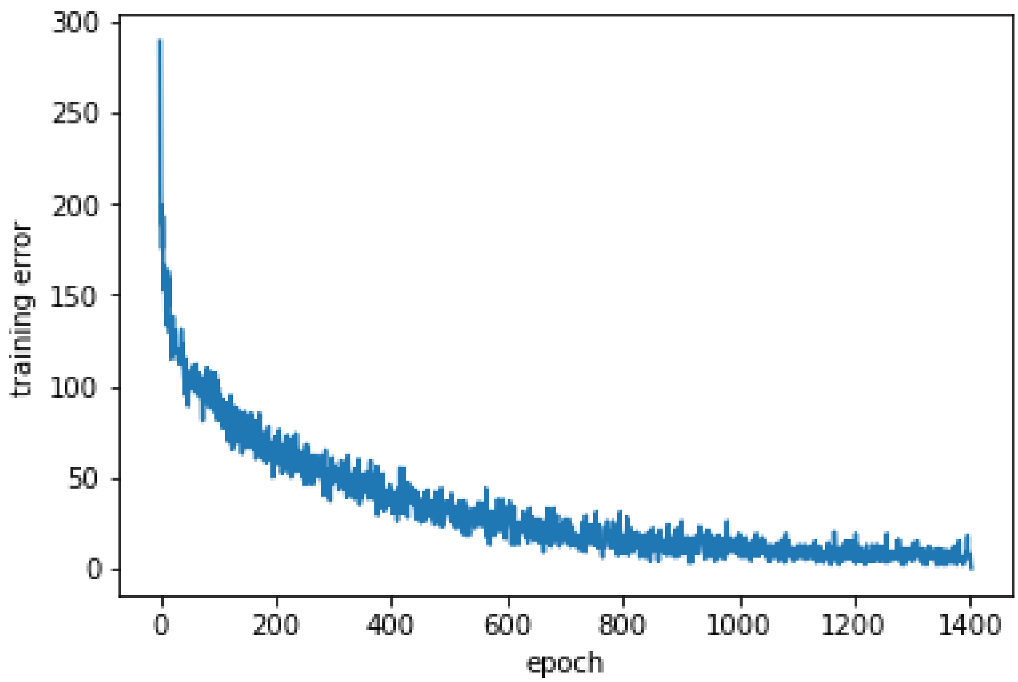
```
Terminated after 1402 iterations.
```

[20]:

```
plt.plot(loss_hist)
plt.xlabel('epoch')
plt.ylabel('training error')
```

[20]:

```
Text(0, 0.5, 'training error')
```



[22]:
```
print('Final accuracy after {} iterations: {}'.format(max_iter, 1-loss_hist[-1]/
 ↪N))
```

```
Final accuracy after 5000 iterations: 1.0
```

[23]:
```
print('Squared 2-norm of w = {}'.format(np.linalg.norm(W)**2))
```

```
Squared 2-norm of w = 177865738033.0
```

(b) The final loss is 0, with a squared 2-norm of w ~= 1.778e11. Since the perceptron algorithm reaches 0 loss, it converges and the data is linearly separable.

[25]:
```
# Compute the percentage of misclassified points in the test data for perceptron
y_hat = np.sign(X_test@W).astype('int')
acc = np.mean(y_hat == y_test)
print('Acc = {}'.format(acc))
```

```
Acc = 0.966
```

(c) 3.4% of the test data are misclassified with the trained perceptron.

[ ]:
```

```

1.3 Train Logistic Regression

In the following cells, you will build a logistic regression. You will implement its loss function, then subsequently train it with gradient descent.

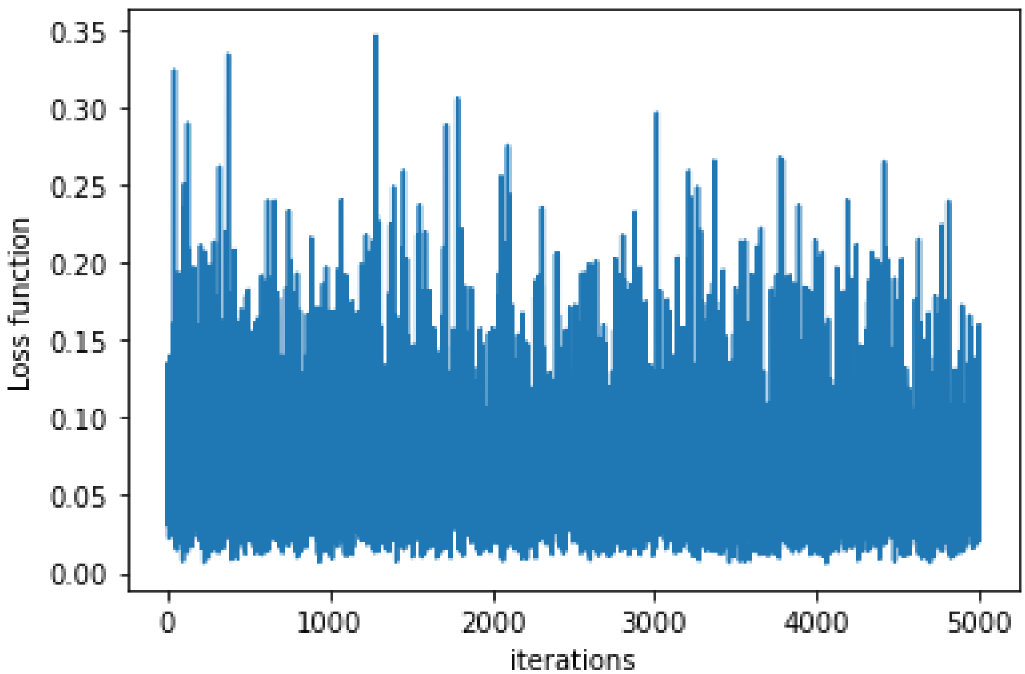
[3]:
```
from Logistic import Logistic
```

[4]:
```
# Complete loss_and_grad function in Logistic.py file and test your results.
N,d = X_train.shape
logistic = Logistic(d=d, reg_param=0)
loss, grad = logistic.loss_and_grad(X_train,y_train)
print('Loss function=',loss)
print(np.linalg.norm(grad,ord=2)**2)
```

```
Loss function= 0.6931471805599454
78885.26903007003
```

(f) The loss is 0.693, with a gradient whose norm is 78885.

[38]:
```
# Complete train_LR function in Logisitc.py file
loss_history, w = logistic.train_LR(X_train,y_train, eta=1e-6,batch_size=50,␣
 ↪num_iters=5000)

fig = plt.figure()
plt.plot(loss_history)
plt.xlabel('iterations')
plt.ylabel('Loss function')
plt.show()
fig.savefig('LR_loss_hist.pdf')
print('squared 2-norm of w:', np.linalg.norm(w,ord=2)**2)
print('loss:', loss_history[4999])
```

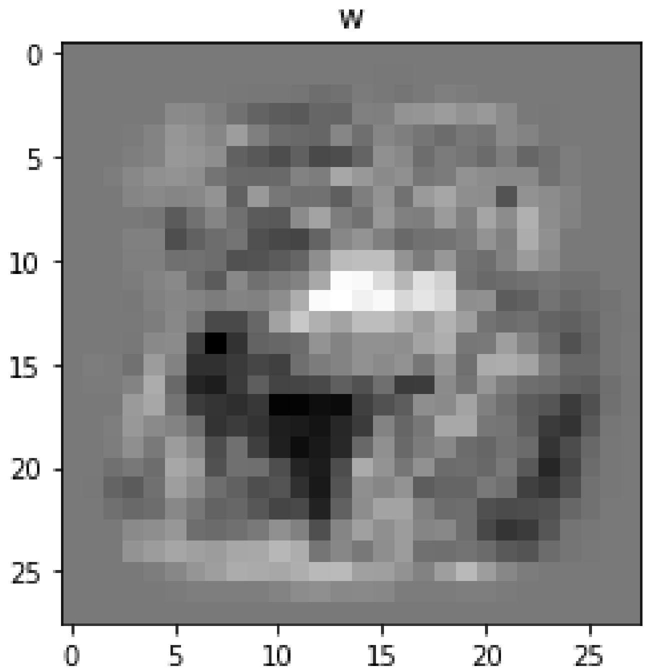



```
squared 2-norm of w: 0.00039567594085311586
loss: 0.020874384528072554
```

(g) The final loss is 0.02, with a squared 2-norm of 0.000396.

[39]:
```
plt.imshow(w[1:].reshape((28, 28)), cmap='gray')
plt.title('w')
```

[39]:
```
Text(0.5, 1.0, 'w')
```



[40]:
```
# final training accuracy
np.mean(logistic.predict(X_train) == (y_train > 0))
```

[40]: 0.9796

[41]:
```
# Complete predict function in Logisitc.py file and compute the percentage of␣
↪misclassified points in the test data
test_acc = np.mean(logistic.predict(X_test) == (y_test > 0))
print('Test error: {:.4}'.format(1-test_acc))
```

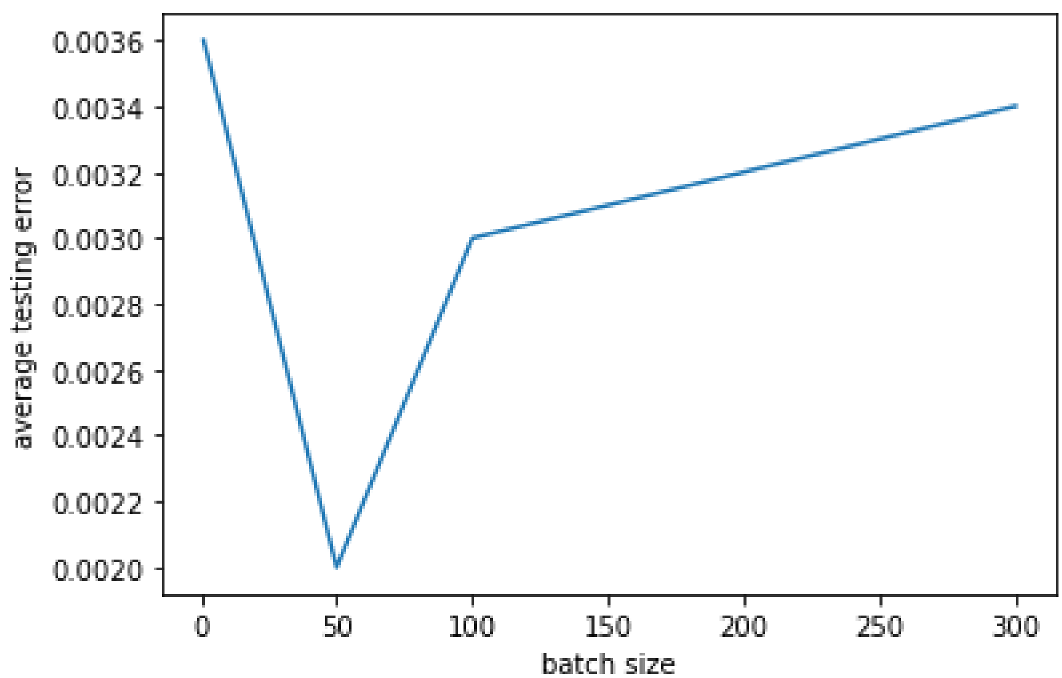
```
Test error: 0.022
```

(h) 2.2% of test data are misclassified using logistic regression.

[48]:
```
Batch = [1, 50 , 100, 300]
test_err = np.zeros((len(Batch),1))
# ================================================================ #
# YOUR CODE HERE:
# Train the Logistic regression for different batch size Avergae the test error␣
↪over 10 times
# ================================================================ #
for t in range (0,1):
    for m in range(0,len(Batch)):
        batch_size = Batch[m]
        eta = 1e-5
```

```
        num_iters = 8000 // batch_size
        logistic = Logistic(d=d, reg_param=0)
        loss_history, w = logistic.train_LR(X_train,y_train,␣
↪eta=eta,batch_size=batch_size, num_iters=num_iters)

        acc = np.mean(logistic.predict(X_test) == (y_test > 0))
        test_err[m, 0] += 0.1*(1-acc)

plt.plot(Batch, test_err[:, 0])
plt.xlabel('batch size')
plt.ylabel('average testing error')
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

[48]: Text(0, 0.5, 'average testing error')



[36]:
```
for i in range(4):
    print('Batch size: ', Batch[i], 'Error: ', test_err[i])
```

```
Batch size:  1 Error:  [0.0036]
Batch size:  50 Error:  [0.002]
Batch size:  100 Error:  [0.003]
Batch size:  300 Error:  [0.0034]
```

(i) Testing error is minimized for a batch size of 50.

[ ]:

```

```

1.4 Train SVM

In the following cells, you will build SVM. You will implement its loss function, then subsequently train it with mini-batch gradient descent. You will choose the learning rate of gradient descent to optimize its classification performance. Finally, you will get the best regularization parameter.

[44]:

```python
from SVM import SVM
```

[45]:

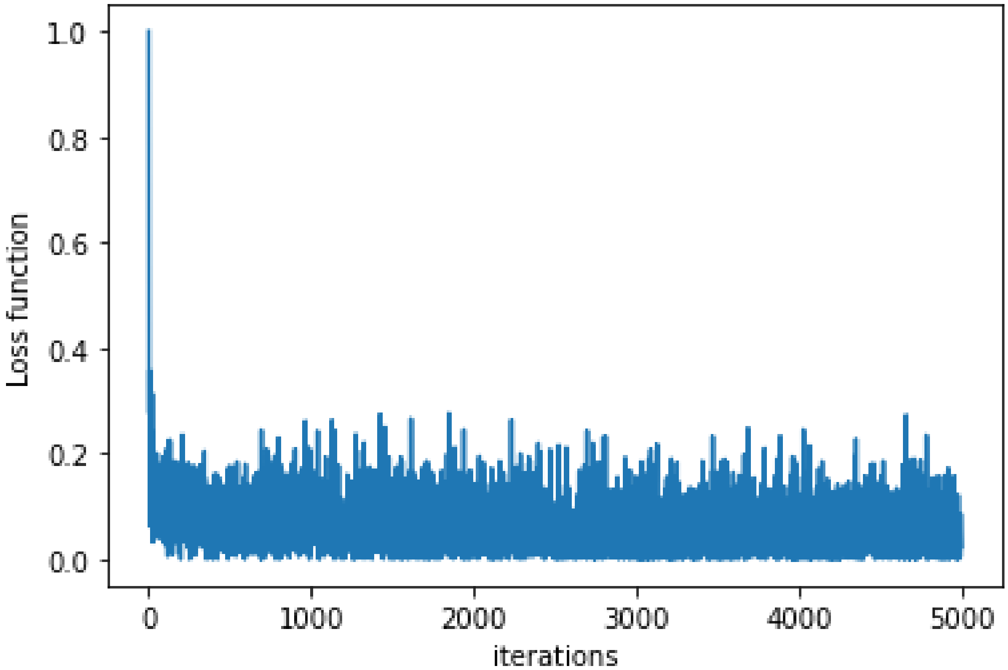
```python
# Complete loss_and_grad function in SVM.py file and test your results.
N,d = X_train.shape
svm = SVM(d=d, reg_param=0)
loss, grad = svm.loss_and_grad(X_train,y_train)
print('Loss function=',loss)
print(np.linalg.norm(grad,ord=2)**2)
```

```
Loss function= 1.0
315541.07612028066
```

(l) The loss is 1 exactly, with a gradient whose squared 2-norm is 315541.

[46]:

```python
# Complete train_svm function in SVM.py file
loss_history, w = svm.train_svm(X_train,y_train, eta=1e-6,batch_size=50,␣
↪num_iters=5000)
fig = plt.figure()
plt.plot(loss_history)
plt.xlabel('iterations')
plt.ylabel('Loss function')
plt.show()
fig.savefig('svm_loss_hist.pdf')
print('squared 2-norm of w:', np.linalg.norm(w,ord=2)**2)
print('loss:', loss_history[4999])
```




```
squared 2-norm of w: 0.0002015896931983999
loss: 0.02292531640000014
```

(m) The final loss is 0.0229, with a squared 2-norm of weight of 0.00020.

[47]:
```
test_acc = np.mean(svm.predict(X_test) == (y_test > 0))
print('Test error: {:.4}'.format(1-test_acc))
```

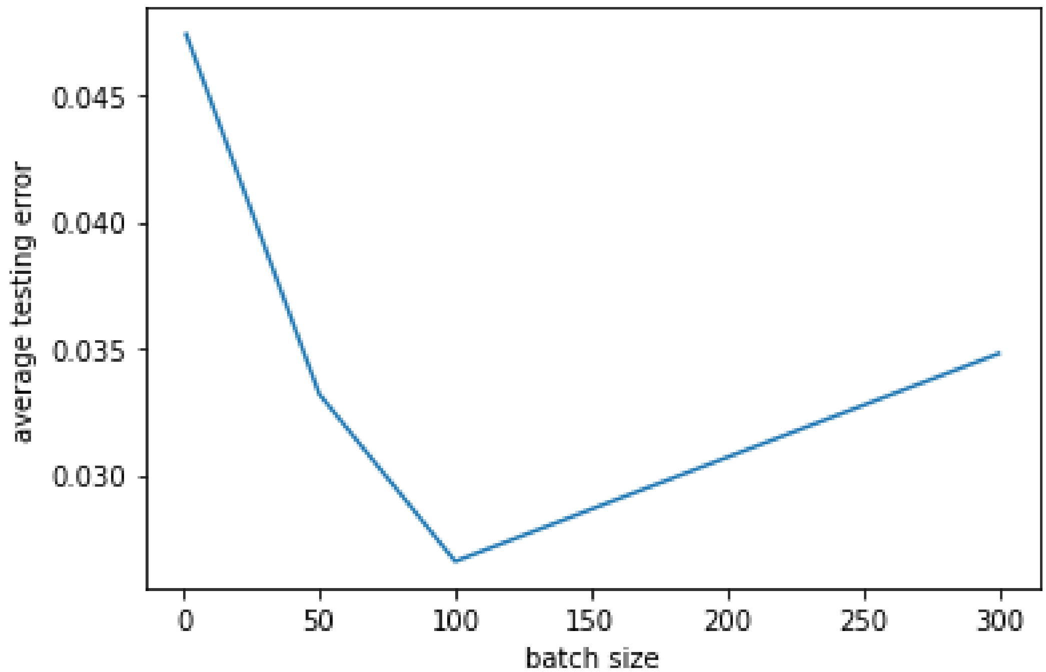
```
Test error: 0.022
```

(n) 2.2% of test data points are misclassified using the trained SVM.

[49]:
```
Batch = [1, 50 , 100, 300]
test_err = np.zeros((len(Batch),1))
# ================================================================ #
# YOUR CODE HERE:
# Train the SVM for different batch size Avergae the test error over 10 times
# ================================================================ #
for t in range (0,10):
    for m in range(0,len(Batch)):
        batch_size = Batch[m]
        eta = 1e-5
        num_iters = 8000 // batch_size
        svm = SVM(d=d, reg_param=0)
        loss_history, w = svm.train_svm(X_train,y_train,␣
 ↪eta=eta,batch_size=batch_size, num_iters=num_iters)
```

```
        acc = np.mean(svm.predict(X_test) == (y_test > 0))
        test_err[m, 0] += 0.1*(1-acc)
plt.plot(Batch, test_err[:, 0])
plt.xlabel('batch size')
plt.ylabel('average testing error')
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

[49]: Text(0, 0.5, 'average testing error')



[50]:
```
for i in range(4):
    print('Batch size: ', Batch[i], 'Error: ', test_err[i])
```

```
Batch size:  1 Error:  [0.0474]
Batch size:  50 Error:  [0.0332]
Batch size:  100 Error:  [0.0266]
Batch size:  300 Error:  [0.0348]
```

(o) Testing error is minimized for a batch size of 100.

[ ]:

Question3

February 19, 2021

[5]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
import csv
from data_load import load
import scipy.io as io
# Load matplotlib images inline
%matplotlib inline
# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/
↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

1 Section #1.3: Multi-Class Logistic Regression and Adaboost

Please follow our instructions in the same order to solve the linear regresssion problem.

Please print out the entire results and codes when completed.

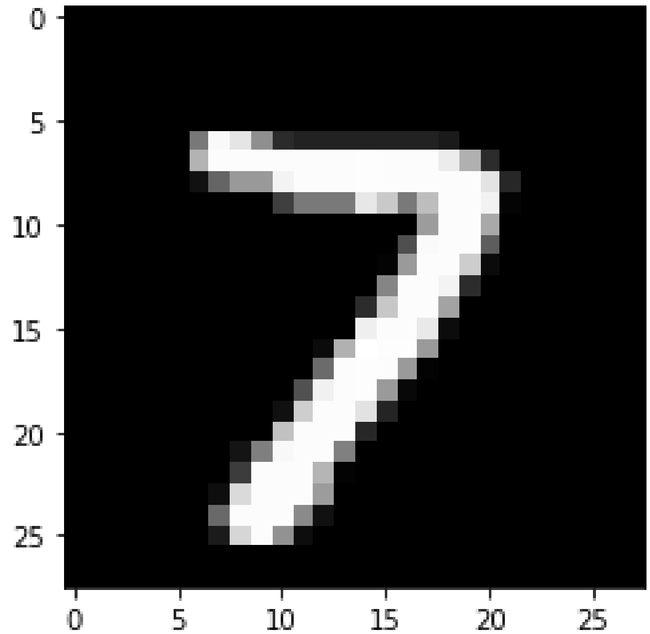
[6]:
```
from data_loadM import load_mnist
X_train,X_test,y_train,y_test=load_mnist()
print('Train data shape: ', X_train.shape)
print('Train target shape: ', y_train.shape)
print('Test data shape: ',X_test.shape)
print('Test target shape: ',y_test.shape)
```

```
Train data shape:  (60000, 784)
Train target shape:  (60000,)
Test data shape:  (10000, 784)
Test target shape:  (10000,)
```

ANSWER The dimension of the training set is: (60000, 784) and the test set is: (10000, 784).

[7]:
```
# To Visualize a point in the dataset
index = 4000
X = np.array(X_train[index], dtype='uint8')
```

```
X = X.reshape((28, 28))
fig = plt.figure()
plt.imshow(X, cmap='gray')
plt.show()
fig.savefig('Sample.pdf')
print('label is', y_train[index])
```




```
label is 7
```

1.1 Train Multi-Class Logistic Regression

In the following cells, you will build a Multi-Class logistic regression. You will implement its loss function, then subsequently train it with gradient descent. You will implement L1 norm regularization, and choose the best regularization parameter.

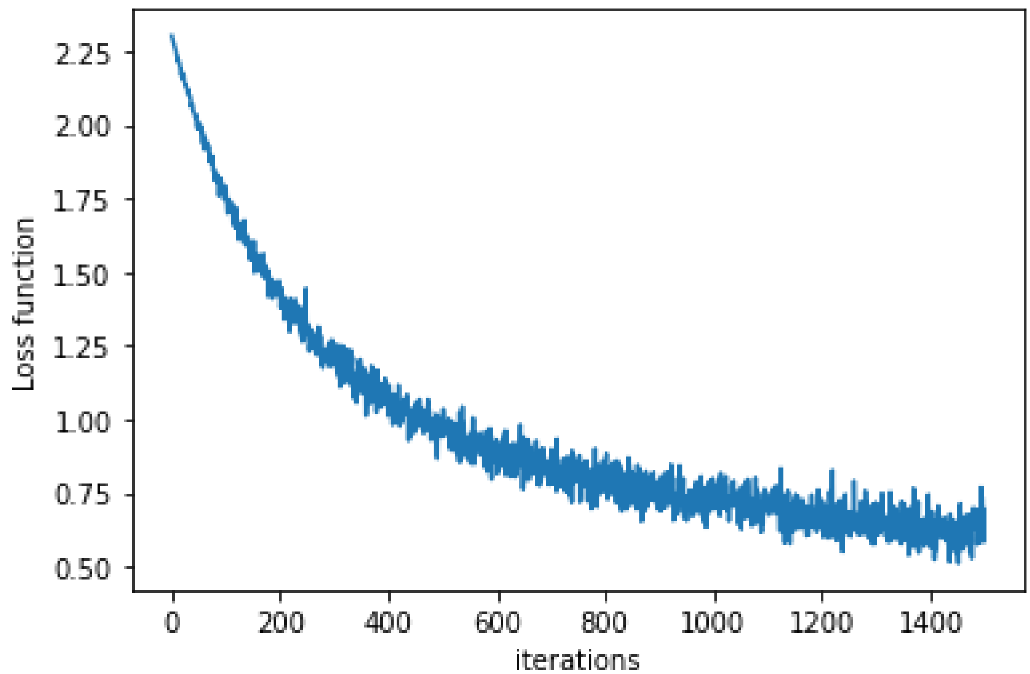
[22]:
```
from MLogistic import MLogistic
```

[23]:
```
## Complete loss_and_grad function in MLogistic.py file and test your results.
num_classes = len(np.unique(y_train))
num_features = X_train.shape[1]

logistic = MLogistic(dim=[num_classes,num_features], reg_param=0)
loss, grad = logistic.loss_and_grad(X_train[:5000],y_train[:5000])
print('Loss function=',loss)
print('Frobenius norm of grad=',np.linalg.norm(grad))
```

```
##
```

```
Loss function= 2.3025850929939837
Frobenius norm of grad= 269.57149388566296
```

[24]:
```
## Complete train_LR function in MLogistic.py file
loss_history, w = logistic.train_LR(X_train,y_train, eta=1e-7,batch_size=200,␣
↪num_iters=1500)
fig = plt.figure()
plt.plot(loss_history)
plt.xlabel('iterations')
plt.ylabel('Loss function')
plt.show()
print(np.linalg.norm(w))
print(loss_history[1499])
```




```
0.013278785268794387
0.6371785831753499
```

ANSWER The final value of the loss function is 0.013 and the value L2 of the weight is 0.637.

[25]:
```
# ================================================================ #
# YOUR CODE HERE:
#Complete predict function in MLogistic.py file and compute the trainin error␣
↪and the test error
```

```
# ================================================================ #

def get_acc(logistic, inputs, target):
    preds = logistic.predict(inputs)
    acc = 100 * sum(preds.astype(int) == target)/len(target)
    return acc

print(f"Train Error: {100 - get_acc(logistic, X_train, y_train)}")
print(f"Test Error: {100 - get_acc(logistic, X_test, y_test)}")

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
60000
Train Error: 13.996666666666667
10000
Test Error: 12.909999999999997
```

Note that here the test error is lower than the train error, which is unexpected. We looked into the dataset and noticed that the last 10,000 samples of the training set (i.e X_train[50000:]) corresponded to X_test, leading to improved performance on the test set.

[20]:

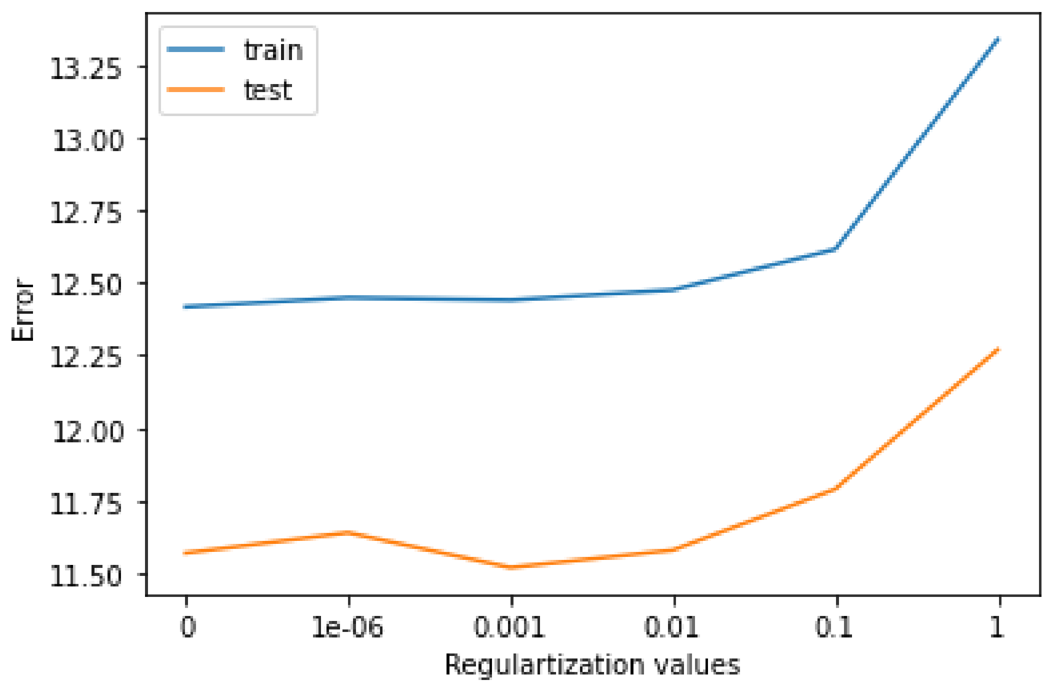
```
reg = [0,1e-6,1e-3,1e-2,1e-1,1]
train_err =np.zeros((len(reg),1))
test_err =np.zeros((len(reg),1))
# ================================================================ #
# YOUR CODE HERE:
# complete the following code to plot both the training and test loss in the␣
↪same plot
# for m range from 1 to 10
# ================================================================ #
for m in range(0,len(reg)):
    logistic = MLogistic(dim=[num_classes,num_features], reg_param=reg[m])
    loss_history, w = logistic.train_LR(X_train,y_train,␣
↪eta=1e-7,batch_size=200, num_iters=3000)
    train_err[m] = get_acc(logistic, X_train, y_train)
    test_err[m] = get_acc(logistic, X_test, y_test)
```

```
60000
10000
60000
10000
60000
10000
60000
10000
```

```
60000
10000
60000
10000
```

[21]:
```python
# Bad naming, here train_err and test_err actually refer to the accuracy
plt.plot(100 - train_err, label='train')
plt.plot(100 - test_err, label='test')
plt.legend()
plt.xlabel('Regulartization values')
results = list(map(str, reg))
plt.xticks(np.arange(len(results)), results)
plt.ylabel('Error')

# ============================================================= #
# END YOUR CODE HERE
# ============================================================= #
```

[21]: Text(0, 0.5, 'Error')



ANSWER The regularization value that seems to work best is 0.001. The minimum value of the test error on the MNIST dataset is around 11.5%. Again, as mentioned earlier, there was overlap between the training and test set, with the data provided, which lead to the unusual instance that the test set error was lower than the training set.

[7]:
```
from sklearn.tree import DecisionTreeClassifier
```

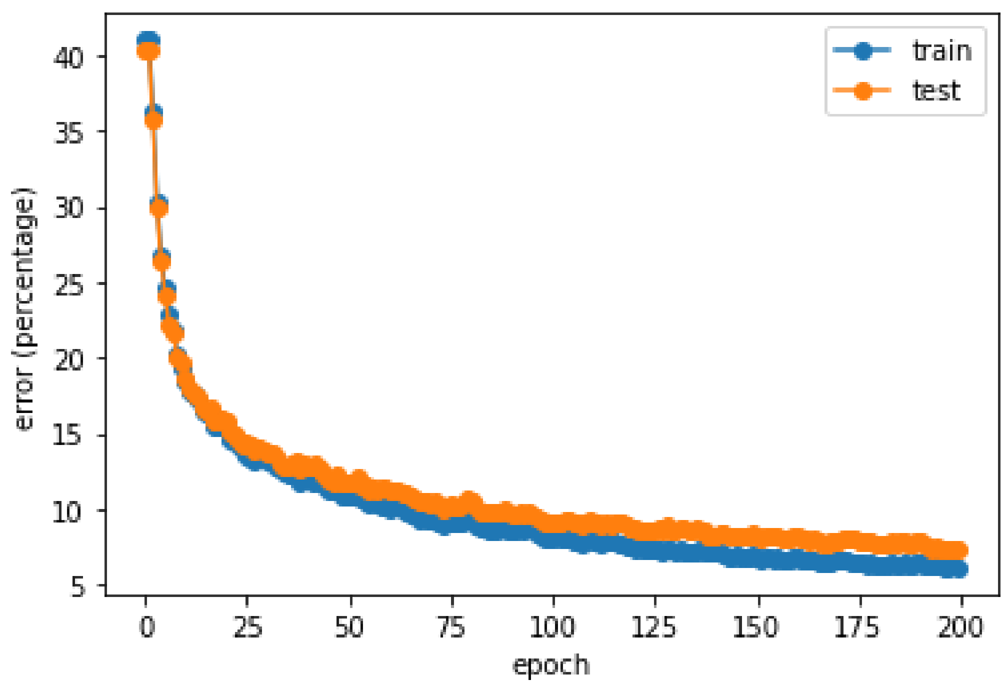
[63]:
```
import pdb

T = 200
N = X_train.shape[0]
num_classes = len(np.unique(y_train))
num_features = X_train.shape[1]
train_err = np.zeros((T,1))
test_err = np.zeros((T,1))
# ================================================================ #
# YOUR CODE HERE:
# complete the following code to plot both the training and test loss in the␣
↪same plot
# as a function of number of classifiers T for Adaboost Algorithm.
# ================================================================ #

#Initialize
D = np.ones_like(y_train)/len(y_train)
alphas = np.zeros((T,1))
classifiers = []
errors = np.zeros((T,1))
total_test_predictions = np.zeros((10000, 10))
total_train_predictions = np.zeros((60000, 10))
train_errors = np.zeros((T,1))
test_errors = np.zeros((T,1))

for t in range(0,T):
    #Train decision Tree
    tree = DecisionTreeClassifier(max_depth = 4).fit(X_train, y_train,␣
↪sample_weight = D)
    classifiers.append(tree)

    #Compute error
    train_predictions = tree.predict(X_train)
    errors[t] = np.sum((train_predictions != y_train) * D)

    #Compute \alpha
    alphas[t] = np.log((1 - errors[t])/errors[t]) + np.log(9) #K=10 classes

    #Update weights
    D = D * np.exp(alphas[t] * (train_predictions != y_train))
    D = D/np.sum(D)

    #Predict using Last t classifiers
    test_predictions = classifiers[t].predict(X_test)
    total_test_predictions[np.arange(10000), test_predictions] += alphas[t]
```

```
    final_test_predictions = np.argmax(total_test_predictions, axis=1)

    total_train_predictions[np.arange(60000), train_predictions] += alphas[t]
    final_train_predictions = np.argmax(total_train_predictions, axis=1)

    #compute test error and train error
    train_errors[t] = np.sum(final_train_predictions != y_train)
    test_errors[t] = np.sum(final_test_predictions != y_test)
```

[66]:

```
# Plot test error and train error in the same plot vs T



plt.plot(100 * train_errors/len(y_train), marker='o', label='train')
plt.plot(100 * test_errors/len(y_test), marker='o', label='test')
plt.legend()
plt.xlabel('epoch')
plt.ylabel('error (percentage)')

print(f"First error: {100 * train_errors[0]/len(y_train)}")
print(f"Last error: {100 * train_errors[-1]/len(y_train)}")

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
First error: [41.01166667]
Last error: [6.20166667]
```



ANSWER The first error was: 41.01% and the last error was: 6.20%

[ ]:

```
```