CS289A_HW03_Prob6c2

February 28, 2017

Load modules to be used in the execution of the problem.

In [1]:
```
%load_ext autoreload
```

In [2]:
```
%autoreload 2
```

In [3]:
```
import math
import HW03_utils as ut
import numpy as np
from matplotlib import pyplot as plt
```

In [4]:
```
def normalize_images(image_vectors):
    # Function to normalize pixel contrast of images

        magnitudes = np.linalg.norm(image_vectors,axis=1)
        normalized_ims = image_vectors/magnitudes[:,None]
        return normalized_ims
```

In [5]:
```
def get_class_bounds(classid,labels):
    # Function to extract index bounds of the specified class from the dataset

    for i in range(len(labels)):
        if labels[i] == classid:
            startindex = i
            break
    stopindex = len(labels)
    for i in range(i,len(labels)):
        if labels[i] != classid:
            stopindex = i
            break

    return startindex,stopindex
```

In [6]:
```
def get_class_from_data(classid,data,labels):
    # Find the start (inclusive) and end (exclusive) of a class within the data

    startindex,stopindex = get_class_bounds(classid,labels)
```

```
            # Separate the specified class
            class_data = data[startindex:stopindex]

            return class_data
```

In [7]:
```
def mean_of_class(classid,data,labels):
    # Calculate the mean value when the class is fit to a normal distribution

        class_data = get_class_from_data(classid,data,labels)
        # Calculate the mean of the class data
        class_mu = np.mean(class_data,axis=0)

        return class_mu
```

In [8]:
```
def cov_of_class(classid,data,labels):
    # Calcualte the covariance matrix when the class is fit to a normal distrib

        class_data = get_class_from_data(classid,data,labels)
        # Calculate the covariance matrix from the class data
        class_Sigma = np.cov(class_data,rowvar=False)

        return class_Sigma
```

In [9]:
```
def Prior(classid,data_labels):

        # Calculate the prior probability
        startindex,stopindex = get_class_bounds(classid,data_labels)
        nPoints = stopindex-startindex
        pi_i = nPoints/len(data_labels)

        return pi_i
```

In [10]:
```
def QDF_solve(X,muC,SigmaC,piC=0.1):
    # Function to solve the linear discriminant function for class C (will co

         QDFs_C = np.zeros(len(X))
         invSigmaC = np.linalg.pinv(SigmaC)
         detSigmaC = np.linalg.det(SigmaC)
         print('det ',detSigmaC)
         lndetSigmaC = np.log(detSigmaC)
         lnpiC = math.log(piC)
         for i in range(len(X)):
             x = X[i]
             QDFs_C[i] = -0.5*np.dot(np.dot((x-muC),invSigmaC),(x-muC))-0.5*lnd

         return QDFs_C
```

In [11]:
```
def maximize_QDFs(quad_disc_fns):
        max_QDF_indices = np.empty(len(quad_disc_fns))
```

```
            for i in range(len(max_QDF_indices)):
                max_QDF_indices[i] = np.argmax(quad_disc_fns[i])

            return max_QDF_indices
```

In [12]:
```
CS_DIR = r"/Users/mitch/Documents/Cal/2 - 2017 Spring/COMPSCI 289A - Intro
```

In [13]:
```
# Load MNIST data
data_array = ut.loaddata("hw3_mnist_dist/hw3_mnist_dist/train.mat",CS_DIR+
```

In [14]:
```
# Shuffle data and set aside validation set
np.random.shuffle(data_array)

trainarray = data_array[:-10000]
valarray = data_array[-10000:]
```

In [15]:
```
def findRedundants(sym_matrix):
    # Take a symmetric matrix and find rows/columns that are redundant

        red_rows = []
        for i in range(len(sym_matrix)):
            if not np.any(sym_matrix[i]):
                red_rows.append(i)

        return red_rows
```

In [16]:
```
def removeRedundants(matrix,red_vecs_inds):
    # Eliminate redundant vectors from a matrix, or elements from
    # a vector corresponding to redundant rows/columns in a matrix

        newlen = len(matrix)-len(red_vecs_inds)
        if len(np.shape(matrix))==2:
            newmatrix = np.empty((newlen,newlen))
            I = 0
            for i in range(len(matrix)):
                if i in red_vecs_inds:
                    continue
                J = 0
                for j in range(len(matrix)):
                    if j in red_vecs_inds:
                        continue
                    newmatrix[I,J] = matrix[i,j]
                    J += 1
                I += 1

            return newmatrix

        if len(np.shape(matrix))==1:
            newvector = np.empty(newlen)
```

```
        I = 0
        for i in range(len(matrix)):
            if i in red_vecs_inds:
                continue
            newvector[I] = matrix[i]
            I+=1

        return newvector
```

In [17]:
```
def main(traindata,trainlabels,valdata,vallabels):
    # Main block of code

        quad_disc_fns = np.empty((len(valdata),10))
        for i in range(10):
            muC = mean_of_class(i,traindata,trainlabels)
            SigmaC = cov_of_class(i,traindata,trainlabels)
            sigvals = []
            for u in SigmaC:
                for v in u:
                    if v!= 0:
                        sigvals.append(v)
            print(sigvals)
            piC = Prior(i,trainlabels)

            RedVarInds = findRedundants(SigmaC)
            newmuC = removeRedundants(muC,RedVarInds)
            newSigmaC = removeRedundants(SigmaC,RedVarInds)
            print(np.shape(newSigmaC))
            newvaldata = np.empty((len(valdata),len(valdata[0])-len(RedVarInds
            for datapointi in range(len(valdata)):
                newvaldata[datapointi] = removeRedundants(valdata[datapointi],

            quad_disc_fns[:,i] = QDF_solve(newvaldata,newmuC,newSigmaC,piC)

            digitPicks = maximize_QDFs(quad_disc_fns)

        count, total = 0,0
        for i in range(len(digitPicks)):
            if digitPicks[i] == vallabels[i]:
                count += 1
            total += 1

        # VERBOSE COMMANDS FOR WATCHING PROGRESS [OPTIONAL]
        #    if total%200 == 0:
        #        print(total,'points evaluated; current score =',count/total)
        print(count,total)
```

```
            score = count/total

            return score
```

In [18]:
```
# Organize array by digit
trainarray_byclass = trainarray[trainarray[:,-1].argsort()]
valarray_byclass = valarray[valarray[:,-1].argsort()]

train_data = trainarray_byclass[:,:-1]
train_labels = trainarray_byclass[:,-1]

val_data = valarray_byclass[:,:-1]
val_labels = valarray_byclass[:,-1]

normalized_traindata = normalize_images(train_data)
normalized_valdata = normalize_images(val_data)
```

In [ ]:
```
samples = [100,200,500,1000,2000,5000,10000,30000,50000]
```

In [ ]:
```
# Train on subsets of full training data set
scores = []
for number in samples:
    trainarraysubset = trainarray[:number]

    # Organize array by digit
    trainarray_byclass = trainarraysubset[trainarraysubset[:,-1].argsort()]
    valarray_byclass = valarray[valarray[:,-1].argsort()]

    # Separate data and labels
    train_data = trainarray_byclass[:,:-1]
    train_labels = trainarray_byclass[:,-1]
    val_data = valarray_byclass[:,:-1]
    val_labels = valarray_byclass[:,-1]

    # Normalize training and validation data
    normalized_train_data = normalize_images(train_data)
    normalized_val_data = normalize_images(val_data)

    print(number,"training samples: ")
    score = main(normalized_train_data,train_labels,normalized_val_data,val
    scores.append(score)
    print(score)
```

In [ ]:
```
errors = np.ones(len(scores))-np.array(scores)

fig = plt.figure(figsize=(15,15))
plt.semilogx(samples,error)
```

```
        plt.xlabel("# Training Points")
        plt.ylabel("Test Error")
        plt.savefig("LDA_errors.jpg")
        plt.show()
```

In [ ]:
```
print(errors)
```

In [ ]:
```

```