Lab 2: Classification (Part 1)

Name: Matthew Wei mswei2

Due September 8, 2023 11:59 pm

Logistics

See Canvas.

Lab submission:

There will be two submissions. Submit your code to gradescope LAB2 CODE section and submit the pdf version of the outputs (the pdf of the Jupyter notebook) to the LAB2 PDF section.

What You Will Need To Know For This Lab

This lab covers some basic classifiers which can be used for M-ary classification.

- k-Nearest Neighbors
- Bayes Classifiers
- Linear Discriminant Analysis

There are some problems which have short answer questions. **Do not write an essay -- a few sentences will suffice.**

Be clear about your answers. For example, if a question asks you "Which classifier would you choose?", be unequivocal about which classifier you would choose (and why); as engineers, part of your job is to make design decisions and justify them in comparison to the alternatives.

Preamble (Don't change this)

In [1]:

```python
from __future__ import division
%pylab inline
import numpy as np
import scipy.spatial.distance as dist
from scipy import stats
```

```
%pylab is deprecated, use %matplotlib inline and import the required libraries.
Populating the interactive namespace from numpy and matplotlib
C:\Users\matt3\AppData\Local\Programs\Python\Python310\lib\site-packages\scipy\__init
__.py:146: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this ver
sion of SciPy (detected version 1.24.2
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

Problem 1 : Bayes Classifiers (20 points)

In this section, you will implement a Bayes classifier for the following M-ary classification problem:

$$H_y : \mathbf{X} \sim \mathcal{N}(\mu_y, \Sigma) \qquad y = 0, \dots, M-1$$

i.e. the data is a d-dimensional Gaussian with a common covariance matrix among all classes, but the means are different (and there is a prior among the classes). Remember, when the mean vectors, covariance matrix and prior probabilities are known, no classifier can do better than the Bayes classifier.

You will write a function which takes in 4 parameters:

- A set of data to classify (with rows as feature vectors) as a (V, d) numpy.ndarray (data)
- A M-length vector with the prior probabilities of each class as a numpy.ndarray (pi)
- A matrix with rows giving the class means as a (M, d) numpy.ndarray (means)
- The common covariance matrix as a (d, d) numpy.ndarray (cov)

It will output a length V numpy.ndarray of the outputs of the classifier (labels). You may not use scikit-learn or similar to implement this. Note that the class labels in this problem are $0, 1, 2$ (not $1, 2, 3$). Since Python uses zero-based indexing, this will allow you to avoid a few +1's in your code.

Some hints which will help you write your code compactly:

- Using some (or all) of the hints, you should be able to write the Bayes classifier by using operations on numpy arrays, rather than loops. This is known as vectorization or array programming, and it can improve both speed and readibility of code. For example, a reasonably vectorized implementation can be done in under 5 lines. If you cannot figure this out, write your code using loops appropriately (Don't spend too much time on trying to write this without loops -- 20 minutes tops!)
- $\mathbf{x}^\top A\mathbf{y} = \mathbf{y}^\top A^\top \mathbf{x}$ for a matrix A and vectors $\mathbf{x}, \mathbf{y}$ of appropriate size.
- Covariance matrices are symmetric -- $\Sigma = \Sigma^\top$
- If $A = \begin{bmatrix} \mathbf{a}_1^\top \\ \vdots \\ \mathbf{a}_N^\top \end{bmatrix}$ has k columns and $\mathbf{v}$ is a vector of length k, then $A\mathbf{v} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{v} \\ \vdots \\ \mathbf{a}_N^\top \mathbf{v} \end{bmatrix}$.
- Some useful functions that you did not see in Lab 1 (*read the documentation before using them!*): numpy.sum, numpy.log, numpy.argmax.
- You may use broadcasting to help simplify your code.

A function prototype is provided below **(10 points)**:

In [2]:
```python
def bayesClassifier(data,pi,means,cov):
    #Put your code here
    y_hat = np.log(pi)+np.dot(means,np.linalg.inv(cov).dot(data.T)).T - 1/2*np.sum(np
    x = np.argmax(y_hat, axis=1)
    return x
```

Now, you will write a function which calculates the error of a classifier using the $0,1$-loss by comparing the true labels and the predicted labels.

The function will take in two parameters:

- A N-vector with the true labels as a numpy.ndarray (truelabels)
- A N-vector with the estimated labels as a numpy.ndarray (estimatedlabels)

The function will return the error (a scalar).

A function prototype is provided below **(5 points)**:

In [3]:
```
def classifierError(truelabels,estimatedlabels):
    # Put your code here
    x = np.sum(truelabels!=estimatedlabels)/len(truelabels)
    return x
```

Now, we will load some sample data, in the format specified above. We have three classes, with

$$\pi_0 = \frac{1}{3}, \pi_1 = \frac{1}{2}, \pi_2 = \frac{1}{6}$$

$$\mu_0 = \begin{bmatrix} 5 \\ 5 \end{bmatrix}, \mu_1 = \begin{bmatrix} 5 \\ 0 \end{bmatrix}, \mu_2 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

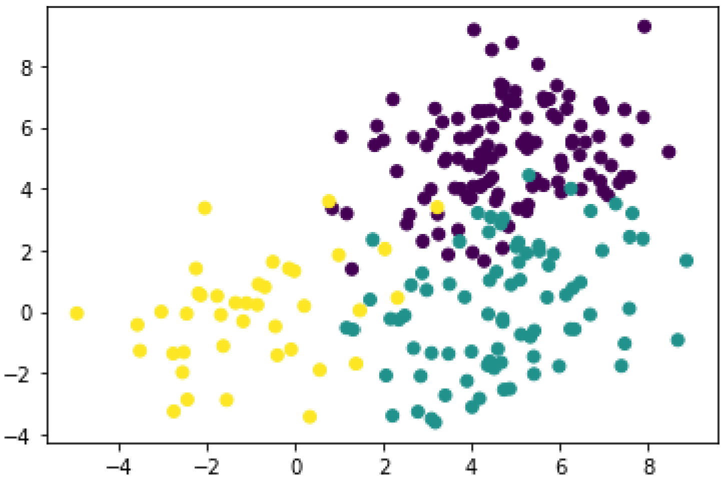
$$\Sigma = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

In [4]:
```
# Loading the data
from numpy import genfromtxt
problem1_tmp= genfromtxt('val.csv', delimiter=',')

#The data which you will use to test the classifier is called "data"
data=problem1_tmp[:,:2]
#The labels are in "truelabels"
truelabels=problem1_tmp[:,2]
```

Let's visualize the data by class. Each class will be in a different color.

In [5]:
```
scatter(data[:,0],data[:,1],c=truelabels)
```

Out[5]:
```
<matplotlib.collections.PathCollection at 0x20fdca837f0>
```



Run the classifier and report the classifier's error rate. **(5 points)**

In [14]:
```
# Put your code here. Store your estimated labels in a variable called "estimatedlabe
pi = np.array([1/3,1/2,1/6])
mean = np.array([[5,5],[5,0],[-1,0]])
cov = np.array([[3,1],[1,3]])
estimatedlabels = bayesClassifier(data, pi, mean, cov)
print(classifierError(truelabels, estimatedlabels))
```

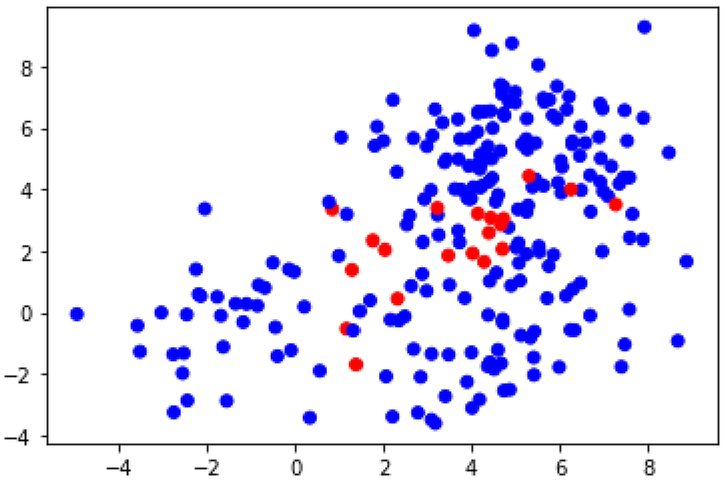
```
0.08333333333333333
```

The classifier's error rate is .0833.

Now, let's see which points were assigned to the correct class (blue) and those which are wrong (red).

In [15]:
```
cmap=['b','r']
colors = np.asarray(cmap)[np.asarray(truelabels!=estimatedlabels).astype(int)]
scatter(data[:,0],data[:,1],c=colors)
```

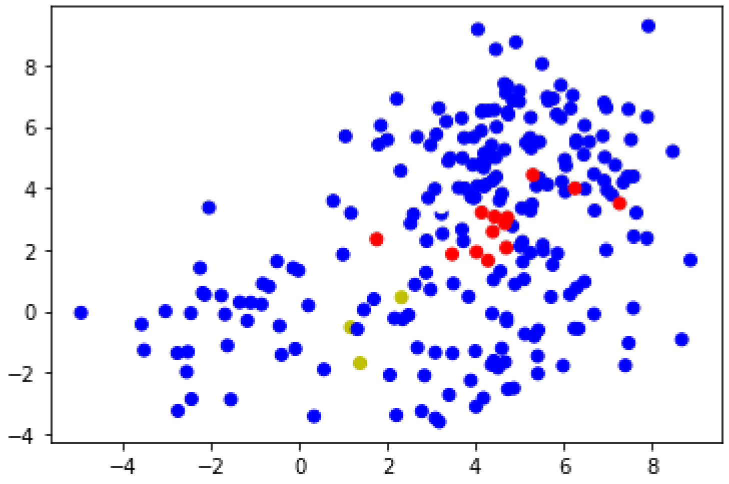
Out[15]:
```
<matplotlib.collections.PathCollection at 0x20fdfc27fa0>
```



And the points which were misclassified (based on which decision boundary they crossed: cmap[i][j] denotes the color for crossing the decision boundary between class i and class j). Note that the decision boundaries in this case are linear, since we have Gaussians with common covariance matrix as our classes.

In [18]:
```
cmap=[['b','r','w'],['r','b','y'],['w','y','b']]
colors = np.asarray(cmap)[truelabels.astype(int), estimatedlabels.astype(int)]
scatter(data[:,0],data[:,1],c=colors)
```

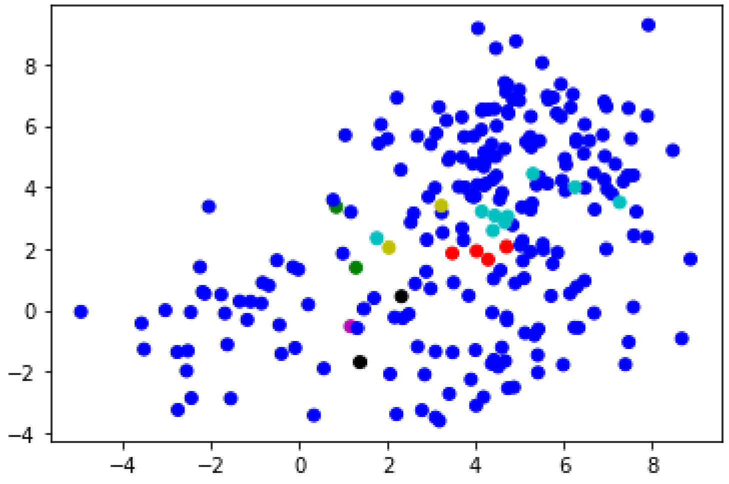
Out[18]:
```
<matplotlib.collections.PathCollection at 0x20fdfc8ba90>
```



And the types of errors made (blue points are correct; cmap[i][j] is the color corresponding to declaring class j when i was the true class):

In [19]:
```
cmap=[['b','r','g'],['c','b','m'],['y','k','b']]
colors = np.asarray(cmap)[truelabels.astype(int), estimatedlabels.astype(int)]
scatter(data[:,0],data[:,1],c=colors)
```

Out[19]:
```
<matplotlib.collections.PathCollection at 0x20fe08762f0>
```



Note that depending on the application, you may need to weigh different types of errors differently (e.g. is it better to convict an innocent man, or let a guilty man go free?). So, it may be worthwhile to visualize the errors made by a classifier separately.

You may want to (but are not required to) read about Type I and Type II errors (false alarms and misses, in ECE313 parlance) and Precision and Recall for some examples/details.

Problem 2 : Linear Discriminant Analysis (20 points)

In this problem, you will implement Linear Discriminant Analysis (LDA). You will use the Bayes classifier from Problem 1 to do this. You will be given:

- Training data feature vectors as a (N, d) numpy.ndarray (trainfeat), where each row is a feature vector.
- Training data labels as a length N numpy.ndarray (trainlabel)

The first function you will write will return a tuple of the estimates of the prior probabilities (as a M length numpy.ndarray), means (as a (M, d) numpy.ndarray) and covariance matrix (as a (d, d) numpy.ndarray) in the LDA model. You may assume that labels $0, \ldots,$trainlabel.max() exist in order to avoid some error checking. **(10 points)**

In [20]:
```
def trainLDA(trainfeat,trainlabel):
    nlabels=int(trainlabel.max())+1 #Assuming all labels up to nlabels exist.
    pi=np.zeros(nlabels)
    means=np.zeros((nlabels,trainfeat.shape[1]))
    cov=np.zeros((trainfeat.shape[1],trainfeat.shape[1]))
    # Put your code here
    for i in range(nlabels):
        pi[i] = trainfeat[trainlabel==i].shape[0]/trainfeat.shape[0]
        means[i] = np.sum(trainfeat[trainlabel == i], axis=0)/trainfeat[trainlabel==i
        cov+=np.dot((trainfeat[trainlabel==i] - means[i]).T, (trainfeat[trainlabel==i
    cov/=trainfeat.shape[0] - means.shape[0]
    return (pi,means,cov)
```

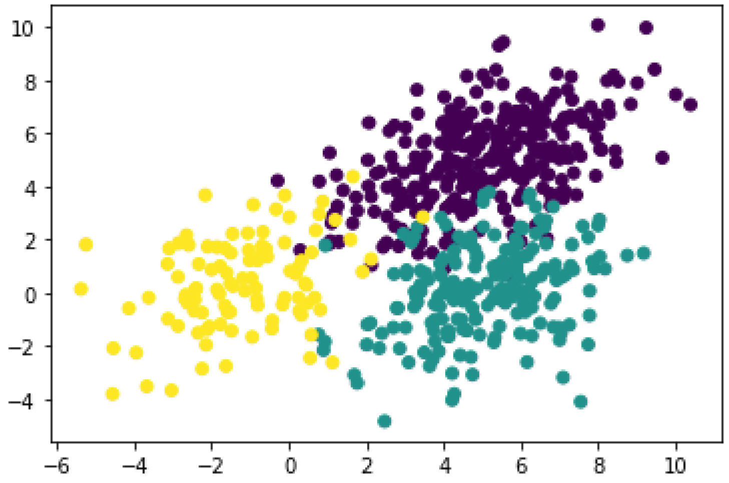
Now, we will load some data to validate our model.

In [21]:
```
# Loading the data
traindata_tmp= genfromtxt('train.csv', delimiter=',')
valdata_tmp= genfromtxt('val.csv', delimiter=',')

#The data which you will use to train LDA is called "trainingdata"
trainingdata=traindata_tmp[:,:2]
#The corresponding labels are in "traininglabels"
traininglabels=traindata_tmp[:,2]

#The data which you will use to validate LDA is called "valdata"
valdata=valdata_tmp[:,:2]
#The corresponding labels are in "vallabels"
vallabels=valdata_tmp[:,2]
```

And we can visualize the training data:

In [22]:
```
scatter(trainingdata[:,0],trainingdata[:,1],c=traininglabels)
```

Out[22]:
```
<matplotlib.collections.PathCollection at 0x20fe08dbbe0>
```



Use the bayesClassifier function you wrote in Problem 1 along with the trainLDA function from Problem 2 to implement the LDA classifier. Train the classifier on the training data, and then run the classifier on the training data and the validation data. Calculate the corresponding errors. Make sure to display the prior, means and covariance estimated in LDA. **(5 points)**

In [31]:
```
# Put your code here
pi, mean, cov = trainLDA(trainingdata, traininglabels)
print("Priors: ", pi)
print("Means: ", mean)
print("Covariances: ", cov)
estimatedLabels = bayesClassifier(trainingdata, pi, mean, cov)
print("Training Error: ", classifierError(traininglabels, estimatedlabels))
estimatedvallabels = bayesClassifier(valdata, pi, mean, cov)
print("Validation Error: ", classifierError(vallabels, estimatedvallabels))
```

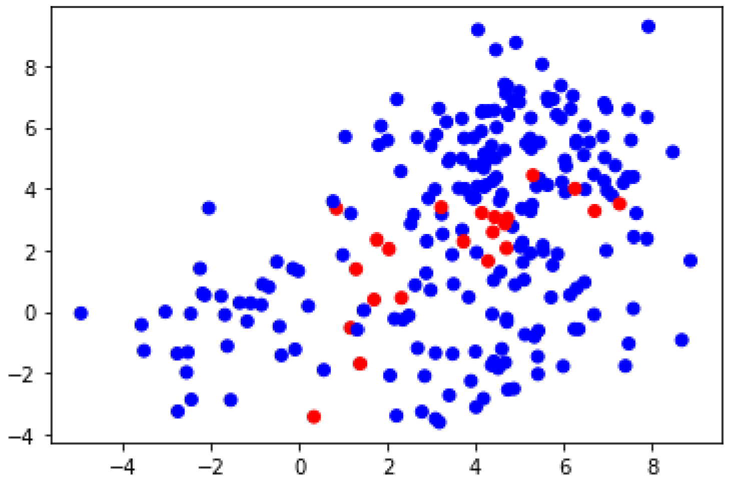
```
Priors:  [0.5        0.33333333 0.16666667]
Means:  [[ 5.04692803  4.97450647]
 [ 5.0352437   0.10765729]
 [-1.21545267  0.36139874]]
Covariances:  [[3.09469421 1.2602846 ]
 [1.2602846  2.91497343]]
Training Error:  0.0016666666666666668
Validation Error:  0.09166666666666666
```

```
C:\Users\matt3\AppData\Local\Temp\ipykernel_44168\3186853537.py:3: DeprecationWarnin
g: elementwise comparison failed; this will raise an error in the future.
  x = np.sum(truelabels!=estimatedlabels)/len(truelabels)
```

We can also visualize the performance of the classifier on the validation data (blue=correct, red=error).

In [32]:
```
#assuming your labels predicted on the validation data are in estimatedvallabels (cha
cmap=['b','r']
colors = np.asarray(cmap)[np.asarray(vallabels!=estimatedvallabels).astype(int)]
scatter(data[:,0],data[:,1],c=colors)
```

Out[32]:
```
<matplotlib.collections.PathCollection at 0x20fe0d003a0>
```



Comment on the training error (error of the classifier on the training data) and the validation error (error of the classifier on the validation data). Which one is higher? Does this make sense? **(5 points)**

The validation error is higher than the training error. This makes sense because validation usually underestimates the data, leading to a higher error rate than the training error value.

Problem 3: k-Nearest Neighbors (20 points)

Implement the k-Nearest Neighbors algorithm.

Your function will take:

- Training data feature vectors as a (N, d) numpy.ndarray (trainfeat), where each row is a feature vector
- Training data labels as a length N numpy.ndarray (trainlabel)
- Test data feature vectors as a (V, d) numpy.ndarray (testfeat), where each row is a feature vector
- The value of k

Use the Euclidean distance (scipy.spatial.distance.cdist) as your dissimilarity measure.

Your function should return a length V numpy.ndarray vector of the estimated labels. This should take around 4 lines of code. Do not use the kNN implementation in scikit-learn or similar.

Some functions which may be useful (read the documentation):

- The numpy.argpartition function can be used to find the k smallest elements of an array (via slicing)
- scipy.stats.mode can find the most common element in an array.

(10 points)

In [41]:
```
import scipy.spatial.distance as dist
from scipy import stats
def kNN(trainfeat,trainlabel,testfeat, k):
    #Put your code here
    d = dist.cdist(trainfeat, testfeat, metric='euclidean')
    i = np.argpartition(d, k, axis=0)[:k,:]
    x = stats.mode(trainlabel[i], axis=0)[0]
    return x
```

Run your k-Nearest Neighbors classifier with the training data from Problem 2, for $k = 1, 3, 4, 5$. Compute the training and validation error rates on the data from Problem 2. **(5 points)**

In [45]:

```
# Put your code here
trainError = np.zeros(4)
valError = np.zeros(4)
k = [1,3,4,5]

for i in range(len(k)):
    labels = kNN(trainingdata, traininglabels, trainingdata, k[i])
    trainError = classifierError(traininglabels, labels)
    labels = kNN(trainingdata, traininglabels, valdata, k[i])
    valError = classifierError(vallabels, labels)
    print("For k = ", k[i])
    print("Training Error: ", trainError)
    print("Validation Error: ", valError)
```

```
For k =  1
Training Error:  0.0
Validation Error:  0.11666666666666667
For k =  3
Training Error:  0.04666666666666667
Validation Error:  0.10833333333333334
For k =  4
Training Error:  0.06333333333333334
Validation Error:  0.10833333333333334
For k =  5
Training Error:  0.05833333333333334
Validation Error:  0.1
```

I will now tell you that the data set was generated according to the parameters given in the final part of problem 1. Compare and contrast the results you got from LDA and k-Nearest Neighbors as well as the Bayes classifier from problem 1. Your answer should consider both computational resources as well as error performance. What is special about the training error when you only have the standard nearest neighbor algorithm? Out of LDA and k-NN, which classifier would you prefer in this problem? Why? **(5 points)**

All three of the methods seem to have low error rates compared to its validation error. The contrast in how accurate they are though. LDA is the most accurate followed by kNN and then the Baysian as the last one. In terms of resources, Baysian requires the least followed by LDA and kNN as the most. We would prefer LDA in this problem because it gives lower error rates and given the simplicity of the data.

Problem 4: LDA and kNN, again **(20 points)**

In many cases, you will be using other people's libraries to implement learning algorithms. In this problem, you will become familiar with scikit-learn's implementation of LDA and kNN.

First, we will load a data set of digits drawn from zip codes written on US mail. This data set was designed to help get good algorithms to sort mail by zip code automatically. It has been preprocessed a bit, with details given here. Each feature vector consists of 16^2 real values representing grayscale values of a 16 by 16 image of a digit. The training data has 7291 samples, while the validation data has 2007 samples. Note that this is not the same dataset built into scikit-learn -- it is much larger.

In [46]:

```
#Loading the Data

#Read in the Training Data
traindata_tmp= genfromtxt('zip.train', delimiter=' ')

#The training labels are stored in "trainlabels", training features in "traindata". R
trainlabels=traindata_tmp[:,0]
traindata=traindata_tmp[:,1:]

#Read in the Validation Data
valdata_tmp= genfromtxt('zip.val', delimiter=' ')

#The validation labels are stored in "vallabels", validation features in "valdata". R
vallabels=valdata_tmp[:,0]
valdata=valdata_tmp[:,1:]
```

Use scikit-learn's sklearn.neighbors.KNeighborsClassifier to run a Nearest Neighbor classifier (1-NN) on the validation data with the provided training set. Note that KNeighborsClassifier defaults to 5-NN.

Measure the time for fitting the model and classification (the %timeit feature or time() or similar will be useful). Try the different algorithms possible to fit the model (ball tree, kd-tree and brute force, and specify the fastest one in your code). Make sure to calculate the error on the validation set. **(5 points)**

In [48]:

```
# Insert code here
from sklearn import neighbors

labels = neighbors.KNeighborsClassifier(n_neighbors=1, algorithm='brute', weights='di
start = time.time()
labels.fit(traindata, trainlabels)
mid = time.time()
pred = labels.predict(valdata)
end = time.time()
error = np.sum(pred!=vallabels)/vallabels.shape[0]
fitTime = mid - start
predTime = end - mid
print("Validation Error: ", error)
print("Fitting Time: ", fitTime)
print("Prediction Time: ", predTime)
```

```
Validation Error:  0.05630293971101146
Fitting Time:  0.01301264762878418
Prediction Time:  0.09900736808776855
```

Now, run LDA on the validation data set with scikit-learn's sklearn.discriminant_analysis. Measure the training time as well as the time used to classify the validation set. Also, calculate the error on the validation set. **(5 points)**

In [49]:

```python
# Insert code here
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

# Copied and pasted from above with minor changes
labels = LDA()
start = time.time()
labels.fit(traindata, trainlabels)
mid = time.time()
pred = labels.predict(valdata)
end = time.time()
error = np.sum(pred!=vallabels)/vallabels.shape[0]
fitTime = mid - start
predTime = end - mid
print("Validation Error: ", error)
print("Fitting Time: ", fitTime)
print("Prediction Time: ", predTime)
```

```
Validation Error:  0.114598903836572
Fitting Time:  0.5483603477478027
Prediction Time:  0.003005504608154297
```

Based on the performance on the validation set, which algorithm would you pick? Your answer should also take into account computational resources required, error on the validation set, and the cost associated with making an error (in real life). **(5 points)**

According to these results, we should pick the kNN algorithm due to its faster time and lower error rate. The data is also much larger and complex making kNN a better opetion to pick. Considering the faster time, the cost would be cheaper as well because more can be run.

Do you think the 0,1-loss is appropriate error measure in this case? If so, why or why not? How can you use domain-specific knowledge to help improve performance for this application?

No because if its dealing with the postal service, a 5% error is a lot of people. The performance can be improved by grouping the postal codes by domain and then running it like that, decreasing the error rate and reducing potential cost if there is any.

In [ ]: