CSE471: Statistical Methods in AI

Assignment 3

Abhinav Moudgil [201331039]

Contents

Naive Bayes classifier for discrete data

Dataset: UCI Bank Marketing Data Set[1]
The data is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls. Often, more than one contact to the same client was required, in order to access if the product (bank term deposit) would be ('yes') or not ('no') subscribed.

Pre processing:
Original data set contained multivariate data. It was trimmed to contain only discrete and categorical details. Following are the details of data set after trimming:
Characteristics: discrete, categorical
Number of instances: 41188
Number of features: 14

Classifier:
Naive Bayes classifier uses the concept of probability to classify new entities. It assumes that all the features are independent of each other. We can thus uncouple each feature and treat each one as independent. That's why it is called *naive* Bayes. We can also call each feature are evidence.
Given multiple evidences or set of features, we have to predict its class from given set of classes.
Thus,

P(outcome/multiple evidence) = P(evidence1/outcome).P(evidence2/outcome) ...... P(evidenceN/outcome).P(multiple evidence)

Or,

P(outcome/evidence) or P(posterior) = P(likelihood of evidence).Prior of evidence/P(evidence)

Naive Bayes decision rule:
Sample belongs to the class which has highest posterior probability.

Tie-break: If posterior probabilities for the classes are equal, sample is assigned the class which has highest prior probability because the class with more prior probability means it came more often in history. We assume that the fashion will follow for future, and hence the decision.

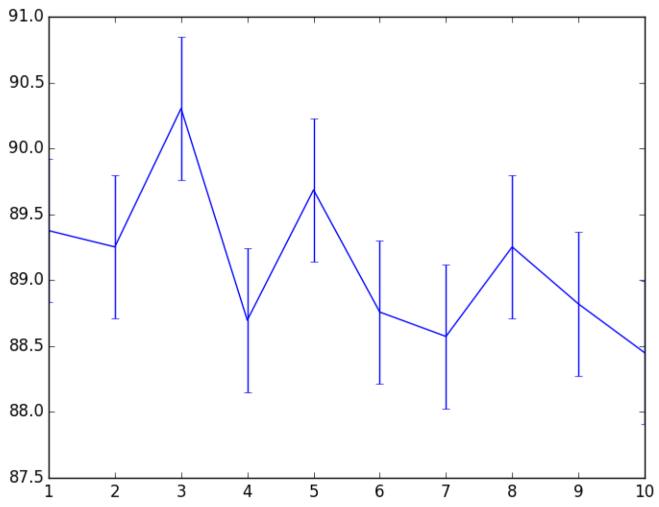
Handling of missing data points:
1. Unknown values are treated as another categorical values. Probability of this value is calculated for each feature and used in calculation of test sample probability. Underlying assumption is that number of unknown values in each feature in testing set will follow the same fashion as in training set.
2. There are some values for each feature, for which probability is not calculated while training as it was not there in training set. In such case, that feature is ignored and other features are used for classification.

Code:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn import preprocessing
import random
import copy
import math

def encode(vector):
    le = preprocessing.LabelEncoder()
    le.fit(vector)
    vector = le.transform(vector)
    return vector

def encodeAll(mat):
    categoricalIndices = [1,2,3,4,5,6,7,8,9,14]
    for i in categoricalIndices:
        mat[:,i] = encode(mat[:,i])
    return mat.astype(np.float)

def deleteFloatData(mat):
    idx_OUT_columns = [10,15,16,17,18,19]
    idx_IN_columns = [i for i in xrange(np.shape(mat)[1]) if i not
    in idx_OUT_columns]
    extractedData = mat[:,idx_IN_columns]
    return extractedData

def findProbabilityMatrix0(featureMatrix, C0):
    l = C0.shape[1]
    p0 = copy.deepcopy(featureMatrix)
    for i in xrange(l - 1):
        featureVector = featureMatrix[i]
        x0 = C0[:, i].tolist()
        j = 0
        for item in featureVector:
            c0 = float(x0.count(item))
            p0[i][j] = c0/float(C0.shape[0])
            j = j + 1
    return p0

def findProbabilityMatrix1(featureMatrix, C1):
    l = C1.shape[1]
    p1 = copy.deepcopy(featureMatrix)
    for i in xrange(0, l - 1):
        featureVector = featureMatrix[i]
        x1 = C1[:, i].tolist()
        j = 0
        for item in featureVector:
            c1 = float(x1.count(item))
            p1[i][j] = c1/float(C1.shape[0])
            j = j + 1
    return p1

def findFeatureMatrix(mat):
    l = mat.shape[1]
    featureMatrix = []
    for i in xrange(0, l - 1):
        featureVector = np.unique(mat[:, i])
        featureMatrix.append(featureVector.tolist())
    return featureMatrix

#Training phase
file = open('bank-additional.csv')
featureVectors = []
i = 0
for line in file :
    vector = line.strip().lower().split(';')
    if i != 0:
        if vector[-1] == '"no"':
            vector[-1] = 0
        else:
            vector[-1] = 1
        featureVectors.append(vector)
    i = i + 1

numberOfRuns = 10
results = np.zeros([numberOfRuns])
for t in xrange(numberOfRuns):
    random.shuffle(featureVectors)
    mat = np.array(featureVectors)
    mat = encodeAll(mat)
    mat = deleteFloatData(mat)
    mat = mat.astype(int)
    N = 2500
    trainData = mat[:N, :]
    testData = mat[N:, :]
    featureMatrix = findFeatureMatrix(trainData)
    C0 = trainData[trainData[:, -1] == 0]
    C1 = trainData[trainData[:, -1] == 1]
    p0 = findProbabilityMatrix0(featureMatrix, C0)
    p1 = findProbabilityMatrix1(featureMatrix, C1)
    pr0 = float(C0.shape[0])/float(trainData.shape[0])
    pr1 = float(C1.shape[0])/float(trainData.shape[0])
    #print pr0
    #print pr1
    if (pr0 > pr1):
        maxPrior = int(0)
    else:
        maxPrior = int(1)
    #Testing phase
    totalValues = testData.shape[0]
    #print totalValues
    #print testData[:, 10]
    myPrediction = np.zeros([totalValues])
    for i in xrange(0, totalValues):
        sample = testData[i, :]
        sample = sample.tolist()
        ans0 = (float(pr0))
        ans1 = (float(pr1))
        count = 0;
        for j in xrange(0, len(sample) - 1):
            flag = 0
            for k in xrange(0, len(featureMatrix[j])):
                if (sample[j] == featureMatrix[j][k]):
                    if (p0[j][k] != 0):
                        ans0 = ans0 * (p0[j][k])
                    else:
                        ans0 = ans0 * p0[j][k]
                    if (p1[j][k] != 0):
                        ans1 = ans1 * (p1[j][k])
                    else:
                        ans1 = ans1 * p1[j][k]
                #   print "Found!"
                    count = count + 1
                    flag = 1
                    break
            #if (flag == 0):
        #       print sample[j], " ", i + trainData.shape[0], " -
    ", j
        print ans0
        print ans1
        if (ans0 > ans1):
            myPrediction[i] = int(0)
        elif(ans0 < ans1):
            myPrediction[i] = int(1)
        else:
            myPrediction[i] = maxPrior

    trueAns = testData[:, -1]
    #print len(myPrediction[myPrediction == 0])
    #print len(myPrediction[myPrediction == 1])
    #print len(trueAns[trueAns == 0])
    #print len(trueAns[trueAns == 1])
    correctValues = 0
    for i in range(totalValues):
        if (myPrediction[i] == trueAns[i]):
            correctValues = correctValues + 1
        #else:
    #       print i + trainData.shape[0]

    correctValues = float(correctValues)
    totalValues = float(totalValues)
    accuracy = correctValues/totalValues * 100
    #print correctValues
    results[t] = accuracy
    #print accuracy

meanAccuracy = np.mean(results)
x = np.array([1,2,3,4,5,6,7,8,9,10])
y = results
plt.figure()
#plt.plot(x,y, marker='o', color='b')
plt.errorbar(x, y, np.std(results, axis = 0))
plt.show()
#stD = np.std(results)
print meanAccuracy
```

Results:

Size of data set	Mean Accuracy	Standard deviation	Time(s)
10%	88.472	0.311	2.630
100%	88.882	0.352	32.244



Observations:

1. Time complexity of this classifier during testing is $O(MN)$, where N is the number of test samples, and M is the number of features.
2. Mean accuracy of classifier can be increased by taking more training samples.

We'll get more unique feature values for which probability can be calculated. Hence less features will be ignored.
3. It assumes that all the features are independent is not true in actual and thus accuracy is less than 90%.

Bayesian parameter estimation derivation

As done in MLE first will start with a simple case with only the mean: μ unknown. As usual we will assume sample x_k is normally distributed as:

$$p(x|\mu) \sim N(\mu, \sigma^2)$$

and the parameter μ has the distribution of:

$$p(\mu) \sim N(\mu_0, \sigma_0^2),$$

as parameter μ is not estimated to be a number but a random variable.
Using Bayes' formula and the corresponding derivation from the previous section the corresponding function could be easily obatined:

$$p(\mu|D) = \alpha \prod_{k=1}^{n} p(x_k|\mu)p(\mu)$$

Where α is introduced as a 'scale' coefficient in order to simplify the derivation. Please note that α is completely independent of μ.
As x_k is normally distributed we update the $p(x_k|\mu)$ and $p(u)$ with the known distribution function:

$$p(x_k|\mu) = \frac{1}{(2\pi\sigma^2)^{1/2}} exp[-\frac{1}{2}(\frac{x_k - \mu}{\sigma})^2]$$

$$p(u) = \frac{1}{(2\pi\sigma_0^2)^{1/2}} exp[-\frac{1}{2}(\frac{\mu - \mu_0}{\sigma_0})^2]$$

Again, substitute $p(x_k|\mu)$ and $p(u)$ in equation $p(\mu|D) = \alpha \prod_{k=1}^{n} p(x_k|\mu)p(\mu)$ we obtained:

$$p(\mu|D) = \alpha \prod_{k=1}^{n} \frac{1}{(2\pi\sigma^2)^{1/2}} exp[-\frac{1}{2}(\frac{x_k - \mu}{\sigma})^2] \frac{1}{(2\pi\sigma_0^2)^{1/2}} exp[-\frac{1}{2}(\frac{\mu - \mu_0}{\sigma_0})^2]$$

$$p(\mu|D) = \alpha \prod_{k=1}^{n} \frac{1}{(2\pi\sigma^2)^{1/2}} \frac{1}{(2\pi\sigma_0^2)^{1/2}} exp[-\frac{1}{2}(\frac{\mu - \mu_0}{\sigma_0})^2 - \frac{1}{2}(\frac{x_k - \mu}{\sigma})^2]$$

Similarly, in order to simplify the derivation we update the scaling factors to α' and α'', and correspondingly,

$$p(\mu|D) = \alpha' exp \sum_{k=1}^{n} (-\frac{1}{2}(\frac{\mu - \mu_0}{\sigma_0})^2 - \frac{1}{2}(\frac{x_k - \mu}{\sigma})^2)$$

$$p(\mu|D) = \alpha'' exp[-\frac{1}{2}(\frac{n}{\sigma^2} + \frac{1}{\sigma_0^2})\mu^2 - 2(\frac{1}{\sigma^2}\sum_{k=1}^{n} x_k + \frac{\mu_0}{\sigma_0^2})\mu]$$

Finally, compare derived $p(u|D)$ to the Gaussian Distribution in the standard form:

$$p(u|D) = \frac{1}{(2\pi\sigma_n^2)^{1/2}} exp[-\frac{1}{2}(\frac{\mu-\mu_n}{\sigma_n})^2]$$

Based on knowledge on Gaussian Distribution,μ_n and σ_n^2 could be obtained accordingly:

$$\mu_n = (\frac{n\sigma_0^2}{n\sigma_0^2+\sigma^2})\bar{x_n} + \frac{\sigma^2}{n\sigma_0^2+\sigma^2}\mu_0$$

$$\sigma_n^2 = \frac{\sigma_0^2\sigma^2}{n\sigma_0^2+\sigma^2}$$

Univariate case:

Given the posteriori density $p(\mu|D)$ successfully derived (variance: σ_n^2 and mean: μ_n now known), the final step is to estimate $p(x|D)$ based on the conclusions above.

$$p(x|\mathcal{D}) = \int p(x|\mu)p(\mu|\mathcal{D})d\mu$$

$$p(x|\mathcal{D}) = \int \frac{1}{\sqrt{2\pi}\sigma}\exp[-\frac{1}{2}(\frac{x-\mu}{\sigma})^2]\frac{1}{\sqrt{2\pi}\sigma_n}\exp[-\frac{1}{2}(\frac{\mu-\mu_n}{\sigma_n})^2]d\mu$$

Finally, substitute σ_n^2 and μ_n the probability function $p(x|\mathcal{D})$ is obtained:

$$p(x|\mathcal{D}) = \frac{1}{2\pi\sigma\sigma_n} exp[-\frac{1}{2}\frac{(x-\mu)}{\sigma^2+\sigma_n^2}]\int exp[-\frac{1}{2}\frac{\sigma^2+\sigma_n^2}{\sigma^2\sigma_n^2}(\mu - \frac{\sigma_n^2\bar{x}_n + \sigma^2\mu_n}{\sigma^2+\sigma_n^2})^2]d\mu$$

Hence, $p(x|D)$ is normally distributed as:

$$p(x|D) \sim N(\mu_n, \sigma^2 + \sigma_n^2)$$

Multivariate case:

The treatment of the multivariate case in which $\sum$ is known but μ is not, is a direct generalization of the univariate case. For this reason we shall only sketch the derivation. As before, we assume that,

$p(x|\sum) \sim N(\mu, \sum)$ and $p(\mu) \sim N(\mu_0, \sum_0)$ We know,

$$p(\mu|D) = \alpha \prod_{k=1}^{n} p(x_k|\mu)p(\mu),$$

Equating coefficients and simplified by knowledge of the matrix identity,

$$(A^-1 + B^-1)^{-1} = A(A+B)^{-1}B = B(A+B)^{-1}A,$$

We obtain the final results,

$$\mu_{\mathbf{n}} = \mathbf{\Sigma_0(\Sigma_0 + \frac{1}{n}\Sigma)^{-1}(\frac{1}{n}\sum_{i=1}^{n} x_i) + \frac{1}{n}\Sigma(\Sigma_0 + \frac{1}{n}\Sigma)^{-1}}\mu_{\mathbf{0}}$$

and,

$$\Sigma_n = \Sigma_0(\Sigma_0 + \frac{1}{n}\Sigma)^{-1}\frac{1}{n}\Sigma$$

Gaussian Naive Bayes classifier with PCA and LDA

Dataset: Arcene[2]

It was obtained by merging three mass-spectrometry datasets to obtain enough training and test data for a benchmark. The original features indicate the abundance of proteins in human sera having a given mass value. Based on those features one must separate cancer patients from healthy patients. We added a number of distractor feature called *probes* having no predictive power. The order of the features and patterns were randomized.

Attribute characteristics: Real
Number of Instances: 900
Number of features:
Real: 7000
Probes: 3000
Total: 10000

Pre processing:

Data set originally contained 10000 features. Number of features are reduced by two approaches:

Principal Component Analysis (PCA).

PCA reduces the dimensions without taking into account the separation of classes. It picks up the top k dimensions with maximum variance, which facilitates separation of classes. The reduction procedure is explained below:

1. Covariance matrix is found for different features. It gives the idea of behaviour of change of one dimension with respect to another.
2. Eigen values of covariance matrix are found which gives the variance across different dimensions. They are sorted in descending order and top k dimensions are picked.
3. New data is found by:

$$NewData = RowFeatureVector.RowDataAdjust$$

where $RowFeatureVector$ is the matrix with the eigen vectors in the columns transposed so that the eigen vectors are now in the rows, with the most significant eigen vector at the top, and $RowDataAdjust$ is the mean-adjusted data transposed, ie. the data items are in each column, with each row holding a separate dimension.

Linear Discriminant Analysis (LDA)
LDA maximizes between class seperation i.e variance along resultant dimension is maximized.
It is done in 5 steps as below:
1. First, mean vector for each class are found.
2. Within class scatter matrix SW and between class scatter matrix SB is found.

$$SW = \sum_{i=1}^{c} \sum_{x=1}^{N} (x - m_i)(x - m_i)^T$$

$$SB = \sum_{i=1}^{c} N_i(m_i - m)(m_i - m)^T$$

3. Calculate the eigen values and eigen vectors of matrix $SW^{-1}SB$
4. Eigen vector are sorted by decreasing eigen values and first one is picked.
5. New data is found by:

$$NewData = RowFeatureVector.RowDataAdjust$$

Classifier:
Same bayesian decision rule is followed i.e class with highest posterior probability is assigned to test the sample. In *gaussian* naive bayes classifier, it is assumed that probability of each feature follows the gaussian distribution. In training phase, mean and variance are found for each feature. During testing, sample is picked and probability of each feature is found using the following gaussian function:

$$G(\mu, \sigma) = e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where μ is mean and σ is variance of the feature.

Results:
PCA

K	Mean Accuracy	Time(m)
10	56.0	16.39
100	56.0	16.39
1000	56.0	17.01

LDA

Mean Accuracy	Time(m)
56.0	24.0

Observations:
1. Mean accuracy is low because of random probes added in data which don't play any role in classifications and reduction of features by PCA and LDA.
2. In gaussian classifier, we have assumed that features will follow the gaussian distribution which is not true in real.
3. Execution time is nearly the same because in both the reduction method, we have calculated all the eigen vectors and picked up top k. The only difference lies in the calculation of new data from old one.
4. Sometimes gaussian function returns the value which is very low ($< 10{-}323$) and python truncates the value to 0. So we can't compare the values, which again results in low accuracy of classifier.
5. LDA calculates the inverse of within class scatter matrix SW. For this dataset, withing class scatter matrix SW is singular. Hence pseudo inverse for scatter matrix is calculated, which results in high execution time than PCA.

PCA

```python
import numpy as np
from numpy import linalg as LA
import math

def gaussian(x, v, M, V):
    G = (math.exp(-(math.pow((v - M[x]),2)/(2 * V[x]))))
    return G

def findVariance(C):
    return np.var(C, axis = 0)

def findMean(C):
    return np.mean(C, axis = 0)

def pcaTransform(data, k):
    data = data - np.mean(data, axis = 0)
    covarianceMatrix = np.cov(np.transpose(data))
    eigenValues, eigenVectors = LA.eig(covarianceMatrix)
    s = np.argsort(eigenValues)[::-1]
    ev = np.zeros(eigenVectors.shape)
    for i in xrange(eigenValues.shape[0]):
        ev[:, i] = eigenVectors[:, s[i]]
    data = np.matrix(data)
    data = np.transpose(data)
    eigenVectors = ev[:, :k]
    eigenVectors = np.matrix(eigenVectors)
    eigenVectors = np.transpose(eigenVectors)
    return eigenVectors
```

```python

def addLabels(data, trainLabels):
    b = np.zeros((data.shape[0], data.shape[1] + 1))
    b[:, :-1] = data
    b[:, -1] = trainLabels
    return b

def mergeData(trainData, testData):
    x = np.zeros((trainData.shape[0] + testData.shape[0], trainData
    .shape[1]))
    x[:trainData.shape[0], :] = trainData
    x[trainData.shape[0]:, :] = testData
    return x

def project(data, eigenVectors):
    data = data - np.mean(data, axis = 0)
    data = np.matrix(data)
    data = np.transpose(data)
    newData = eigenVectors * data
    newData = np.transpose(newData)
    newData = np.array(newData)
    return newData


def getDataMatrix(file, intOrFloat):
    #intOrFloat decides whether data should be int or float
    if (intOrFloat == 1):
        featureVectors = []
        for line in file :
            vector = line.strip().lower().split(' ')
            featureVectors.append(vector)
        data = np.array(featureVectors)
        data = data.astype(float)
    else:
        trainLabels = []
        for line in file :
            vector = line
            trainLabels.append(vector)
        data = np.array(trainLabels)
        data = data.astype(int)
    return data

file = open('arcene_train.data.txt')
data = getDataMatrix(file, 1)
file = open('arcene_train.labels.txt')
trainLabels = getDataMatrix(file, 0)
file = open('arcene_valid.data.txt')
testData = getDataMatrix(file, 1)
file = open('arcene_valid.labels.txt')
testLabels = getDataMatrix(file, 0)
#PCA
k = 1000
ev = pcaTransform(data, k)
trainData = project(data, ev)
testData = project(testData, ev)
trainData = addLabels(trainData, trainLabels)
testData = addLabels(testData, testLabels)
```

```

C0 = trainData[trainData[:, -1] == -1]
C1 = trainData[trainData[:, -1] == 1]
V0 = findVariance(C0[:, :-1])
V1 = findVariance(C1[:, :-1])
M0 = findMean(C0[:, :-1])
M1 = findMean(C1[:, :-1])
pr0 = float(C0.shape[0])/float(trainData.shape[0])
pr1 = float(C1.shape[0])/float(trainData.shape[0])
if (pr0 > pr1):
    maxPrior = int(-1)
else:
    maxPrior = int(1)
print maxPrior
L = math.pow(10, -323)
MAX = -math.pow(10, 300)

#Testing phase
totalValues = testData.shape[0]
myPrediction = np.zeros([totalValues])
for i in xrange(0, totalValues):
    sample = testData[i, :]
    sample = sample.tolist()
    ans0 = math.log(float(pr0))
    ans1 = math.log(float(pr1))
    count = 0;
    for j in xrange(0, len(sample) - 1):
        g1 = gaussian(j, sample[j], M0, V0)
        g2 = gaussian(j, sample[j], M1, V1)
        if (g1 < L):
            ans0 = MAX
        if(g2 < L):
            ans1 = MAX
        if(ans0 > MAX):
            ans0 = ans0 + math.log(g1)
        if(ans1 > MAX):
            ans1 = ans1 + math.log(g2)
    print "ans0 ", ans0
    print "ans1 ", ans1
    if (ans0 > ans1):
        myPrediction[i] = int(-1)
    elif(ans1 > ans0):
        myPrediction[i] = int(1)
    elif(ans1 == ans0):
        print "Max - ", maxPrior
        myPrediction[i] = maxPrior
trueAns = testData[:, -1]
correctValues = 0
for i in range(totalValues):
    if (myPrediction[i] == trueAns[i]):
        correctValues = correctValues + 1

correctValues = float(correctValues)
totalValues = float(totalValues)
accuracy = correctValues/totalValues * 100
print accuracy
```

LDA

```python
import numpy as np
from numpy import linalg as LA
import math
from sklearn import preprocessing

def gaussian(v, M, V):
    G = (math.exp(-(math.pow((v - M),2)/(2 * V))))
    return G

def findVariance(C):
    return np.var(C, axis = 0)

def findMean(C):
    return np.mean(C, axis = 0)

def mergeData(trainData, testData):
    x = np.zeros((trainData.shape[0] + testData.shape[0], trainData.shape[1]))
    x[:trainData.shape[0], :] = trainData
    x[trainData.shape[0]:, :] = testData
    return x

def ldaTransform(data):
    C0 = data[data[:, -1] == -1]
    C1 = data[data[:, -1] == 1]
    C0 = C0[:, :-1]
    C1 = C1[:, :-1]
    S0 = np.cov(np.transpose(C0))
    S1 = np.cov(np.transpose(C1))
    SW = S0 + S1
    Mu0 = np.mean(C0, axis = 0)
    Mu1 = np.mean(C1, axis = 0)
    Mu = np.mean(data, axis = 0)
    Mu = Mu[:-1]
    Mu = np.matrix(Mu)
    Mu0 = np.matrix(Mu0)
    Mu1 = np.matrix(Mu1)
    SB = C0.shape[0] * np.transpose(Mu0 - Mu) * (Mu0 - Mu) + C1.shape[0] * np.transpose(Mu1 - Mu) * (Mu1 - Mu)
    #t = Mu0 - Mu1
    #t = np.matrix(t)
    #SB = np.transpose(t) * t
    Swin = LA.pinv(SW) #costly
    Swin = np.matrix(Swin)
    SwinSB = Swin * SB #costly
    e, v = LA.eig(SwinSB) #costly
    s = np.argsort(e)[::-1]
    v = np.array(v)
    ev = np.zeros(v.shape)
    for i in xrange(e.shape[0]):
        ev[:, i] = v[:, s[i]]
    w = ev[:, 0]
    w = np.matrix(w)
    return w

def project(data, w):
    data = np.matrix(data)
    data = np.transpose(data)
    newData = w * data
    newData = np.transpose(newData)
    newData = np.array(newData)
    return newData

def addLabels(data, trainLabels):
    b = np.zeros((data.shape[0], data.shape[1] + 1))
    b[:, :-1] = data
    b[:, -1] = trainLabels
    return b

def getDataMatrix(file, intOrFloat):
    #intOrFloat decides whether data should be int or float
    if (intOrFloat == 1):
        featureVectors = []
        for line in file :
            vector = line.strip().lower().split(' ')
            featureVectors.append(vector)
        data = np.array(featureVectors)
        data = data.astype(float)
    else:
        trainLabels = []
        for line in file :
            vector = line
            trainLabels.append(vector)
        data = np.array(trainLabels)
        data = data.astype(int)
    return data


file = open('arcene_train.data.txt')
data = getDataMatrix(file, 1)
file = open('arcene_train.labels.txt')
trainLabels = getDataMatrix(file, 0)
file = open('arcene_valid.data.txt')
testData = getDataMatrix(file, 1)
file = open('arcene_valid.labels.txt')
testLabels = getDataMatrix(file, 0)

#LDA
trainData = addLabels(data, trainLabels)
ev = ldaTransform(trainData)
trainData = trainData[:, :-1]
trainData = project(trainData, ev)
testData = project(testData, ev)
trainData = addLabels(trainData, trainLabels)
testData = addLabels(testData, testLabels)
C0 = trainData[trainData[:, -1] == -1]
C1 = trainData[trainData[:, -1] == 1]
V0 = findVariance(C0[:, :-1])
V1 = findVariance(C1[:, :-1])
M0 = findMean(C0[:, :-1])
M1 = findMean(C1[:, :-1])
pr0 = float(C0.shape[0])/float(trainData.shape[0])
pr1 = float(C1.shape[0])/float(trainData.shape[0])
L = math.pow(10, -323)
```

```
MAX = -math.pow(10, 300)

#Testing phase
totalValues = testData.shape[0]
myPrediction = np.zeros([totalValues])
j = 0
for i in xrange(0, totalValues):
    sample = testData[i, :]
    sample = sample.tolist()
    ans0 = math.log(float(pr0))
    ans1 = math.log(float(pr1))
    count = 0;
    g1 = gaussian(sample[j], M0, V0)
    g2 = gaussian(sample[j], M1, V1)
    if (g1 < L):
        ans0 = MAX
    if(g2 < L):
        ans1 = MAX
    if(ans0 > MAX):
        ans0 = ans0 + math.log(g1)
    if(ans1 > MAX):
        ans1 = ans1 + math.log(g2)
    print "ans0 ", ans0
    print "ans1 ", ans1
    if (ans0 > ans1):
        myPrediction[i] = int(-1)
    else:
        myPrediction[i] = int(1)

trueAns = testData[:, -1]
correctValues = 0
for i in range(totalValues):
    if (myPrediction[i] == trueAns[i]):
        correctValues = correctValues + 1
correctValues = float(correctValues)
totalValues = float(totalValues)
accuracy = correctValues/totalValues * 100
print accuracy
```

References

1. S. Moro, P. Cortez and P. Rita. A Data-Driven Approach to Predict the Success of Bank Telemarketing. Decision Support Systems, Elsevier, 62:22-31, June 2014
2. Isabelle Guyon, Steve R. Gunn, Asa Ben-Hur, Gideon Dror, 2004. Result analysis of the NIPS 2003 feature selection challenge.