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, categorcial
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 this 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:

```
Time taken - 33.201s
Written in 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]
```

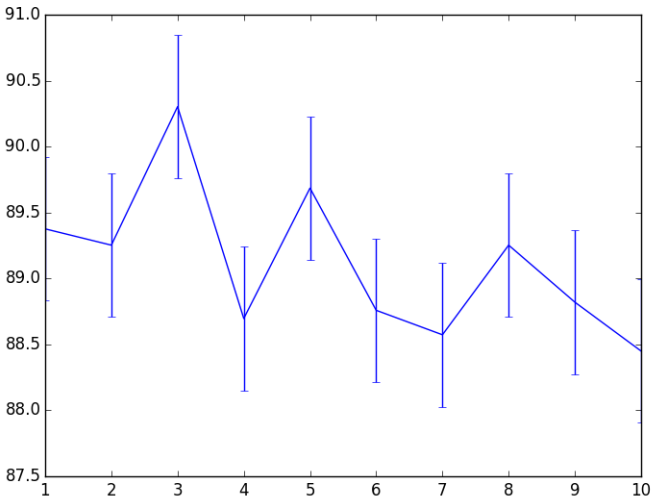
```
extractedData = mat[:,idx_IN_columns]
return extractedData

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-full.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)

#Training phase
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]
myPrediction = np.zeros([totalValues])
for i in xrange(0, totalValues):
sample = testData[i, :]
sample = sample.tolist()
ans0 = (float(pr0))
ans1 = (float(pr1))
for j in xrange(0, len(sample) - 1):
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]
break
if (ans0 > ans1):
myPrediction[i] = int(0)
elif(ans0 < ans1):
myPrediction[i] = int(1)
else:
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
results[t] = accuracy

meanAccuracy = np.mean(results)
stD = np.std(results)
print meanAccuracy
print stD
```

Results:

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



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

Bayesian estimation techniques to calculate the a posterior density $p(|D)$ and the desired probability density $p(x|D)$ for the case where $p(x|) \sim N(\mu, \sum)$.

Univariate case:

Consider case where μ is the only unknown parameter.

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

We assume that whatever prior knowledge we might have about can be expressed by a known prior density $p(\mu)$.
Assumption:

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

where both μ_0 and σ_0^2 are known.
A value is drawn for μ from a population governed by the probability law $p(\mu)$. Once this value is drawn, it becomes the true value of μ and completely determines the density for x. Samples x1, ..., xn are independently drawn from the resuling population. Letting $D = x_1, ..., x_n$, we use Bayes' formula to obtain:

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

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

where α is a normalization factor that depends on D but is independent of μ. Since, $p(\mu) \sim N(\mu_0, \sigma_0^2)$, and $p(x_k|\mu) \sim N(\mu, \sigma^2)$ we have,

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

which implies,

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

where factors that do not depend on μ have been absorbed into the constants α,$\alpha^{'}$, and $\alpha^{''}$. If we write density$p(\mu|D) \sim N(\mu_n, \sigma_n^2)$, then n and n2 can be found by equating coefficients in above equation with corresponding coefficients in the generic Gaussian of the form:

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

Identifying coefficients in this way yields,

$$\frac{1}{\sigma_n^2} = \frac{n}{\sigma^2} + \frac{1}{\sigma_0^2},$$

and

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

where $\bar{x_n}$ is the sample mean.
We solve explicitly for μ_n and σ_n^2 and obtain

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

and

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

Using these values of μ and σ, we get

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

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,

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

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	58.0	16.39
1000	58.0	17.01

LDA

Mean Accuracy	Time(m)
58.0	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

```
 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):

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)
newData = eigenVectors * 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

#Preprocessing
file = open('arcene_train.data.txt')
featureVectors = []
for line in file :
vector = line.strip().lower().split(' ')
featureVectors.append(vector)
data = np.array(featureVectors)
data = data.astype(float)
file = open('arcene_train.labels.txt')
trainLabels = []
for line in file :
```

```
vector = line
trainLabels.append(vector)
trainLabels = np.array(trainLabels)
trainLabels = trainLabels.astype(int)
file = open('arcene_valid.data.txt')
featureVectors = []
for line in file :
vector = line.strip().lower().split(' ')
featureVectors.append(vector)
testData = np.array(featureVectors)
testData = data.astype(float)
file = open('arcene_train.labels.txt')
trainLabels = []
for line in file :
vector = line
trainLabels.append(vector)
trainLabels = np.array(trainLabels)
trainLabels = trainLabels.astype(int)
file = open('arcene_valid.labels.txt')
testLabels = []
for line in file :
vector = line
testLabels.append(vector)
testLabels = np.array(testLabels)
testLabels = testLabels.astype(int)
#PCA
k = 1000
trainData = pcaTransform(data, k)
trainData = addLabels(trainData, trainLabels)
testData = pcaTransform(testData, k)
testData = addLabels(testData, testLabels)
f = open('trainData.txt', w)
for i in xrange(trainData.shape[0]):
for j in xrange(trainData.shape[1]):
f.write(str(trainData[i,j]) + " ")
f.write("\n")
f.close()

f = open('testData.txt', w)
for i in xrange(testData.shape[0]):
for j in xrange(testData.shape[1]):
f.write(str(testData[i,j]) + " ")
f.write("\n")
f.close()

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])
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)
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
```

LDA

```
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 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
print "SW - "
print SW
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)
print "SB -"
print SB
Swin = LA.pinv(SW) #costly
Swin = np.matrix(Swin)
SwinSB = Swin * SB #costly
print "SWinSB - "
```

```
print SwinSB
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)
l = data[:, -1]
data = data[:, :-1]
data = np.matrix(data)
data = np.transpose(data)
newData = w * data
newData = np.transpose(newData)
newData = np.array(newData)
newData = addLabels(newData, l)
print 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


file = open('arcene_train.data.txt')
featureVectors = []
for line in file :
vector = line.strip().lower().split(' ')
featureVectors.append(vector)
data = np.array(featureVectors)
data = data.astype(float)
preprocessing.scale(data, axis=0, with_mean=True, with_std=True, copy=False)
file = open('arcene_valid.data.txt')
featureVectors = []
for line in file :
vector = line.strip().lower().split(' ')
featureVectors.append(vector)
testData = np.array(featureVectors)
testData = data.astype(float)
preprocessing.scale(data, axis=0, with_mean=True, with_std=True, copy=False)
file = open('arcene_train.labels.txt')
trainLabels = []
for line in file :
```

```
vector = line
trainLabels.append(vector)
trainLabels = np.array(trainLabels)
trainLabels = trainLabels.astype(int)
file = open('arcene_valid.labels.txt')
testLabels = []
for line in file :
vector = line
testLabels.append(vector)
testLabels = np.array(testLabels)
testLabels = testLabels.astype(int)
#LDA
trainData = addLabels(data, trainLabels)
trainData = ldaTransform(trainData)
testData = addLabels(testData, testLabels)
testData = ldaTransform(testData)

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.