ENEL4AA - Design and Analysis of Algorithms
Assignment 2: Naive Bayes Classifier

Name: Keshav Jeewanlall **Student Number: 213508238** **Date: 10-05-2019**

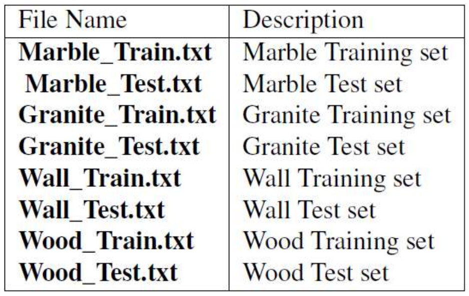
Abstract:* *This report entails a detailed description of a C++ based software that uses Naïve Bayes classifier that can be trained and used to recognise items for which features have been extracted and saved as vectors in a text file. The report covers all theory required to implement this program as well as well explained examples. The algorithms and methods used in this program are explicitly documented. Testing and results are recorded and tabulated in order for a thorough analysis.

1. INTRODUTION

Naive Bayes is a probabilistic machine learning algorithm that can be used in a wide variety of classification tasks. It assumes the features that go into the model are independent of each other. That is changing the value of one feature, does not directly influence or change the value of any of the other features used in the algorithm [1].

This report discusses the design, analysis and implementation of a general purpose system that uses Naïve Bayes classifier that can be trained and used to recognise items for which features have been extracted and saved as vectors in a text file. The system is then evaluated using a dataset, which is split into training and testing data subsets.

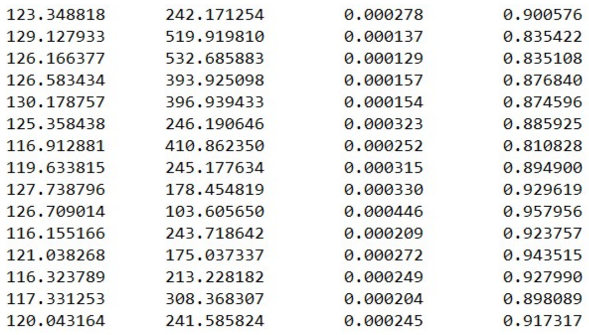
The dataset used consists of a collection of features extracted from a set of textures. These are textures of Granite, Marble, Wood and Wall. They are arbitrarily divided in 8 ASCII text files as follows [2]:



File Name	Description
Marble_Train.txt	Marble Training set
Marble_Test.txt	Marble Test set
Granite_Train.txt	Granite Training set
Granite_Test.txt	Granite Test set
Wall_Train.txt	Wall Training set
Wall_Test.txt	Wall Test set
Wood_Train.txt	Wood Training set
Wood_Test.txt	Wood Test set

Figure 1: List of ASCII text files used [2]

Each of these files is made of records of 4 components (real numbers). A file will look like in Figure 2.

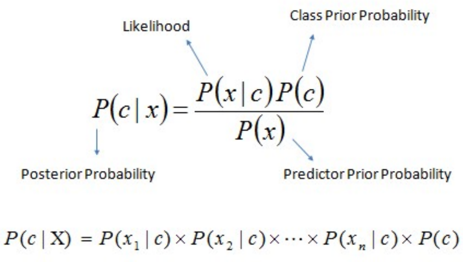


```
123.348818     242.171254     0.000278     0.900576
129.127933     519.919810     0.000137     0.835422
126.166377     532.685883     0.000129     0.835108
126.583434     393.925098     0.000157     0.876840
130.178757     396.939433     0.000154     0.874596
125.358438     246.190646     0.000323     0.885925
116.912881     410.862350     0.000252     0.810828
119.633815     245.177634     0.000315     0.894900
127.738796     178.454819     0.000330     0.929619
126.709014     103.605650     0.000446     0.957956
116.155166     243.718642     0.000209     0.923757
121.038268     175.037337     0.000272     0.943515
116.323789     213.228182     0.000249     0.927990
117.331253     308.368307     0.000204     0.898089
120.043164     241.585824     0.000245     0.917317
```

Figure 2: Example of representation characteristics of textures

2. MATERIALS AND METHODS

Bayes theorem provides a way of calculating the posterior probability, $P(c|x)$, from $P(c)$, $P(x)$, and $P(x|c)$. Naive Bayes classifier assume that the effect of the value of a predictor (x) on a given class (c) is independent of the values of other predictors. This assumption is called class conditional independence [3].



$$P(c\mid x)=\frac{P(x\mid c)P(c)}{P(x)}$$

$$P(c\mid X) = P(x_1\mid c)\times P(x_2\mid c)\times\cdots\times P(x_n\mid c)\times P(c)$$

Figure 3: [3]

- $P(c|x)$ is the posterior probability of *class* (*target*) given *predictor* (*attribute*).
- $P(c)$ is the prior probability of *class*.
- $P(x|c)$ is the likelihood which is the probability of *predictor* given *class*.
- $P(x)$ is the prior probability of *predictor*.

2.1. Training Stage

2.1.1. Computation of Priori Probabilities

If we consider that the classification problem is made of *n* classes and that $C_1, C_2, \ldots, C_n$ are the *n* classes, and $|C_i|$ is the number of items in C_i. We can then compute the total number of items as [2]:

$$TotalNbrOfItems = \sum_{i=1}^{n} |C_i| \qquad where\ i = 1,2,\ldots,n$$

The probability of each class can then be obtained as follows [2]:

$$P(C_i) = \frac{|C_i|}{TotalNbrOfItems} \qquad where\ i = 1,2,\ldots,n$$

2.1.2. Computation of Conditional Probabilities

If the components of the feature vectors are continuous, we can assume that the values have a Gaussian distribution with a mean m and a standard deviation σ defined by [2]

$$g(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} \exp(-\frac{(x-\mu)^2}{2\sigma^2})$$

So that given a component x_k of the feature vector $x = (x_1, x_2, \ldots, x_n)$ we have [2]

$$p(x_k|C_i) = g(x_k, \mu_{C_i}, \sigma_{C_i})$$

We need to compute μ_{C_i} and σ_{C_i}, which are the mean and standard deviation of the values of attribute A_k for training samples of class C_i [2].

2.1.3. Code Used to Construct the Knowledge Base Model

The following code is used for training to construct the knowledge base model

```
int train(texture* textureClass, char* trainingSet)
{
enum eMeanVariance { muOld, sigmaOld, muNew,
sigmaNew
};
double x[NumberOfAttributes] = { 0, };
double mu[NumberOfAttributes] = { 0, };
double sigma[NumberOfAttributes] = { 0, };
attrib* attribVector;

numberOfClassSamples = 0;
ifstream trainingData(trainingSet);

if (!trainingData)
{
printf("\n\rError opening training set file!\n\r");
system("PAUSE");
return -1;
}

while (trainingData >> x[0] >> x[1] >> x[2] >> x[3])
{
numberOfClassSamples++;
totalNumberofSamples++;
for (int i = 0; i < NumberOfAttributes; i++)
{
mu[i] += x[i];
}
}
(*textureClass).setOccurrences(numberOfClassSamples)
;

(*textureClass).setTextureProbability((double)number
OfClassSamples);
for (int i = 0; i < NumberOfAttributes; i++)
{
mu[i] = mu[i] / numberOfClassSamples;
}
trainingData.close();

trainingData.open(trainingSet);
if (!trainingData)
{
printf("\n\rError opening training set file!\n\r");
system("PAUSE");
return -1;
}

while (trainingData >> x[0] >> x[1] >> x[2] >> x[3])
{
for (int i = 0; i < NumberOfAttributes; i++)
{
sigma[i] += pow(x[i] - mu[i], 2);
}
}

for (int i = 0; i < NumberOfAttributes; i++)
{
sigma[i] = sqrt(sigma[i] / numberOfClassSamples);
}
trainingData.close();

attribVector = (*textureClass).X.getAttributes();
for (int i = 0; i < NumberOfAttributes; i++)
{
attribVector[i].setMean(mu[i]);

attribVector[i].setStandardDeviation(sigma[i]);
}

return 0;
}
```

1.1. Testing Stage

1.1.1. Classifying Items

A new item from the test set which is represented by the vector $X = (x_1, x_2, \ldots, x_n)$, X is predicted to belong to the class C_i if and only if [2]

$$P(C_i|X) > P(C_j|X) \quad 1 \leq j \leq m, j \neq i$$

As mentioned above, Bayes' theorem enables the computation of a posteriori class probabilities as follows: [2,3]

$$P(C_i|X) = \frac{P(X|C_i)P(C_i)}{P(X)}$$

The class label of X is C_i if and only if it is the class that maximizes $P(X|C_i)P(C_i)$ [2].

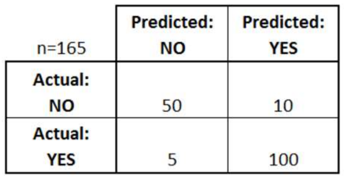
1.1.2. Code Used to Construct the Classification Model.

The following code is used for testing to construct the classification model

```
int classify(texture* TextureClass, double
x[NumberOfAttributes])
{
int mosTrueProbableClass = -1;
double posTrueProbability = -1;
double maxPosTrueProbability = -1;

for (int i = 0; i < NumberOfClasses; i++)
{
TextureClass[i].X.setMeasuredValue(x);
                posTrueProbability =
(TextureClass[i].getTextureProbability()) *
(TextureClass[i].X.ConditionallyIndependenTrueProbab
ilty());

if (posTrueProbability > maxPosTrueProbability)
{
maxPosTrueProbability = posTrueProbability;
mosTrueProbableClass = i;
}
}

return mosTrueProbableClass;
}
```

1.2. Confusion Matrix

A confusion matrix is a matrix which is used for determining the performance of a classification model [4].



n=165	Predicted: NO	Predicted: YES
Actual: NO	50	10
Actual: YES	5	100

Figure 4: Confusion Matrix for binary classifier [6]

Figure 4 shows the confusion matrix for a binary classifier. It can be seen from the figure that rows represent the actual values while the columns represent the predicted values.

The confusion matrix can be used to calculate performance metrics of the classification model such as [4]:

- True positives (TP) – predicted that it was in a class and it was.
- True negatives (TN) – predicted that it was not in a class and it wasn't.
- False positives (FP) – predicted that it was in a class and it wasn't.
- False negatives (FN) – predicted that it wasn't in a class and it was.
- True positive rate (TPR) – How often does it predict that it is in a class and it is.
$$\frac{TP}{P}$$
- False positive rate (FPR) – How often does it predict that it is in a class when it isn't.
$$\frac{FP}{N}$$
- Accuracy – How often is the prediction correct
$$\frac{(TP + TN)}{Total}$$

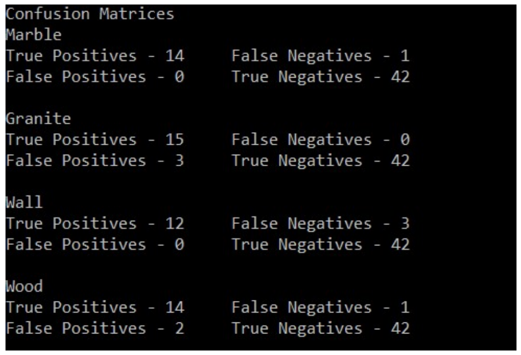
1.3. Visual Studio

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. Visual Studio 2017 is used in this project for programming and running the software. The software is to be developed using the C++ programming language

2. ANALYSIS AND DISCUSSION

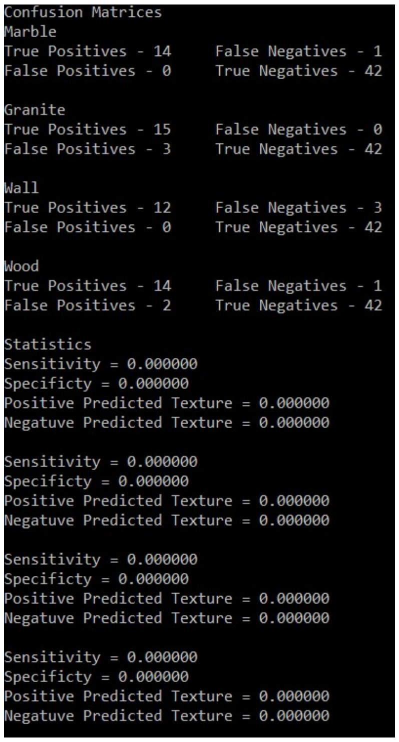
2.1. Program Output

Each texture had 60 test features in each test. Figure 5 displays the confusion matrices for the results of the tests for all 4 features and figure 6 shows the entire output of the program.



```
Confusion Matrices
Marble
True Positives - 14     False Negatives - 1
False Positives - 0     True Negatives - 42

Granite
True Positives - 15     False Negatives - 0
False Positives - 3     True Negatives - 42

Wall
True Positives - 12     False Negatives - 3
False Positives - 0     True Negatives - 42

Wood
True Positives - 14     False Negatives - 1
False Positives - 2     True Negatives - 42
```

Figure 5: Confusion Matrices Output



```
Confusion Matrices
Marble
True Positives - 14     False Negatives - 1
False Positives - 0     True Negatives - 42

Granite
True Positives - 15     False Negatives - 0
False Positives - 3     True Negatives - 42

Wall
True Positives - 12     False Negatives - 3
False Positives - 0     True Negatives - 42

Wood
True Positives - 14     False Negatives - 1
False Positives - 2     True Negatives - 42

Statistics
Sensitivity = 0.000000
Specificty = 0.000000
Positive Predicted Texture = 0.000000
Negatuve Predicted Texture = 0.000000

Sensitivity = 0.000000
Specificty = 0.000000
Positive Predicted Texture = 0.000000
Negatuve Predicted Texture = 0.000000

Sensitivity = 0.000000
Specificty = 0.000000
Positive Predicted Texture = 0.000000
Negatuve Predicted Texture = 0.000000

Sensitivity = 0.000000
Specificty = 0.000000
Positive Predicted Texture = 0.000000
Negatuve Predicted Texture = 0.000000
```

Figure 6

2.2. Analysis of Confusion Matrices

Marble:

Table 1: Marble Confusion Matrix

n=60	**Predicted: False**	**Predicted: True**
Actual: False	TN = 42	FP = 0
Actual: True	FN = 1	TP = 14

From the confusion matrix (Table 1), the following metrics are calculated:

$$FP\ rate = \frac{FP}{N} = \frac{0}{42} = 0$$
$$TP\ rate = \frac{TP}{P} = \frac{14}{15} = 0.933$$
$$Accuracy = \frac{TP + TN}{Total} = \frac{14 + 42}{60} = 0.933 \ \approx 93.3\%$$

Granite:

Table 2: Granite Confusion Matrix

n=60	**Predicted: False**	**Predicted: True**
Actual: False	TN = 42	FP = 3
Actual: True	FN = 0	TP = 15

From the confusion matrix (Table 2), the following metrics are calculated:

$$FP\ rate = \frac{FP}{N} = \frac{3}{45} = 0.067$$
$$TP\ rate = \frac{TP}{P} = \frac{15}{15} = 1$$
$$Accuracy = \frac{TP + TN}{Total} = \frac{15 + 42}{60} = 0.95 \ \approx 95\%$$

Wall:

Table 3: Wall Confusion Matrix

n=60	**Predicted: False**	**Predicted: True**
Actual: False	TN = 42	FP = 0
Actual: True	FN = 3	TP = 12

From the confusion matrix (Table 1), the following metrics are calculated:

$$FP\ rate = \frac{FP}{N} = \frac{0}{42} = 0$$
$$TP\ rate = \frac{TP}{P} = \frac{12}{15} = 0.8$$
$$Accuracy = \frac{TP + TN}{Total} = \frac{14 + 42}{60} = 0.933 \ \approx 93.3\%$$

Wood:

Table 4: Wood Confusion Matrix

n=60	**Predicted: False**	**Predicted: True**
Actual: False	TN = 42	FP = 0
Actual: True	FN = 1	TP = 14

From the confusion matrix (Table 1), the following metrics are calculated:

$$FP\ rate = \frac{FP}{N} = \frac{0}{42} = 0$$
$$TP\ rate = \frac{TP}{P} = \frac{14}{15} = 0.933$$
$$Accuracy = \frac{TP + TN}{Total} = \frac{14 + 42}{60} = 0.933 \ \approx 93.3\%$$

From the results above it can be seen that the Naïve Bayes classifier has a high classification accuracy.

3. CONCLUSION

The system was able to analyse the feature characteristics of each texture and classify them into the four categories. From the analysis of the accuracy of Naïve Bayes classifier it was found that the software is able to classify the features with an average of 93.7%. These results prove that Naive Bayes is an effective machine learning solution to the problem of classification.

4. REFERENCES

[1] "How Naive Bayes Algorithm Works?," Machine Learning Plus, [Online]. Available: https://www.machinelearningplus.com/predictive-modeling/how-naive-bayes-algorithm-works-with-example-and-full-code/. [Accessed 30 April 2019].

[2] J. R. Tapamo, "ENEL4AA - Design and Analysis of Algorithms - 2018. Assignment 2: Naive Bayes Classifier," 2019.

[3] "Naive Bayesian," [Online]. Available: https://www.saedsayad.com/naive_bayesian.htm. [Accessed 30 April 2019].

[4] "Simple guide to confusion matrix terminology," [Online]. Available: https://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/. [Accessed 7 May 2019].

5. APPENDIX

```
/***************************************************
*********************
* ENEL4AA - Design and Analysis of Algorithms
* Assignment 2: Naive Bayes Classifier
* Keshav Jeewanlall 213508238
* 10 May 2019
***************************************************
********************/

#ifndef __ATTRIB_H__
#define __ATTRIB_H__

#define _USE_MATH_DEFINES
#include<math.h>

class attrib
{
private: double mu;
private: double sigma;
private: char* attribName;


public: attrib(void)
{
	setAttributeName("Unkown Attribute");
	setMean(0);
	setStandardDeviation(1);
}

public: attrib(char* name)
{
	setAttributeName(name);
	setMean(0);
	setStandardDeviation(1);
}

public: void setAttributeName(char* attrName)
{
	attribName = attrName;
}

public: char* getAttributeName(void)
{
	return attribName;
}

public: void setMean(double mean)
{
	mu = mean;
}


public: double getMean(void)
{
	return mu;
}

public: void setStandardDeviation(double
standardDeviation)
{
	sigma = standardDeviation;
}

public: double getStandardDeviation(void)
{
	return sigma;
}

public: double gaussian(double x)
{

	double z = (x - mu) / (sigma);
	return (1 / sqrt(2 *
M_PI*sigma*sigma))*(exp(-(pow(z, 2)) / 2));
}

public: void printAttributeData()
{
	printf("\n\rName:%s\nMean:%f\nStandard
Deviation:%f",
		getAttributeName(), getMean(),
getStandardDeviation());
}
};

#endif

/***************************************************
*********************
* ENEL4AA - Design and Analysis of Algorithms
* Assignment 2: Naive Bayes Classifier
* Keshav Jeewanlall 213508238
* 10 May 2019
***************************************************
********************/

#ifndef __FEATURE_H__
#define __FEATURE_H__

#include "attributes.h"

class feature
{
#define NumberOfAttributes	(4)

private: attrib attributes[NumberOfAttributes];
private: double
measuredFeatureVector[NumberOfAttributes];


public: feature(void)
{

}

public: void setMeasuredValue(double *sample)
{
	for (int attributeCounter = 0;
attributeCounter < NumberOfAttributes;
attributeCounter++)
	{

	measuredFeatureVector[attributeCounter] =
sample[attributeCounter];
	}
}

public: void getMeasuredValue(double *returnVector)
{
	for (int attributeCounter = 0;
attributeCounter < NumberOfAttributes;
attributeCounter++)
	{
		returnVector[attributeCounter] =
measuredFeatureVector[attributeCounter];
	}
}

public: int geTrueNumberofAttributes(void)
{
	return NumberOfAttributes;
}

public: double
ConditionallyIndependenTrueProbabilty(void)
{
        double P = 1;
        for (int attributeCounter = 0;
attributeCounter < NumberOfAttributes;
attributeCounter++)
        {
                P *=
attributes[attributeCounter].gaussian(measuredFeatur
eVector[attributeCounter]);
        }
        return P;
}

public: attrib* getAttributes(void)
{
        return attributes;
}


public: void printFeatureData(void)
{
        for (int attributeCounter = 0;
attributeCounter < NumberOfAttributes;
attributeCounter++)
        {

        attributes[attributeCounter].printAttributeD
ata();
        }
}
};

#endif

/***************************************************
*********************
* ENEL4AA - Design and Analysis of Algorithms
* Assignment 2: Naive Bayes Classifier
* Keshav Jeewanlall 213508238
* 10 May 2019
****************************************************
********************/


#include "features.h"

class texture
{
public: feature X;
private: char* textureName;
private: double probablityOfTexture;
private: int occurrences;

public: texture(void)
{
        setTextureName("Unkown Texture");
        setTextureProbability(0);
}


public: texture(char* name)
{
        setTextureName(name);
        setTextureProbability(0);
}


public: void setTextureName(char* name)
{
        textureName = name;
}

public: char* getTextureName(void)
{
        return textureName;
}

public: void setTextureProbability(double prob)
{
        if (prob >= 0 && prob <= 1)
        {
                probablityOfTexture = prob;
        }
        else
        {
                probablityOfTexture = -1;
        }
}


public: double getTextureProbability(void)
{
        return probablityOfTexture;
}

public: void setOccurrences(int numberOfTimes)
{
        if (numberOfTimes >= 0)
        {
                occurrences = numberOfTimes;
        }
        else
        {
                occurrences = 0;
        }
}


public: int getOccurrences(void)
{
        return occurrences;
}

public: void printTextureData(void)
{
        printf("\n\rName of Texture:%s\nProbablity
of Texture:%f", getTextureName(),
getTextureProbability());
}
};


/***************************************************
*********************
* ENEL4AA - Design and Analysis of Algorithms
* Assignment 2: Naive Bayes Classifier
* Keshav Jeewanlall 213508238
* 10 May 2019
****************************************************
********************/

#include <stdio.h>
#include <fstream>
#include "attributes.h"
#include "features.h"
#include "textures.h"

using namespace std;

#define NumberOfClasses (4)
#define False                   (0)
#define True                    (1)

enum TextureType { MARBLE, GRANITE, WALL, WOOD };

int train(texture* textureClass, char* trainingSet);
int classify(texture* TextureClass, double
x[NumberOfAttributes]);
```

```
int confusion_matrix(char* testFile, texture*
TextureClass, int
confusion_Matrix[NumberOfClasses][2][2], int
classType);
double sensitivity(int confusion_Matrix[2][2]);
double specificity(int confusion_Matrix[2][2]);
double positive_predictive_value(int
confusion_Matrix[2][2]);
double negative_predictive_value(int
confusion_Matrix[2][2]);

int totalNumberofSamples = 0;
int numberOfClassSamples = 0;

int main(void)
{
        texture classSet[NumberOfClasses];
        attrib* attribVector;
        int positives[NumberOfClasses][2][2] = { 0,
};

        classSet[MARBLE].setTextureName("Marble");
        classSet[GRANITE].setTextureName("Granite");
        classSet[WALL].setTextureName("Wall");
        classSet[WOOD].setTextureName("Wood");

        train(&(classSet[MARBLE]),
"train_data/Marble_Train.txt");
        train(&(classSet[GRANITE]),
"train_data/Granite_Train.txt");
        train(&(classSet[WALL]),
"train_data/Wall_Train.txt");
        train(&(classSet[WOOD]),
"train_data/Wood_Train.txt");

        for (int i = 0; i < NumberOfClasses; i++)
        {

        classSet[i].setTextureProbability((double)cl
assSet[MARBLE].getOccurrences() /
totalNumberofSamples);
        }

        confusion_matrix("test_data/Marble_Test.txt"
, &(classSet[MARBLE]), positives, MARBLE);
        confusion_matrix("test_data/Granite_Test.txt
", &(classSet[MARBLE]), positives, GRANITE);
        confusion_matrix("test_data/Wall_Test.txt",
&(classSet[MARBLE]), positives, WALL);
        confusion_matrix("test_data/Wood_Test.txt",
&(classSet[MARBLE]), positives, WOOD);

        printf("Confusion Matrices");
        for (int i = 0; i < NumberOfClasses; i++)
        {
                printf("\n%s",
classSet[i].getTextureName());
                printf("\nTrue Positives - %d
False Negatives - %d", positives[i][0][0],
positives[i][0][1]);
                printf("\nFalse Positives - %d
True Negatives - %d", positives[i][1][0],
positives[1][1][1]);
                printf("\n");
        }

        printf("\nStatistics");
        for (int i = 0; i < NumberOfClasses; i++)
        {
                printf("\nSensitivity = %f",
classSet[i].getTextureName(), ">",
sensitivity(positives[i]));
                printf("\nSpecificty = %f",
classSet[i].getTextureName(), ">",
specificity(positives[i]));
                printf("\nPositive Predicted Texture
= %f", classSet[i].getTextureName(), ">",
positive_predictive_value(positives[i]));
                printf("\nNegatuve Predicted Texture
= %f", classSet[i].getTextureName(), ">",
negative_predictive_value(positives[i]));
                printf("\n");
        }

        printf("\n");

        system("PAUSE");
        return 0;
}

int train(texture* textureClass, char* trainingSet)
{
        enum eMeanVariance { muOld, sigmaOld, muNew,
sigmaNew };
        double x[NumberOfAttributes] = { 0, };
        double mu[NumberOfAttributes] = { 0, };
        double sigma[NumberOfAttributes] = { 0, };
        attrib* attribVector;

        numberOfClassSamples = 0;
        ifstream trainingData(trainingSet);

        if (!trainingData)
        {
                printf("\n\rError opening training
set file!\n\r");
                system("PAUSE");
                return -1;
        }

        while (trainingData >> x[0] >> x[1] >> x[2]
>> x[3])
        {
                numberOfClassSamples++;
                totalNumberofSamples++;
                for (int i = 0; i <
NumberOfAttributes; i++)
                {
                        mu[i] += x[i];
                }
        }

        (*textureClass).setOccurrences(numberOfClass
Samples);

        (*textureClass).setTextureProbability((doubl
e)numberOfClassSamples);
        for (int i = 0; i < NumberOfAttributes; i++)
        {
                mu[i] = mu[i] /
numberOfClassSamples;
        }
        trainingData.close();

        trainingData.open(trainingSet);
        if (!trainingData)
        {
                printf("\n\rError opening training
set file!\n\r");
                system("PAUSE");
                return -1;
        }

        while (trainingData >> x[0] >> x[1] >> x[2]
>> x[3])
        {
```

```
		for (int i = 0; i <
NumberOfAttributes; i++)
		{
			sigma[i] += pow(x[i] -
mu[i], 2);
		}
	}

	for (int i = 0; i < NumberOfAttributes; i++)
	{
		sigma[i] = sqrt(sigma[i] /
numberOfClassSamples);
	}
	trainingData.close();

	attribVector =
(*textureClass).X.getAttributes();
	for (int i = 0; i < NumberOfAttributes; i++)
	{
		attribVector[i].setMean(mu[i]);

	attribVector[i].setStandardDeviation(sigma[i
]);
	}

	return 0;
}

int classify(texture* TextureClass, double
x[NumberOfAttributes])
{
	int mosTrueProbableClass = -1;
	double posTrueProbability = -1;
	double maxPosTrueProbability = -1;

	for (int i = 0; i < NumberOfClasses; i++)
	{

	TextureClass[i].X.setMeasuredValue(x);
		posTrueProbability =
(TextureClass[i].getTextureProbability()) *
(TextureClass[i].X.ConditionallyIndependenTrueProbab
ilty());

		if (posTrueProbability >
maxPosTrueProbability)
		{
			maxPosTrueProbability =
posTrueProbability;
			mosTrueProbableClass = i;
		}
	}

	return mosTrueProbableClass;
}

int confusion_matrix(char* testFile, texture*
TextureClass, int
confusion_Matrix[NumberOfClasses][2][2], int
classType)
{
	double sampleValue[NumberOfAttributes] = {
0, };
	ifstream testData(testFile);

	if (!testData)
	{
		printf("\n\rError opening test set
file!\n\r");
		system("PAUSE");
		return -1;
	}

	int s = 2;

	while (testData >> sampleValue[0] >>
sampleValue[1] >> sampleValue[2] >> sampleValue[3])
	{
		int classifiedAs =
classify(TextureClass, sampleValue);

		if (classifiedAs == classType)
		{

	(confusion_Matrix[classType][0][0])++;
			for (int i = 0; i <
NumberOfClasses; i++)
			{
				if (i != classType)
				{

	(confusion_Matrix[i][1][1])++;
				}
			}
		}
		else
		{

	(confusion_Matrix[classifiedAs][1][0])++; //
it is False positive for the class that got
classified

	(confusion_Matrix[classType][0][1])++;// it
is a False negative for the class we're classifying
for
			for (int i = 0; i <
NumberOfClasses; i++) //and it's a True negative for
the remaining classes
			{
				if ((i !=
classifiedAs) && (i != classType))

	(confusion_Matrix[i][1][1])++;
			}
		}
	}

	testData.close();
	return 0;
}

double sensitivity(int confusion_Matrix[2][2])
{
	int TrueP = 0;
	int TrueP_FalseN = 0;


	TrueP = confusion_Matrix[0][0];
	TrueP_FalseN = confusion_Matrix[0][0] +
confusion_Matrix[0][1];

	if (TrueP_FalseN == 0)
	{
		return -1;
	}
	else
	{
		return (double)TrueP / TrueP_FalseN;
	}
}

double specificity(int confusion_Matrix[2][2])
{
	int TrueN = 0;
	int TrueN_FalseP = 0;


	TrueN = confusion_Matrix[1][1];
```

```
        TrueN_FalseP = confusion_Matrix[1][1] +
confusion_Matrix[1][0];

        if (TrueN_FalseP == 0)
        {
                -1;
        }
        else
        {
                return (double)TrueN / TrueN_FalseP;
        }
}

double positive_predictive_value(int
confusion_Matrix[2][2])
{
        int TrueP = 0;
        int TrueP_FlaseP = 0;


        TrueP = confusion_Matrix[0][0];
        TrueP_FlaseP = confusion_Matrix[0][0] +
confusion_Matrix[1][0];

        if (TrueP_FlaseP == 0)
        {
                return -1;
        }
        else
        {
                return (double)TrueP / TrueP_FlaseP;
        }
}

double negative_predictive_value(int
confusion_Matrix[2][2])
{
        int TrueN = 0;
        int TrueN_FlaseN = 0;


        TrueN = confusion_Matrix[1][1];
        TrueN_FlaseN = confusion_Matrix[1][1] +
confusion_Matrix[0][1];

        if (TrueN_FlaseN == 0)
        {
                return -1;
        }
        else
        {
                return (double)TrueN / TrueN_FlaseN;
        }
}
```