University of Glasgow

Artificial Intelligence – Assessed Exercise

Student:
Enzo Roiz - 2161561R

Lecturers:
Dr Alessandro Vinciarelli
Dr Maurizio Filippone

Introduction

During the semester a system was developed in the laboratory sessions in order to realise experiments covering the two main blocks of the course, sensing and reasoning. For the former, was expected to develop a system able to extract information such as energy, magnitude and zero crossing rate from audio signals given as a list of integer numbers in text files. For the latter, the system was to be able to represent the audio data considering properties such as average energy, average magnitude and average zero crossing rate, and also was to be capable of inferring if an audio sample is either speech or silence via statistical approaches. This report is divided into three chapters and an appendix, described below.

Chapter 1 – Design:
In this chapter is explained what the four *PEAS* framework elements are, and also is given a description of the experiments in *PEAS* terms.

Chapter 2 – Theory:
A detailed explanation of the signal processing approaches is given in this chapter, in particular for the extraction of energy, magnitude and zero crossing rate signals. Equations and plots are shown for a better understanding. Furthermore, the adopted statistical approaches are explained in detail, including equations and formulas, as well as the theory, assumptions and decision rule for applying the Naive Bayes Classifier.

Chapter 3 – Experiments:
In this chapter the adopted setup and results of the experiments are explained, taking into account the used cross validation and the performance metric adopted to measure the efficiency of the approaches used.

Appendix A – Code:
This appendix contains the source code of the developed system.

Chapter 1 – Design:

To design an intelligent agent, first is needed to specify its settings, which are defined in terms of four elements, in short, called *PEAS.* The acronym *PEAS* stands for **P**erformance, **E**nvironment, **A**ctuators and **S**ensors which are described below.

Performance: It defines how "successful" an agent is in performing its tasks. It must be objective, that is, it must be represented as a number.

Environment: It limits the circumstances that an agent can deal with, in other words, it is what the agent is interacting with. As there are several different environments for different agents, they can be distinguished by its characteristics as follows:

- *Fully (Accessible) vs Partially observable (Inaccessible):*
    - An environment is said to be fully observable when the agent's sensors can detect all the relevant information from the environment for choosing an action. Otherwise it is partially observable.
- *Deterministic vs Stochastic:*
    - If the next state of the environment is completely predictable from the current state and from the actions taken by an agent, the environment is said to be deterministic. Otherwise it is stochastic.
- *Episodic vs Sequential:*
    - If the agent's experience is divided into episodes, that is, the agent perceiving and then acting, and the actions taken by the agent depends only on the episode itself and does not affect future decisions, the environment is episodic. Otherwise it is sequential.
- *Static vs Dynamic:*
    - If the environment may to change over time, it is dynamic for the agent. Otherwise it is static. If the environment does not change over time, but the performance score does, the environment is then said to be semidynamic.
- *Discrete vs Continuous:*
    - If there are a limited number of steps, defined percepts and actions to take, then the environment is discrete. Otherwise it is continuous.
- *Single vs Multi agent:*
    - If an agent is operating by itself in an environment, the environment is said to be single agent. If there are other agents, then the environment is multi agent.

Actuators: An actuator is anything that can change the environment. It limits what the agent can do within the environment.

Sensors: It limits what the agent can know about the environment. The sensors are the input from the environment to the agent.

Considering the agent developed in the laboratory sessions, it is described below in terms of the *PEAS* framework:

Performance: As the goal of the agent developed is to infer via statistical approaches if an audio sample is either speech or silence, the adopted performance measurement was the percentage of correct inferences done by the agent.

Environment: The developed agent deals with text files with a list of integer numbers representing an audio signal. The environment characteristics are described in the table below:

Environment	**Accessible**	**Deterministic**	**Episodic**	**Static**	**Discrete**	**Single Agent**
Audio Inference System	Yes	No	No	Yes	Yes	Yes

Actuators: Monitor. The agent changes the environment by showing messages.

Sensors: File readers. The agent gets information from the environment by reading files.

Chapter 2 – Theory:

This chapter is divided into two parts. The first one consider the signal processing approaches adopted for extracting properties from the audio signals and the second one deals with the statistical approaches used to create an classifier to infer the type of an audio sample.

Signal Processing Approaches:

In this assessed exercise was of particular interest the *short term analysis.* In the short term analysis the extraction of information from the audio signals takes into account short segments (for this exercise: 30ms) that can be considered as sustained sound with stable properties, once the signals' properties change relatively slowly over the time and different segments of the signal have different properties. Mathematically talking, a property *Q[n]* at time *nT*, where *T = 1/F* is the sampling period can be expressed by:

$$Q[n] = \sum_{m=-\infty}^{\infty} K(s[m])w[n-m]$$

considering that *K* is a transform, that can be linear or non-linear and *w[n]* is the analysis window. For this exercise was used the *rectangular window* that is defined as follows:

$$w[n] = \begin{cases} 1 : 0 \leq l \leq N-1 \\ 0 : l < 0 \\ 0 : l \geq N \end{cases}$$

The window size used in the experiments was of, as said, 30ms which is the average duration of a phoneme, which are "atoms" composing sounds during speech. Moving on to the information extraction in a short-term approach, this exercise intends to extract some important properties from an audio signal. They are:

Short-time Energy:

The short-time energy can be extracted through the following convolution:

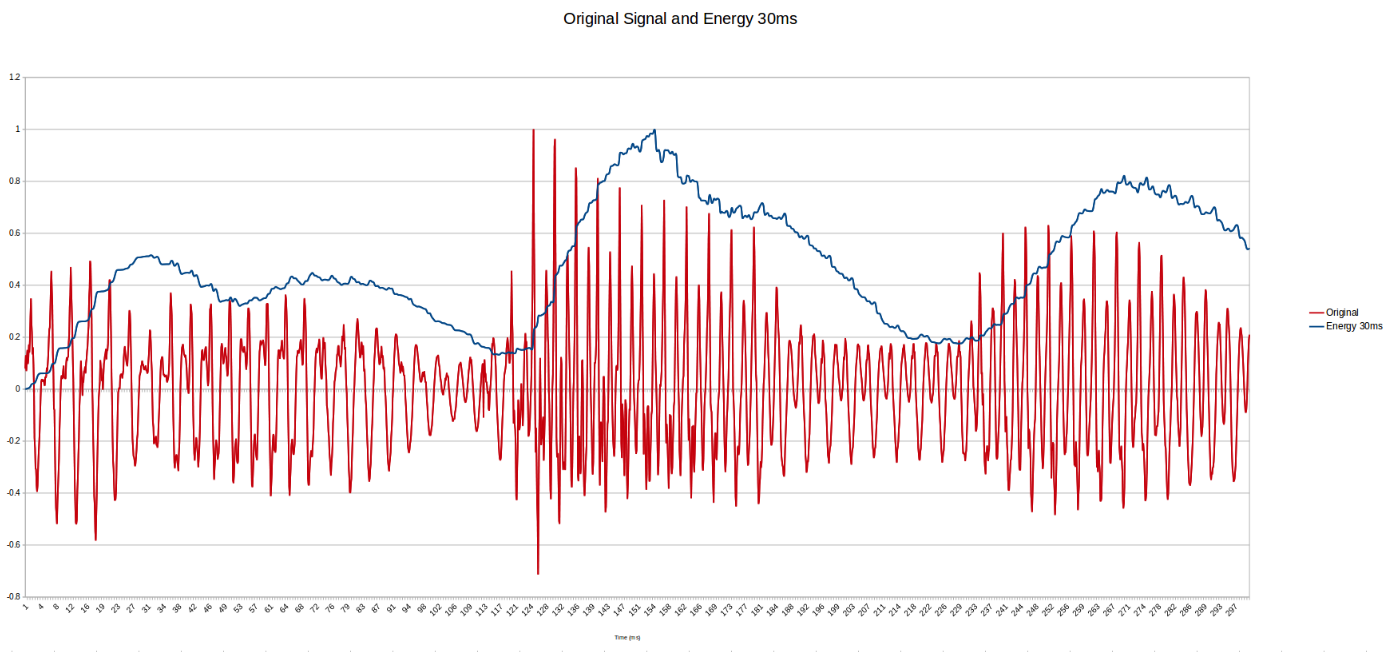
$$E[n] = \sum_{m=-\infty}^{\infty} s^2[n]w[n-m].$$

Short-time Average Magnitude:

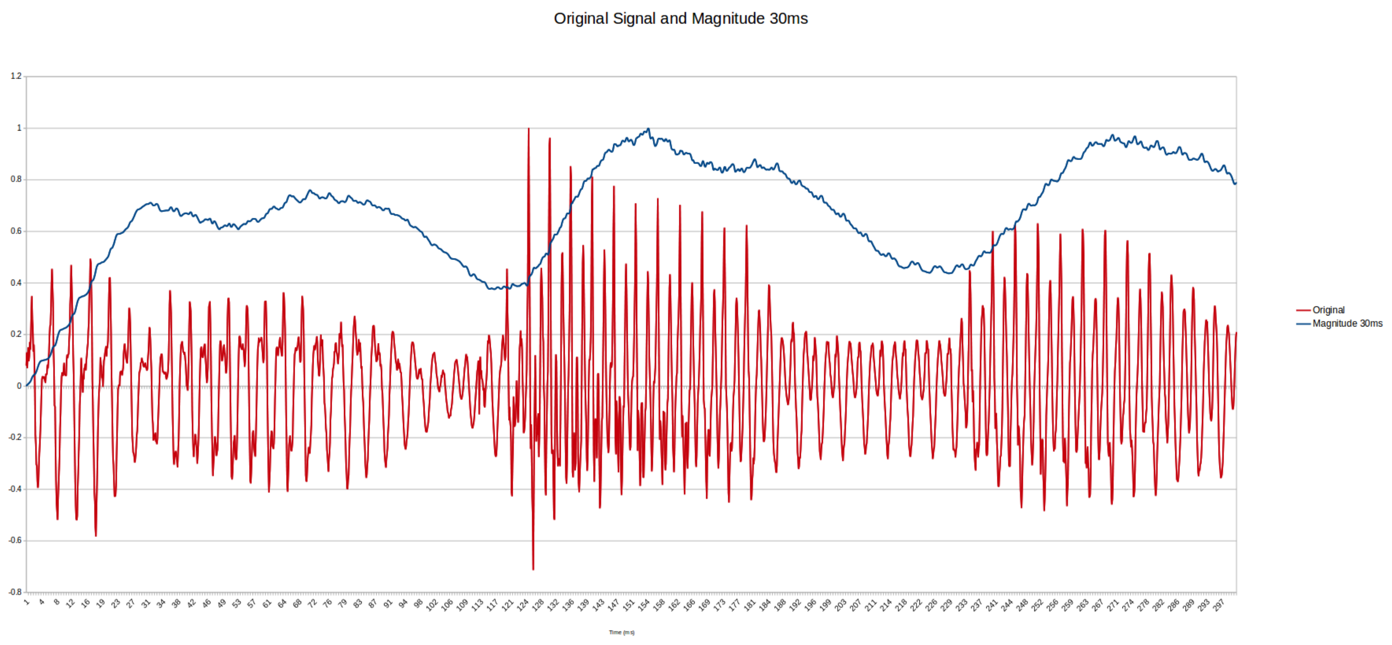
The short-time average magnitude can be extracted through the following convolution:

$$M[n] = \sum_{m=-\infty}^{\infty} |s[n]w[n-m]|.$$

Both of the properties, energy and average magnitude have the same type of information and are important on detecting if a sound is either voiced or unvoiced. However, the average magnitude is less sensitive to local fluctuations. Because of the use of the square E[n] tends to be more sensitive to the highest values of s[n] and supress the lowest values. Due to it, E[n] is often substituted by M[n], which has a smaller dynamic range and smoother differences if compared to E[n], what can be seen in the normalized graphs below that stands for E[n] and M[n] respectively compared to the original signal.



Graph 2.1



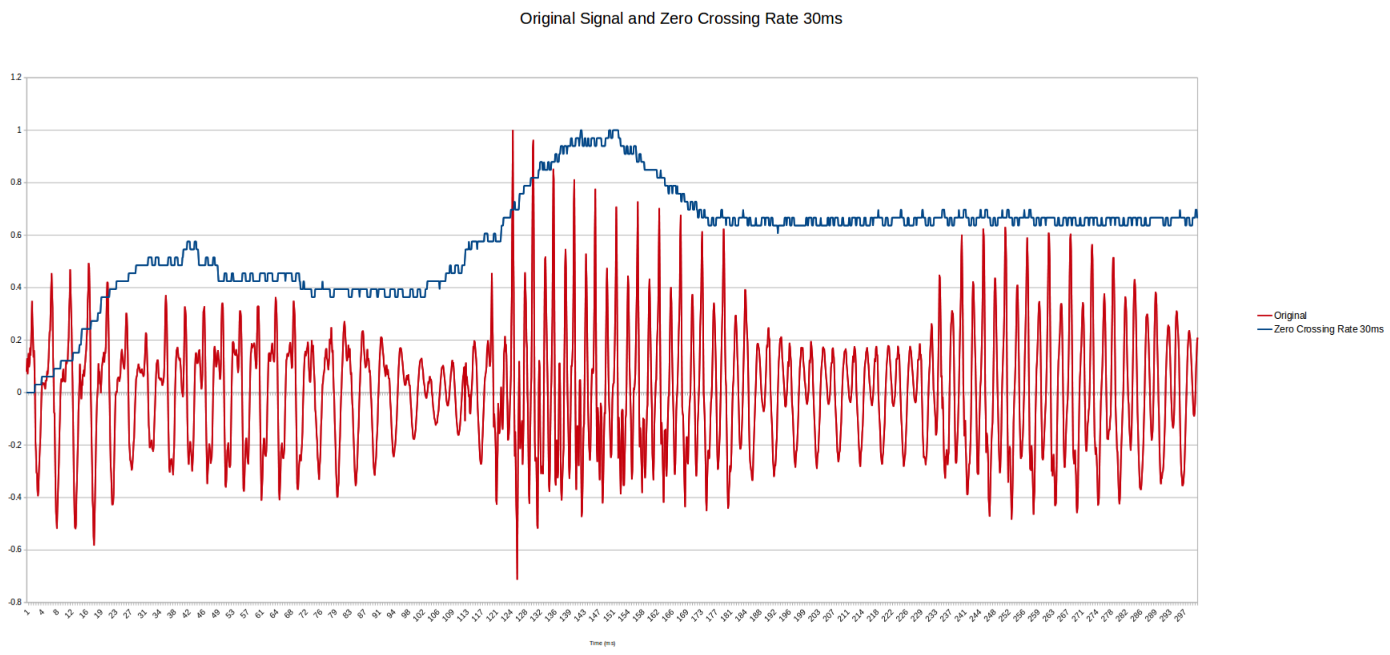
Graph 2.2

Short-time Average Zero Crossing Rate:

The average zero crossing rate can be obtained as follows:

$$Z[n] = \frac{1}{2N} \sum_{m=-\infty}^{\infty} |sign(s[m]) - sign(s[m-1])| w[n-m]$$

where *N* is the length of the window. The ZCR, allows us to have a rough idea of the frequencies represented in the data, since it is a rate of the signal changes along the signal. It has been used in different signal processing areas, for example, word-boundaries, speech music discrimination and audio classification. Below is the graph that shows the original signal and its zero crossing rate, both normalized.



Graph 2.3

Statistical Approaches:

Moving on to the statistical approaches adopted in this exercise, a Naive Bayes Classifier was developed for inferring if an audio sample is silence or speech. The Naive Bayes Classifier consists of applying the Bayes' theorem with independence assumptions between the attributes, that is, the value of a particular attribute is not related to any other attribute. The Bayes' theorem is as follows:

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}.$$

or in its normalized form, that gives 0 < *P(A|B)* < 1,

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B|A)P(A) + P(B|\neg A)P(\neg A)}.$$

For the exercise these attributes were the *short-time energy, the short-time magnitude* and the *short-time average zero crossing rate*, and the categories were *Silence* and *Speech*. Moreover, it was assumed that the prior probability of a sample to be silence or speech was *P(silence) = P(speech) = 0.5.* Considering the assumptions above, the conditional models are then given by:

$$p(Silence|Energy, Magnitude, ZCR)$$

and

$$p(Speech|Energy, Magnitude, ZCR)$$

Once the modelling is going to be the same for both, only the first one will be taken into account for the explanation of the model. Using the Bayes' theorem it can be written as:

$$p(Silence|Energy, Magnitude, ZCR) = \frac{p(Silence)p(Energy, Magnitude, ZCR|Silence)}{p(Energy, Magnitude, ZCR)}$$

Once the numerator is the joint distribution of

$$p(Silence, Energy, Magnitude, ZCR)$$

it can be rewritten, using the chain rule, as:

$$= p(Silence)p(Energy, Magnitude, ZCR|Silence)$$

$$= p(Silence)p(Energy|Silence)p(Magnitude, ZCR|Silence, Energy)$$

$$= p(Silence)p(Energy|Silence)p(Magnitude|Silence, Energy)p(ZCR|Silence, Energy, Magnitude)$$

However, as stated before, with the independence assumption, every attribute is independent from the others given *Silence* or *Speech*, which gives, for example:

$$p(Energy|Silence, Magnitude) = P(Energy|Silence)$$

$$p(Energy|Silence, Magnitude, ZCR) = P(Energy|Silence)$$

thus, finally, the conditional distribution can be written, using the normalized form of the Bayes' theorem, as:

$$posterior(Silence) = \frac{P(Silence)p(Energy|Silence)p(Magnitude|Silence)p(ZCR|Silence)}{evidence}$$

and applying the same for the Speech category:

$$posterior(Speech) = \frac{P(Speech)p(Energy|Speech)p(Magnitude|Speech)p(ZCR|Speech)}{evidence}$$

Equation 2.1

where the evidence is:

$$P(Silence)p(Energy|Silence)p(Magnitude|Silence)p(ZCR|Silence)+$$
$$P(Speech)p(Energy|Speech)p(Magnitude|Speech)p(ZCR|Speech)$$

Equation 2.2

As it was assumed that the values of the attributes (energy, magnitude, zero crossing rate) associated with a category (silence, speech) are distributed according to a Gaussian distribution, the Gaussian (or Normal) distribution was used to calculate each attribute probability in relation to a category. It is given by:

$$\frac{1}{\sqrt{2\pi\sigma^2}} e^{\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)}$$

Equation 2.3

where *x* is an attribute value, σ^2 is the variance of an attribute associated with a category, given by:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mu)^2$$

and *μ* is the mean of an attribute associated with a category.

$$\frac{1}{n}\sum_{i=1}^{n} x_i$$

Consider this hypothetical scenario, based on a previously known audio samples, the following information was obtained:

Audio Type	**E – mean**	**E – variance**	**M – mean**	**M – variance**	**Z – mean**	**Z – variance**
Silence	3.4448	0.2502	4.743	0.0471	1.381	0.1487
Speech	6.2174	1.4173	6.1252	0.3854	0.8012	0.1016

Suppose that a sample to be classified has the following attributes:

Energy: 3.7072
Magnitude: 4.8972
Zero Crossing Rate: 1.8845

In order to infer the type of the sample given, the speech and silence posteriors are calculated as follows:

P(Silence) = P(Speech) = 0.5 as assumed.
From Equation 2.3:

$$p(Energy|Silence) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(3.7072-\mu)^2}{2\sigma^2}} = 0.6950$$

where $\mu = 3.448$ and $\sigma^2 = 0.2502$.

By doing the same for the other attributes and for speech class, was obtained:

p(Magnitude|Silence) = 1.4281
p(ZCR|Silence) = 0.4411
p(Energy|Speech) = 0.0363
p(Magnitude|Speech)= 0.0908
p(ZCR|Speech) = 0.0039

From Equation 2.2, *evidence = 0.2189*

Using the Equation 2.1, the result is:

***posterior(Silence)** = 0.99 or 99%*
***posterior(Speech)** = 0.01 or 1%*

The adopted decision rule was the *maximum a posteriori* decision rule, in which is selected the hypothesis with a greater posterior. Thus, is inferred that the sample to be classified is a **silence** audio.

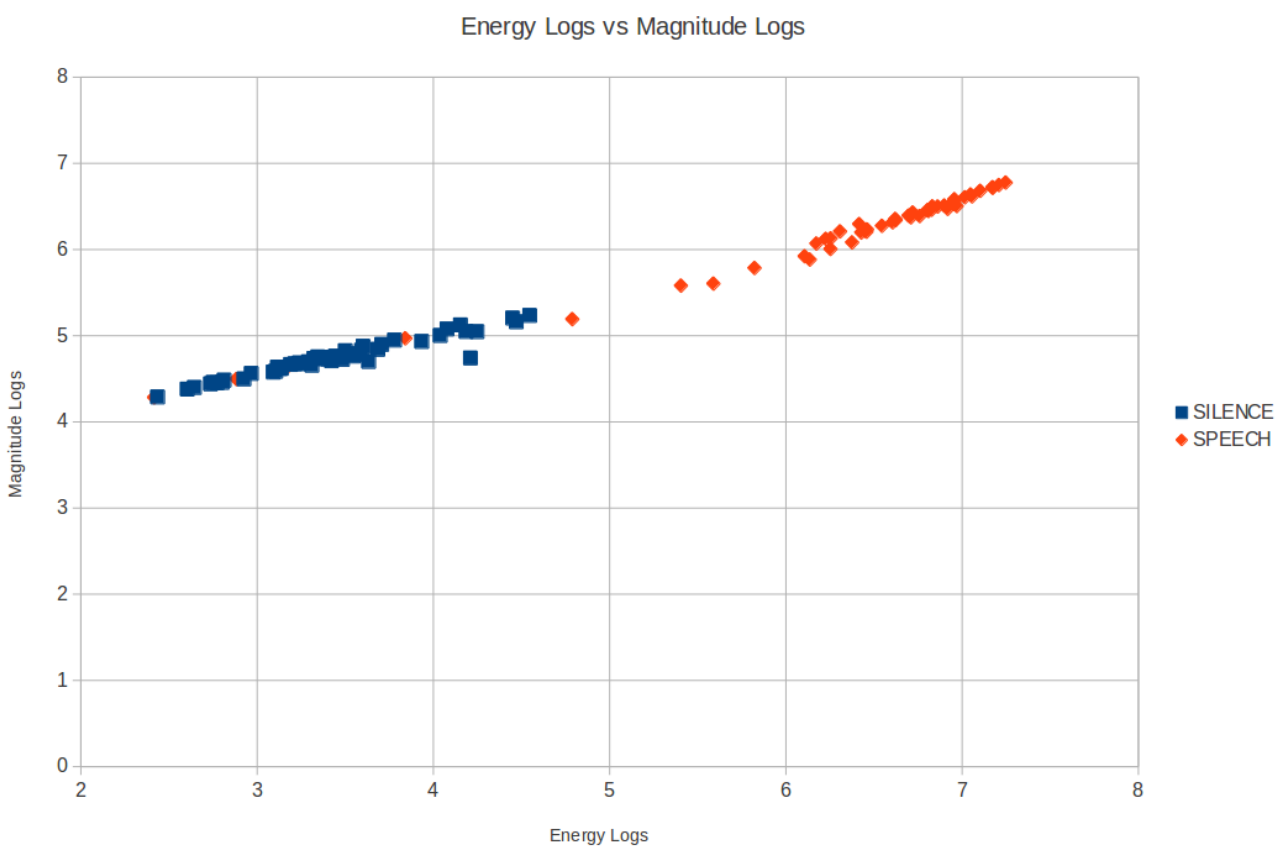
Chapter 3 – Experiments:

For this exercise was provided a sample of 100 audio signals files, given as a list of integer numbers which has 50 silence signals and 50 speech signals, labelled properly. In order to test the effectiveness of the approach applied, the tests were done in a cross-validation fashion, using the K-Fold validation process, with K = 10. In this scenario the data sample was divided into 10 subsets, with each subset having 10 samples, 5 silence, 5 speech. In order to test the classification effectiveness, one subset is kept as a test subset while the remaining subsets form a training subset, which is divided into silence and speech samples. The K-Fold cross-validation process iterates over the subsets. Thus, for the first iteration the samples from [0] to [4], from both silence and speech signals, form the test subset while the samples from [5] to [49] of speech and silence form the training subset. For the second iteration the test subset goes from [5] to [9] of silence and speech signals while the remaining samples form the training set. The process ends when it finishes its 10th iteration. For each iteration, the Naive Bayes classification is applied to the training set and the number of times it classified a sample rightly is counted in order to measure its accuracy. At the end of all the iterations the number of right inferences is divided by the total of samples tested, giving a percentage of accuracy.

Below is the output of the system developed:

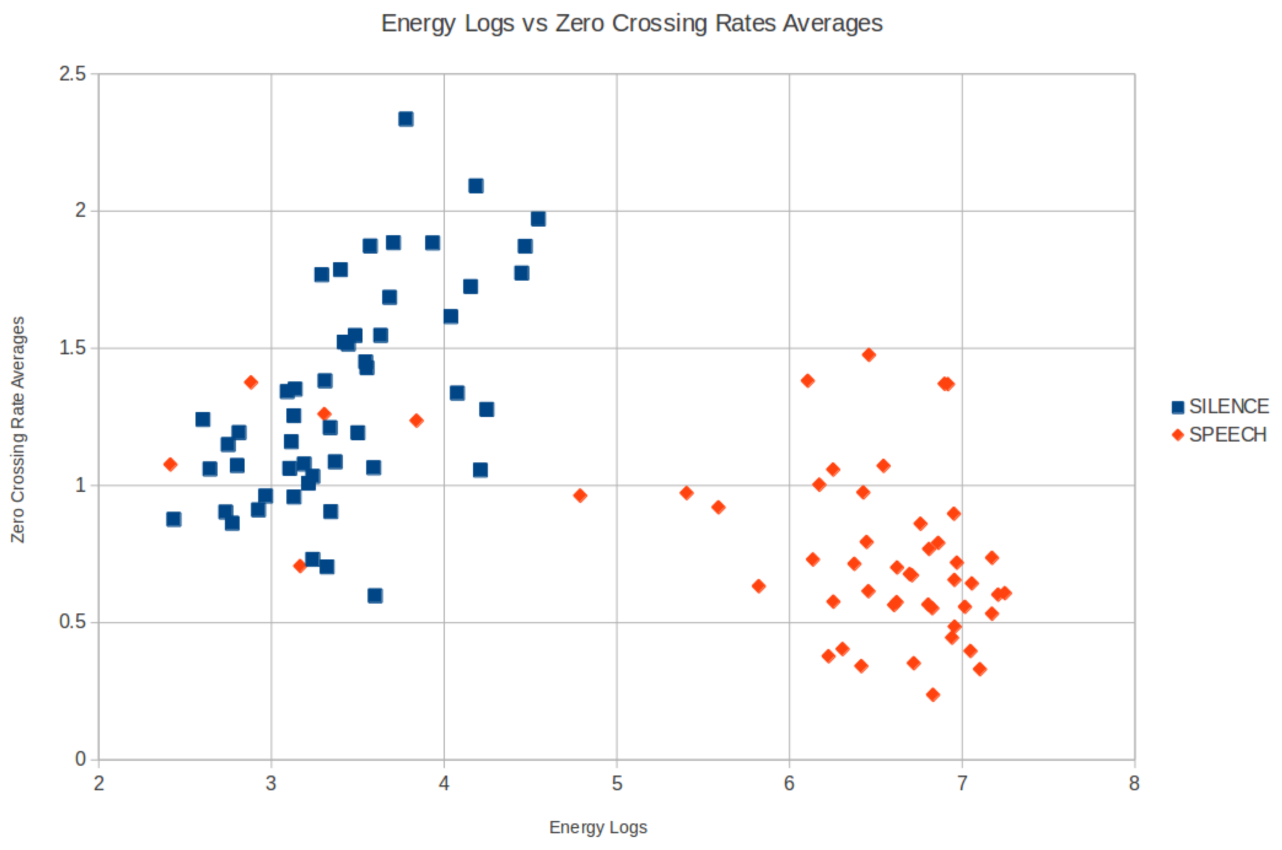
```
The results for iteration were: [10.0, 10.0, 9.0, 9.0, 9.0, 10.0, 10.0, 8.0, 10.0, 10.0]
The accuracy for the test was: 95.0%
```

In the first line, for each fold iteration, it prints out the number of correct classifications in a total of 10 samples. In the second line the accuracy in percentage, 95%, is shown. This result can be seen in practice in the following graphs *Energy vs Magniitude, Energy Logs vs Zero Crossing Rate Averages* and *Magnitude Logs vs Zero Crossing Rate Averages* that follows.

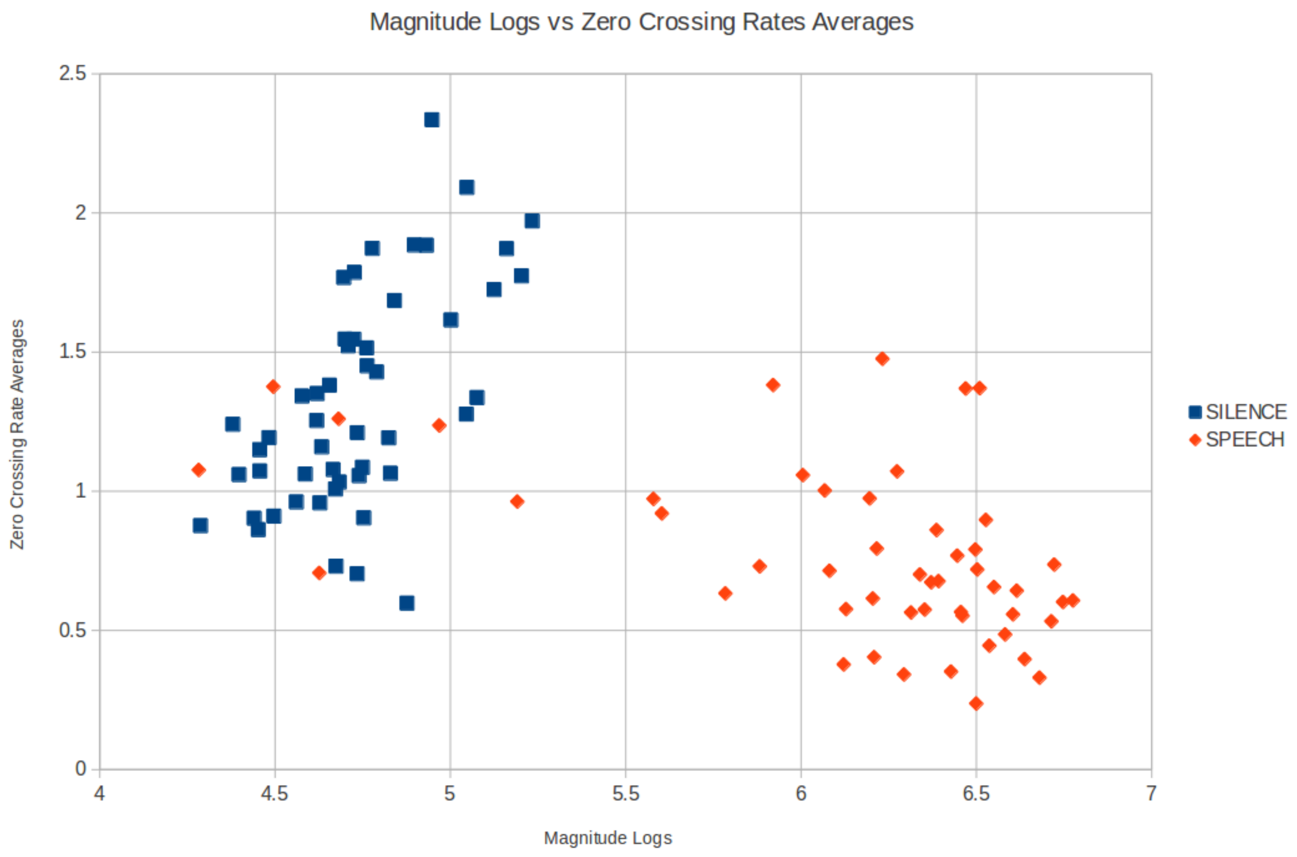


Graph 3.1

In this first graph is possible to see that there are a clear separation between the two categories, silence and speech. The speech samples are on the right side of the graph, while silence ones are in the left



Graph 3.2



Graph 3.3

By the two last graphs, more than the categories separations, looking to them carefully is possible to see that there are five “intruders” speech samples between the silence samples. They are probably the cause of the 5% error on the classifier, given that they represent 5% of the whole sample and its signals seems to be a silence signal for the classifier, that see them as silence signals.

References:

Camastra, F. and Vinciarelli, A. (2008). *Machine learning for audio, image and video analysis.* London: Springer.

Russell, S. and Norvig, P. (1995). *Artificial intelligence*. Englewood Cliffs, N.J.: Prentice Hall.

En.wikibooks.org, (2014). *Artificial Intelligence/AI Agents and their Environments - Wikibooks, open books for an open world.* [online] Available at: http://en.wikibooks.org/wiki/Artificial_Intelligence/AI_Agents_and_their_Environments [Accessed 26 Nov. 2014].

Visualstudiomagazine.com, (2013). *Understanding and Using K-Fold Cross-Validation for Neural Networks -- Visual Studio Magazine*. [online] Available at: http://visualstudiomagazine.com/articles/2013/10/01/understanding-and-using-kfold.aspx [Accessed 26 Nov. 2014].

Wikipedia, (2014). *Bayes' theorem*. [online] Available at: http://en.wikipedia.org/wiki/Bayes%27_theorem [Accessed 26 Nov. 2014].

Wikipedia, (2014). *Naive Bayes classifier*. [online] Available at: http://en.wikipedia.org/wiki/Naive_Bayes_classifier [Accessed 26 Nov. 2014].

Wikipedia, (2014). *Normal distribution*. [online] Available at: http://en.wikipedia.org/wiki/Normal_distribution [Accessed 26 Nov. 2014].

Appendix A – Code:

Observations:
For running the program is needed to provide the directory where the signal samples are.
Below is the code used to develop the program:

Main.java:

```
import java.util.ArrayList;

/**
 * Main class of the program initialise and executes everything needed
 * read information about the signals from a file
 * extract the signals and computes
 * the accuracy on predicting sample silence or speech
 *
 * @author enzoroiz
 */
public class Main {
	private final static int k = 10;
	public static void main(String[] args) {
		String entry;
		// Receive command line parameters
		try {
			entry = args[0];
		} catch (Exception e) {
			e.printStackTrace();
			System.out.println("Parameters problem");
			System.exit(0);
			return;
		}
		// Read all files in the specified directory and receive a matrix of the
		// samples read as a return
		FileReader fileReader = new FileReader(entry);
		ArrayList<ArrayList<Double>> samples = fileReader.readAllFiles();
		// Split the data into silence and speech signals
		ArrayList<ArrayList<Double>> silenceSignals = new ArrayList<>();
		ArrayList<ArrayList<Double>> speechSignals = new ArrayList<>();
		silenceSignals.addAll(samples.subList(0, samples.size() / 2));
		speechSignals
				.addAll(samples.subList(samples.size() / 2, samples.size()));
		// Process the signal information for all the read files
		ArrayList<SignalInfo> silenceSignalsInfo = SignalProcessing
				.processSignals(silenceSignals, 300, 30);
		ArrayList<SignalInfo> speechSignalsInfo = SignalProcessing
				.processSignals(speechSignals, 300, 30);
		// Compute the acuuracy in a cross-validation fashion, using K-fold
		// validation with K=10
		Classifier classifier = new Classifier(silenceSignalsInfo,
				speechSignalsInfo, k);
		classifier.test();

		// Do the same described above, but shuffle the samples and test it "n"
		// times in order to have different sets
		 //classifier.shuffleTest();

	}
}
```

FileReader.java:

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * Class that helps in reading files.
 *
 * @author enzoroiz
 *
 */
public class FileReader {
	/**
	 * Instance fields
	 */
	private File filesPath;
	private File[] filesToRead;

	/**
	 * Construtor informing the entryPath List the files of the given path or
	 * throws an error if it is not a directory
	 *
	 * @param entryPath
	 */
	public FileReader(String entryPath) {
		filesPath = new File(entryPath);
		try {
			if (!filesPath.isDirectory()) {
				throw new IOException("The path is not valid. ");
			} else {
				filesToRead = new File(entryPath).listFiles();
			}
		} catch (Exception e) {
			System.out.println(e.getMessage());
		}
	}
	/**
	 * Read the files listed in the constructor
	 *
	 * @return the samples to be processed as an matrix
	 */
	public ArrayList<ArrayList<Double>> readAllFiles() {
		// Read the silence ones first
		ArrayList<ArrayList<Double>> samples = new ArrayList<>();
		for (File file : filesToRead) {
			if (!file.isDirectory()
					&& file.getName().toLowerCase().contains("silence")) {
				samples.add(readFile(file));
			}
		}

		// And then the speech ones
		for (File file : filesToRead) {
			if (!file.isDirectory()
					&& file.getName().toLowerCase().contains("speech")) {
				samples.add(readFile(file));
			}
		}
		return samples;
	}
	/**
	 * Read the file passed as parameter and store the information in an array
	 * list
	 *
	 * @param file
	 * @return array list containing information read
	 */
	private ArrayList<Double> readFile(File file) {
		ArrayList<Double> sample = new ArrayList<Double>();
		try {
			double x;
			Scanner scanner = new Scanner(file);
			sample.add(Double.valueOf(String.valueOf(scanner.nextLine())));
			while (scanner.hasNextLine()) {
				x = Double.valueOf(String.valueOf(scanner.nextLine()));
				sample.add(x);
			}
			scanner.close();
		} catch (Exception e) {
			e.printStackTrace();
		}
		return sample;
	}
}
```

SignalInfo.java:

```
/**
 * Class that stores informations extracted from a read signal
 *
 * @author enzoroiz
 */
public class SignalInfo {
	/**
	 * Instance fields
	 */
	public static int NUM_INFO = 3;
	public static String ENERGY = "ENERGY";
	public static String MAGNITUDE = "MAGNITUDE";
	public static String ZCR = "ZERO CROSSING RATE";
	private double energy;
	private double magnitude;
	private double zeroCrossingRate;
	/**
	 * Constructor passing as parameter
	 *
	 * @param energyLog
	 * @param magnitudeLog
	 * @param zeroCrossingRateAverage
	 */
	public SignalInfo(double energy, double magnitude, double zeroCrossingRate) {
		this.energy = energy;
		this.magnitude = magnitude;
		this.zeroCrossingRate = zeroCrossingRate;
	}
	/**
	 * Getter to return the specified information 0 if energyLog 1 if
	 * magnitudeLog 2 if zeroCrossingRateAverage
	 *
	 * @param index
	 * @return the information passed as index
	 */
	public double get(int index) {
		switch (index) {
		case 0:
			return energy;
		case 1:
			return magnitude;
		case 2:
			return zeroCrossingRate;
		default:
			return Double.NaN;
		}
	}

	@Override
	public String toString() {
		return ("[" + energy + ", " + magnitude + ", " + zeroCrossingRate + "]");
	}

	@Override
	public boolean equals(Object obj) {
		SignalInfo signalInfo = (SignalInfo) obj;
		return (signalInfo.get(0) == energy && signalInfo.get(1) == magnitude && signalInfo
					.get(2) == zeroCrossingRate);
	}

}
```

SignalProcessing.java:

```
import java.util.ArrayList;
/**
 * Class used to process signals and extract its informations
 *
 * @author enzoroiz
 *
 */
@SuppressWarnings("unchecked")
public class SignalProcessing {
	// Attributes
	private ArrayList<Double> sample;
	private int audioSizeInMS;
	/**
	 * @param sample
	 *            to analyse
	 * @param audioSizeInMS
	 */
	public SignalProcessing(ArrayList<Double> sample, int audioSizeInMS) {
		this.sample = sample;
		this.audioSizeInMS = audioSizeInMS;
	}
```

```
/**
 * @return the original signal normalized with the peak
 */
public ArrayList<Double> originalNormalized() {
	ArrayList<Double> originalNormalized = new ArrayList<Double>();
	originalNormalized = (ArrayList<Double>) sample.clone();
	return normalizeSignal(originalNormalized);
}
/**
 * Ideal delay
 *
 * @param delay
 * @return samples shifted by delay
 */
public ArrayList<Double> idealDelay(int delay) {
	ArrayList<Double> shiftedSample = new ArrayList<Double>();
	shiftedSample = (ArrayList<Double>) sample.clone();
	int shiftSize = (sample.size() * delay) / audioSizeInMS;
	for (int i = 0; i < shiftSize; i++) {
		shiftedSample.add(0, 0.0);
		shiftedSample.remove(shiftedSample.size() - 1);
	}
	return normalizeSignal(shiftedSample);
}

/**
 * Moving Average
 *
 * @param limit
 * @return the moving average sample
 */
public ArrayList<Double> movingAverage(int window) {
	ArrayList<Double> movingAverage = new ArrayList<Double>();
	int averageSize = (sample.size() * window) / audioSizeInMS;
	int i;
	double sampleWindowSum = 0;
	// Calculate moving average
	for (i = 0; i < averageSize / 2; i++) {
		sampleWindowSum += sample.get(i);
	}

	for (; i < averageSize; i++) {
		movingAverage.add(sampleWindowSum / averageSize);
		sampleWindowSum += sample.get(i);
	}

	for (; i < sample.size(); i++) {
		movingAverage.add(sampleWindowSum / averageSize);
		sampleWindowSum -= sample.get(i - averageSize);
		sampleWindowSum += sample.get(i);
	}

	for (int j = 0; j < averageSize / 2; j++) {
		movingAverage.add(sampleWindowSum / averageSize);
		sampleWindowSum -= sample.get(sample.size() + j - averageSize);
	}

	return normalizeSignal(movingAverage);
}
```

```java
/**
 * Convolution
 *
 * @param window
 * @return the convolved signal
 */
public ArrayList<Double> convolution(int window) {
		ArrayList<Double> convolution = new ArrayList<Double>();
		int windowSize = (sample.size() * window) / audioSizeInMS;
		int i;
		double sampleWindowSum = 0;
		// Calculate the convolution
		for (i = 0; i < windowSize; i++) {
				sampleWindowSum += (sample.get(i));
				convolution.add(sampleWindowSum);
		}

		for (; i < sample.size(); i++) {
				sampleWindowSum -= (sample.get(i - windowSize));
				sampleWindowSum += (sample.get(i));
				convolution.add(sampleWindowSum);
		}

		return normalizeSignal(convolution);
}

/**
 * Energy
 *
 * @param window
 * @return the energy of the original signal
 */
public ArrayList<Double> energy(int window) {
		ArrayList<Double> energy = new ArrayList<Double>();
		int windowSize = (sample.size() * window) / audioSizeInMS;

		int i;
		double sampleWindowSum = 0;

		// Calculate energy
		for (i = 0; i < windowSize; i++) {
				sampleWindowSum += (Math.pow(sample.get(i), 2) / Math.pow(10, 4));
				energy.add(sampleWindowSum);
		}

		for (; i < sample.size(); i++) {
				sampleWindowSum -= (Math.pow(sample.get(i - windowSize), 2) / Math
								.pow(10, 4));
				sampleWindowSum += (Math.pow(sample.get(i), 2) / Math.pow(10, 4));
				energy.add(sampleWindowSum);
		}

		// return normalizeSignal(energy);
		return energy;
}
```

```
/**
 * Magnitude
 *
 * @param window
 * @return the magnitude of the original signal
 */
public ArrayList<Double> magnitude(int window) {
	ArrayList<Double> magnitude = new ArrayList<Double>();
	int windowSize = (sample.size() * window) / audioSizeInMS;
	int i;
	double sampleWindowSum = 0;

	// Calculate magnitude
	for (i = 0; i < windowSize; i++) {
		sampleWindowSum += Math.abs(sample.get(i));
		magnitude.add(sampleWindowSum);
	}
	for (; i < sample.size(); i++) {
		sampleWindowSum -= Math.abs(sample.get(i - windowSize));
		sampleWindowSum += Math.abs(sample.get(i));
		magnitude.add(sampleWindowSum);
	}
	// return normalizeSignal(magnitude);
	return magnitude;
}

/**
 * Zero Crossing Rate
 *
 * @param window
 * @return the ZCR
 */
public ArrayList<Double> zeroCrossingRate(int window) {
	ArrayList<Double> zcr = new ArrayList<Double>();
	ArrayList<Double> zcrAux = new ArrayList<Double>(30);
	int windowSize = (sample.size() * window) / audioSizeInMS;
	int i;
	double zcrSampleWindowSum = 0;
	boolean sampleBeforeIsPositive = true;
	boolean sampleNowIsPositive;

	// Calculate zero crossing rate
	for (i = 0; i < windowSize; i++) {
		if (sample.get(i) >= 0) {
			sampleNowIsPositive = true;
		} else {
			sampleNowIsPositive = false;
		}

		if (sampleNowIsPositive != sampleBeforeIsPositive) {
			zcrSampleWindowSum += 1;
			zcrAux.add(1.0);
		} else {
			zcrAux.add(0.0);
		}

		sampleBeforeIsPositive = sampleNowIsPositive;

		zcr.add(zcrSampleWindowSum/(2 * window));
	}
```

```
        for (; i < sample.size(); i++) {
            if (sample.get(i) >= 0) {
                sampleNowIsPositive = true;
            } else {
                sampleNowIsPositive = false;
            }

            if (sampleNowIsPositive != sampleBeforeIsPositive) {
                zcrSampleWindowSum += 1;
                zcrSampleWindowSum -= zcrAux.get(0);
                zcrAux.add(1.0);
                zcrAux.remove(0);
            } else {
                zcrSampleWindowSum -= zcrAux.get(0);
                zcrAux.add(0.0);
                zcrAux.remove(0);
            }

            sampleBeforeIsPositive = sampleNowIsPositive;
            zcr.add(zcrSampleWindowSum/(2 * window));
        }

        // return normalizeSignal(zcr);
        return zcr;
    }

    /**
     * Normalize the signal by the maximum absolute value
     *
     * @param arrayToNormalize
     * @return a normalized array
     */
    public ArrayList<Double> normalizeSignal(ArrayList<Double> arrayToNormalize) {
        int maxValueIndex = 0;
        double maxValue;
        double aux;
        // Get the max absolute value of the array
        for (int i = 0; i < arrayToNormalize.size(); i++) {
            if (Math.abs(arrayToNormalize.get(i)) > Math.abs(arrayToNormalize
                    .get(maxValueIndex))) {
                maxValueIndex = i;
            }
        }
        maxValue = Math.abs(arrayToNormalize.get(maxValueIndex))
                / Math.pow(10.0, 1.0);

        // Normalize the other values
        for (int i = 0; i < arrayToNormalize.size(); i++) {
            aux = arrayToNormalize.get(i) / maxValue;
            aux = Math.round(aux * 100000.0) / 100000.0;
            arrayToNormalize.set(i, aux);
        }
        return arrayToNormalize;
    }
```

```
	/**
	 * @param samples
	 *            read from files
	 * @param audioSizeInMS
	 *            of the files
	 * @return a list containing log from the average of energy and magnitude
	 *         signals, and the average of zero crossing rate for each of the
	 *         signals
	 */
	public static ArrayList<SignalInfo> processSignals(
			ArrayList<ArrayList<Double>> samples, int audioSizeInMS,
			int windowSize) {
		ArrayList<SignalProcessing> signalProcessing = new ArrayList<>();
		ArrayList<SignalInfo> signalInfos = new ArrayList<>();

		// For each signal
		for (int i = 0; i < samples.size(); i++) {
			signalProcessing.add(new SignalProcessing(samples.get(i),
					audioSizeInMS));
			signalInfos.add(signalProcessing.get(i).getSignalInfos(
					signalProcessing.get(i), windowSize));
		}
		return signalInfos;
	}

	/**
	 * Create Stat objects to compute the log and the mean of the signals
	 *
	 * @param signalProcessing
	 *            object
	 * @param windowSize
	 * @return SignalInfo containing log from the average of energy and
	 *         magnitude signals, and the average of zero crossing rate for each
	 *         of the signals
	 */
	private SignalInfo getSignalInfos(SignalProcessing signalProcessing,
			int windowSize) {
		double energyLog = new Stat(signalProcessing.energy(windowSize))
				.getLog();
		double magnitudeLog = new Stat(signalProcessing.magnitude(windowSize))
				.getLog();
		double zeroCrossingRateMean = new Stat(
				signalProcessing.zeroCrossingRate(windowSize)).getMean();
		SignalInfo signalInfo = new SignalInfo(energyLog, magnitudeLog,
				zeroCrossingRateMean);

		return signalInfo;
	}

}
```

Stat.java:

```
import java.util.ArrayList;

/**
 * Class that computes statistics
 *
 * @author enzoroiz
 *
 */
public class Stat {
	/**
	 * instance fields
	 */
	private ArrayList<Double> signalNormalized;
	private double mean;
	private double log10;
	private double variance;

	/**
	 * Constructor given
	 *
	 * @param signalNormalized
	 *          array list of double
	 */
	@SuppressWarnings("unchecked")
	public Stat(ArrayList<Double> signalNormalized) {
		this.signalNormalized = (ArrayList<Double>) signalNormalized.clone();
		this.mean = mean();
		this.log10 = log(mean);
	}

	/**
	 * @return the mean of the numbers in the array list
	 */
	private double mean() {
		double sum = 0;
		for (int i = 0; i < signalNormalized.size(); i++) {
			sum += signalNormalized.get(i);
		}
		return Math.round(sum / signalNormalized.size() * 10000.0) / 10000.0;
	}

	/**
	 *
	 * @param mean
	 * @return the of the mean
	 */
	private double log(double mean) {
		return Math.round(Math.log10(mean) * 10000.0) / 10000.0;
	}

	/**
	 * @return the variance of the numbers in the array list given the mean
	 */
	public double getVariance() {
		double sum = 0;
		for (int i = 0; i < signalNormalized.size(); i++) {
			sum += Math.pow(signalNormalized.get(i) - mean, 2.0);
		}
		return Math.round(sum / signalNormalized.size() * 10000.0) / 10000.0;
	}
	/**
	 * @param parameter
	 *            to be analysed
	 * @return the normal or the Gaussian distribution given the mean and the
	 *         variance
	 */
	public double getNormalDistribution(double parameter) {
		this.variance = getVariance();
		double firstPart = (1 / Math.sqrt(2 * Math.PI * variance));
		double secondPart = -Math.pow(parameter - mean, 2) / (2 * variance);

		return firstPart * Math.pow(Math.E, secondPart);
	}

	/**
	 * @param parameter
	 *            to be analysed
	 * @param mean
	 * @param variance
	 * @return the normal or the Gaussian distribution given the mean and the
	 *         variance
	 */
	public static double getNormalDistribution(double parameter, double mean, double variance) {
		double firstPart = (1 / Math.sqrt(2 * Math.PI * variance));
		double secondPart = -Math.pow(parameter - mean, 2) / (2 * variance);

		return firstPart * Math.pow(Math.E, secondPart);
	}

	/**
	 * @return the mean
	 */
	public double getMean() {
		return this.mean;
	}

	/**
	 * @return the log in the basis of ten
	 */
	public double getLog() {
		return this.log10;
	}

	/**
	 * @param number
	 *            to round
	 * @return a number rounded up to 4 decimal places
	 */
	public static double round(double number) {
		return Math.round(number * 100000.0) / 100000.0;
	}

	@Override
	public String toString() {
		return ("Mean: " + mean + " Log 10: " + log10 + " Variance " + getVariance());
	}
}
```

Classifier.java:

```
import java.util.ArrayList;
import java.util.Collections;

/**
 * Class that computes the accuracy of predicting either speech or silence
 * signal by doing a kfold validation approach, applying the Naive Bayes
 * Classifier
 *
 * @author enzoroiz
 *
 */
public class Classifier {
	/**
	 * Final fields
	 */
	// Probability Speech or Silence
	private final double PSILENCE = 0.5;
	private final double PSPEECH = 0.5;

	/**
	 * Instance fields
	 */
	private ArrayList<SignalInfo> silenceProcessedSignals;
	private ArrayList<SignalInfo> speechProcessedSignals;
	private ArrayList<ArrayList<Stat>> statsSilence;
	private ArrayList<ArrayList<Stat>> statsSpeech;
	private int processedSignalsSize;
	private int signalInfosSize;
	private int kFold;

	/**
	 * Constructor
	 *
	 * @param silenceProcessedSignals
	 * @param speechProcessedSignals
	 * @param kFold
	 *            times
	 */
	public Classifier(ArrayList<SignalInfo> silenceProcessedSignals,
			ArrayList<SignalInfo> speechProcessedSignals, int kFold) {
		this.silenceProcessedSignals = silenceProcessedSignals;
		this.speechProcessedSignals = speechProcessedSignals;
		this.processedSignalsSize = silenceProcessedSignals.size(); // 50
		this.signalInfosSize = SignalInfo.NUM_INFO; // 3
		this.kFold = kFold;

		// Initialise Stats Arrays
		this.statsSilence = new ArrayList<>();
		this.statsSpeech = new ArrayList<>();
		for (int i = 0; i < signalInfosSize; i++) {
			this.statsSilence.add(new ArrayList<Stat>());
			this.statsSpeech.add(new ArrayList<Stat>());
		}
	}

	/**
	 * Realise the Kfold cross validation by splitting the data between speech
	 * and silence signals For each one of the Kfold iteration keeps a
	 * k/processedSignalsSize as test set and the remaining as training set for
	 * both speech and silence samples. Computes the statistics for the training
 * set and put it in an array list
 */
private void kFold() {
        int lowerBound;
        int upperBound;

        ArrayList<Double> silenceInfo;
        ArrayList<Double> speechInfo;

        // For each type of information: E - M - Z //3
        for (int i = 0; i < signalInfosSize; i++) {
                // For each folding //10
                for (int fold = 0; fold < kFold; fold++) {
                        silenceInfo = new ArrayList<>();
                        speechInfo = new ArrayList<>();
                        lowerBound = fold * (processedSignalsSize / kFold);
                        upperBound = (fold + 1) * (processedSignalsSize / kFold);

                        // For each information of the processed signal //50
                        for (int j = 0; j < processedSignalsSize; j++) {
                                if (!(j >= lowerBound && j < upperBound)) { // Adding

                // training sets
                                        silenceInfo.add(silenceProcessedSignals.get(j).get(i));
                                        speechInfo.add(speechProcessedSignals.get(j).get(i));
                                }
                        }

                        this.statsSilence.get(i).add(new Stat(silenceInfo));
                        this.statsSpeech.get(i).add(new Stat(speechInfo));
                }
        }
}

/**
 * For each of the Kfold iteration uses the test set in order to compute the
 * accuracy in predicting if the signal analysed is either from a speech or
 * silence sample. Do it by applying the Naive Bayes approach to the
 * corresponding training set. Compute the accuracy by measuring how many
 * times the classification was right.
 */
public void test() {
        double posteriorSilenceForSilenceSignal = PSILENCE;
        double posteriorSpeechForSilenceSignal = PSPEECH;
        double posteriorSilenceForSpeechSignal = PSILENCE;
        double posteriorSpeechForSpeechSignal = PSPEECH;
        double evidenceSilenceSignal;
        double evidenceSpeechSignal;
        double right;
        int lowerBound;
        int upperBound;

        ArrayList<Double> guessed = new ArrayList<>();

        kFold();
```

```
		// For each fold iteration
		for (int i = 0; i < kFold; i++) {// 10
			right = 0;
			lowerBound = i * (processedSignalsSize / kFold);
			upperBound = (i + 1) * (processedSignalsSize / kFold);
			// For each subset
			for (int j = lowerBound; j < upperBound; j++) {// 5
				posteriorSilenceForSilenceSignal = PSILENCE;
				posteriorSpeechForSilenceSignal = PSPEECH;
				posteriorSilenceForSpeechSignal = PSILENCE;
				posteriorSpeechForSpeechSignal = PSPEECH;
				// For each type of information
				for (int m = 0; m < signalInfosSize; m++) {// 3
					// Calculate the posterior for silence signals
					posteriorSilenceForSilenceSignal *= statsSilence
							.get(m)
							.get(i)
							.getNormalDistribution(
							silenceProcessedSignals.get(j).get(m));
					posteriorSpeechForSilenceSignal *= statsSpeech
							.get(m)
							.get(i)
							.getNormalDistribution(
							silenceProcessedSignals.get(j).get(m));
					// Calculate the posterior for speech signals
					posteriorSilenceForSpeechSignal *= statsSilence
							.get(m)
							.get(i)
							.getNormalDistribution(
							speechProcessedSignals.get(j).get(m));
					posteriorSpeechForSpeechSignal *= statsSpeech
							.get(m)
							.get(i)
							.getNormalDistribution(
							speechProcessedSignals.get(j).get(m));
				}

				// Calculate the evidences
				evidenceSilenceSignal = posteriorSilenceForSilenceSignal
						+ posteriorSpeechForSilenceSignal;
				evidenceSpeechSignal = posteriorSilenceForSpeechSignal
						+ posteriorSpeechForSpeechSignal;
				// Calculate the posterior for silence signals in %
				posteriorSilenceForSilenceSignal = posteriorSilenceForSilenceSignal
						/ evidenceSilenceSignal * 100;
				posteriorSpeechForSilenceSignal = posteriorSpeechForSilenceSignal
						/ evidenceSilenceSignal * 100;
				// Calculate the posterior for speech signals in %
				posteriorSilenceForSpeechSignal = posteriorSilenceForSpeechSignal
						/ evidenceSpeechSignal * 100;
				posteriorSpeechForSpeechSignal = posteriorSpeechForSpeechSignal
						/ evidenceSpeechSignal * 100;
				if (posteriorSilenceForSilenceSignal > posteriorSpeechForSilenceSignal) {
					right++;
				}
				if (posteriorSpeechForSpeechSignal > posteriorSilenceForSpeechSignal) {
					right++;
				}
			}
			guessed.add(right);
		}
		outputResults(guessed);
	}
	/**
	 * Prints the accuracy got in the test
	 *
	 * @param guessed
	 */
	private void outputResults(ArrayList<Double> guessed) {
		System.out.println("The results for iteration were: " + guessed.toString());
		System.out.println("The accuracy for the test was: "
				+ Stat.round(new Stat(guessed).getMean() * 10) + "%");
	}

	/**
	 * test the accuracy for "n" times shuffling the processed signals
	 *
	 * @param times
	 *            to shuffle and test compute the accuracy
	 */
	public void shuffledSampleTest(int times) {
		for (int i = 0; i < times; i++) {
			shuffle();
			test();
		}
	}

	/**
	 * Shuffle the processed Signals in order to produce different training and
	 * test sets
	 */
	private void shuffle() {
		ArrayList<SignalInfo> silenceTemp = new ArrayList<>(
				silenceProcessedSignals);
		ArrayList<SignalInfo> speechTemp = new ArrayList<>(
				speechProcessedSignals);

		ArrayList<Integer> index = new ArrayList<>();
		for (int i = 0; i < processedSignalsSize; i++) {
			index.add(i);
		}

		Collections.shuffle(index);

		for (int i = 0; i < processedSignalsSize; i++) {
			silenceTemp.set(i, silenceProcessedSignals.get(index.get(i)));
			speechTemp.set(i, speechProcessedSignals.get(index.get(i)));
		}
		this.silenceProcessedSignals = silenceTemp;
		this.speechProcessedSignals = speechTemp;
	}
}
```