Design of sleep events monitoring device system

Álvaro Carrera Cardeli
s172254

Federico Medea
s193612

Walid Khaled Hussein
s181263

***Abstract*—The obstructive sleep apnea (OSA) is a severe sleep disorder characterized by frequent upper airway obstruction. This report presents a simple solution to monitor OSA with an android application for smartphones that is able to estimate the percentage and time slots of sleep apnea from a recording of a subject while he/she is sleeping. Firstly, ten features were extracted from time and frequency domain with Matlab, for classification of *normal breathing*, *snoring* and *apnea*. Then, *K-nearest neighbor* algorithm was implemented to work as classifier, which was also trained in Matlab to obtain the most optimal model. This classifier works together with a database generated in the study that will be the reference samples to classify. Finally, the models developed were translated into Java language to be used in the android application. The Java methods used for the app development were compared to the ones developed in Matlab in order to be validated. Lastly, some tests were done in the application.**

***Keywords*—KNN, Java, Matlab, Android Studio, FFT, apnea.**

I. Introduction

Sleep apnea is considered the most prevalent sleep disturbance. There are three types of sleep apnea, including central, obstructive, and mixed [1]. *Central Sleep Apnea (CSA)* occurs when the brain does not periodically activate the breathing muscles in the chest, whereas *obstructive sleep apnea (OSA)* occurs when air is physically blocked from flowing through the lungs during sleep. *Mixed Sleep Apnea (MSA)* is a mixture of central and obstructive. OSA is the most common of the three while CSA and MSA are considerably rarer. The patient is often unaware of the breath stoppage because the body does not induce a full awakening (see Fig. 1).

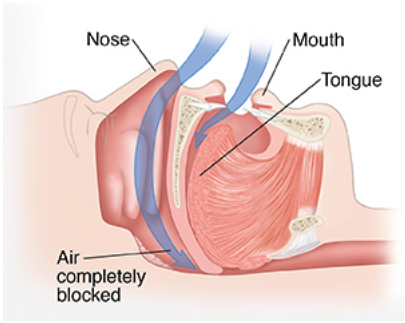

Fig. 1: Graphical representation of obstructive sleep apnea (OSA) [2].

The *National Sleep Foundation (NSF)* reported needing to receive seven to eight sleeping hours each night for adults to work healthily [1]. Frequent airflow obstructions during this time have a significant impact on the human output during the day. This induces chronic sleepiness, non-restorative sleep, high blood pressure, cardiovascular disorders, issues with memory loss, erectile dysfunction, changes in personality and depression.

Attended overnight Polysomnography (PSG) is considered the gold standard for analysis of sleep apnea [1]. Generally to get an OSA test, a patient needs to remain for more than one night in a specialized sleep laboratory, with 22 wires connected to his body to monitor and analyze multiple neurological and cardiovascular signals. This procedure affects patients with considerable anxiety, and they may not be able to sleep well during the night. Lastly, very few hospitals can give the PSG check.

Due to the above-mentioned difficulties of using PSG, a need for portable devices with reasonable accuracy has arisen. This paper is oriented to introduce a portable system that is inexpensive, reliable and effective in a comparison to PSG.

A. *State of the art*

In the last few years, several recent studies about detection of sleep quality, snoring and sleep apnea have been published. As a result of new development in healthcare technology, most of these works are connected to the use of smartphones. In 2007 Oliver et al. implemented HealthGear, a set of non-invasive physiological sensors wirelessly connected via Bluetooth to a cell phone [3]. Using a blood oximeter to monitor the user's blood oxygen level and pulse while sleeping, it was possible to detect sleep apnea events.

In another study by Dafna, Eliran, et al. a whole-night snore detector based on non-contact microphone was designed [4]. The audio signal recordings were used in order to give an estimation of sleep apnea. The *Apnea-Hypoapnea Index (AHI)* estimation can be calculated by dividing the number of apnea events by the number of hours of sleep.

Al-Mardini et al. published an article in 2014 whose objective was to classify obstructive sleep apnea (OSA) using a smartphone [1]. Three built-in sensors of a smartphone were used, which were an oxymeter to measure the oxygen level, a microphone to record the respiratory effort and an accelerometer to detect body's movement, to screen OSA.

In 2015 Nandakumar et al. developed a contactless system that was able to detect sleep apnea events on smartphones [5]. By transforming the phone into an active sonar system that emitted frequency-modulated sound signals and listened to their reflections, the systems monitored the minute changes to these reflections to extract the chest movements caused by breathing. The study was able to identify various sleep apnea

events such as obstructive apnea, central apnea, and hypopnea from the sonar reflections.
In another similar study conducted by Lion et al. in 2019, a non-contact sensing system based on Sonar technology as a *Sleep Disordered Breathing (SDB)* screener was implemented [6]. The respiration and movements of a subject could be measured through a smartphone equipped with a custom app that identified sleep stages and detected SDB patterns.

B. *Research approach*

For this research, the objective is to create an application on smarthphone that can record breathing of a subject by using a microphone. Based on this system, it is possible to give an estimation of sleep apnea. As stated in literature, the polysomnography test (PSG) is traditionally used to diagnose sleep apnea and other sleep disorders. To obtain patient's sleeping patterns, different sensors are used in the test: a chest and abdomen belt to measure breathing movements, a nasal pressure transducer, a snore microphone, a pulse oximeter to measure oxygen saturation, a movement sensor on each leg to detect movements and five electroencephalography (EEG) sensors to measure brain activity [5]. Therefore multiple measurements are necessary to diagnose sleep apnea. Nevertheless this study would like to introduce a smartphone application that is able to give a possible estimation of sleep apnea based on recording of breathing. Indeed, by using this application at home, a patient can get reliable information about his sleep quality and disorders in a very easy and comfortable way in order to prevent future possible complications related to them. In the following sections a description of the methods applied in the study is given. After that, the obtained results are discussed and analysed. Lastly, final conclusions about the research are given.

II. Methods

In this section the methods used to classify the different events are described on details. A flowchart of the followed algorithm is shown in Fig. 2. After that, the algorithm used for the implementation of the smartphone application on Android is also discussed.

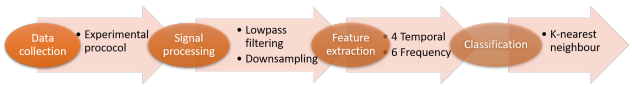

Fig. 2: Flowchart of the algorithm development.

A. *Data collection and Experimental protocol*

Six healthy subjects with no sleep or cardiac disorders participated in the study. Since multiple measurements were taken from each subject, the original database was composed of 13 recordings in total. The experiment was conducted by using a microphone ($f_{sample} = 44.1\,kHz$) integrated in an Android smartphone (Samsung Galaxy S8). The participant, who was lain down on the bed, placed the smartphone close to him and started the recording.
Each recording was composed of three different phases:

1) *Normal breathing*: (duration 1 min. 30 secs.) in which the subject breathed normally.
2) *Snoring*: (duration 1 min. 30 secs) in which the subject breathed normally for the first 20 seconds, in order to control the breathing, and then started snoring.
3) *Sleep apnea*: (duration 2 mins.) in which the subject breathed normally for the first 10 seconds and then started faking sleep apnea. In the last 10 seconds the patient was breathing normally again.

As a result of this experiment, 39 samples were collected (3 samples per subject), firstly, to be used in Matlab to develop the signal processing algorithm and, then, to generate the database for classification.

B. *Signal processing*

1) Pre-processing: as stated previously, the audio signal is recorded with a sampling frequency equal to $44.1\,kHz$, which means, that the frequency range recorded is up to $22.05\,kHz$ (*Nyquist theorem*). Since the bandwidth of breath sound that originates from trachea goes from $60\,Hz$ to $4\ kHz$ [7], it was decided to down-sample the recorded audio in order to reduce the number of points and, thus, increase the efficiency of the processing. Hence, the new sampling frequency was set to $8\,kHz$, meaning that $4\,kHz$ is the maximum frequency recorded (maximum frequency generated by the trachea).
In order to avoid aliasing in the down-sampled signal, firstly, a low-pass filter with a chosen frequency cut of $4\ kHz$ is applied to the original audio. After filtering, the signal can be down-sampled to $8\ kHz$. This sampling rate is used due to *Nyquist–Shannon sampling theorem* that states, a signal can be unambiguously reconstructed when it is sampled with a sampling frequency that is at least twice higher than the frequency max [8]. Therefore, the new signal has a sampling rate that is approximately five-fold lower than the original sampling frequency. In Fig. 3 it is shown the original signal and the signal after being down-sampled of a sample randomly picked in time and frequency domain.

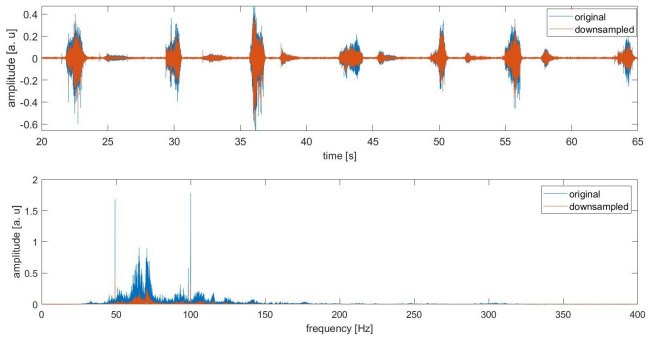

Fig. 3: Comparison plot between an original signal randomly recorded (blue) and the signal after down-sampling (red) in time domain (top) and frequency domain (bottom).

2) Feature extraction: to classify the different events, which are normal breathing, snoring and sleep apnea, a total number of 10 features is established. The first 4 features are related

to the time domain, whereas the other 6 features are obtained from the frequency domain. In table I it is shown the statistical analysis of the extracted attributes.

	Mean	Std	Min	Max	Range
Mean	0.0078	0.0150	0.0008	0.0945	0.0937
Std	0.0172	0.0253	0.0011	0.1427	0.1416
Energy mean	0.4466	1.5058	0.0006	9.7855	9.7849
Number of peaks	3.5385	2.8499	0	11	11
PSD	167.4139	564.3831	0.2134	3666.1	3665.9
VLF power	143.6674	480.9644	0.1558	3054.8	3054.7
LF power norm	0.5167	0.2871	0.0976	0.9958	0.8981
HF power norm	0.4833	0.2871	0.0042	0.9024	0.8981
Ratio1	0.2733	0.2463	0.0237	1.3699	1.3462
Ratio2	0.1275	0.1506	0.0002	0.6368	0.6365

TABLE I: Statistical measurements of all attributes.

A brief description of these features is given in the following paragraph.

Time domain features

- **Mean**: arithmetic mean of the signal in absolute value.
- **Standard deviation**: standard deviation of the signal.
- **Energy mean**: using a window size of $60\,ms$ with an overlapping of 75%, the energy is calculated as the sum of the squared points in the window. The window is used to smooth the signal and highlight the events (peaks). The selected attribute is the arithmetic mean of this energy computed (see Fig. 4).
- **Number of peaks**: a fixed threshold calculated as 180% of the energy mean is established. A signal fragment can be considered as peak every time it is above the threshold. However, to avoid possible mistakes in the peak detection, two further conditions are introduced. A peak is regarded as an actual peak when its duration is higher than $300\,ms$ and when the time difference between consecutive peaks is higher than $3.5\,s$. In Fig. 4 two peaks can be clearly seen. They are above the threshold lasting sufficiently and having a time difference between them that is bigger than $3.5\,s$.

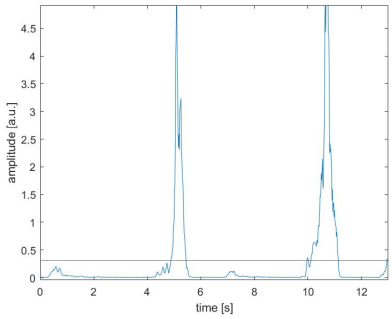

Fig. 4: Energy of a random down-sampled signal (snoring class). A fixed threshold (horizontal line) is set as 180% of energy mean. Two peaks are detected at 5 and $10.7\,s$.

Frequency domain features

Before extracting frequency domain features, the Fast Fourier Transform (FFT) is applied in order to obtain the spectrum of the signal. Then, the six following features in frequency domain are mined.

- **Power Spectral Density (PSD)**: total power of the spectrum of the signal (from 0 to $4\,kHz$).
- **VLF power**: power spectrum in Very Low Frequency (VLF). This frequency range is defined from 0 to $200\,Hz$, where it was considered that the PSD is mostly distributed.
- **LF power normalized**: low frequency (LF) power spectrum normalized. The frequency range is defined from 200 to $600\,Hz$. It is computed as

$$LF_{powerNorm} = \frac{PSD_{LF}}{PSD - PSD_{VLF}}$$

- **HF power normalized**: high frequency (HF) power spectrum normalized. The frequency range is defined from 600 to $4\,kHz$. It is computed as

$$HF_{powerNorm} = \frac{PSD_{HF}}{PSD - PSD_{VLF}}$$

- **Ratio1**: since most of the frequency components for normal breathing are below $120\,Hz$, the PSD in $0-120\,Hz$ was considered to create a feature. Furthermore, most of the frequency components in the three classes are below $400\,Hz$, so the PSD in the range $120-400\,Hz$ was used to create a ratio with the frequency range from 0 to $120\,Hz$. In Fig. 5 it is possible to see a comparison between the frequency components of normal breathing and snoring from a subject. We can denote how snoring has frequency components above $120\,Hz$ in contrast to normal breathing, where most of its important frequencies are below $120\,Hz$. The ratio is computed as

$$ratio1 = \frac{PSD_{120-400}}{PSD_{0-120}}$$

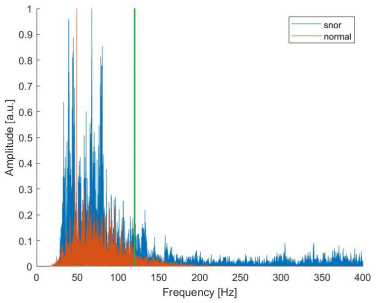

Fig. 5: Frequency components of two random samples of normal breathing and snoring. The spectrum is divided by the green line at $120\,Hz$.

- **Ratio2**: frequencies above $500\,Hz$ were also taken into account. In Fig. 6 it is shown the spectrograms for normal breathing and snoring of a subject. It is easy to denote how the spectrogram of snoring (right) shows important frequency components above $500\,Hz$ while in the case of normal breathing (left) it can barely be seen components above $500\,Hz$. Then the ratio is defined as

$$ratio2 = \frac{PSD_{500-4000}}{PSD}$$

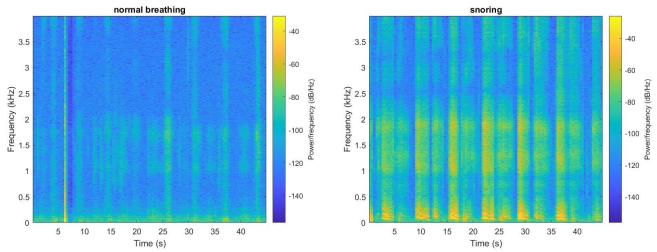

Fig. 6: Spectrograms (plot of time vs. frequency domain) of two random samples of normal breathing (left) and snoring (right).

C. *Classification*

A database of 78 samples was created out of the 39 original samples collected from the patients in order to have a bigger dataset for classification. This new database was obtained by using a window size of $45\,s$ for each sample and by only using pure fragments (in each $45\,s$ fragment there is only one class processed). It is interesting to notice that the database is well balanced since we have 26 samples for each class (*normal breathing*, *snoring* and *sleep apnea*).

Since the target value is a nominal attribute with three possible classes, it was decided to use a machine learning model orientated to classification. Thus, *K-Nearest Neighbor (KNN)* classifier was chosen to classify the input recordings. This method uses all the samples available in the database and classifies the input signals (fragments of $45\,s$) based on a measure of similarity with the samples in the database [9]. A event is classified by a majority vote of its neighbors and it is assigned to the most common class of its nearest K neighbors by a distance function (measure of similarity). The Euclidean distance was used as the measure of similarity and it is calculated with the following equation:

$$d = \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2}$$

The KNN method was chosen as the classifier for this research because it is an algorithm simple to implement, effective and low time-consuming for the decision (due to the small size of the database, 78 samples). Indeed, by only computing the distance between samples and then looking for the K minimum distances among them the method is able to classify the event with high efficiency and accuracy. However, before applying the model to the input signal, a normalization of its 10 attributes is necessary in order to avoid possible bias in distance computation. In fact, as table I shows, there is a big variability in the range of the attributes, i.e. PSD and VLF power have statistical values that are much bigger than the other attributes. Therefore the normalization provides the model with lower distance differences between attributes, allowing all the attributes to have the same importance in the classification (otherwise attributes like PSD and VLF would have a higher importance in the classification due to its bigger orders).

Lastly, in order to decide the best KNN model, leave-one-out cross-validation was used to evaluate the best K value (parameter that defines the model). So, the K with the highest accuracy classifying the samples of the database would be the final model. In this special case of cross-validation, the number of folds is equal to the number of samples in the database, thus all the samples role as test once. Therefore, the learning algorithm is applied once per each sample (test set), using all the other samples as the training set. Leave-one-out was used to cross-validate instead of other alternatives more practiced due to the small size of the database.

The models considered of KNN were the range of K from 2 to 10 (9 models) because the size of the database limits to use a higher number of neighbors. So in summary, the 9 models were trained and tested 78 times, using once each sample as the test and the remaining 77 samples as training set. As a result, it was obtained that the optimal number of neighbours was K equal to 5. This model reported an accuracy of 84.6% in the classification of the database (only 12 out of 78 samples were miss-classified).

D. *App development*

Since it is being tried to develop a user-friendly solution to allow the patient to monitor his/her sleep without being in the hospital, a simple android application was created so that it could be accessible for everybody. The application, that is named *Sleep Disease Tracer* (SDT), has an intuitive user interface and the core (logic of the application) is based on the signal processing and machine learning models developed in *Matlab* that were previously analysed.

Besides the libraries and folders that are created by default when it is started a new project in *Android Studio*, the application is composed of three static resources, one library for the graphs, four activities and six classes, all of them listed in table II.

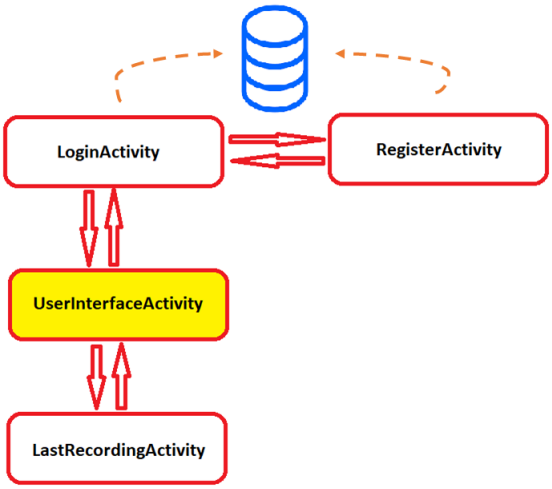

Fig. 7: Outline of the activities swapping in the android application.

Figure 7 shows how are organized the activities of the application. When the application starts running *LoginActivity*

Resource type	Name	Description
Activity	*UserInterfaceActivity*	Main Activity of the app. that controls start/stop recording, show results and log out
	LoginActivity	Activity to log in the app. that is connected with the database of Android Studio (SQLite)
	RegisterActivity	Activity to register in the app. that is required to access the application, also connected with SQLite
	LastRecordingActivity	Activity to show the results obtained in the last recording
Library	*com.jjoe64:graphview:4.2.2*	Library to load the classes needed to plot figures in Android Studio
Static Resource	*data.csv*	CSV with the database created for the machine learning classification
	datamean.csv	Mean of each attribute of the database
	datastd.csv	Standard deviation of each attribute of the database
Class	*Complex*	Class that defines the structure of a complex number
	DiscreteFourierTransform	Class that calculates the FFT of an input vector
	SignalProcessing	Class that process the signal and runs the machine learning to obtain a classification
	sleepSample	Class that defines the attributes used for the signal processing
	SoundDataUtils	Class to convert the bytes of the files (.pcm) into arrays of double values
	SQLite_helper	Class to access to and communicate with the SQLite database

TABLE II: Resources of the application.

(launcher activity) is the first screen that can be seen. This activity together with the registration activity (*RegisterActivity*) are the only ones that are in communication with the Android Studio database (*SQLite*), since their function is to verify if a user exists in order to access the application or register a new user to subsequently be able to enter the application. Once logged-in, there is a change of activity to the main activity (*UserInterfaceActivity*) where all the actions to record, process, log out and show results, are controlled. Lastly, *LastRecordingActivity* is used to show the results obtained out of the last recording.

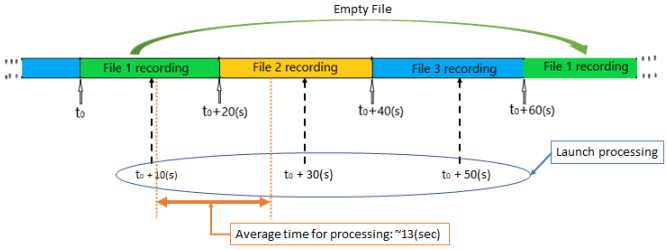

Fig. 8: Operating scheme of real-time processing, along with the three files that store the audio signal.

As regards the core of the application, it is controlled and launched from the main activity. The recording is carried out by the class *AudioRecord* that defines the sensor of the phone used (MIC), frequency sampling (8000 Hz), channel (input mono-channel) and the encoding (PCM, 16 bits). These recordings are saved in three files that allow the application to process the signal with certain overlapping without stopping the recording. Figure 8 shows how the recording process works: every 20 seconds a new file starts recording, since a 45-second chunk is being processed every 20 seconds and processing takes an average of 13 seconds, it will be able to continue recording the audio while the chunks are processed maintaining a 25-second overlap with the previous fragment. It is noteworthy that it is processed a fragment and recorded a file with the same period (20 seconds) to avoid the overlap between files that would make crashing the application.

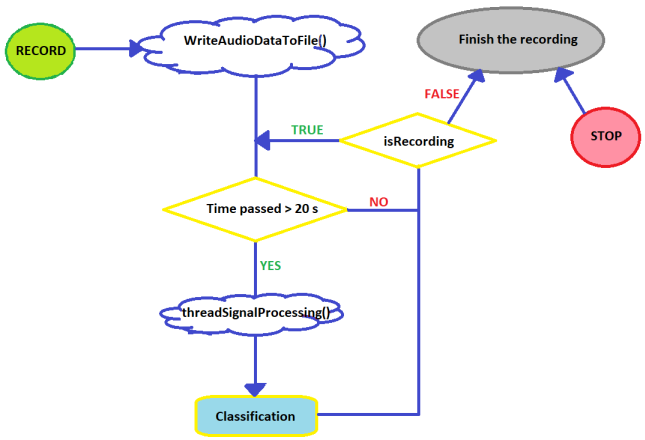

Fig. 9: Scheme of the actions that are running during the time audio is being recorded.

Figure 9 shows the flow-chart of the recording and processing control. Whenever the *record* button is pressed, a thread with *WriteAudioDataToFile()* method starts running. In this method, not only the recording process is controlled but also every 20 seconds is launched the signal processing in an inner thread. This loop keeps working while the *stop* button is not pressed. When this happens, the threads are set to null, the *AudioRecord* is stopped and the results (classification vector) are prepared to be used when the *LastRecordingActivity* will require them. Finally, the signal processing was based on the model developed and tested in *Matlab*, but some changes had to be done due to the different coding language. The *Fast Fourier Transform* (FFT) was implemented with the aid of some online research. On the other hand, *K-Nearest Neighbor* method for machine learning was also implemented from scratch. Lastly, since the recorder used allows to set the frequency sampling, it was avoid the block known as *resample* in Matlab that dealt with low-pass filtering and down-sampling (setting frequency sampling to 8000 Hz provides the app with the desired signal ready to process). In next section, it will be tested if all the changes listed above make differ the results from the ones obtained in *Matlab*.

III. Results

The results obtained will be divided into three parts. Firstly, results obtained in Matlab are commented. Then results got

from Java coding are compared with the desired results of Matlab. Lastly, the final results given by testing the application on the smartphone are presented.

A. *Matlab*

1) Signal processing: In Fig. 10 it is shown the results related to a subject taking into account a window size of 45 s. In order to make a comparison between the different events (classes), a normalization of the three signals was carried out after they had been filtered. Moreover, the energy of these signals was calculated by scrolling a $60\,ms$ window, as explained in methods section, and displayed in the bottom of the figure. This sort of figures were of great aid to do feature abstraction. Firstly, it is possible to notice how only a single event of sleep apnea happens in the examined time window and it is the one with the biggest time width in comparison to the other events, therefore the number of peaks (events) can be a useful attribute to discriminate this class (apnea). Secondly, it is easy to denote a similar number of events for normal breathing and snoring. Regarding their relative energies we can see how snoring energy is generally higher than normal breathing energy.

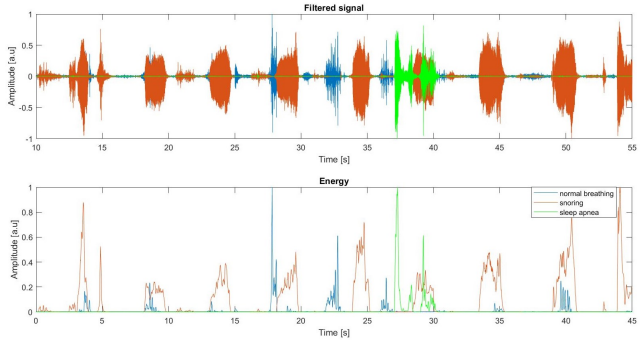

Fig. 10: (Top) Three samples normalized of normal breathing (blue), snoring (red) and sleep apnea (green) recordings from the same patient in time domain. (Bottom) Obtained energy out of each of the signals.

On the other hand, a comparison between the three signals for the same subject in frequency domain is shown in Fig. 11. As the previous case, a normalization was carried out to be able to compare the results with perspective. Firstly, we can notice that normal breathing frequency components are centered around a range that goes from 20 to $120\,Hz$ with a peak at $60\,Hz$. Regarding the sleep apnea signal it is possible to denote that its frequency range goes from 50 to $160\,Hz$ with a peak at $80\,Hz$. Lastly, the snoring signal is the one with the largest frequency range since it has frequency components from 50 to $220\,Hz$. The highest peak can be seen at $70\,Hz$.

Comparing the frequencies of the three classes treated is also of great help when it comes obtaining attributes, since it can be clearly seen how each class (normal, snore, apnea) moves in a certain frequency range.

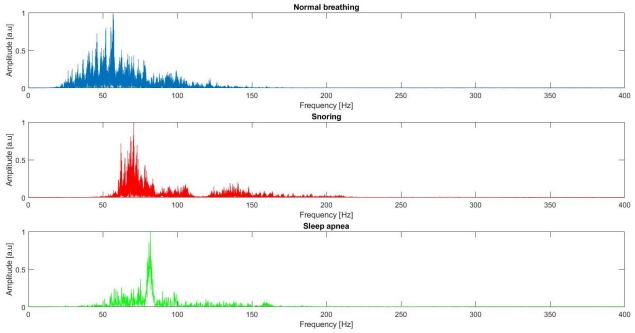

Fig. 11: Normalized spectrum of each of the signals represented in Fig. 10. In blue normal breathing, in red snoring and in green sleep apnea.

It is important to mention that all classes have an outstanding frequency at $50\,Hz$, which is believed to be noise generated by the device, although in this case of study it was considered as part of the signal, so filtering that frequency has been avoided.

2) Classification: In Table III are shown the results obtained by using leave-one-out cross-validation for a number of neighbours K that goes from 2 to 10. K equal to 1 was not taken into account because some samples were taken from different fragments of the same recording, thus it was considered that this would help to increase the accuracy of this model (K = 1) partially.

K	Accuracy	K	Accuracy	K	Accuracy
2	62.82%	5	84.62%	8	78.21%
3	83.33%	6	82.05%	9	80.77%
4	80.77%	7	82.05%	10	79.49%

TABLE III: Accuracy obtained with leave-one-out cross-validation for different K neighbours.

As the table lists, the highest accuracy is given by K equals to 5, since the accuracy in classification is 84.62%. As a result of this, a number of neighbours K equals to 5 was chosen as the model for classification.

B. *Comparison between Matlab and Java*

In this part of the section, the results obtained from Matlab and Java are shown and compared. The following results are related to the Fast Fourier Transform (FFT), attributes and classification. The FFT and attributes results are compared by using one sample during sleep apnea.

1) Fast Fourier Transform (FFT): as stated in methods section, the FFT implementation in Java needs to be validated. As a result of this, it is important to compare these results with the ones obtained for the FFT in Matlab in order to have a confirmation that the Java method works correctly.

Fig. 12 shows the FFT results from the same sample in Matlab and in Java. The Fourier transform of a real discrete signal results in a periodic spectrum and symmetric in the origin. In the sample of study, the *FFT* recovered in Java has a periodicity of $8000\,Hz$ and the period returned is from 0 to $8000\,Hz$, thus the fragment of the spectrum that goes from 4000 to $8000\,Hz$ was shifted in order to center the spectrum at in the origin. Although it is aware that there are no negative

frequencies, this displacement serves to validate the symmetry with respect to the origin of the recovered Fourier transform (this explains the frequency range plotted from -4000 to $4000\,Hz$). Moreover, the signal recovered in Matlab is also centered in the origin, therefore, the negative frequencies help to validate that the spectrum shift is applied properly. As figure shows, it is possible to denote that the FFT calculated in Java fits perfectly to the one computed in Matlab. Therefore it can be concluded that the Java method is working in the same way as Matlab one.

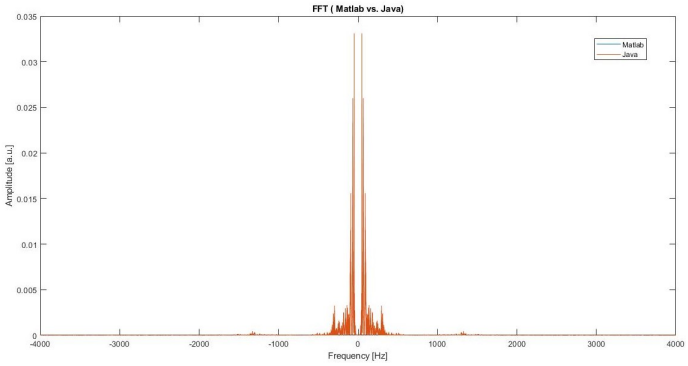

Fig. 12: Comparison of the FFT obtained from one sample in Java (red) and Matlab (blue).

2) Attributes: since all the coding in Java was created from scratch, it is necessary to evaluate the goodness of these methods implemented compared to Matlab. In order to do this, a comparison between the measurements of the 10 attributes obtained from Matlab and Java of the sample used to validate was carried out.

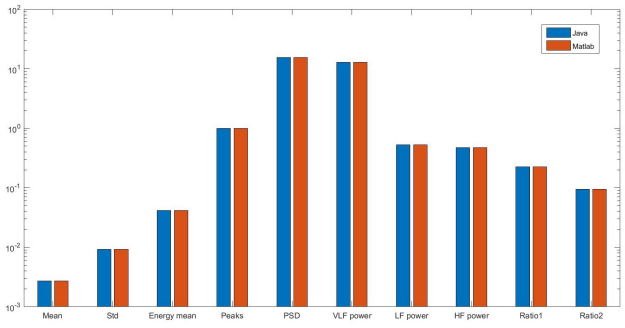

Fig. 13: Measurements obtained from one random sample of the 10 attributes extracted in Java (blue) and Matlab (red).

Fig. 13 shows the values of the 10 attributes computed in Matlab and Java from the test sample (sleep apnea recording). The y-axis is presented on a logarithmic scale to make it possible to appreciate all the attributes, if, on the contrary, a normal scale had been used, the difference in orders of magnitude between attributes would mean that the smallest could not be practically appreciated. It is easy to denote that the values are exactly the same for all the attributes in both programming languages. Therefore it is possible to conclude that the methods implemented in Java to calculate the features work properly.

3) Classification: a comparison between machine learning results obtained in Matlab and Java is carried out in order to test the correctness of the Java methods used for classification. To do this, 10 random samples compiling the three classes were considered for evaluation. With the KNN algorithm implemented in Java it was possible to obtain the same classification for all the samples as in Matlab (meaning that the two KNN classify correctly and wrongly in the same way and the same samples). As a result of this, it can be assumed that the KNN model implemented and used in the android application (Java) works in the same way as the one used in Matlab and, thus, the accuracy of the model K equal to 5 in Java is 84.6% (like in Matlab). So in conclusion, the Java method for classification is validated positively and with great precision.

C. App testing

Once validated all the coding developed in Java, in last instance it is needed to test the performance of the android application developed in *Android Studio* with Java. In order to do so, a demo was programmed simulating the recording process and processing the signal at the same time that the recording is running. Therefore, a vector of 180 seconds and frequency sampling of $8\,kHz$ was composed and loaded into the application as a static resource.

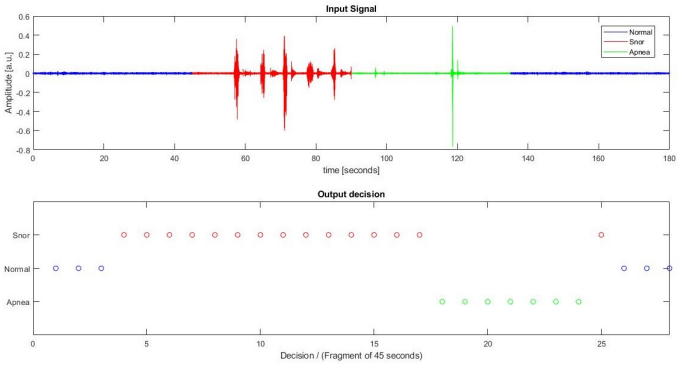

Fig. 14: Result of the demo carried out by the application with an input audio with a duration of three minutes and the different alternating classes (normal breathing in blue, snoring in red, apnea in green). (Top) Input signal of the demo. (Bottom) Classification of the input signal processing fragments of 45 seconds and 40 seconds of overlap.

The recording consists of four fragments of 45 seconds each as shown in the top part of the Fig 14, where the first and the last fragments (blue) are different recordings of normal breathing from the same patient. While, the second fragment (red) and the third fragment (green) are snore and apnea, respectively, from the same patient.

Due to the limitation of static resource size by android studio, a longer demo time could not be generated. However, it has been decided to increase the window overlap to almost 90% (40 seconds) in order to obtain a greater number of classifications. Therefore, the results of the real-time classification are shown

in the bottom part of the Fig 14. It can be seen that the classification is in accordance with the order in which the different classes are introduced in the demo vector, apart from one clear miss-classification in the 25th fragment (classified snore when it should be classified either apnea or normal). Nonetheless, there is an evident majority of classifications of the snore class because it is in the middle of the audio vector and there is a great overlap, so the contribution of the snore signal has a great presence, coming to decide a favorable classification for this class.

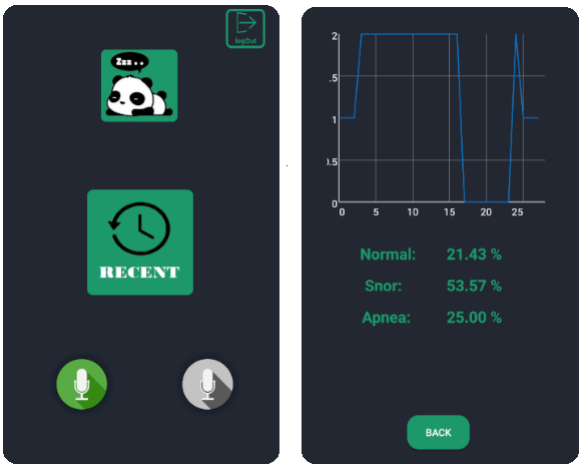

Fig. 15: (Left) Main activity of the final application. (Right) Activity that shows the results of the recordings done previously.

As a last instance, Fig 15 shows the user interface of the developed application. The image on the left corresponds to the main screen of the application (main activity), while the right corresponds to the graphic design that was decided to show the results. In this case, the same results are being shown as in Fig 14. Finally, it is noteworthy that the design of the application has been made in such a way that it adapts to the hours in which the user is predisposed to sleep and, therefore, it has been decided to use dark shades that favor sleep both for the background of the application as for the rest of the graphics that make up the user interface.

IV. Conclusions

The results and conclusions obtained throughout the project have generally been positive. On the one hand, an attempt has been made to search for simplicity when processing the signal or to select the classification model, but without neglecting to seek high efficiency and accuracy in the proposed solution. On the other hand, the short period of time provided for the development of the project required obtaining solutions as soon as possible, thus leaving room for maneuver in the event that the tests were to fail in any of the developments. Therefore, having achieved an efficiency in the classification (main objective of the project along with detection of the event) of almost 85% can be considered a great result.

Moreover, some attributes generated throughout the study could be of great importance in a more consolidated application if the way to obtain them were made more precise. For example, it can be seen at a glance that event detection could be a very favorable characteristic for apnea discrimination, so the development of a better and adaptive threshold could be useful.

However, due to the short time in which the project has been developed, several things have remained without being perfected or, even, without a final implementation. Therefore, a series of possible improvements to the application will be proposed, with mind focused on future developments.

Firstly, the recovery of the data stored in the files with the extension *'pcm'* was returning erroneous values despite being well recorded, so it would be the main objective of improvement to make the application work 100% independently.

On the other hand, the signal processing implemented in real time works with a very large vector and a very low overlap due to the computational load. It would be a great improvement to minimize the size of the window (currently 45 seconds) to reduce the vector of points and thus be able to reduce the processing time of each window (currently around 13 seconds). Another possible option to reduce the number of points could be to reduce the sampling frequency, ignoring the higher frequencies generated by the trachea (since they have almost no influence on the characterization of the signals). If the processing time is reduced it can be increased the overlapping and, thus, increase the resolution of the prediction.

To conclude, despite the fact that the results achieved are very positive considering the time allocated, the application lacks a maturation that, over time, can significantly improve the detection and classification of the audio signals.

References

[1] M. Al-Mardini, F. Aloul, A. Sagahyroon, and L. Al-Husseini, "Classifying obstructive sleep apnea using smartphones," *Journal of biomedical informatics*, vol. 52, pp. 251–259, 2014.

[2] "https://en.wikipedia.org/wiki/obstructive_sleep_apnea."

[3] N. Oliver and F. Flores-Mangas, "Healthgear: Automatic sleep apnea detection and monitoring with a mobile phone." *JCM*, vol. 2, no. 2, pp. 1–9, 2007.

[4] E. Dafna, A. Tarasiuk, and Y. Zigel, "Automatic detection of whole night snoring events using non-contact microphone," *PloS one*, vol. 8, no. 12, 2013.

[5] R. Nandakumar, S. Gollakota, and N. Watson, "Contactless sleep apnea detection on smartphones," in *Proceedings of the 13th annual international conference on mobile systems, applications, and services*, 2015, pp. 45–57.

[6] G. Lyon, R. Tiron, A. Zaffaroni, A. Osman, H. Kilroy, K. Lederer, I. Fietze, and T. Penzel, "Detection of sleep apnea using sonar smartphone technology," in *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE, 2019, pp. 7193–7196.

[7] K. N. Priftis, L. J. Hadjileontiadis, and M. L. Everard, "Breath sounds," *From Basic Science to Clinical Practice. 1st Ed., Cham: Springer*, 2018.

[8] E. J. Candès *et al.*, "Compressive sampling," in *Proceedings of the international congress of mathematicians*, vol. 3. Madrid, Spain, 2006, pp. 1433–1452.

[9] L. E. Peterson, "K-nearest neighbor," *Scholarpedia*, vol. 4, no. 2, p. 1883, 2009.

Appendices

1 Figures

Due to the low resolution shown in some figures in the report, it was decided to add them in the appendix and, thus clarify the results and conclusions obtained.

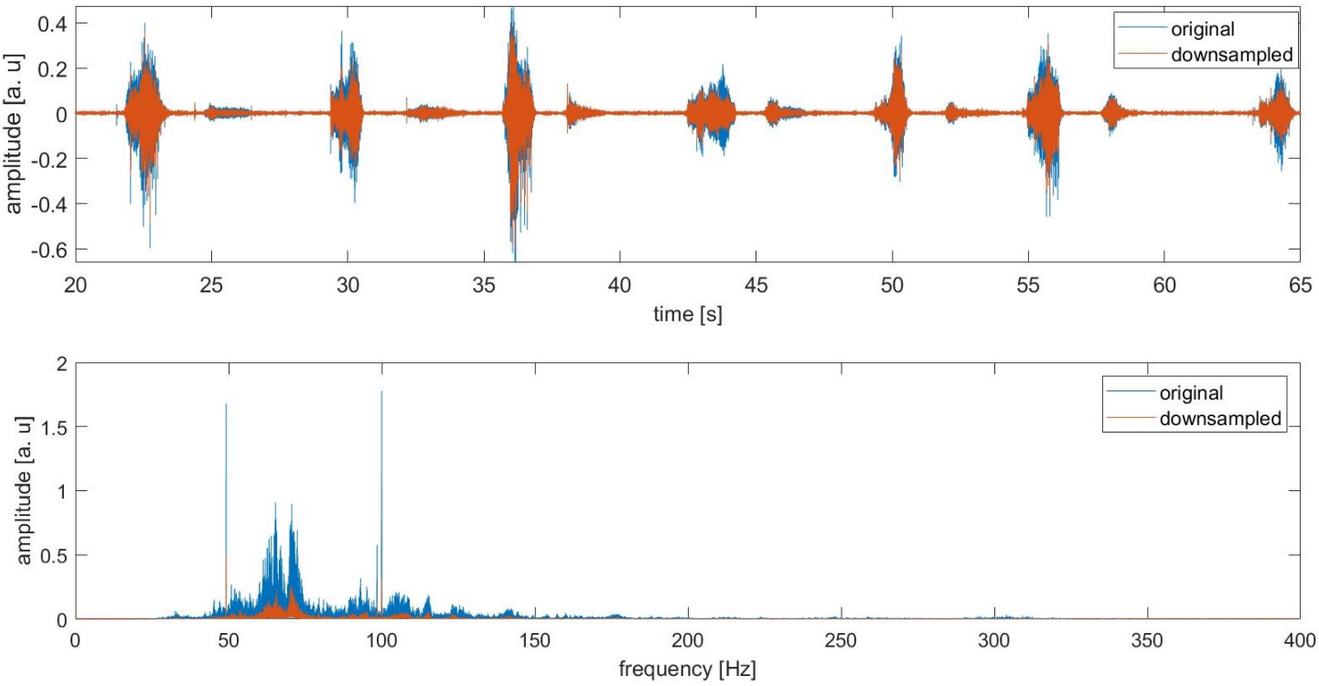

Figure 3: Comparison plot between an original signal randomly recorded (blue) and the signal after down-sampling (red) in time domain (top) and frequency domain (bottom).

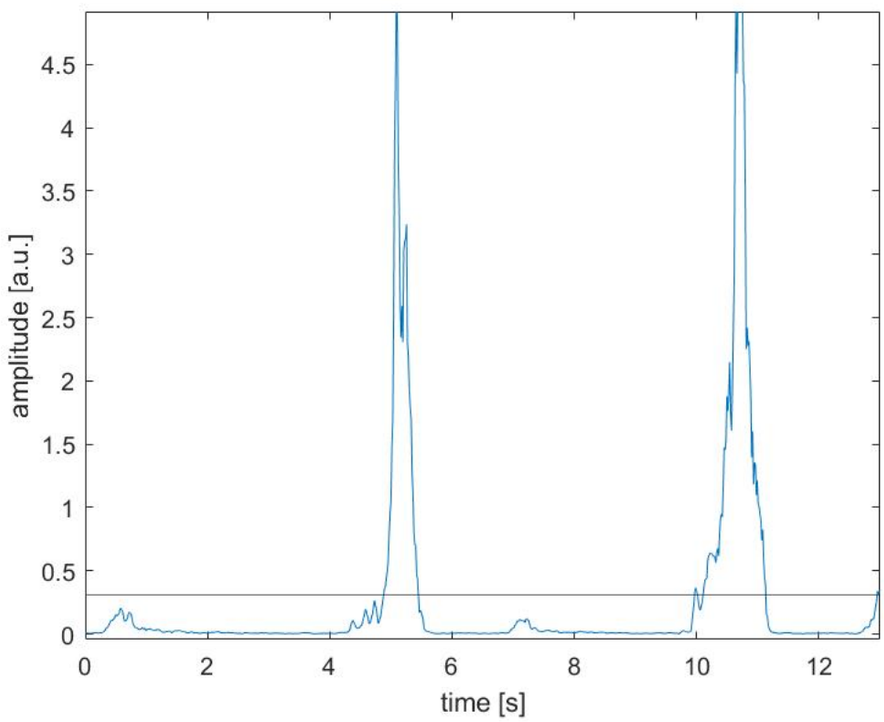

Figure 4: Energy of a random down-sampled signal (snoring class). A fixed threshold (horizontal line) is set as 180% of energy mean. Two peaks are detected at 5 and 10.7 s.

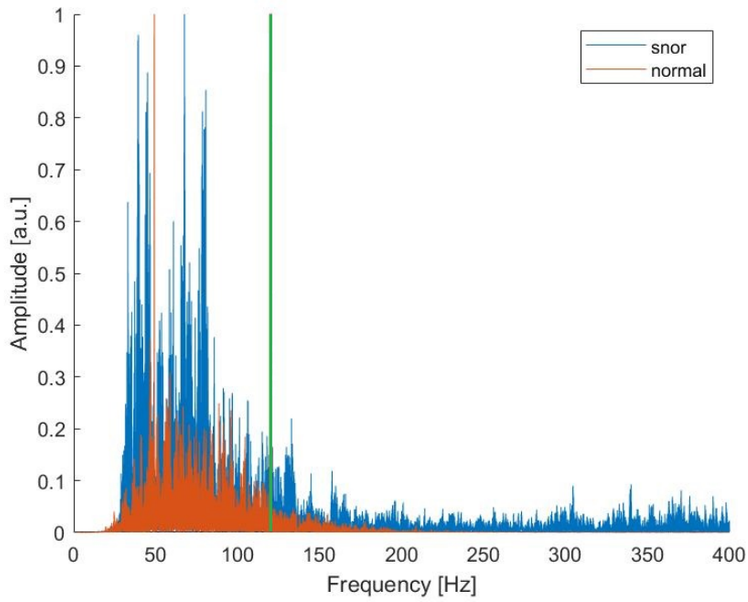

Figure 5: Frequency components of two random samples of normal breathing and snoring. The spectrum is divided by the red line at $120\,Hz$.

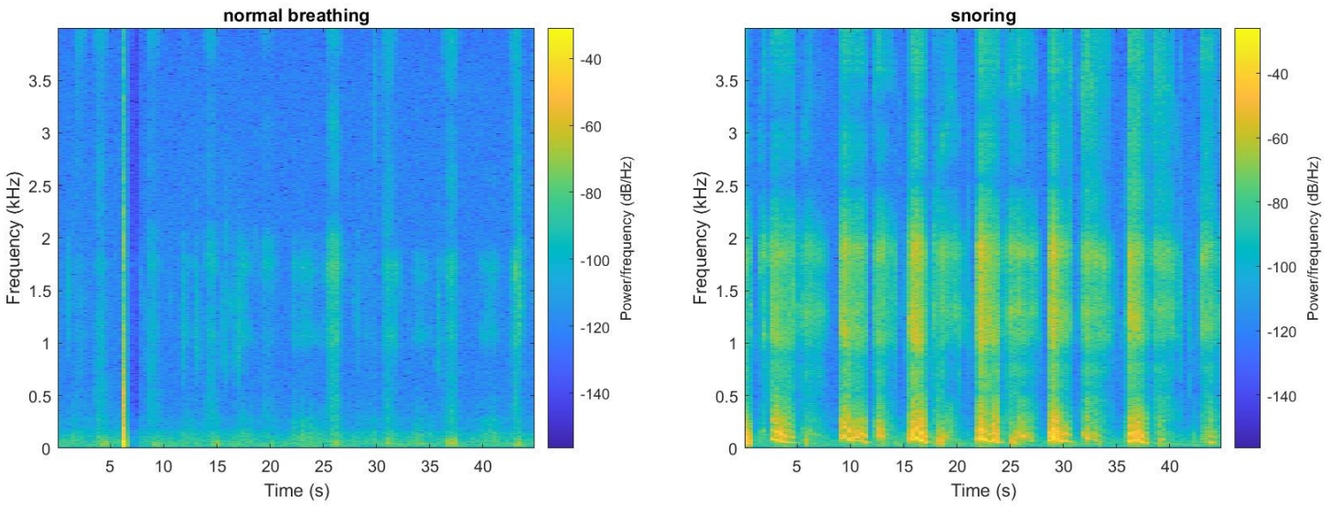

Figure 6: Spectrograms (plot of time vs. frequency domain) of two random samples of normal breathing (left) and snoring (right).

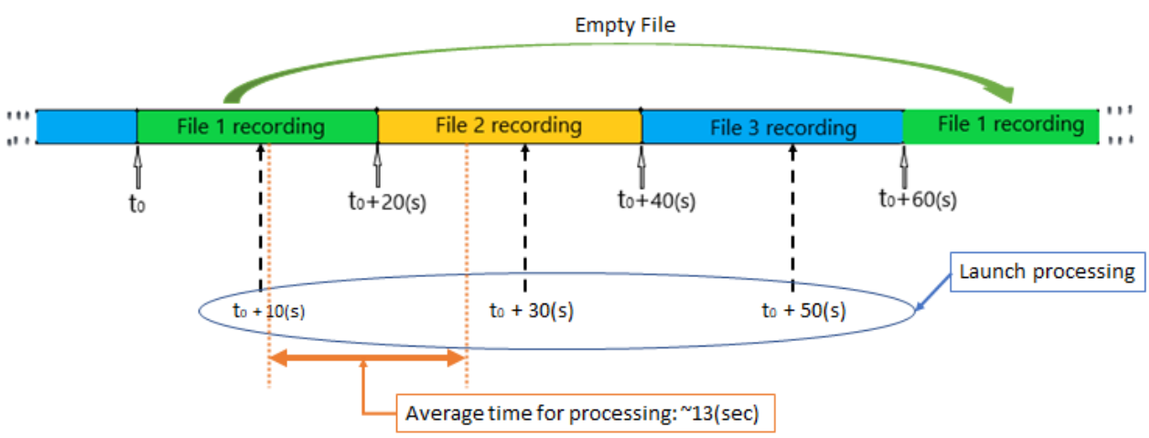

Figure 8: Operating scheme of real-time processing, along with the three files that store the audio signal.

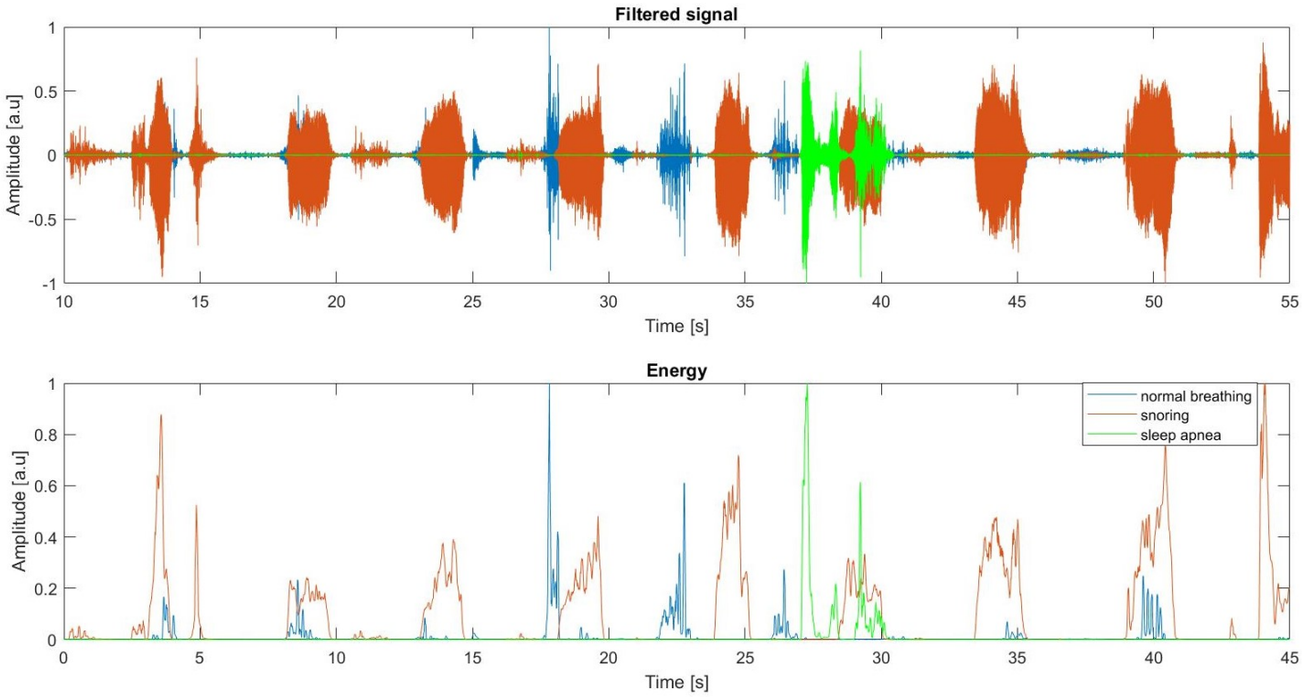

Figure 10: (Top) Three samples normalized of normal breathing (blue), snoring (red) and sleep apnea (green) recordings from the same patient in time domain. (Bottom) Obtained energy out of each of the signals.

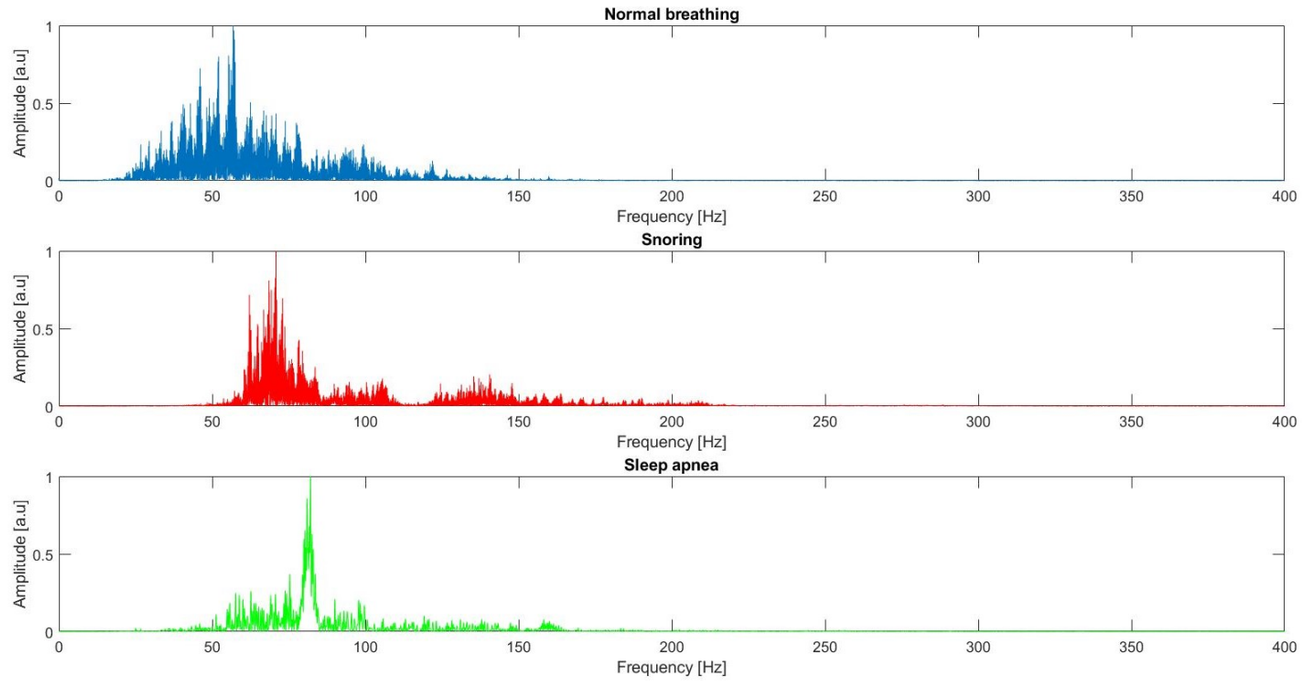

Figure 11: Normalized spectrum of each of the signals represented in Fig. 10. In blue normal breathing, in red snoring and in green sleep apnea.

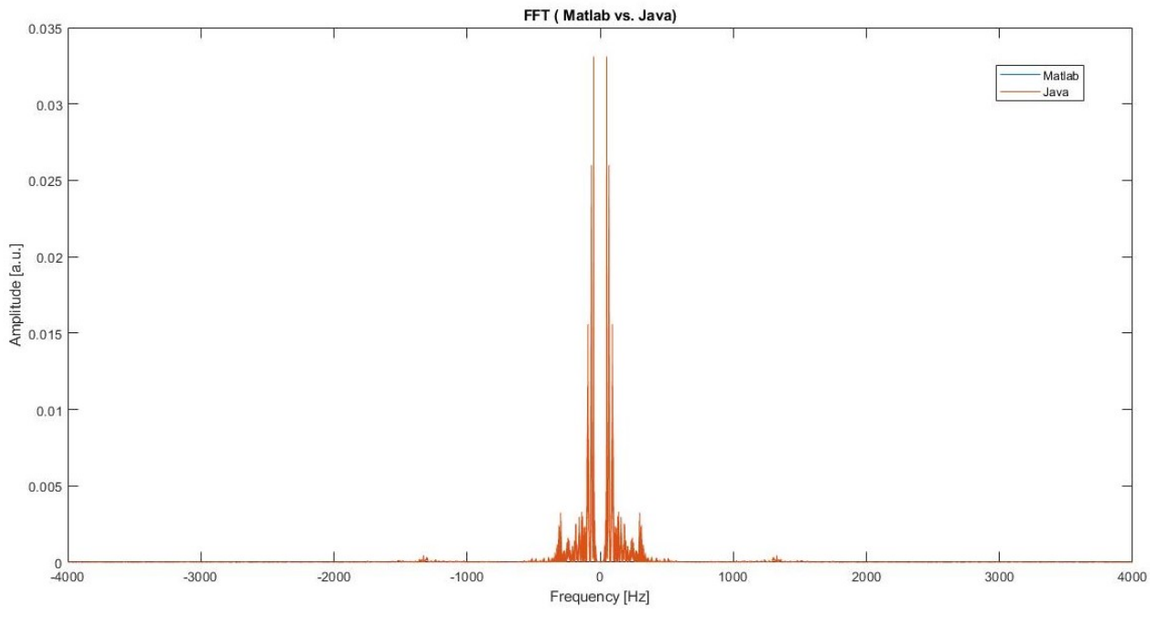

Figure 12: Comparison of the FFT obtained from one sample in Java (red) and Matlab (blue).

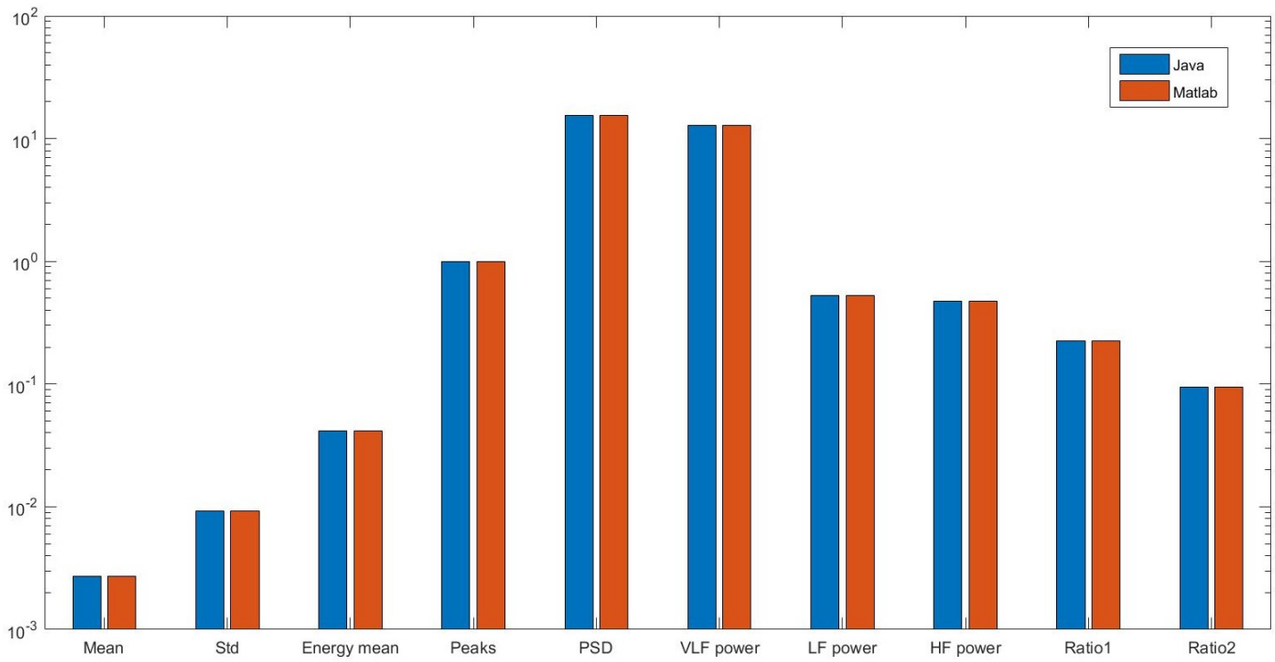

Figure 13: Measurements obtained from one random sample of the 10 attributes extracted in Java (blue) and Matlab (red).

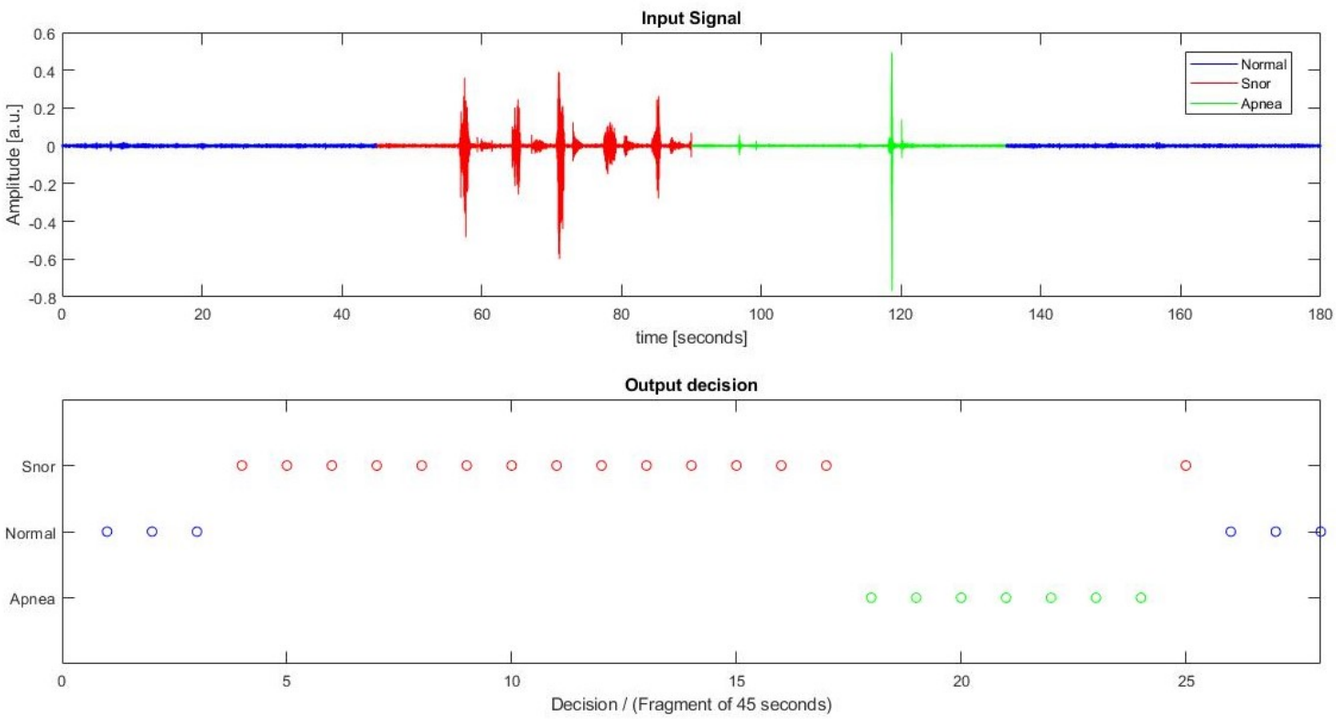

Figure 14: Result of the demo carried out by the application with an input audio with a duration of three minutes and the different alternating classes (normal breathing in blue, snoring in red, apnea in green). (Top) Input signal of the demo. (Bottom) Classification of the input signal processing fragments of 45 seconds and 40 seconds of overlap.

2 Java Scripts

```
package com.portable.sleepdiseasetracer;

import androidx.annotation.RequiresApi;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;

import android.Manifest;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.media.AudioFormat;
import android.media.AudioRecord;
import android.media.MediaRecorder;
import android.os.Build;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import android.view.View;
import android.widget.ImageButton;
import android.widget.Toast;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOError;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.nio.charset.Charset;
import java.util.ArrayList;

/* Main Activity of the application.
*   It starts running after the user logs in. It is controlling 4 ImageButtons that run the main actions of the app.
*       1) The record ImageButton: starts the audio recording after preparing the AudioRecord. The recording is run
*   in a thread. Meanwhile, every 20 seconds fragments of 45 seconds of the recording are being processed by the class
*   'SignalProcessing.java'. It uses 3 .pcm files to control the overlap of the fragments and keep a track of all the recording.
*       2) The stop ImageButton: it is active after pressing the record ImageButton and whenever is pressed: it stops the recording,
*   sets to null the recording a processing threads and prepares the results obtained from the processing (decisions).
*       3) The last ImageButton: launches an activity that show graphically the results obtained in the last recording run.
*       4) The logout ImageButton: whenever is pressed the user logs out of the application and the activity is changed to the
*   log in activity.
*
*   Last Modified: 07/05/2020
* */
public class UserInterfaceActivity extends AppCompatActivity {
    private ImageButton stop;
    private ImageButton record;
    private static final String LOG_TAG = "AudioRecord";
    private static File path = null;
    private double[] signal;
    // It will be used 3 files and 3 fileOutputStream to control the overlap of the fragments processed:
    private File fileName;
    private File fileName2;
    private File fileName3;
    private FileOutputStream createFile;
    private FileOutputStream createFile2;
    private FileOutputStream createFile3;
    // Defining the characteristics of the AudioRecord (with a frequency sampling of 8kHz):
    private static final int RECORDER_SAMPLERATE = 8000;
    private static final int RECORDER_CHANNELS = AudioFormat.CHANNEL_IN_MONO;
    private static final int RECORDER_AUDIO_ENCODING = AudioFormat.ENCODING_PCM_16BIT;
    int BufferElements2Rec = 1024; // want to use a buffer of 2048 (2K) since each record has 2 bytes, we use a size of 1024
    int BytesPerElement = 2; // 2 bytes in 16bit format
    // Variables for AudioRecord and threads:
    private AudioRecord recorder2 = null;
    private Thread recordingThread = null;
    private Thread processingThread = null;
    private boolean isRecording = false;
    private boolean isProcessing = false;
    // Variable to control the order to reconstruct the signal vector from the different files:
    int fragmentsCount = 1;
    // Variables of the results obtained after the processing:
    private int[] decisionVec;
    private ArrayList<Integer> decisions = new ArrayList<>();
    // Boolean to set demo scenario:
    private static boolean isDemo = true;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_userinterface);
        // Start the control of the 4 ImageButtons:
        stop = (ImageButton) findViewById(R.id.stop);
        record = (ImageButton) findViewById(R.id.record);
        final ImageButton last = (ImageButton) findViewById(R.id.last);
        ImageButton logout = (ImageButton) findViewById(R.id.logout);
```

```
        stop.setEnabled(false); // Disable stop ImageButton until it is pressed the record button.
        // Record to the external storage directory for visibility:
        path = Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);
        fileName = new File(path,"/voice8K16bitmono.pcm");
        fileName2 = new File(path,"/voice8K16bitmono2.pcm");
        fileName3 = new File(path,"/voice8K16bitmono3.pcm");
        // Request for the needed permissions:
        try {
            ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.RECORD_AUDIO},
    PackageManager.PERMISSION_GRANTED);
            ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.
    WRITE_EXTERNAL_STORAGE}, PackageManager.PERMISSION_GRANTED);
            ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.READ_EXTERNAL_STORAGE
    }, PackageManager.PERMISSION_GRANTED);
        }catch (IOError e){
            Log.e(LOG_TAG, "Error granting permissions: " + e);
        }

        record.setOnClickListener(new View.OnClickListener() {
            @RequiresApi(api = Build.VERSION_CODES.LOLLIPOP)
            @Override
            public void onClick(View v) {
                // Set record button to grey and stop to red:
                record.setBackground(getDrawable(R.drawable.mic_grey));
                stop.setBackground(getDrawable(R.drawable.micb));
                Toast.makeText(getApplicationContext(), "Recording started", Toast.LENGTH_LONG).show();
                // If it is running the demo we use a file already recorded:
                if (isDemo){
                    Log.e(LOG_TAG, "Is demo!");
                    try {
                        runDemo();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                } else {
                    try {
                        createFile = new FileOutputStream(fileName);
                        createFile2 = new FileOutputStream(fileName2);
                        createFile3 = new FileOutputStream(fileName3);
                    } catch (FileNotFoundException e) {
                        e.printStackTrace();
                        Log.e(LOG_TAG, "Error creating files: " + e);
                    }
                    // Prepare and start the AudioRecord:
                    recorder2 = new AudioRecord(MediaRecorder.AudioSource.MIC,
                            RECORDER_SAMPLERATE, RECORDER_CHANNELS,
                            RECORDER_AUDIO_ENCODING, BufferElements2Rec * BytesPerElement);

                    recorder2.startRecording();
                    isRecording = true;
                    // Prepare a thread to control the audio recording:
                    recordingThread = new Thread(new Runnable() {
                        public void run() {
                            try {
                                writeAudioDataToFile();
                            } catch (FileNotFoundException e) {
                                e.printStackTrace();
                            }
                        }
                    }, "AudioRecorder Thread");
                    // Launch the thread (run writeAudioDataToFile() method):
                    recordingThread.start();
                    // Enable stop and disable record:
                }
                stop.setEnabled(true);
                record.setEnabled(false);
            }
        });

        stop.setOnClickListener(new View.OnClickListener() {
            @RequiresApi(api = Build.VERSION_CODES.LOLLIPOP)
            @Override
            public void onClick(View v) {
                Log.e(LOG_TAG, "Finish record!");
                // Set record button to green and stop to grey:
                record.setBackground(getDrawable(R.drawable.micr));
                stop.setBackground(getDrawable(R.drawable.mic_grey));
                // If it is running the demo we use a file already recorded:
                if (!isDemo){
                    // Stop the recording activity:
                    if (null != recorder2) {
                        isRecording = false;
                        recorder2.stop();
                        recorder2.release();
                        recorder2 = null;
                        recordingThread = null;
                        processingThread = null;
                    }
                }
                Log.d(LOG_TAG, "Decision Vector : "  + decisions.size());
                // Prepare the result to use them in other activities:
                decisionVec = new int[decisions.size()];
                for (int i = 0; i < decisions.size(); i++){
                    decisionVec[i] = decisions.get(i);
                }
                // Enable record and disable stop:
                record.setEnabled(true);
                stop.setEnabled(false);
```

```
                Toast.makeText(getApplicationContext(), "Audio Recorded successfully", Toast.LENGTH_LONG).show();
            }
        });

        last.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Changes to LastRecordingActivity and it passes the decisions obtained in the processing:
                Intent i = new Intent(getApplicationContext(),LastRecordingActivity.class); // Show the last recording
                if(decisionVec != null){
                    i.putExtra("decisionVector",decisionVec);
                }
                startActivity(i);
            }
        });

        logout.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Logs out from the main activity and changes to LoginActivity:
                Intent i = new Intent(getApplicationContext(),LoginActivity.class); // Logs out of the application
                startActivity(i);
            }
        });
    }


    private void writeAudioDataToFile() throws FileNotFoundException {
        /*  Method used to record audio with the microphone while the boolean 'isRecording' is set to true.
         *   In addition, it launches the signal processing for fragments of 45 seconds: the first fragment is launched
         *   after 50 seconds of recording, the rest every 20 seconds.
         *
         *   Last Modified: 07/05/2020
         * */
        short sData[] = new short[BufferElements2Rec];
        int fileNumber = 1;
        long t0 = System.currentTimeMillis(); // Time to control the recording (every 20 sec. it starts to record in a diferent file)
        long tProcessing = System.currentTimeMillis(); // Time to control the processing
        boolean firstTime = true; // Boolean to control that the first process starts after 50 sec.
        // Prepare thread to run the processing while the app. keeps recording:
        processingThread = new Thread(new Runnable() {
            public void run() {
                try {
                    threadProcess();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }, "SignalProcessing Thread");

        while (isRecording) {
            // First processing after 50 sec. the rest after 20 secs., in the case that 20 sec. have passed
            // and the previous processing didn't finish we skip the next fragment to avoid the thread from crashing:
            if (System.currentTimeMillis() - tProcessing >= 50000 && firstTime){
                tProcessing = System.currentTimeMillis();
                firstTime = false;
                processingThread.start();
            } else if (System.currentTimeMillis() - tProcessing >= 20000 && !firstTime && !isProcessing){
                tProcessing = System.currentTimeMillis();
                processingThread.start();
                isProcessing = true;
                Log.e(LOG_TAG, "Launch thread! ");
            } else if (System.currentTimeMillis() - tProcessing >= 20000 && !firstTime && isProcessing){
                signal = null;
                if (fragmentsCount == 1){
                    createFile = new FileOutputStream(fileName);
                    fragmentsCount = 2;
                } else if (fragmentsCount == 2){
                    createFile2 = new FileOutputStream(fileName2);
                    fragmentsCount = 3;
                } else {
                    createFile3 = new FileOutputStream(fileName3);
                    fragmentsCount = 1;
                }
            }
            // Loop to control in which file we have to record every 20 sec:
            if (System.currentTimeMillis() - t0 >= 20000) {
                t0 = System.currentTimeMillis();
                if (fileNumber == 1){
                    fileNumber = 2;
                } else if (fileNumber == 2){
                    fileNumber = 3;
                } else if (fileNumber == 3){
                    fileNumber = 1;
                }
            } else {
                // Get the voice output from microphone to byte format.
                // Buffer data:
                recorder2.read(sData, 0, BufferElements2Rec);
```

```
                System.out.println("Short wirting to file" + sData.toString());
                try {
                    // Writes the data to file from buffer and stores the voice buffer:
                    byte bData[] = short2byte(sData);
                    if(fileNumber == 1){
                        createFile.write(bData, 0, BufferElements2Rec * BytesPerElement);
                    } else if(fileNumber == 2){
                        createFile2.write(bData, 0, BufferElements2Rec * BytesPerElement);
                    } else{
                        createFile3.write(bData, 0, BufferElements2Rec * BytesPerElement);
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }


    private byte[] short2byte(short[] sData) {
        /*  Method to convert short to byte.
         *
         *   Last Modified: 05/05/2020
         * */
        int shortArrsize = sData.length;
        byte[] bytes = new byte[shortArrsize * 2];
        for (int i = 0; i < shortArrsize; i++) {
            bytes[i * 2] = (byte) (sData[i] & 0x00FF);
            bytes[(i * 2) + 1] = (byte) (sData[i] >> 8);
            sData[i] = 0;
        }
        return bytes;
    }


    public void threadProcess () throws IOException {
        /*  Method to prepare and launch the signal processing. Reads the data that is stored in the 3 .pcm files and
         *  prepares the signal vector taking into account the order of recording of the files.
         *  In addition, it only allows files of 45 sec (8000*45 = 360000 points) which is the size used to prepare the database.
         *
         *   Last Modified: 07/05/2020
         * */
        SoundDataUtils util = new SoundDataUtils();
        // Convert the data of the files into doubles arrays:
        double[] extra = null;
        double[] signal1 = util.load16BitPCMRawDataFileAsDoubleArray(fileName);
        double[] signal2 = util.load16BitPCMRawDataFileAsDoubleArray(fileName2);
        double[] signal3 = util.load16BitPCMRawDataFileAsDoubleArray(fileName3);

        int totalPoints = signal1.length + signal2.length + signal3.length;
        // If there are more points than 360000 in the files, we remove the first part of the recording to make sure that the fragment size
        // is of 45 seconds:
        int shift = totalPoints - 360000;
        Log.d(LOG_TAG, "Points -> signal1: " + signal1.length + " - signal2: " + signal2.length + " - signal3: " + signal3.length + " - shift: " + shift);
        // If there are less points than 360000 we copy the last points of the previous signal to make 45 seconds:
        if(shift < 0 && signal != null){
            extra = new double[Math.abs(shift)];
            System.arraycopy(signal,(signal.length - Math.abs(shift)),extra,0,Math.abs(shift));
            shift = 0;
        }
        // Concatenate the files to form the signal vector (size of 360000):
        double[] signalAux = new double[totalPoints - shift];
        if (fragmentsCount == 1){
            System.arraycopy(signal1,shift,signalAux,0,(signal1.length - shift));
            System.arraycopy(signal2,0,signalAux,(signal1.length - shift),signal2.length);
            System.arraycopy(signal3,0,signalAux,(signal1.length + signal2.length - shift),signal3.length);
            createFile = new FileOutputStream(fileName);
            fragmentsCount = 2;
        } else if (fragmentsCount == 2){
            System.arraycopy(signal2,shift,signalAux,0,(signal2.length - shift));
            System.arraycopy(signal3,0,signalAux,(signal2.length - shift),signal3.length);
            System.arraycopy(signal1,0,signalAux,(signal2.length + signal3.length - shift),signal1.length);
            createFile2 = new FileOutputStream(fileName2);
            fragmentsCount = 3;
        } else {
            System.arraycopy(signal3,shift,signalAux,0,(signal3.length - shift));
            System.arraycopy(signal1,0,signalAux,(signal3.length - shift),signal1.length);
            System.arraycopy(signal2,0,signalAux,(signal3.length + signal1.length - shift),signal2.length);
            createFile3 = new FileOutputStream(fileName3);
            fragmentsCount = 1;
        }
        // If there was less points than 360000 in the files, we add a the beginning of the signal file the point from the previous fragment:
        if (extra != null){
            signal = new double[signalAux.length + extra.length];
            System.arraycopy(extra,0,signal,0,extra.length);
            System.arraycopy(signalAux,0,signal,extra.length,signalAux.length);
        } else {
            signal = signalAux;
        }
```

```
        Log.d(LOG_TAG, "Signal length:" + signal.length); // Has to be 360000
        try {
            // Prepare the static resources needed and launch the SignalProcessing class:
            InputStream database = getResources().openRawResource(R.raw.data);
            InputStream meanVector = getResources().openRawResource(R.raw.datamean);
            InputStream stdVector = getResources().openRawResource(R.raw.datastd);
            SignalProcessing sp = null;
            if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.N) {
                sp = new SignalProcessing(database,meanVector,stdVector,signal);
            }
            // Result obtained from the signal processing (classification: 0 = Apnea; 1 = Normal; 2 = Snor
    ):
            Log.d(LOG_TAG, "After signal processing, decision: " + sp.decisionValue);
            decisions.add(sp.decisionValue);
            // Prepare a new thread to launch the processing again for the next fragment:
            processingThread = null;
            processingThread = new Thread(new Runnable() {
                public void run() {
                    try {
                        threadProcess();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }, "SignalProcessing Thread");
            isProcessing = false;
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    public void runDemo() throws IOException {
        /*  Method to run the signal processing in demo mode. The signal is load from an static resource
    that
         *  was creates with 3 minutes of recording. It fragments the signal in 45 seconds with overlap of
     40 seconds.
         *
         *   Last Modified: 10/05/2020
         * */
        long tStart = System.currentTimeMillis(); // measure the time that takes process 3 minutes of
    signal
        double[] signalDemo = new double[360000];
        double[] windowSignal = new double[40000];
        int count = 0;
        int countFirstFragment = 0;
        boolean isFirstFragment = true;

        // Prepare the static resources needed run the demo:
        InputStream signalVector = getResources().openRawResource(R.raw.datademo);
        String line = "";
        // Obtain input signal and process it in fragments of 45 seconds with 35 seconds of overlap:
        BufferedReader csvSamples = new BufferedReader(new InputStreamReader(signalVector, Charset.forName
    ("UTF-8")));
        while ((line = csvSamples.readLine()) != null){
            if (isFirstFragment){
                signalDemo[countFirstFragment] = Double.valueOf(line);
                countFirstFragment++;
                if(countFirstFragment == signalDemo.length){
                    isFirstFragment = false;
                    // Prepare the static resources needed and launch the SignalProcessing class:
                    InputStream database = getResources().openRawResource(R.raw.data);
                    InputStream meanVector = getResources().openRawResource(R.raw.datamean);
                    InputStream stdVector = getResources().openRawResource(R.raw.datastd);
                    SignalProcessing sp = null;
                    if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.N) {
                        Log.d(LOG_TAG, "Signal size: " + signalDemo.length);
                        sp = new SignalProcessing(database,meanVector,stdVector,signalDemo);
                    }
                    // Result obtained from the signal processing (classification: 0 = Apnea; 1 = Normal;
    2 = Snor):
                    Log.d(LOG_TAG, "After signal processing, decision: " + sp.decisionValue);
                    decisions.add(sp.decisionValue);
                }
            } else {
                if (count < windowSignal.length - 1){
                    windowSignal[count] = Double.valueOf(line);
                    count++;
                } else {
                    windowSignal[count] = Double.valueOf(line);
                    System.arraycopy(signalDemo,windowSignal.length,signalDemo,0,(signalDemo.length-
    windowSignal.length));
                    System.arraycopy(windowSignal,0,signalDemo,(signalDemo.length-windowSignal.length),
    windowSignal.length);
                    // Prepare the static resources needed and launch the SignalProcessing class:
                    InputStream database = getResources().openRawResource(R.raw.data);
                    InputStream meanVector = getResources().openRawResource(R.raw.datamean);
                    InputStream stdVector = getResources().openRawResource(R.raw.datastd);
                    SignalProcessing sp = null;
                    if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.N) {
                        Log.d(LOG_TAG, "Signal size: " + signalDemo.length);
                        sp = new SignalProcessing(database,meanVector,stdVector,signalDemo);
                    }
                    // Result obtained from the signal processing (classification: 0 = Apnea; 1 = Normal;
    2 = Snor):
                    Log.d(LOG_TAG, "After signal processing, decision: " + sp.decisionValue);
                    decisions.add(sp.decisionValue);
                    // Record another fragment of 10 seconds:
```

```
                    count = 0;
                }
            }
        }
        Log.d(LOG_TAG, "TIME PROCESSING: " + (System.currentTimeMillis() - tStart));
        stopDemo();
    }

    public void stopDemo(){
        /*  Method to ...
         *
         *   Last Modified: 10/05/2020
         * */
        Log.d(LOG_TAG, "Before call onClick ");
        stop.callOnClick();
    }
}
```

Listing 1: UserInterfaceActivity.java

```
package com.portable.sleepdiseasetracer;

import androidx.appcompat.app.AppCompatActivity;
import android.content.Intent;
import android.database.Cursor;
import android.database.SQLException;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import DataBase.SQLite_helper;

/* Activity that launches the application.
 *   It controls that the user credentials are correct when it comes logging in, if so, it allows the user to enter the app.
 *   To control this, there is an access to the database (SQLite) to check that the user exists and his/her password is correct.
 *   It also controls the option to swap to the register activity.
 *
 *   Last Modified: 22/04/2020
 * */
public class LoginActivity extends AppCompatActivity {

    Button btnRegister;
    Button btnLogin;

    // Create an instance of the database:
    SQLite_helper helper = new SQLite_helper(this,"DB1",null,1);

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_login);

        btnRegister = (Button) findViewById(R.id.btnregister);

        btnRegister.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent i = new Intent(getApplicationContext(),RegisterActivity.class); // Switch to Register activity
                startActivity(i);
            }
        });

        btnLogin = (Button) findViewById(R.id.btnlogin);

        btnLogin.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                EditText txtUser = (EditText) findViewById(R.id.userText);
                EditText txtPassword = (EditText) findViewById(R.id.passwordText);

                try{
                    // Checks that the user is registered and the password is correct (it access to the SQLite database to do so):
                    Cursor userCursor = helper.consultUser(txtUser.getText().toString(),txtPassword.getText().toString());
                    // If we get one result back from the database the login credentials are correct, thus enters in the app.:
                    if(userCursor.getCount()>0){
                        Intent i = new Intent(getApplicationContext(),UserInterfaceActivity.class); // Enter the application
                        startActivity(i);
                    } else{
                        Toast.makeText(getApplicationContext(),
                                "Log in credentials are wrong. Verify that both user and password are well-written.",
                                Toast.LENGTH_LONG).show();
                    }
                    // If the credentials are incorrect, reset the input values of name and password:
                    txtUser.setText("");
                    txtPassword.setText("");
                    txtUser.findFocus();

                }catch (SQLException e){
                    e.printStackTrace();
```

```
                }
            }
        });
    }

}
```

Listing 2: LoginActivity.java

```
package com.portable.sleepdiseasetracer;

import androidx.appcompat.app.AppCompatActivity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import DataBase.SQLite_helper;

/* Activity for user registration.
 *   It controls the user registration, by taking the input values of user and password and saving them
 *   in the database (SQLite).
 *
 *   Last Modified: 22/04/2020
 * */
public class RegisterActivity extends AppCompatActivity {

    Button regUser;
    Button cancel;
    EditText txtUser, txtPassword;

    // Create an instance of the database:
    SQLite_helper helper = new SQLite_helper(this,"DB1",null,1);

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_register);

        regUser = (Button) findViewById(R.id.btnregister);
        cancel = (Button) findViewById(R.id.btncancel);
        txtUser = (EditText) findViewById(R.id.userText);
        txtPassword = (EditText) findViewById(R.id.passwordText);

        regUser.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Save the new user in the data base:
                helper.openDatabase();
                helper.insertRegister(String.valueOf(txtUser.getText()),String.valueOf(txtPassword.getText
    ()));
                helper.closeDatabase();

                Toast.makeText(getApplicationContext(),"You've been successfully registered",Toast.
    LENGTH_LONG).show();

                Intent i = new Intent(getApplicationContext(),LoginActivity.class); // Switch login
    activity
                startActivity(i);
            }
        });

        cancel.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent i = new Intent(getApplicationContext(),LoginActivity.class); // Switch login
    activity
                startActivity(i);
            }
        });
    }
}
```

Listing 3: RegisterActivity.java

```
package com.portable.sleepdiseasetracer;

import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;
import com.jjoe64.graphview.GraphView;
import com.jjoe64.graphview.series.DataPoint;
import com.jjoe64.graphview.series.LineGraphSeries;

/* Activity 'LastRecording'.
 *   It is an activity that shows the results obtained in the recordings.
 *      1) Shows a graph with the state of the patient during the recording.
 *      2) Shows the percentages of each state during the recording
 *
 *   Last Modified: 09/05/2020
 * */
public class LastRecordingActivity extends AppCompatActivity {

    Button back;
```

```
    TextView normalValue, snorValue, apneaValue;
    LineGraphSeries<DataPoint> series;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lastrecording);

        // Get the results of the processing from the main activity (UserInterfaceActivity):
        int[] decisionVec = getIntent().getIntArrayExtra("decisionVector");

        back = (Button) findViewById(R.id.back);
        normalValue = (TextView) findViewById(R.id.normalValue);
        snorValue = (TextView) findViewById(R.id.snorValue);
        apneaValue = (TextView) findViewById(R.id.apneaValue);
        GraphView graph = (GraphView) findViewById(R.id.graph);
        series = new LineGraphSeries<DataPoint>();

        double x = 0.0;
        int normal = 0;
        int snor = 0;
        int apnea = 0;
        assert decisionVec != null;
        for(int i = 0; i < decisionVec.length; i++){
            series.appendData(new DataPoint(x,decisionVec[i]), true, decisionVec.length);
            x = x + 1.0;
            // Calculate the number of time that is detected each state:
            if (decisionVec[i] == 0){
                apnea++;
            } else if (decisionVec[i] == 1){
                normal++;
            } else{
                snor++;
            }
        }
        // Set the parameters of the graph:
        graph.getViewport().setXAxisBoundsManual(true);
        graph.getViewport().setYAxisBoundsManual(true);
        graph.getViewport().setMinX(0);
        graph.getViewport().setMaxX(x);
        graph.getViewport().setMinY(0);
        graph.getViewport().setMaxY(2);
        graph.addSeries(series);

        double normalResult = (double) (100*normal)/decisionVec.length;
        double snorResult = (double) (100*snor)/decisionVec.length;
        double apneaResult = (double) (100*apnea)/decisionVec.length;
        normalValue.setText( String.format("%1.2f", normalResult) + " %");
        snorValue.setText( String.format("%1.2f", snorResult)+ " %");
        apneaValue.setText( String.format("%1.2f", apneaResult)+ " %");

        back.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent i = new Intent(getApplicationContext(),UserInterfaceActivity.class); // Switch UI
    activity
                startActivity(i);
            }
        });
    }

    public static double round(double value, int places) {
        /* Method that rounds to n (places) decimals a double.
         *
         *   Last Modified: 10/05/2020
         * */
        if (places < 0) throw new IllegalArgumentException();

        long factor = (long) Math.pow(10, places);
        value = value * factor;
        long tmp = Math.round(value);
        return (double) tmp / factor;
    }
}
```

Listing 4: LastRecordingActivity.java

```
package com.portable.sleepdiseasetracer;

import android.os.Build;
import android.util.Log;
import androidx.annotation.RequiresApi;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.nio.charset.Charset;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/* Class 'SignalProcessing'.
 *   Core of the application that receives as an input one double array (signal recorded)
 *   and process it to obtain the features of the recording. Once done this, the machine learning, based
    on
 *   K-nearest neighbor algorithm, is launched to classify each input fragment of 45 seconds.
 *
```

```
 *   Last Modified: 09/05/2020
 * */
public class SignalProcessing {

    public List<sleepSample> trainSleepSamples = new ArrayList<>();
    public sleepSample trainSleepSamplesMean;
    public sleepSample trainSleepSamplesStd;
    public int decisionValue;

    @RequiresApi(api = Build.VERSION_CODES.N)
    public SignalProcessing (InputStream database, InputStream meanVector, InputStream stdVector, double[]
     signal) throws FileNotFoundException {
        /* Method that controls all the processing run in the signal.
         *  Firstly, it launches a method 'loadData' to load the static resources used to classify the
    signal.
         *
         *   Last Modified: 01/05/2020
         * */
        Log.d("AudioRecord", "IN signal processing");
        loadData(database,meanVector,stdVector); // loading static resources (database, mean and std of
    the attributes of the database)
        decisionValue = processData(signal); // launch signal processing
        // This fragment commented is to split the signal into fragments if the input vector is bigger
    than 45 seconds:
     /*   Log.d("AudioRecord", "Before fragments");
        double[][] signalFragments = fragmentSignal(signal); // split signal into fragments of 45 seconds!
        if (signalFragments != null){
            decisionValue = new int[signalFragments.length];
            for (int i = 0; i < signalFragments.length; i++){
                decisionValue[i] = processData(signalFragments[i]);
            }
        }*/
    }

    private double[][] fragmentSignal(double[] signal) {
        /* Method that splits the signal into fragments of 45 seconds (size of the fragments that where
    used to create the database)
         *
         *   Last Modified: 01/05/2020
         * */
        // The signal will be processed in fragments of 45 seconds with an overlap of 35 seconds
        int windowSize = 8000*45; // freq sampling of 8KHz and 45 sec of length of fragment
        int windowDisplacement = 8000*10; // the window will be displaced 10 seconds every time
        int fragmentsNumber = (int) Math.round((signal.length - windowSize)/windowDisplacement + 0.5); //
    number of fragment that will be
        double[][] signalFragmentation; // array of arrays, to store all the fragments
        if (fragmentsNumber < 1){
            signalFragmentation = null;
        } else{
            signalFragmentation = new double[fragmentsNumber][windowSize];
            int index = 0;

            for (int i = 0; i < signalFragmentation.length; i++){
                System.arraycopy(signal,index,signalFragmentation[i],0,signalFragmentation[i].length);
                index = index + windowDisplacement;
            }
        }
        return signalFragmentation;
    }

    private void loadData (InputStream database, InputStream meanVector, InputStream stdVector) throws
    FileNotFoundException {
        /* Method that loads the static resources that will be used in the signal processing
         *
         *   Last Modified: 01/05/2020
         * */
        String line = "";
        try {
            // Obtain database:
            BufferedReader csvSamples = new BufferedReader(new InputStreamReader(database, Charset.forName
    ("UTF-8")));
            while ((line = csvSamples.readLine()) != null){
                String[] token = line.split(","); // split by ','
                // read the data:
                sleepSample sample = new sleepSample(token);
                trainSleepSamples.add(sample); // variable accessible to all the methods
            }
            // Obtain means of the database:
            line = "";
            csvSamples = new BufferedReader(new InputStreamReader(meanVector, Charset.forName("UTF-8")));
            while ((line = csvSamples.readLine()) != null){
                String[] token = line.split(",");
                sleepSample sample = new sleepSample(token);
                trainSleepSamplesMean = sample;// variable accessible to all the methods
            }
            // Obtain std of the database:
            line = "";
            csvSamples = new BufferedReader(new InputStreamReader(stdVector, Charset.forName("UTF-8")));
            while ((line = csvSamples.readLine()) != null){
                String[] token = line.split(",");
                sleepSample sample = new sleepSample(token);
                trainSleepSamplesStd = sample;// variable accessible to all the methods
            }
        } catch (IOException e) {
            Log.wtf("AudioRecord", "Error reading the data file:" + line, e);
            e.printStackTrace();
        }
    }
```

```
    @RequiresApi(api = Build.VERSION_CODES.N)
    private int processData (double[] fragment){
        /* Method that controls the data processing. It carries out all the feature abstraction.
         *
         *   Last Modified: 01/05/2020
         * */
        double[] token = new double[11];
        int index, prev_index;
        int decision = -1;

        // pre-processing:
        double[] energy = eventDetector(fragment); // calculate the energy in windows of 60 ms.
        int peaks = countPeaks(energy); // count the peaks that are above a fixed threshold

        // frequency domain:
        double[] frequency = DiscreteFourierTransform.calculateDFT(fragment); // obtain fft
        double[] freq_axis = new double[frequency.length]; // This two variables ar coming from FFT (call
    a method)
        // xaxis of the spectrum:
        double x = -4000;
        for(int w = 0; w < freq_axis.length; w++){
            freq_axis[w] = x;
            x = x + 0.0152588; // step size of: 1/8000
        }

        try{
            index = LookForIndex(freq_axis,0); // look for positive frequencies
            double[] positive_freq_axis = new double[freq_axis.length - index];
            System.arraycopy(freq_axis, index,positive_freq_axis,0,positive_freq_axis.length);
            double[] positive_frequency = new double[frequency.length - index];
            System.arraycopy(frequency, index,positive_frequency,0,positive_frequency.length);
            // look for the VLF (0 - 200 Hz):
            index = LookForIndex(positive_freq_axis,200); // look for frequencies bellow 200Hz
            double[] VLF = new double[index+1];
            System.arraycopy(positive_frequency,0,VLF,0,VLF.length);
            // look for the LF (200 - 600 Hz):
            prev_index = index;
            index = LookForIndex(positive_freq_axis,600); // look for frequencies between 200-600Hz
            double[] LF = new double[index - prev_index];
            System.arraycopy(positive_frequency,prev_index,LF,0,LF.length);
            // look for the HF (600 - 4000 Hz):
            double[] HF = new double[positive_frequency.length - index];
            System.arraycopy(positive_frequency,index,HF,0,HF.length);
            // look for frequencies above 500 Hz:
            index = LookForIndex(positive_freq_axis,500); // look for frequencies above 500Hz
            double[] freq_500 = new double[positive_frequency.length - index];
            System.arraycopy(positive_frequency,index,freq_500,0,freq_500.length);
            // look for frequencies bellow 120 Hz:
            index = LookForIndex(positive_freq_axis,120); // look for frequencies bellow 120Hz
            double[] freq_120 = new double[index+1];
            System.arraycopy(positive_frequency,0,freq_120,0,freq_120.length);
            // look for frequencies between 120 - 400 Hz:
            prev_index = index;
            index = LookForIndex(positive_freq_axis,400); // look for frequencies between 120-400Hz
            double[] freq_400 = new double[index - prev_index];
            System.arraycopy(positive_frequency,prev_index,freq_400,0,freq_400.length);
            // Obtain all the attributes:
            token[0] = sumAbsVector(fragment)/fragment.length;
            token[1] = standardDeviation(fragment);
            token[2] = sumVector(energy)/energy.length;
            token[3] = peaks;
            token[4] = Arrays.stream(positive_frequency).sum();
            token[5] = Arrays.stream(VLF).sum();
            token[6] = Arrays.stream(LF).sum()/(token[4]-token[5]);
            token[7] = Arrays.stream(HF).sum()/(token[4]-token[5]);
            token[8] = Arrays.stream(freq_400).sum()/Arrays.stream(freq_120).sum();
            token[9] = Arrays.stream(freq_500).sum()/token[4];
            token[10] = -1;

            // launch machine learning:
            sleepSample sampleTest = new sleepSample(token);
            Log.d("AudioRecord", "ATTRIBUTES: "+ sampleTest.toString());
            decision = MachineLearningDecision(sampleTest);
        } catch (Exception e) {
            e.printStackTrace();
            Log.d("AudioRecord", "Error: "+ e);
        }
        return decision;
    }

    @RequiresApi(api = Build.VERSION_CODES.N)
    private int MachineLearningDecision(sleepSample sampleTest) {
        /* Method that controls all the machine learning process.
         *  Firstly, it calculates the euclidean distance with all the samples of the database.
         *  Then applied K-nearest neighbor for the decision.
         *
         *   Last Modified: 01/05/2020
         * */
        sleepSample sampleTestNormalized = normalizeSample(sampleTest);
        double[] EuclideanDistances = new double[trainSleepSamples.size()];
        List<sleepSample> trainingDecision = trainSleepSamples;
        // Calculate the euclidean distance of the fragment with the 78 samples of the database:
        for (int i = 0; i < trainSleepSamples.size(); i++){
            EuclideanDistances[i] = obtainEuclideanDistance(sampleTestNormalized, trainSleepSamples.get(i)
    );
        }
        // Lauch k-nearest method to decide with the 5 closest samples:
```

```
        int decision = KnearestAlgorithm (EuclideanDistances, 5, trainingDecision);
        Log.d("AudioRecord", "Decision:" + decision);
        return decision;
    }

    private sleepSample normalizeSample(sleepSample sample) {
        /* Method that normalizes the input fragment using the means and std of the static resources.
         *
         *   Last Modified: 01/05/2020
         * */
        Log.d("AudioRecord", "Means: " + trainSleepSamplesMean.getAbs_mean() + " - std : " +
    trainSleepSamplesStd.getAbs_mean());
        double[] token = new double[11];
        token[0] = (sample.getAbs_mean() - trainSleepSamplesMean.getAbs_mean()) / trainSleepSamplesStd.
    getAbs_mean();
        token[1] = (sample.getStd() - trainSleepSamplesMean.getStd()) / trainSleepSamplesStd.getStd();
        token[2] = (sample.getEnergy_mean() - trainSleepSamplesMean.getEnergy_mean()) /
    trainSleepSamplesStd.getEnergy_mean();
        token[3] = (sample.getPeaks() - trainSleepSamplesMean.getPeaks()) / trainSleepSamplesStd.getPeaks
    ();
        token[4] = (sample.getPSD() - trainSleepSamplesMean.getPSD()) / trainSleepSamplesStd.getPSD();
        token[5] = (sample.getVLF_power() - trainSleepSamplesMean.getVLF_power()) / trainSleepSamplesStd.
    getVLF_power();
        token[6] = (sample.getLF_power_norm() - trainSleepSamplesMean.getLF_power_norm()) /
    trainSleepSamplesStd.getLF_power_norm();
        token[7] = (sample.getHF_power_norm() - trainSleepSamplesMean.getHF_power_norm()) /
    trainSleepSamplesStd.getHF_power_norm();
        token[8] = (sample.getRatio_120_400() - trainSleepSamplesMean.getRatio_120_400()) /
    trainSleepSamplesStd.getRatio_120_400();
        token[9] = (sample.getRatio_500() - trainSleepSamplesMean.getRatio_500()) / trainSleepSamplesStd.
    getRatio_500();
        token[10] = sample.getType();
        sleepSample sampleNorm = new sleepSample(token);

        return sampleNorm;
    }

    @RequiresApi(api = Build.VERSION_CODES.N)
    private int KnearestAlgorithm(double[] euclideanDistances, int k, List<sleepSample> trainingDecision)
    {
        /* Method that calculates the k nearest neighbors to the input fragment. It receives as an input
         *  all the distances with the database. the numbers of neighbor that we are looking for (k) and
         *  classification vector of the samples in the database.
         *
         *   Last Modified: 01/05/2020
         * */
        List<sleepSample> saveList = new ArrayList<>();
        int[] neighbors = new int[k];
        for(int i = 0; i < k; i++){
            // Look for the k minimum distances:
            double minVal = euclideanDistances[0];
            int index = 0;
            for(int j = 1; j < euclideanDistances.length; j++){
                if(euclideanDistances[j] < minVal){
                    minVal = euclideanDistances[j];
                    index = j;
                }
            }
            // Remove the closest neighbor to keep looking for the following neighbors:
            neighbors[i] = trainingDecision.get(index).getType();
            saveList.add(trainingDecision.get(index));
            trainingDecision.remove(index);
            euclideanDistances = removeTheElement(euclideanDistances,index);
        }
        for (int i = 0; i < saveList.size(); i++){
            trainingDecision.add(saveList.get(i));
        }
        return mode(neighbors);
    }

    private double obtainEuclideanDistance(sleepSample sampleTest, sleepSample sampleTrain) {
        /* Method that calculates the euclidean distance.

         *   Last Modified: 01/05/2020
         * */
        double distance;
        double distanceSquare;
        distanceSquare = (sampleTest.getAbs_mean()-sampleTrain.getAbs_mean())*(sampleTest.getAbs_mean()-
    sampleTrain.getAbs_mean());
        distanceSquare =+ (sampleTest.getStd()-sampleTrain.getStd())*(sampleTest.getStd()-sampleTrain.
    getStd());
        distanceSquare =+ (sampleTest.getEnergy_mean()-sampleTrain.getEnergy_mean())*(sampleTest.
    getEnergy_mean()-sampleTrain.getEnergy_mean());
        distanceSquare =+ (sampleTest.getPeaks()-sampleTrain.getPeaks())*(sampleTest.getPeaks()-
    sampleTrain.getPeaks());
        distanceSquare =+ (sampleTest.getPSD()-sampleTrain.getPSD())*(sampleTest.getPSD()-sampleTrain.
    getPSD());
        distanceSquare =+ (sampleTest.getVLF_power()-sampleTrain.getVLF_power())*(sampleTest.getVLF_power
    ()-sampleTrain.getVLF_power());
        distanceSquare =+ (sampleTest.getLF_power_norm()-sampleTrain.getLF_power_norm())*(sampleTest.
    getLF_power_norm()-sampleTrain.getLF_power_norm());
        distanceSquare =+ (sampleTest.getHF_power_norm()-sampleTrain.getHF_power_norm())*(sampleTest.
    getHF_power_norm()-sampleTrain.getHF_power_norm());
        distanceSquare =+ (sampleTest.getRatio_120_400()-sampleTrain.getRatio_120_400())*(sampleTest.
    getRatio_120_400()-sampleTrain.getRatio_120_400());
        distanceSquare =+ (sampleTest.getRatio_500()-sampleTrain.getRatio_500())*(sampleTest.getRatio_500
    ()-sampleTrain.getRatio_500());
        distance = Math.sqrt(distanceSquare);
```

```
        return distance;
    }

    public static double[] removeTheElement(double[] arr,int index){
        /* Method that removes an element from an array.
         *
         *   Last Modified: 01/05/2020
         * */
        // Create another array of size one less
        double[] anotherArray = new double[arr.length - 1];
        // Copy the elements except the index from original array to the other array
        for (int i = 0, k = 0; i < arr.length; i++) {
            if (i == index) {
                continue;
            }
            anotherArray[k++] = arr[i];
        }
        return anotherArray;
    }

    public static int mode(int[] sequence){
        /* Method that calculates the mode.
         *
         *   Last Modified: 01/05/2020
         * */
        int maxValue = 0;
        int maxCount = 0;

        for (int i = 0; i < sequence.length; ++i) {
            int count = 0;
            for (int j = 0; j < sequence.length; ++j){
                if (sequence[j] == sequence[i])
                    ++count;
            }
            if (count > maxCount){
                maxCount = count;
                maxValue = sequence[i];
            }
        }
        return maxValue;
    }

    private int LookForIndex(double[] freq_axis, int threshold) {
        /* Method that looks for the index of the value that is above the threshold.
         *
         *   Last Modified: 01/05/2020
         * */
        for (int j = 0; j < freq_axis.length; j++){
            if(freq_axis[j] >= threshold){
                return j;
            }
        }
        return 0;
    }

    public static double[] eventDetector (double[] y){
        /* Method that calculates the energy of the fragment with a widow size of 60 ms and
         * 75% of overlap.
         *
         *   Last Modified: 01/05/2020
         * */
        int fs = 8000;
        double windowSize = 0.06;
        double n = (y.length/fs)/windowSize;
        int n_overlap = (int) Math.round(4*n - 3);
        int windowsPoints = (int) Math.round(windowSize*fs);
        int windowStep = (int) Math.round(windowsPoints*0.25 - 0.5);
        int low_ind = 0;
        double[] energy = new double[n_overlap];
        for(int i = 0; i < n_overlap; i++ ){
            double[] y_window = new double[windowsPoints];
            System.arraycopy(y,low_ind,y_window,0,y_window.length);
            energy[i] = sumSquareVector(y_window);
            low_ind = low_ind + windowStep;
        }
        return energy;
    }

    public static int countPeaks (double[] energy){
        /* Method that counts the number of peaks that are above a fixed threshold.
         * This threshold is 180% the mean of the input vector.
         *
         *   Last Modified: 01/05/2020
         * */
        double threshold = 1.8*sumVector(energy)/energy.length;
        boolean upward = true;
        boolean downward = false;
        double t_min = -1;
        double t_max = -1;
        double min_duration = 0.3;
        double max_duration = 3.5;
        double t_min_origin = -max_duration;
        int count = 0;

        double[] t = new double[energy.length];
        double time = 0;
        for(int i = 0; i < energy.length; i++){
            t[i] = time;
```

```
            time = time + 0.015015; // step size: 45/energy.length
        }
        for (int i = 0; i < energy.length; i++){
            if(upward){
                if(energy[i] >= threshold){
                    t_min = t[i];
                    upward = false;
                    downward = true;
                }
            } else if (downward){
                if(energy[i] <= threshold){
                    t_max = t[i];
                    upward = true;
                    downward = false;
                }
            }

            if(t_min != -1 && t_max != -1){
                if(t_max - t_min >= min_duration && t_max - t_min_origin >= max_duration){
                    count = count + 1;
                    t_min_origin = t_min;
                }
                t_min = -1;
                t_max = -1;
            }
        }
        return count;
    }

    public static double sumVector (double[] vector){
        /* Method that calculates the sum of a vector.
         *
         *   Last Modified: 01/05/2020
         * */
        double sum = 0;
        for(int i = 0; i < vector.length; i++){
            sum = sum + vector[i];
        }
        return sum;
    }

    private double sumAbsVector(double[] vector) {
        /* Method that calculates the sum of the absolute value of a vector.
         *
         *   Last Modified: 01/05/2020
         * */
        double sum = 0;
        for(int i = 0; i < vector.length; i++){
            sum = sum + Math.abs(vector[i]);
        }
        return sum;
    }

    public static double sumSquareVector (double[] vector){
        /* Method that calculates the sum of square value of a vector.
         *
         *   Last Modified: 01/05/2020
         * */
        double sum = 0;
        for(int i = 0; i < vector.length; i++){
            sum = sum + vector[i]*vector[i];
        }
        return sum;
    }

    private double standardDeviation(double[] fragment) {
        /* Method that calculates std.
         *
         *   Last Modified: 01/05/2020
         * */
        double[] deviation = new double[fragment.length];
        double mean = sumVector(fragment)/fragment.length;
        for (int i = 0; i < fragment.length; i++){
            deviation[i] = (fragment[i] - mean) * (fragment[i] - mean);
        }
        return Math.sqrt(sumVector(deviation)/deviation.length);
    }
}
```

Listing 5: SignalProcessing.java

```
package com.portable.sleepdiseasetracer;

import android.util.Log;
import java.util.ArrayList;

/* Class DiscreteFourierTransform.
 *   It calculates the Fast Fourier Transform (although the name can lead to confusion) of an input vector.
 *   It is composed of two methods, 'calculateDFT' that prepares the input vector to be transformed, and
 *   'fft' which is the method that calculates the Fourier transform in an optimal way.
 *
 *   Last Modified: 07/05/2020
 * */
public class DiscreteFourierTransform {

    public static double[] calculateDFT (double[] signal){
        /*  Method used to prepare the input vector (signal) to be transformed.
```

```
         *  The required vector size to apply the fft has to be to the power of two, thus this method is in charge
         *  to add zeros, if needed, to the input vector in order to obtain this size.
         *  After running the fft method receives the return and shifts the vector to center the spectrum in the origin.
         *
         *   Last Modified: 07/05/2020
         * */
        Log.wtf("AudioRecord", "In FFT");
        ArrayList<Complex> stringArrayList = new ArrayList<Complex>();
        int numbits = 0;
        int addzeros = 0;
        // The length of the vector needs to be to the power of two. Loop to look for the best power depending on the length of the signal:
        for(int i = 15; i < 22; i++){
            numbits = (int) Math.pow(2, i); //PAD WITH THIS MANY ZEROS 2^i
            if(numbits > signal.length){
                i = 23; // To get out of the loop.
            }
        }
        // We have to add zeros to the signal to have a length to the power of two:
        addzeros = numbits - signal.length;
        for (int k = 0; k < signal.length; k++) {
            stringArrayList.add(new Complex(signal[k],0.0));
        }
        for (int b = 0; b < addzeros; b++) {
            stringArrayList.add(new Complex(0.0,0.0));
        }
        Complex[] input = new Complex[numbits];
        for (int i = 0; i < numbits; i++) {
            input[i] = stringArrayList.get(i);
        }
        // Launch fft method:
        Complex[] freq = fft(input);
        double[] freq_module = new double[freq.length];
        // Calculate the module of the spectrum:
        for (int i = 0; i < freq.length; i++){
            freq_module[i] = (freq[i].abs()*freq[i].abs())/input.length;
        }
        // Shift the vector:
        double[] freq_shifted = new double[freq_module.length];
        System.arraycopy(freq_module, freq_shifted.length/2,freq_shifted,0,freq_shifted.length/2);
        System.arraycopy(freq_module, 0,freq_shifted,freq_shifted.length/2,freq_shifted.length/2);
        return freq_shifted;
    }

    public static Complex[] fft(Complex[] x) {
        /*  Method that calculates the Fast Fourier Tranform of an input vector by dividing the vector into
         *  even and odd index. This division help to optimize the fft and, thus, obtain a result faster.
         *
         *   Last Modified: 07/05/2020
         * */
        int n = x.length;
        // Base case (to finish the loop of the fft calling):
        if (n == 1) return new Complex[] { x[0] };
        // Control that the length of the signal is a power of two:
        if (n % 2 != 0) {
            throw new IllegalArgumentException("n is not a power of 2");
        }

        // Compute FFT of even terms:
        Complex[] even = new Complex[n/2];
        for (int k = 0; k < n/2; k++) {
            even[k] = x[2*k];
        }
        Complex[] evenFFT = fft(even);

        // Compute FFT of odd terms:
        Complex[] odd  = even;  // Reuse the array (to avoid n log n space)
        for (int k = 0; k < n/2; k++) {
            odd[k] = x[2*k + 1];
        }
        Complex[] oddFFT = fft(odd);

        // Combine the even and odd terms:
        Complex[] y = new Complex[n];
        for (int k = 0; k < n/2; k++) {
            // Transformation:
            double kth = -2 * k * Math.PI / n;
            Complex wk = new Complex(Math.cos(kth), Math.sin(kth));
            y[k] = evenFFT[k].plus (wk.times(oddFFT[k]));
            y[k + n/2] = evenFFT[k].minus(wk.times(oddFFT[k]));
        }
        return y;
    }
}
```

Listing 6: DiscreteFourierTransform.java

```
package com.portable.sleepdiseasetracer;

import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

```

```java
/* Java class 'SoundDataUtils'.
 *   It is a class that transforms the values saved in a .pcm file (bits) into double.
 *   Each double value has to be defined by 16 bit in the file.
 *
 *   Last Modified: 07/05/2020
 * */
public class SoundDataUtils {
    public static double[] load16BitPCMRawDataFileAsDoubleArray(File file) {
        /*  Method that reads the input file and launches the transformation method.
         *
         *   Last Modified: 07/05/2020
         * */
        InputStream in;
        if (file.isFile()) {
            long size = file.length();
            try {
                in = new FileInputStream(file);
                return readStreamAsDoubleArray(in, size);
            } catch (Exception e) {
            }
        }
        return null;
    }

    public static double[] readStreamAsDoubleArray(InputStream in, long size) throws IOException {
        /*  Method that transforms an array of bits into an array of double, with the relation: 16 bits =
    1 double.
         *
         *   Last Modified: 07/05/2020
         * */
        int bufferSize = (int) (size / 2);
        // Two bytes define one double, that's way it is the half of the file size:
        double[] result = new double[bufferSize];
        DataInputStream is = new DataInputStream(in);
        for (int i = 0; i < bufferSize; i++) {
            // readShort() returns 16 bits:
            result[i] = is.readShort() / 32768.0;
        }
        return result;
    }
}
```

Listing 7: SoundDataUtils.java

```java
package DataBase;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper; // Class to communicate the database with the android app
    .
import androidx.annotation.Nullable;

/* Class SQLite_helper.
 *   It is the interface to communicate the application (Login and register activity)
 *   with the Android Studio's database (SQLite). It has a method to define the structure of the database,
 *   one to save one registration in the database and another to validate that the input credentials (
    login)
 *   are in the database.
 *
 *   Last Modified: 22/04/2020
 * */
public class SQLite_helper extends SQLiteOpenHelper {

    // Constructor of the class:
    public SQLite_helper(@Nullable Context context, @Nullable String name, @Nullable SQLiteDatabase.
    CursorFactory factory, int version) {
        super(context, name, factory, version);
    }

    // Method to create the database:
    @Override
    public void onCreate(SQLiteDatabase db) {
        // Create a simple database with the structure: User - Password:
        String query = "create table users(_ID integer primary key autoincrement, Name text, Password text
    );";
        db.execSQL(query);
    }

    // Method to apply changes to the database
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

    }

    // Method to open the database
    public void openDatabase(){
        this.getWritableDatabase();
    }

    // Method to close the database
    public void closeDatabase(){
        this.close();
    }

    // Method to insert registers in the user's table
```

```
    public void insertRegister(String name, String password){
        ContentValues values = new ContentValues();
        values.put("Name", name);
        values.put("Password", password);
        this.getWritableDatabase().insert("users",null,values);
    }

    // Method to validate if the user exists
    public Cursor consultUser(String user, String password) throws SQLException {
        Cursor mcursor = null;
        // Structure of the query: Database - Attributes - Conditions (WHERE):
        mcursor = this.getReadableDatabase().query("users",new String[]{"_ID","Name","Password"},
                "Name like '"+ user + "' and Password like '" + password + "'",
                null,null,null,null);
        return mcursor;
    }
}
```

Listing 8: SQLite_helper.java

```
package com.portable.sleepdiseasetracer;

/* Java class 'sleepSample'.
 *   It is a class that defines the structure of the attributes of our database:
 *      1) It has a constructor that creates a variable of the type 'sleepComplex' with the input of the attributes.
 *      2) It contains the methods 'get' and 'set' for each of the variables of the structure.
 *
 *   Last Modified: 27/04/2020
 * */
class sleepSample {
    private double abs_mean;
    private double std;
    private double energy_mean;
    private double peaks;
    private double PSD;
    private double VLF_power;
    private double LF_power_norm;
    private double HF_power_norm;
    private double ratio_120_400;
    private double ratio_500;
    private int type;

    public sleepSample(String[] token) {
        /*  Constructor of the class 'sleepSample': creates a new object with the given attributes with String[] input.
         *
         *   Last Modified: 27/04/2020
         * */
        this.abs_mean = Double.parseDouble(token[0]);
        this.std = Double.parseDouble(token[1]);
        this.energy_mean = Double.parseDouble(token[2]);
        this.peaks = Double.parseDouble(token[3]);
        this.PSD = Double.parseDouble(token[4]);
        this.VLF_power = Double.parseDouble(token[5]);
        this.LF_power_norm = Double.parseDouble(token[6]);
        this.HF_power_norm = Double.parseDouble(token[7]);
        this.ratio_120_400 = Double.parseDouble(token[8]);
        this.ratio_500 = Double.parseDouble(token[9]);
        this.type = Integer.parseInt(token[10]);
    }

    public sleepSample(double[] token) {
        /*  Constructor of the class 'sleepSample': creates a new object with the given attributes with double[] input.
         *
         *   Last Modified: 27/04/2020
         * */
        this.abs_mean = token[0];
        this.std = token[1];
        this.energy_mean = token[2];
        this.peaks = token[3];
        this.PSD = token[4];
        this.VLF_power = token[5];
        this.LF_power_norm = token[6];
        this.HF_power_norm = token[7];
        this.ratio_120_400 = token[8];
        this.ratio_500 = token[9];
        this.type = (int) token[10];
    }

    public double getAbs_mean() {
        return abs_mean;
    }

    public void setAbs_mean(double abs_mean) {
        this.abs_mean = abs_mean;
    }

    public double getStd() {
        return std;
    }

    public void setStd(double std) {
        this.std = std;
    }

    public double getEnergy_mean() {
```

```
        return energy_mean;
    }

    public void setEnergy_mean(double energy_mean) {
        this.energy_mean = energy_mean;
    }

    public double getPeaks() {
        return peaks;
    }

    public void setPeaks(double peaks) {
        this.peaks = peaks;
    }

    public double getPSD() {
        return PSD;
    }

    public void setPSD(double PSD) {
        this.PSD = PSD;
    }

    public double getVLF_power() {
        return VLF_power;
    }

    public void setVLF_power(double VLF_power) {
        this.VLF_power = VLF_power;
    }

    public double getLF_power_norm() {
        return LF_power_norm;
    }

    public void setLF_power_norm(double LF_power_norm) {
        this.LF_power_norm = LF_power_norm;
    }

    public double getHF_power_norm() {
        return HF_power_norm;
    }

    public void setHF_power_norm(double HF_power_norm) {
        this.HF_power_norm = HF_power_norm;
    }

    public double getRatio_120_400() {
        return ratio_120_400;
    }

    public void setRatio_120_400(double ratio_120_400) {
        this.ratio_120_400 = ratio_120_400;
    }

    public double getRatio_500() {
        return ratio_500;
    }

    public void setRatio_500(double ratio_500) {
        this.ratio_500 = ratio_500;
    }

    public int getType() {
        return type;
    }

    public void setType(int type) {
        this.type = type;
    }

    @Override
    public String toString() {
        /*  Method that return a string representation of the invoking sleepSample object.
         *
         *   Last Modified: 27/04/2020
         * */
        return "sleepSample{" +
                "abs_mean=" + abs_mean +
                ", std=" + std +
                ", energy_mean=" + energy_mean +
                ", peaks=" + peaks +
                ", PSD=" + PSD +
                ", VLF_power=" + VLF_power +
                ", LF_power_norm=" + LF_power_norm +
                ", HF_power_norm=" + HF_power_norm +
                ", ratio_120_400=" + ratio_120_400 +
                ", ratio_500=" + ratio_500 +
                ", type=" + type +
                '}';
    }
}
```

Listing 9: sleepSample.java

```
package com.portable.sleepdiseasetracer;

/* Java class 'Complex'.
```

```
 *   It is a class that defines the structure of a complex number (real and imaginary parts):
 *      1) It has a constructor that creates a variable of the type 'Complex' with the input real and imaginary part.
 *      2) It contains the methods 'get' and 'set' for each of the variables of the structure.
 *      3) It also contains some mathematical operations that can be done with complex numbers.
 *
 *   Last Modified: 07/05/2020
 * */
public class Complex {
    private double re;   // the real part
    private double im;   // the imaginary part

    public Complex(double real, double imag) {
        /*  Constructor of the class 'Complex': creates a new object with the given real and imaginary parts
         *
         *   Last Modified: 07/05/2020
         * */
        re = real;
        im = imag;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }


    public String toString() {
        /*  Method that return a string representation of the invoking Complex object.
         *
         *   Last Modified: 07/05/2020
         * */
        if (im == 0) return re + "";
        if (re == 0) return im + "i";
        if (im < 0) return re + " - " + (-im) + "i";
        return re + " + " + im + "i";
    }

    public double abs() {
        /*  Method that returns the abs/modulus/magnitude of a complex number.
         *
         *   Last Modified: 07/05/2020
         * */
        return Math.hypot(re, im);
    }

    public double phase() {
        /*  Method that returns the angle/phase/argument, normalized to be between -pi and pi.
         *
         *   Last Modified: 07/05/2020
         * */
        return Math.atan2(im, re);
    }

    public Complex plus(Complex b) {
        /*  Method that returns a new Complex object whose value is (this + b).
         *
         *   Last Modified: 07/05/2020
         * */
        Complex a = this;  // invoking object
        double real = a.re + b.re;
        double imag = a.im + b.im;
        return new Complex(real, imag);
    }

    public Complex minus(Complex b) {
        /*  Method that returns a new Complex object whose value is (this - b).
         *
         *   Last Modified: 07/05/2020
         * */
        Complex a = this;
        double real = a.re - b.re;
        double imag = a.im - b.im;
        return new Complex(real, imag);
    }

    public Complex times(Complex b) {
        /*  Method that returns a new Complex object whose value is (this * b).
         *
         *   Last Modified: 07/05/2020
         * */
        Complex a = this;
        double real = a.re * b.re - a.im * b.im;
        double imag = a.re * b.im + a.im * b.re;
        return new Complex(real, imag);
    }
```

```
100 }
```

Listing 10: Complex.java

3 Matlab Scripts

```
close all;
clear all; clc
%% Description:
% Code created by Alvaro Carrera Cardeli, Federico Medea and Walid Khaled Hussein
% Main code to obtain all the attributes of a fragment of 45 seconds.
% 4 of the attributes are from the time domain and 6 from the frequency
% domain.
%% Obtain the recordings (input)
cdir = fileparts(mfilename('fullpath'));
% Nomral breathing input.
file = fullfile(cdir,'recordings/10 normal.m4a');
[y,Fs] = audioread(file);
%% INPUTS:
f_down = 8000; % Frequency sampling that will be used
fragment_origin = 40; % second where it will be set the origin of the 45 sec. fragment
fragment_size = 45; % 45 seg
windowSize = 0.06; % 60 ms
max_VLF = 200; % 200 Hz
max_LF = 600; % 600 Hz

%% Filtering (Low-pass filter + Downsample)
t =(0:length(y)-1)/Fs;
% Take a fragment of the sample:
ind_inf = find(t <= fragment_origin);
ind_sup = find(t > (fragment_origin + fragment_size),1);
y_frag = y(ind_inf(end):ind_sup-1);
t_frag = t(ind_inf(end):ind_sup-1);
% Filtering:
[y_down,t_down] = resample(y_frag,t_frag,f_down); % Lowpass filter + downsampling
[energy,t_energy] = EventDetector(y_down,f_down,windowSize); % method to obtain the energy of a 60 ms
    window
peaks = countPeaks(energy,t_energy); % method to count the peaks that are above a fixed threshold under
    certain conditions
% Frequency domain:
[freq_axis, freq] = frequencyDomain(f_down,y_down); % method that returns fft
% Positive frequency values:
f_index = find(freq_axis <= 0);
positive_frequency_axis = freq_axis(f_index(end):end);
positive_freq = freq(f_index(end):end);
% VLF - Record from 0 to max_VLF:
VLF_f_index = find(positive_frequency_axis <= max_VLF);
VLF_frequency_axis = positive_frequency_axis(1:length(VLF_f_index));
VLF_freq = positive_freq(1:length(VLF_f_index));
% LF - Record from max_VLF to max_LF:
LF_f_index = find(positive_frequency_axis > max_VLF & positive_frequency_axis <= max_LF);
LF_frequency_axis = positive_frequency_axis(LF_f_index(1):LF_f_index(length(LF_f_index)));
LF_freq = positive_freq(LF_f_index(1):LF_f_index(length(LF_f_index)));
% HF - Record from max_LF to 4000:
HF_f_index = find(positive_frequency_axis <= max_LF);
HF_frequency_axis = positive_frequency_axis(HF_f_index(end):end);
HF_freq = positive_freq(HF_f_index(end):end);
% Power above 500 Hz:
f_500Hz_index = find(positive_frequency_axis <= 500);
f_500Hz_axis = positive_frequency_axis(f_500Hz_index(end):end);
f_500Hz = positive_freq(f_500Hz_index(end):end);
power_above_500 = sum(f_500Hz);
% Power between 0 - 120 Hz:
f_120Hz_index = find(positive_frequency_axis <= 120);
f_120Hz_axis = positive_frequency_axis(1:length(f_120Hz_index));
f_120Hz = positive_freq(1:length(f_120Hz_index));
p_0_120_Hz = sum(f_120Hz);
% Power between 120 - 400 Hz:
f_400Hz_index = find(positive_frequency_axis > 120 & positive_frequency_axis <= 400);
f_400Hz_axis = positive_frequency_axis(f_400Hz_index(1):f_400Hz_index(length(f_400Hz_index)));
f_400Hz = positive_freq(f_400Hz_index(1):f_400Hz_index(length(f_400Hz_index)));
p_120_400_Hz = sum(f_400Hz);

%% Results:
% (1) Mean of the absolut window:
mean_abs_1 = sum(abs(y_down))/length(y_down);
% (2) Standard deviation of the original input:
std_2 = std(y_down);
% (3) Energy mean:
energy_mean_3 = sum(energy)/length(energy);
% (4) Number of peaks:
num_peak_4 = peaks;
% (5) Power Spectrum Density:
PSD_5 = sum(positive_freq);
% (6) Very Low Frequency power Normalization:
VLF_power_6 = sum(VLF_freq);
% (7) Low Frequency power Normalization:
LF_power_norm_7 = sum(LF_freq)/(PSD_5 - VLF_power_6);
% (8) High Frequency power:
HF_power_norm_8 = sum(HF_freq)/(PSD_5 - VLF_power_6);
% (9) Ratio Power(120-400Hz)/Power(0-120Hz):
ratio_9 = p_120_400_Hz/p_0_120_Hz;
% (10) Power above 500 Hz:
ratio_500Hz_10 = power_above_500/PSD_5;

% Print the sample attributes:
sample = mean_abs_1+","+std_2+","+energy_mean_3+","+num_peak_4+","+PSD_5+","+VLF_power_6+","+
    LF_power_norm_7+","+HF_power_norm_8+","+ratio_9+","+ratio_500Hz_10
%% PLOTS:
% Plot of the spectogram:
%figure;
```

```
%spectrogram(y_down,0.5*f_down,0.1*f_down,0:f_down/2-1,f_down,'yaxis')

% Time domain:
figure;
subplot(3,2,1);
plot(t_down,y_down); % Original
subplot(3,2,2);
plot(t_down,abs(y_down)); % Absolut value
subplot(3,2,3);
plot(t_down,abs(y_down)/max(abs(y_down))); % Normalized
subplot(3,2,4);
plot(t_energy,energy); % energy
hold on;
yline(2*mean(energy));
% Frequency domain:
subplot(3,2,5);
plot(positive_frequency_axis,positive_freq)
xlim([0,400]);
subplot(3,2,6);
plot(positive_frequency_axis,positive_freq/max(positive_freq)) % Normalized
xlim([0,400]);
```

Listing 11: data_mining.m

```
% Code created by Alvaro Carrera Cardeli, Federico Medea and Walid Khaled Hussein
% Method 'EventDetector':
%   Get as an input a vector in the time domain and calculates the energy
%   in this vector by using a window of 60 ms size and an overlap of 75%
%   (45 ms)
function [energy_return,t_return] = EventDetector(y,fs,windowSize)
    %t = 0:1/fs:length(y)*(1/fs)- 1/fs; % time vector of the input signal
    n = (length(y)*(1/fs))/windowSize; % number of fragments with the windowsSize of the vector y
    n_overlap = round(4*n-3); % number of windows if there is an overlap of 75%
    % BLOCK 1: energy vector calculation.
    windowsPoints = round(windowSize*fs); % number of points of the window
    windowStep = round(windowsPoints*0.25 - 0.5); % 75% of overlap
    low_ind = 1;
    high_ind = windowsPoints + low_ind - 1;
    energy = nan(n_overlap,1);
    % Loop simulating the displacement of the window through the vector:
    for j = 1:(n_overlap)
        y_window = y(low_ind:high_ind);
        energy(j) = sum(y_window.^2);
        low_ind = low_ind + windowStep;
        high_ind = windowsPoints + low_ind - 1;
    end

    energy_return=energy;
    t_return = 0:45/length(energy):45-(1/length(energy));
end
```

Listing 12: EventDetector.m

```
function num_peak = countPeaks(energy,t)
    threshold = 1.8*mean(energy); % 180% of the energy mean
    upward = 1;
    downward = 0;
    t_min = -1;
    t_max = -1;
    min_duration = 0.3; % 300 ms
    max_duration = 3.5; % 3.5 sec
    t_min_origin = -max_duration;
    count = 0;

    for j = 1 : length(energy)
        if (upward)
            if(energy(j) >= threshold)
                t_min = t(j);
                upward = 0;
                downward = 1;
            end
        elseif (downward)
            if(energy(j) <= threshold)
                t_max = t(j);
                upward = 1;
                downward = 0;
            end
        end

        if (t_min ~= -1 && t_max ~= -1)
            if (t_max - t_min >= min_duration && t_max - t_min_origin >= max_duration)
                count = count + 1;
                t_min_origin = t_min;
            end
            t_min = -1;
            t_max = -1;
        end
    end

    num_peak = count;
end
```

Listing 13: countPeaks.m

```
% Code created by Alvaro Carrera Cardeli, Federico Medea and Walid Khaled Hussein
% Method 'frequencyDomain'
%   Calculates the Fats Fourier Transform (fft) of an input vector and
```

```
%   returns the spectrum power centreing the frequency vector in the
%   origin.
function [freq_axis, freq_values] = frequencyDomain(freq_samp,y)
    y_clean=y(~isnan(y));
    freq_values=abs((fftshift(fft(y_clean)).^2)/length(y_clean)); % fft
    freq_axis=-freq_samp/2:freq_samp/(length(freq_values)):(freq_samp/2-freq_samp/(length(freq_values)));
    % shift to centre the spectrum in the origin
end
```

Listing 14: frequencyDomain.m

```
% Code created by Alvaro Carrera Cardeli, Federico Medea and Walid Khaled Hussein
% Main code that controls the machine learning training.
% The model that it is used is K-nearest neighbor and it is been used
% leaveout cross validation to look for the optimum value of K (neighbors). The range
% of K goes from 2 to 10 due to the size of the database.
close all; clear all; clc;

%% Data Loading
[X,y,attributeNames,N,M] = loadData(); % load the data from the database.

%% Prepare Models:
%K-NEAREST NEIGHBOR:
K_near = 2:10; % Number of neighbors
Distance = 'euclidean'; % Distance measure

%% Cross-validation
CV = cvpartition(N, 'Leaveout'); % leaveout
K = CV.NumTestSets;

% Initialize variables for K-NEAREST NEIGHBOR:
T = length(K_near);
prediction = nan(K,T);
y_true = nan(K,1); % Real clasiffication

% Loop to look for the optimal K value:
for k = 1 : K
    fprintf('Crossvalidation fold %d/%d\n', k, CV.NumTestSets);

    % Extract training and test set
    X_train = X(CV.training(k), :);
    y_train = y(CV.training(k));
    X_test = X(CV.test(k), :);
    y_test = y(CV.test(k));

    y_true(k,1) = y_test(1);

    % Standardization:
    mu = mean(X_train);
    sigma = std(X_train);
    X_train_norm = (X_train - mu) ./ sigma;
    X_test_norm = (X_test - mu) ./ sigma;

    % --------------- MODEL 1: K-NEAREST NEIGHBOR.--------------------
    for j = 1 : T
        knn=fitcknn(X_train_norm, y_train, 'NumNeighbors', K_near(j), 'Distance', Distance);
        prediction(k,j) = predict(knn, X_test_norm);
    end
end

accuracy = nan(T,1);
% Calculate the accuaracy of all the models (all the K):
for s = 1 : T
    num_succeed = sum(prediction(:,s) == y_true);
    accuracy(s) = num_succeed/length(y_true);
end
[accuaricy_max, ind] = max(accuracy);
% Best model:
Knn_opt = K_near(ind)
```

Listing 15: MachineLearningTraining.m

```
% Code created by Alvaro Carrera Cardeli, Federico Medea and Walid Khaled Hussein
% Method 'loadData'
%   Loads and prepares the data from the database 'database_new.csv' to run
%   the machine learning with it.
function [X,y,attributeNames,N,M] = loadData()
    % Get the data from a CSV file
    cdir = fileparts(mfilename('fullpath'));
    file_path = fullfile(cdir,'database_new.csv');
    % Load the data into matlab using readtable.
    SAHDS_table = readtable(file_path);

    % Extract the rows and columns (attributes) corresponding to the dimensions that are not nominal.
    X = table2array(SAHDS_table(:, [2:end-1]));
    % Extract attribute names.
    attributeNames = SAHDS_table.Properties.VariableNames([2:end-1])';

    % Extract the nominal attribute (type) from the 12th column.
    class = table2cell(SAHDS_table(:,12));
    % Get unique values of the attribute (type) -> Apnea, Normal, Snor.
    class_values = unique(class);

    % Replace by numbers [1 2 3] the class labels (type) that match the class names (class_values).
    [~,class_transformation] = ismember(class, class_values);
    % Since we want to assign numerical values to the classes starting from
    % a zero and not a one.(Apnea = 0, Nomral = 1, Snor = 2)
    class_transformation = class_transformation-1;
```

```
    % Lastly, we determine the number of attributes M, the number of observations N.
    y = class_transformation;
    [N, M] = size(X);
end
```

Listing 16: loadData.m

4 Database

row_names	abs_mean	std	energy_mean	peaks	PSD	VLF_power
1	0.0011702	0.001473	0.0010418	0	0.3908	0.38078
2	0.0015454	0.0020569	0.0020317	4	0.76175	0.75587
3	0.0022312	0.0044009	0.0093037	3	3.4864	2.6978
4	0.0040442	0.010436	0.052239	3	19.6039	17.9208
5	0.0014246	0.0025707	0.0031731	1	1.1898	1.1115
6	0.0013838	0.0025741	0.0031797	2	1.1929	1.0475
7	0.0015839	0.002845	0.0038816	2	1.4572	1.361
8	0.0021269	0.0034733	0.0057953	2	2.1717	1.3819
9	0.0040031	0.010314	0.051114	4	19.1491	12.4946
10	0.0050016	0.012374	0.073573	4	27.5628	16.2194
11	0.0024933	0.0069811	0.023384	3	8.7728	6.9503
12	0.0070411	0.017099	0.14031	3	52.6273	48.9206
13	0.00083659	0.0010883	0.00056918	5	0.21344	0.15579
14	0.0011438	0.0025778	0.0031925	1	1.1963	1.1513
15	0.0029717	0.0058926	0.016681	6	6.2503	2.4191
16	0.002969	0.0055191	0.014635	6	5.4832	2.4026
17	0.0011688	0.0018636	0.0016683	1	0.62528	0.3155
18	0.0013905	0.0023893	0.0027424	2	1.0277	0.31307
19	0.0026071	0.0050194	0.012102	5	4.5351	4.4968
20	0.0023555	0.005033	0.01217	5	4.5596	4.5346
21	0.080764	0.13047	8.1705	7	3063.869	2620.6049
22	0.09449	0.14271	9.7855	5	3666.0703	3054.8269
23	0.0028694	0.012022	0.069447	2	26.0172	24.836
24	0.002685	0.00744	0.026596	3	9.9637	9.7927
25	0.0077532	0.020225	0.19654	9	73.6289	72.4736
26	0.0048284	0.013574	0.08849	5	33.1638	32.4995
27	0.0048471	0.01362	0.088892	6	33.3908	16.5016
28	0.0062199	0.015888	0.12094	10	45.4383	21.8931
29	0.0024473	0.0066466	0.021226	3	7.9521	7.8242
30	0.0051458	0.016609	0.13255	2	49.6557	49.0917
31	0.0017845	0.0026331	0.0033292	3	1.2482	1.2206
32	0.0015991	0.002238	0.0024044	3	0.90178	0.87547
33	0.0079822	0.024294	0.28329	7	106.2364	64.8469
34	0.0098846	0.02794	0.37509	11	140.5206	76.095
35	0.0056348	0.020431	0.20056	3	75.1345	71.6237
36	0.0032531	0.013488	0.087417	1	32.7487	31.3941
37	0.024733	0.061799	1.8349	9	687.4432	587.3261
38	0.01726	0.044816	0.96488	8	361.5194	339.6908
39	0.0051683	0.011398	0.062416	9	23.3827	20.9503

row_names	LF_power_norm	HF_power_norm	ratio_120_400	ratio_500	type
1	0.80948	0.19052	0.047325	0.0054886	"normal"
2	0.68868	0.31133	0.023665	0.0025409	"normal"
3	0.24172	0.75828	0.16538	0.17742	"snor"
4	0.26456	0.73545	0.053379	0.064667	"snor"
5	0.35628	0.64373	0.027146	0.046165	"apnea"
6	0.36063	0.6394	0.059412	0.082334	"apnea"
7	0.83125	0.16884	0.11989	0.021905	"normal"
8	0.82571	0.17429	0.27851	0.1688	"normal"
9	0.28329	0.71672	0.25363	0.26981	"snor"
10	0.30698	0.69302	0.33956	0.30436	"snor"
11	0.25567	0.74433	0.18662	0.1597	"apnea"
12	0.37866	0.62135	0.13568	0.045426	"apnea"
13	0.19744	0.80259	0.11323	0.2318	"normal"
14	0.43564	0.56436	0.068418	0.022856	"normal"
15	0.098828	0.90117	0.24495	0.57237	"snor"
16	0.11915	0.88086	0.26725	0.51657	"snor"
17	0.13511	0.86489	0.13515	0.43675	"apnea"
18	0.097644	0.90236	0.2729	0.63679	"apnea"
19	0.81213	0.18787	0.031937	0.0017396	"normal"
20	0.81631	0.18369	0.053633	0.0010706	"normal"
21	0.7342	0.2658	0.5656	0.044613	"snor"
22	0.73957	0.26044	0.57826	0.050801	"snor"
23	0.60621	0.39379	0.12591	0.018357	"apnea"
24	0.42237	0.57763	0.061311	0.010258	"apnea"
25	0.98744	0.012558	0.14318	0.00024733	"normal"
26	0.95864	0.041364	0.16931	0.00091879	"normal"
27	0.28938	0.71062	0.36877	0.38672	"snor"
28	0.27492	0.72508	0.32835	0.4019	"snor"
29	0.64224	0.35776	0.05155	0.0060851	"apnea"
30	0.86532	0.13468	0.094268	0.0016588	"apnea"
31	0.77129	0.22872	0.037886	0.0054787	"normal"
32	0.80345	0.19655	0.053414	0.006256	"normal"
33	0.22415	0.77585	0.21175	0.31812	"snor"
34	0.21481	0.78521	0.25764	0.3803	"snor"
35	0.62638	0.37362	0.23743	0.017785	"apnea"
36	0.37449	0.62551	0.10997	0.026494	"apnea"
37	0.99224	0.0077708	0.82513	0.0031262	"normal"
38	0.99576	0.0042448	0.48804	0.00045314	"normal"
39	0.28026	0.71979	0.06475	0.080663	"snor"

row_names	abs_mean	std	energy_mean	peaks	PSD	VLF_power
40	0.005658	0.011439	0.062856	10	23.5515	17.515
41	0.0036397	0.013085	0.08227	3	30.8207	27.6687
42	0.0062861	0.019813	0.18858	2	70.6587	60.3028
43	0.0024149	0.0031654	0.0048115	0	1.8038	1.5268
44	0.0027857	0.0036827	0.0065126	0	2.4414	2.1191
45	0.0058609	0.012773	0.078376	5	29.3679	20.4235
46	0.0062178	0.011411	0.062522	5	23.4379	15.6709
47	0.0030369	0.011975	0.068904	1	25.8139	20.9998
48	0.0027211	0.0092799	0.041376	1	15.5012	12.7469
49	0.0034063	0.0043598	0.0091273	0	3.4216	2.6169
50	0.0033177	0.0041628	0.00832	0	3.1194	2.3444
51	0.0076734	0.016651	0.13316	6	49.9086	33.1217
52	0.0061283	0.012922	0.080177	4	30.0558	20.3124
53	0.0027025	0.0077325	0.028724	1	10.7627	7.697
54	0.0026243	0.0084565	0.034358	1	12.8722	9.933
55	0.003242	0.0040819	0.0079947	0	2.9993	2.3
56	0.0031257	0.0039084	0.0073305	0	2.7498	2.1206
57	0.0086343	0.021589	0.22381	4	83.8947	52.8977
58	0.0082028	0.017931	0.15448	6	57.8767	31.8899
59	0.0027502	0.010741	0.055427	0	20.7655	16.115
60	0.0031079	0.0091768	0.04046	2	15.1586	8.4329
61	0.0021411	0.0027192	0.0035507	2	1.3312	0.69441
62	0.0020634	0.0025977	0.0032393	0	1.2149	0.64527
63	0.0057296	0.016173	0.12567	3	47.0816	30.3325
64	0.0071102	0.016056	0.12384	7	46.4041	32.1712
65	0.0026715	0.0056078	0.015107	1	5.6605	3.7632
66	0.0039652	0.011013	0.058278	3	21.8335	15.744
67	0.0024117	0.0060318	0.017481	3	6.5492	6.4244
68	0.0026236	0.010793	0.055957	0	20.9705	17.9944
69	0.031046	0.063451	1.9324	9	724.6894	683.809
70	0.033544	0.071301	2.4373	8	915.0908	838.6563
71	0.0047168	0.023586	0.26729	1	100.1323	95.0423
72	0.0029302	0.017534	0.14772	1	55.3408	51.8204
73	0.0013802	0.0034375	0.0056774	1	2.1271	2.0334
74	0.0022331	0.0066141	0.021019	4	7.8744	7.3995
75	0.034929	0.075957	2.77	6	1038.5168	950.37
76	0.032449	0.069096	2.2939	5	859.3703	792.5324
77	0.0048797	0.018963	0.17278	2	64.7263	58.631
78	0.0029916	0.011016	0.05831	1	21.8447	17.8428

row_names	LF_power_norm	HF_power_norm	ratio_120_400	ratio_500	type
40	0.25432	0.74571	0.097678	0.20583	"snor"
41	0.41974	0.58027	0.24489	0.060751	"apnea"
42	0.45866	0.54134	0.36646	0.081323	"apnea"
43	0.97683	0.02317	0.30068	0.0049414	"normal"
44	0.95375	0.046249	0.24051	0.008019	"normal"
45	0.30309	0.69692	0.24352	0.2211	"snor"
46	0.33569	0.66431	0.34987	0.23258	"snor"
47	0.50465	0.49535	0.35146	0.10485	"apnea"
48	0.52582	0.47419	0.22531	0.095145	"apnea"
49	0.94965	0.050349	0.39239	0.014793	"normal"
50	0.95139	0.048614	0.41117	0.015086	"normal"
51	0.25349	0.74651	0.28561	0.26157	"snor"
52	0.27181	0.72819	0.28294	0.24549	"snor"
53	0.38591	0.6141	0.18048	0.19123	"apnea"
54	0.38182	0.61819	0.17824	0.15319	"apnea"
55	0.96311	0.036889	0.39696	0.010581	"normal"
56	0.97497	0.025034	0.38182	0.0076377	"normal"
57	0.15732	0.84268	0.23434	0.32027	"snor"
58	0.1418	0.8582	0.22809	0.39582	"snor"
59	0.54834	0.45167	0.37811	0.1075	"apnea"
60	0.47641	0.52359	0.86615	0.25757	"apnea"
61	0.93842	0.061577	1.3687	0.034028	"normal"
62	0.98908	0.010916	1.3699	0.0072058	"normal"
63	0.25625	0.74376	0.46725	0.27677	"snor"
64	0.25031	0.7497	0.22808	0.24172	"snor"
65	0.51624	0.48377	0.32687	0.17576	"apnea"
66	0.52759	0.47242	0.31423	0.14751	"apnea"
67	0.62354	0.37647	0.063797	0.0081431	"normal"
68	0.83871	0.16129	0.30952	0.024626	"normal"
69	0.72758	0.27242	0.38429	0.015879	"snor"
70	0.83258	0.16743	0.52903	0.014958	"snor"
71	0.44168	0.55832	0.19623	0.028668	"apnea"
72	0.16863	0.83138	0.054512	0.053462	"apnea"
73	0.13817	0.86183	0.028747	0.038488	"normal"
74	0.39936	0.60064	0.17375	0.036759	"normal"
75	0.51621	0.48379	0.42632	0.041352	"snor"
76	0.4243	0.5757	0.39347	0.044987	"snor"
77	0.24663	0.75338	0.21993	0.07459	"apnea"
78	0.15914	0.84086	0.17615	0.15769	"apnea"