# Project: Spectrogram clustering

# Topic # 3

# Signal Processing


**Students:**

Hugo Frazão

Saad Ejaz

**Faculty of Electronics and Information Technology**

Warsaw University of Technology

February – 2022

# Table of Contents

# Task Description

The project involves analyzing a signal which consists of two major parts: a constant factory noise and an occasional useful signal. Our task is to differentiate between different signals using spectrogram clustering. This can be done by first applying windowed Fast Fourier Transform (FFT) to the signal to construct a spectrogram. This spectrogram is analyzed and a clustering of Fourier-based features is performed to differentiate between different signals. Our task is to determine appropriate windowing and number of clusters, to visualize the time-domain signal and its spectrogram at both the intermediate and final stages of processing, and cluster the signal into two clusters: *useful* and *useless*.

# Algorithm and Results

First we will provide a brief summary of the procedure and the report will later on include details on specific portions of the algorithms, the results, and their analysis.

The algorithm can be broken down into the following steps:

1. FFT of the provided signal to generate a spectrogram containing timestamps, each of which represents a magnitude-frequency plot.
2. Clustering the timestamps resulting from the FFT into two clusters: useful and useless/discarded.
3. Perform post-processing on the clustering to remove irregularities and inconsistencies.
4. Smoothing and de-windowing the binary clustering to get an output signal containing only the useful parts of the original signal.
5. Manually annotate the original signal to get a *ground truth* to compare our results with. This will allow us to introduce a performance index i.e. the Jaccard similarity between the ground truth and our output. Using this, we can tweak the parameters in the previous steps to improve our results.
6. Run the algorithm on different signals (generated by ourselves) to see if the solution is universal.

# Signal Statistics

Our sample data is a *.wav* file that contains two parts, one being noise and other being the useful signal. The signal is initially plotted in MATLAB. Some of the signal characteristics of "glas-11025-fixed.wav" are given below:

Sampling rate (Fs): 11025 Hz

Length of wav. file in time: 8.9826s

Sampling period: Ts = 1/Fs = 1/11025 = 9.0703e-05s

Total number of samples: N = 8.9826s/(9.0703e-05s) = 99033 samples

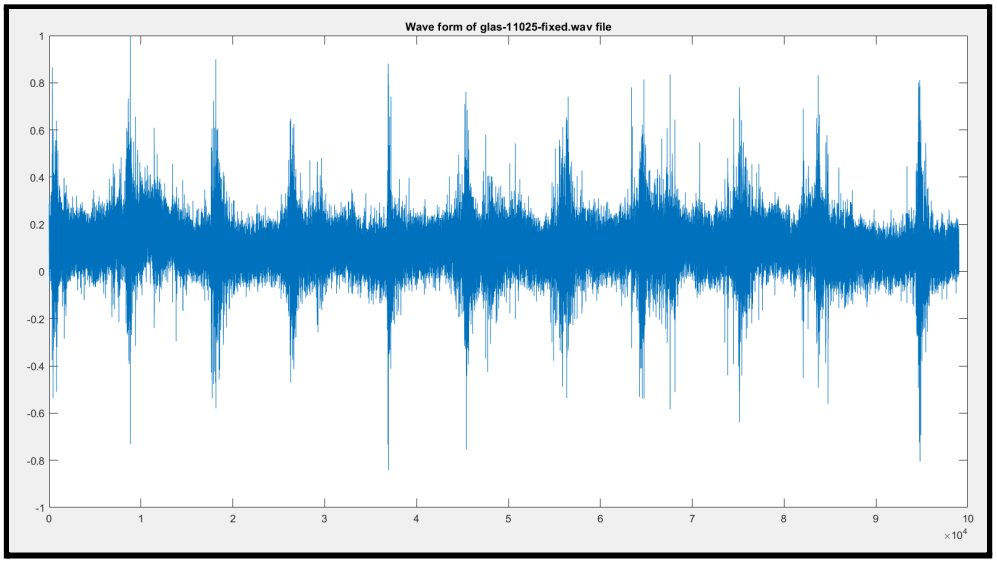


Figure 1 - Wave form of glas-11025-fixed.wav file

A small portion of the above signal (only 250 samples out of 99033) is plotted to better understand the discrete nature of the signal:

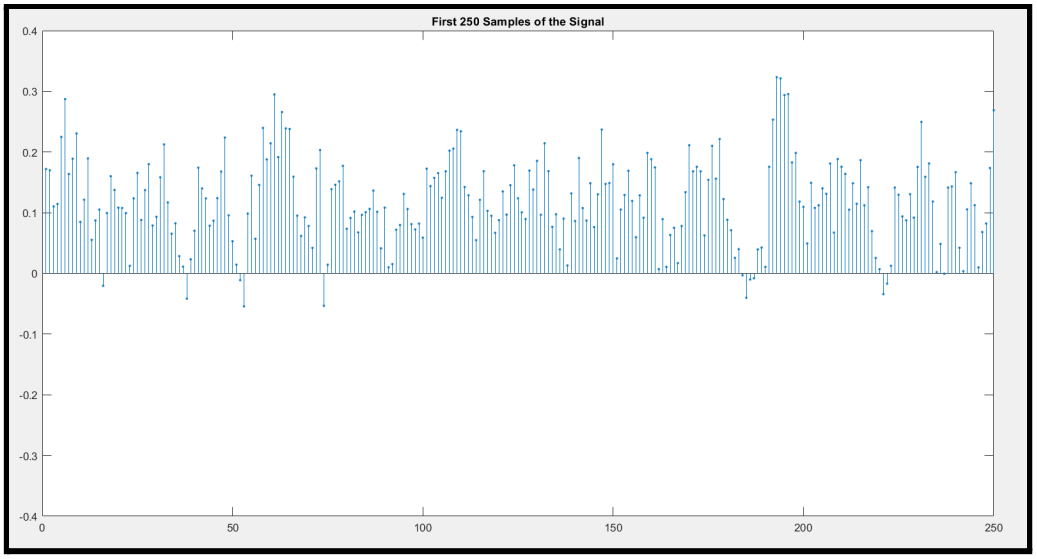


Figure 2 - First 250 samples of the signal

# Spectrogram - Windowed FFT

Since our audio signal is in the time domain and we need to perform clustering of Fourier-based features, we need to work in the frequency domain. For this purpose, we will use Fast Fourier Transform (FFT). A Fourier transform breaks down a signal in the time domain to a multitude of sinusoidal signals with different amplitudes and frequencies. An FFT is a faster way to compute Discrete Fourier Transform (DFT) on a signal and can be implemented on MATLAB using the *fft(...)* function. However, it is important to note that unless the measured signal is an integer number of periods (something that cannot be positively ensured in real life), using FFT may cause spectral leakage. This is presented in the form of high frequency peaks (arising from discontinuity because of noninteger periods) in the resulting frequency domain signal which are absent in the original time domain signal. Therefore, to prevent spectral leakage, a windowing function is used. Windowing diminishes the effect of the amplitude discontinuities at the borders of each finite sequence. This can be implemented in MATLAB using the *spectrogram(...)* function which has three main parameters to consider. These are listed below.

1. *The windowing function*: The *Hanning Window* is chosen as it works well in most cases [1].
2. *The length of the window:* We used the performance index to help us select a suitable value for the window size i.e. 128.
3. *Overlap:* This parameter determines the overlap between successive windows and is usually specified in the form of a percentage. Overlap is used to slightly reduce the effect of windowing when it comes to loss of information at the ends of the window. Overlap for our spectrogram was set to 75% of window size.

## Results

The Hanning Window function is shown below in both the time and frequency domains:

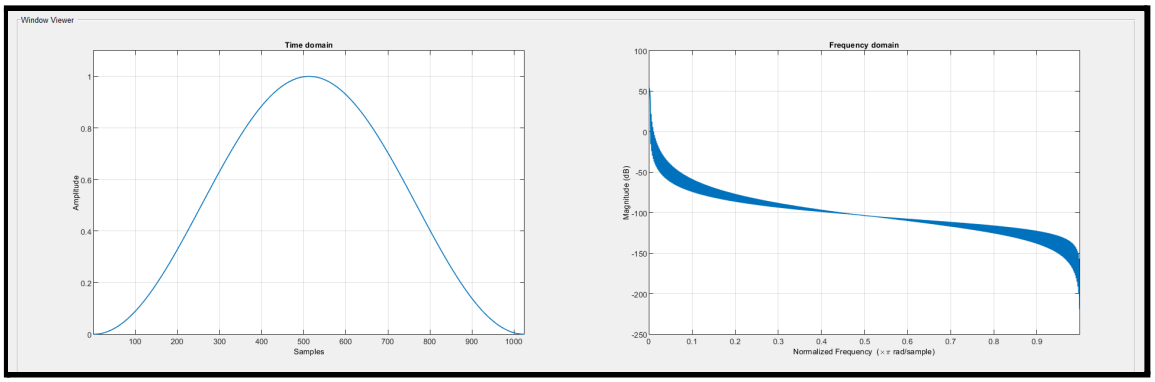


Figure 3 - Hanning Window in time and frequency domain

Applying this Hanning function to the signal, we can see the output below. A considerable amount of information is lost at the ends because of the shape of the windowing function (the ends have amplitudes close to zero). This is the reason we will use an overlap when generating the spectrogram.

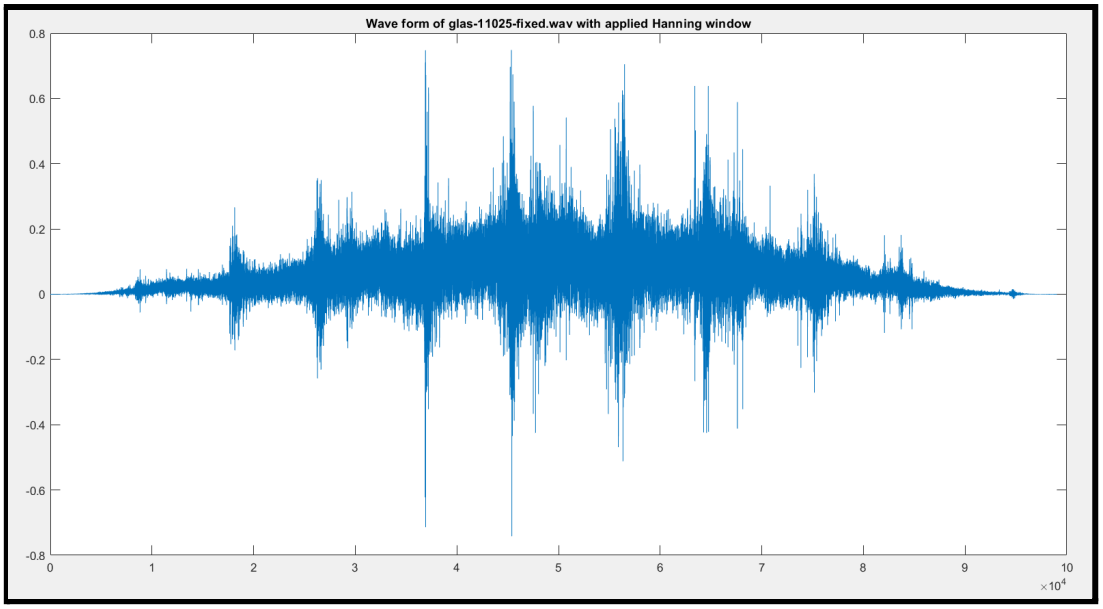


Figure 4 - Wave form of glas-11025-fixed.wav with applied Hanning window

Moreover, to have a better understanding of the signal we plotted it as a spectrogram so that we could have a 3D view of the signal in terms of time domain, frequency domain and power.

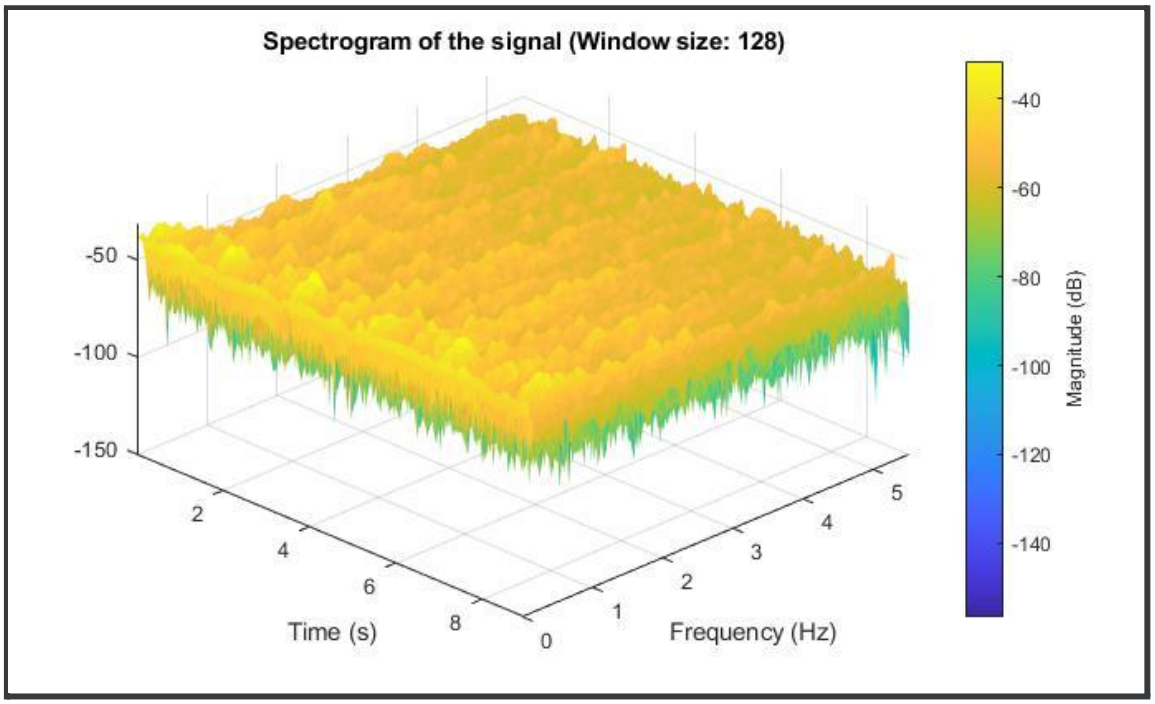


Figure 5 - Spectrogram of the Signal (Window size: 128)

## Analysis of Results

As a result of the FFT, we got *2603* timestamps each corresponding to magnitude values (S) in dB over *129* (window size + 1) frequencies. These timestamps will be the basis of our clustering as similar sounds will have similar magnitudes with respect to similar frequencies.

# Clustering

Clustering is a group of unsupervised learning algorithms that group together data points that are 'close' to each other. Closeness is determined by a distance function with the features of the data points as input arguments. The algorithm we plan to use in our project is the K-means clustering algorithm. For this algorithm, we start off with a specific number of centroids/mean-points (this number is denoted by K), and after each iteration the distance of each point to each centroid is calculated. Following this, the points are allotted to the cluster described by the nearest centroid. The centroid is recalculated (it is the mean of all the points in a cluster), and the process repeated. Many different distance functions may be used to determine closeness, but the *Euclidean distance* is the most popular and is what we will use in our project. Initialization of centroids is integral for good clustering and even convergence. The K-means algorithm is implemented in MATLAB in the *kmeans(...)* function which we will use in our project. The Fourier-based features of the signal (*frequency and amplitude*) will be considered for the clustering.

For our project, we will divide the timestamps we got from our spectrogram analysis (plots of magnitude over frequency) into two clusters: *useful* (1) and *useless* (0). However, it must be noted that clustering is a random process (the results depend on the position of the initial clusters - which are positioned randomly). To counter this, we do the following:T

1. Perform K-means clustering multiple times (70 in our project). The input to the clustering is a vector of magnitudes (S) corresponding to *129* frequencies. Moreover, since the S values are complex numbers, their absolute value will be used for clustering.
2. Determine the loss in each of the clustering (Loss is defined as the aggregate distance of each point from its assigned cluster. In MATLAB, this is an optional output of the *kmeans* function).
3. Choose the clustering with the least loss. This will be our final clustering.
4. Lastly, since we know that the useful part of the signal is *occasional,* we set the index of the smaller cluster to 1. This will ease our further calculations and will also be better for viewing purposes.

## Results

After the clustering is completed, we get the following binary vector. As stated before, 1 represents a useful signal while 0 represents a useless signal.

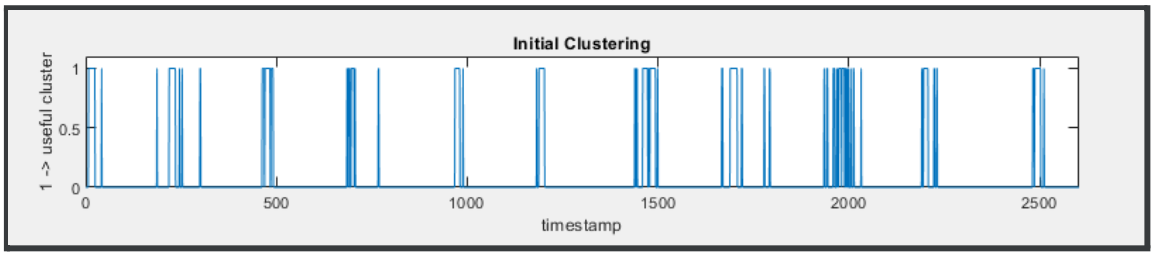


Figure 6 - Initial Clustering results

## Analysis of Results

The results shown in Figure 6 show decent results as we can clearly identify what timestamps the useful signal (i.e., approximately periodic) occur in. However, using this clustering will not yield good results as there are many inconsistencies. The peaks are very sparse with many timestamps of useless signals embedded within the useful ones. Moreover, the length of the set of timestamps of the useful cluster is very low, which would make the resulting output too short to be comprehensible. To rectify this we pass the clustering results to some post-processing steps that will better help isolate useful signals in the form of *regions* of timestamps.

# Post-Processing of Clustering Results

To alleviate the randomness and uncertainty in the original clustering, we perform three post-processing steps, which are as follows:

1. *Merging:* Small groups of useless timestamps embedded within largely useful cluster timestamps are converted to useful. This effect is seen as a merge of *infiltrating* useless signals. The variable (in the code), *merge_threshold* determines the minimum length of the useless signal *pulse* as a percentage of the total number of timestamps of the signal. A merge threshold of 1.5% is taken in this project.
2. *Weeding:* This processing step removes very small pulses of useful cluster portions. The variable *weeding_threshold* determines the minimum length of the useful signal pulse as a percentage of the total number of timestamps of the signal. A weeding threshold of 0.2% is taken in this project.
3. *Smooth De-windowing:* When converting from the windowed signal back to the original signal, the *smoothing* process occurs. Each pulse of the useful signal is extended on both sides (determined by *smoothing_extension* which is also a percentage of the total number of timestamps in the signal) smoothly. The output is an amplitude scale vector that has 1s for the useful pulses and a smooth *parabolic* curve on the extended portion of each end of every pulse. The amplitude in the parabolic/smooth region varies between 0 and 1, depending on the variable *parabolic_center* that is used to fit a parabola (it the the amplitude at midway of the extension). Following this, the amplitude scale is multiplied (element-wise) with the original signal (and optionally trimmed) to get the output containing the useful part of the signal. A smoothing extension of 2.5% is taken in this project.

## Results

The results after these three steps can be seen in the following plot. Moreover, a spectrogram is also plotted which depicts the useful (blue) and useless clusters (red).

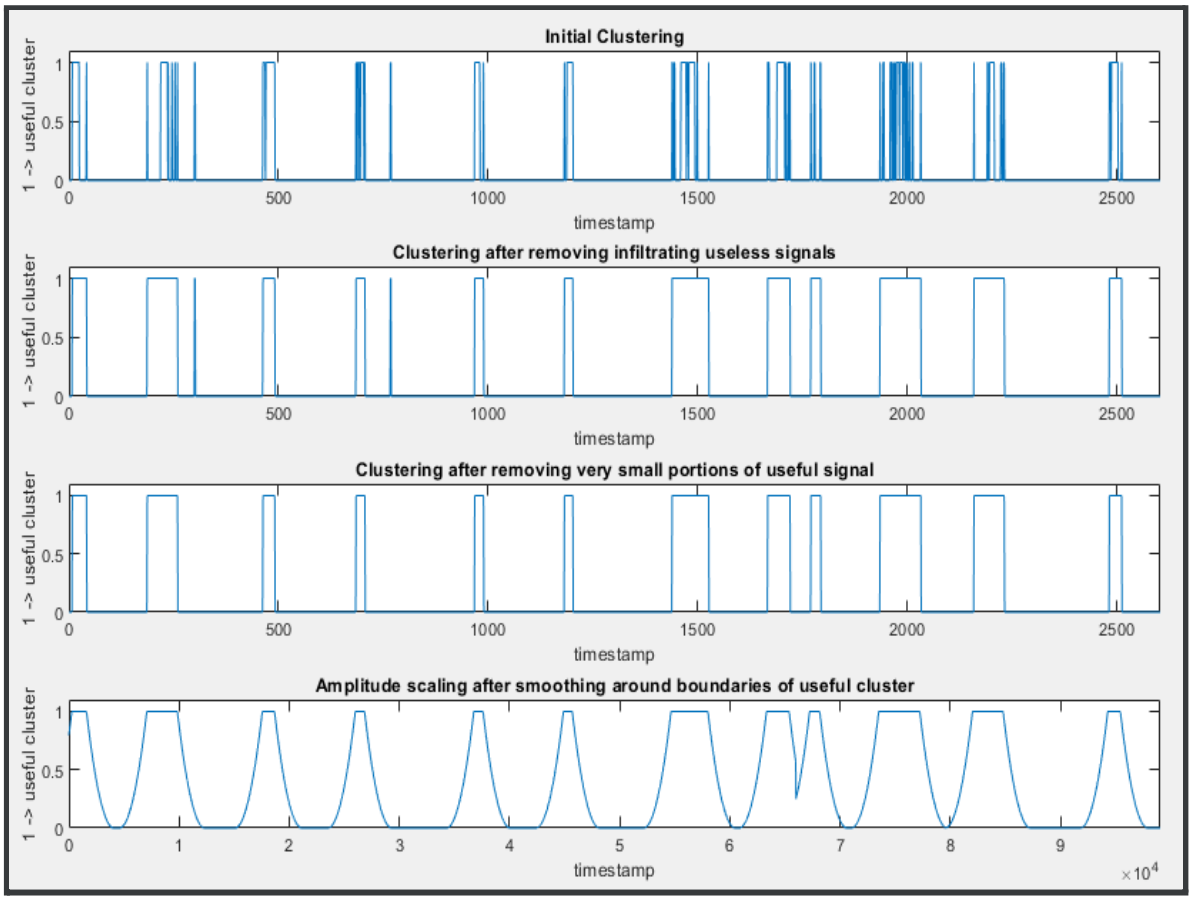


Figure 7 - Clustering and intermediate processing stages

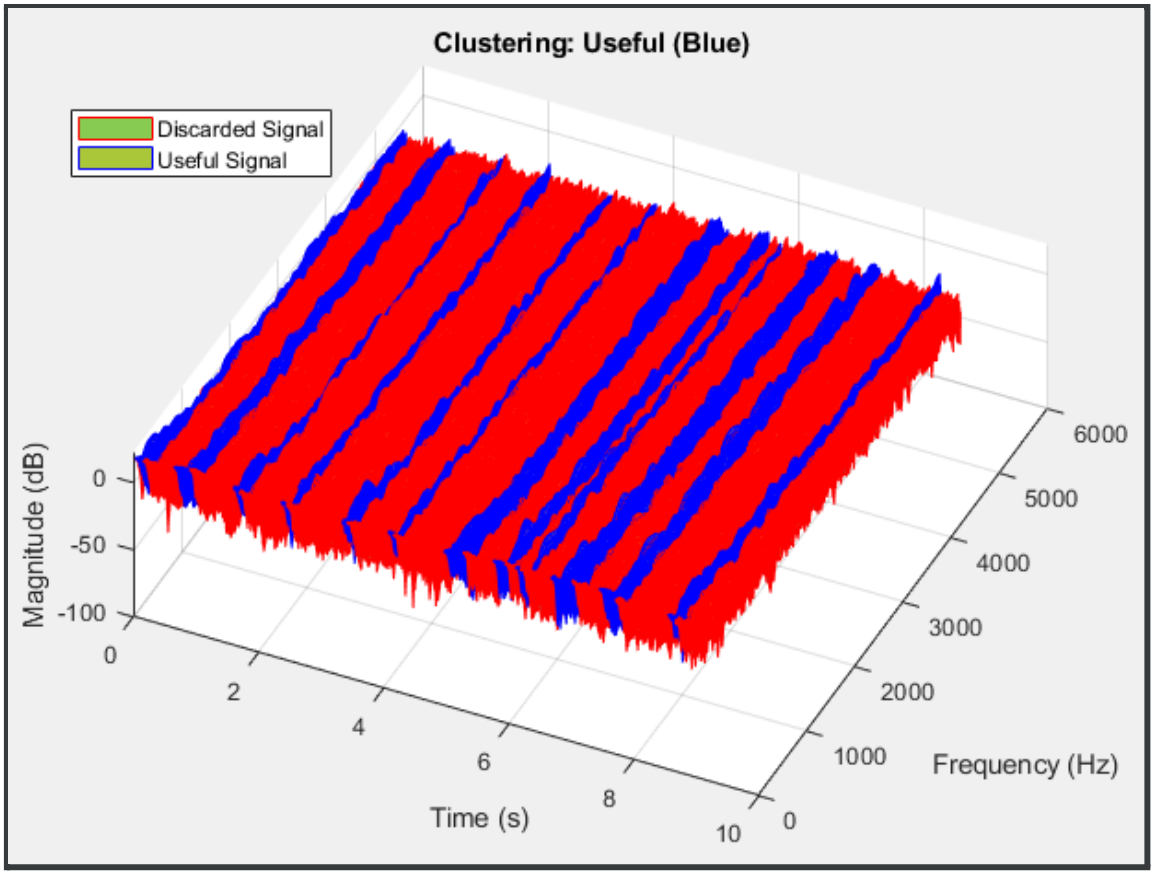


Figure 8 - Spectrogram of the final result

## Analysis of Results

The post-processing steps allowed us to mold the clustering into well defined regions of useful signals (that can be compared by listening to the original signal), rather than inconsistent peaks. This will allow for a better chance to verify our results as it is easier to distinguish chunks of timestamps than single timestamps. Moreover, the smoothing process ensures that the useful portion of the signal is not sudden and that it gradually builds up. This significantly improved the quality of the audio output (stored in the folder *useful_audios*). However, verifying our results is largely qualitative uptill now. We need to develop a method to quantify performance with which we can tune the many parameters involved in the process. The next section discusses that in detail.

# Performance Measurement and Improvement

In order to measure and improve the performance of our algorithm it was necessary to determine what the true clustering was (also known as the *ground truth*). This was done by manually annotating the original signal and marking the points that correspond to the start and end of the useful signal pulse. All annotated signals are stored in the folder *annotated_audios*.

The first step to produce this binary vector was to listen to the audio file and select the interval of samples of which we consider the useful part. For that we used *Audacity* to listen and mark time samples.

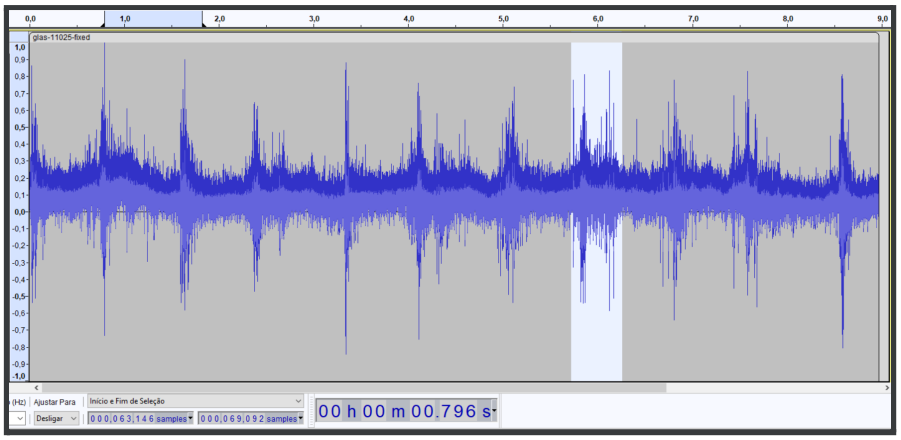


Figure 9 - Selection of the useful signal's sample interval

In the next figure in green we show all the intervals that we consider as being useful.

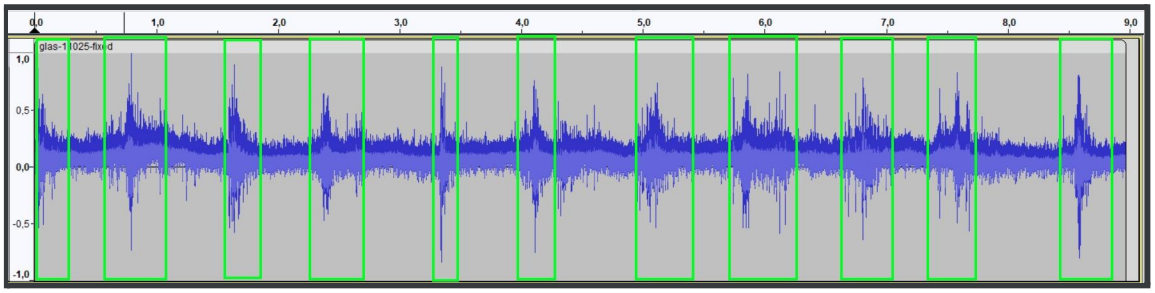

Figure 10 - Signal manually clustered

After taking all the intervals we inserted the ground truth matrix into matlab and plotted it to see if it matches with the previous selection.

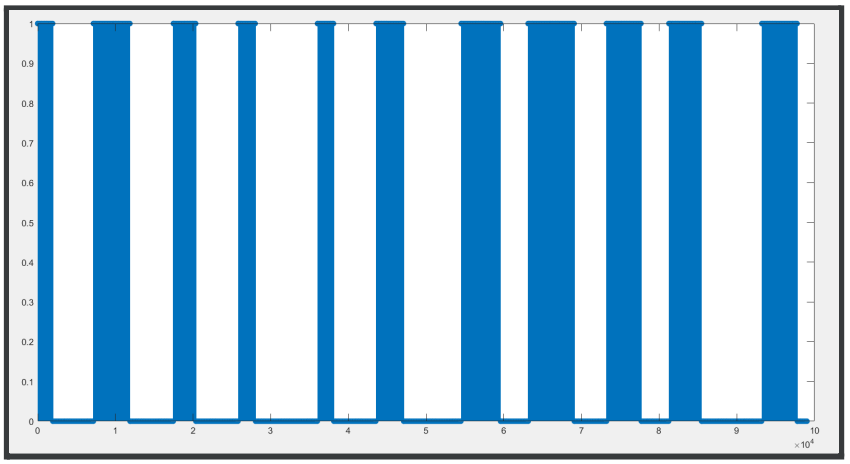

Figure 11 - Ground truth binary vector plotted in MATLAB

Comparing our results with the ground truth, we get the following plot:

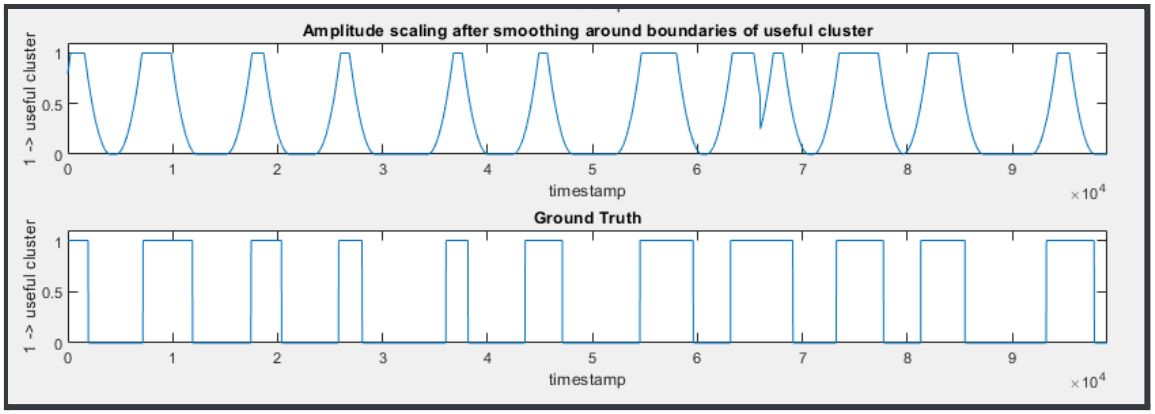


Figure 12 - Final clustering vs ground truth

We can clearly see that from a visual perspective the clustering results coincides fairly accurately with the manually determined ground truth signal. The results are quite satisfactory.

As we have the ground truth as the expected output for our algorithm, we can use it as a guideline to adjust the parameters in the code. To measure the performance of the algorithm we used the Jaccard similarity coefficient as a metric. This coefficient is used to compare the ground truth with the output data and see how much of both are similar. The Jaccard coefficient for two sets A and B is expressed as shown below. The *jaccard* MATLAB function is used in this project.

$$Jaccard\ coefficient\ = \frac{|A \cap B)|}{|(A \cup B)|}$$

The Jaccard index/coefficient for our clustering (given the manually determined ground truth) is 0.7368, which is considerably good as the Jaccard also penalizes useless signal parts identified as useful (which happens a lot in our project due to the nature of our processing algorithms, mainly *merging* and s*moothing*. Before calculating the Jaccard index, the amplitude scaling vector was converted to a binary vector by a simple MATLAB comparison operator i.e., every amplitude greater than 50% was considered 1, while the rest 0.

To have a better visual idea of the data used to calculate the Jaccard index we can subtract the result output to the ground truth and plot it to see the difference between the two clusters.

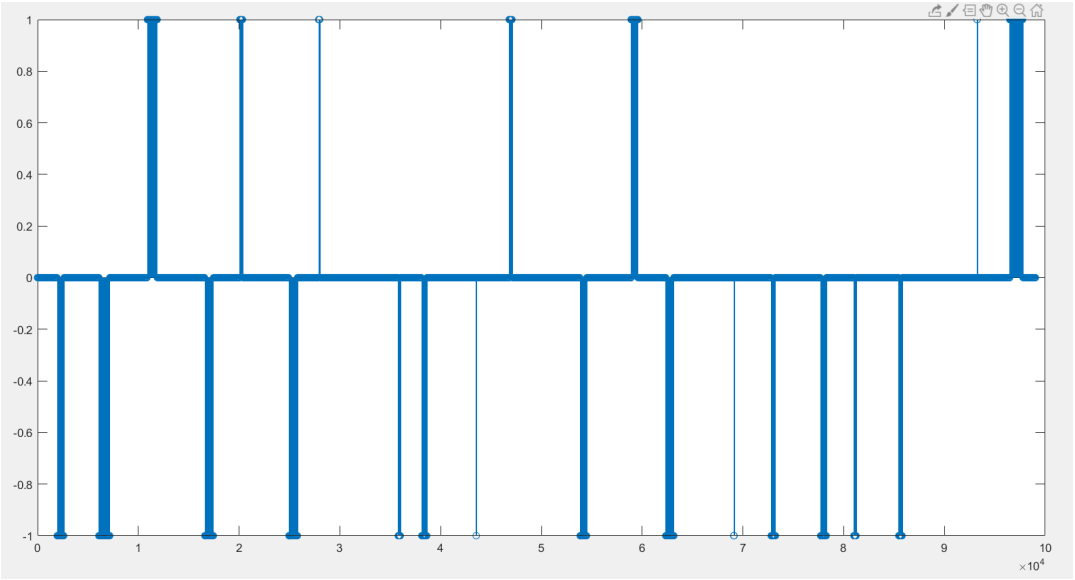

Figure 13 - Difference between ground truth and output result

Moreover, we can also use the Jaccard similarity index to find the best set of parameters among a large parameter space. An example to do that is noted in the text file accompanying the code (*example_parameter_search.txt)*, the screenshot of which can also be seen on the next page.

The example in this case used a subset of the set of parameters (*window_size, overlap_ratio, smoothing_extenstion)* each with their own domain. For each combination of values, the clustering was run and the Jaccard index noted. This would give a sense of what parameters are good for the clustering process.

```
Jaccard Similarity Indices for windowSize, overlap_ratio, smoothing_extension (%)

Jaccard: 0.69084 for 64 0.25 0.5
Jaccard: 0.76312 for 64 0.25 1
Jaccard: 0.77067 for 64 0.25 1.5
Jaccard: 0.73052 for 64 0.25 2
Jaccard: 0.67388 for 64 0.25 2.5
Jaccard: 0.7105 for 64 0.5 0.5
Jaccard: 0.75125 for 64 0.5 1
Jaccard: 0.73658 for 64 0.5 1.5
Jaccard: 0.68426 for 64 0.5 2
Jaccard: 0.62744 for 64 0.5 2.5
Jaccard: 0.71818 for 64 0.75 0.5
Jaccard: 0.73867 for 64 0.75 1
Jaccard: 0.70379 for 64 0.75 1.5
Jaccard: 0.65485 for 64 0.75 2
Jaccard: 0.60986 for 64 0.75 2.5
Jaccard: 0.49617 for 128 0.25 0.5
Jaccard: 0.59454 for 128 0.25 1
Jaccard: 0.66158 for 128 0.25 1.5
Jaccard: 0.67516 for 128 0.25 2
Jaccard: 0.65729 for 128 0.25 2.5
Jaccard: 0.51406 for 128 0.5 0.5
Jaccard: 0.63316 for 128 0.5 1
Jaccard: 0.69398 for 128 0.5 1.5
Jaccard: 0.68836 for 128 0.5 2
Jaccard: 0.67224 for 128 0.5 2.5
Jaccard: 0.57949 for 128 0.75 0.5
Jaccard: 0.66008 for 128 0.75 1
Jaccard: 0.72762 for 128 0.75 1.5
Jaccard: 0.70419 for 128 0.75 2
Jaccard: 0.67189 for 128 0.75 2.5
Jaccard: 0.40451 for 256 0.25 0.5
Jaccard: 0.73224 for 256 0.25 1
Jaccard: 0.64439 for 256 0.25 1.5
Jaccard: 0.70346 for 256 0.25 2
Jaccard: 0.65013 for 256 0.25 2.5
Jaccard: 0.6602 for 256 0.5 0.5
Jaccard: 0.72606 for 256 0.5 1
Jaccard: 0.72136 for 256 0.5 1.5
Jaccard: 0.69334 for 256 0.5 2
Jaccard: 0.64151 for 256 0.5 2.5
Jaccard: 0.43751 for 256 0.75 0.5
Jaccard: 0.54726 for 256 0.75 1
Jaccard: 0.62591 for 256 0.75 1.5
Jaccard: 0.65087 for 256 0.75 2
Jaccard: 0.64584 for 256 0.75 2.5
```

Figure 14 - Partial screenshot of an example parameter search process

**Note:** For our project, we evaluated the performance both quantitatively (by using the Jaccardian similarity index) and qualitatively (by listening to the output). Therefore we did not only rely on the Jaccardian results. The parameters we ended up choosing initially belonged to a smaller parameter space (than in Figure 12), and the values of each parameter are noted in their respective sections.

# Test on Additional Noisy Signals

To check whether our solution is generic/universal, we test it on other signals of similar type. A noisy audio signal is recorded in which the constant noise is a vacuum cleaner in the background of a bottle hitting a chair repeatedly. The noisy audios can be located in the folder: *noisy_audios.* The signal and its useful parts (manually determined) can be seen in the figures below:

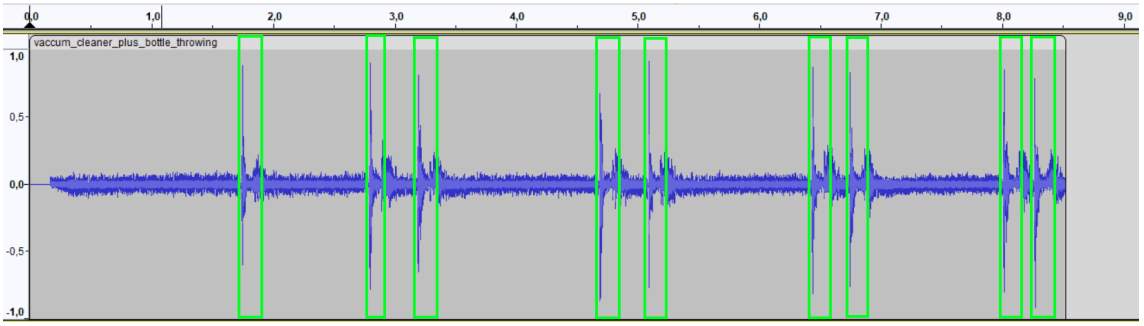


Figure 15 - New signal manually clustered

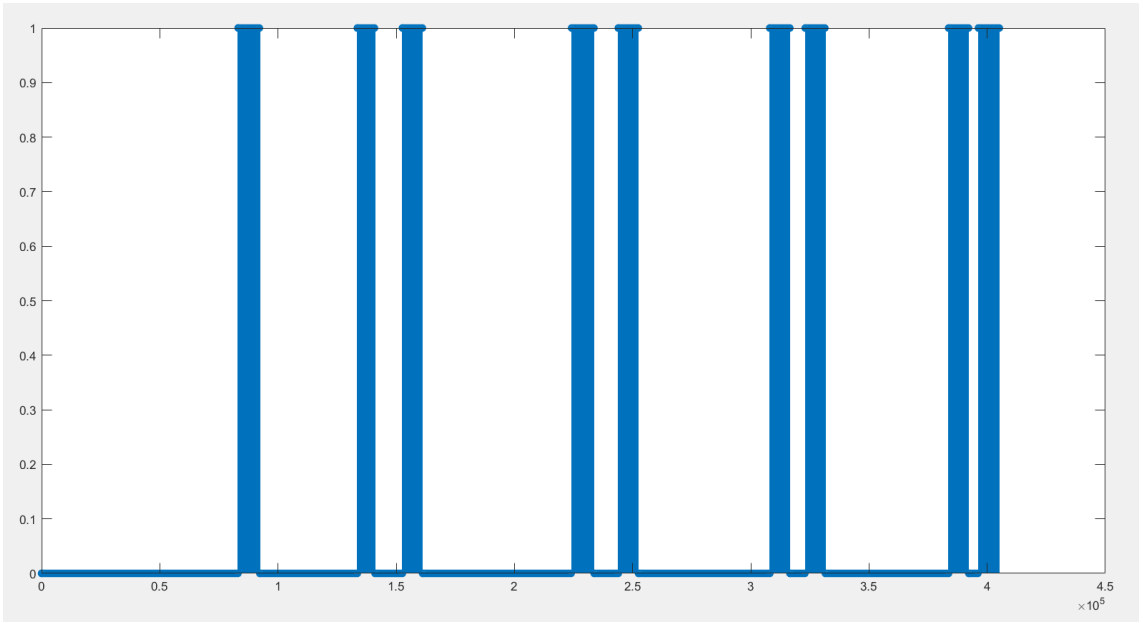

Figure 16 - New signal binary vector plotted in MATLAB

The audio signal named *vaccum_cleaner_plus_bottle_throwing.wav* is clustered and processed the same way as the given signal was and the results can be seen in the next figure. Moreover, since the signal was also manually clustered, we also have the ground truth for this case. Therefore, the Jaccard's similarity is also determined and noted.

The Jaccard similarity for this signal was 58.22% which may seem low, but qualitatively speaking, the output signal extracted sounds promising. The reason for the low Jaccard score is that it equally penalizes false positives (useless signal clustered as useful) and false negatives (vice versa), whereas our code is designed to extract as much of the useful signal as possible, even if it means that some noise is retained.

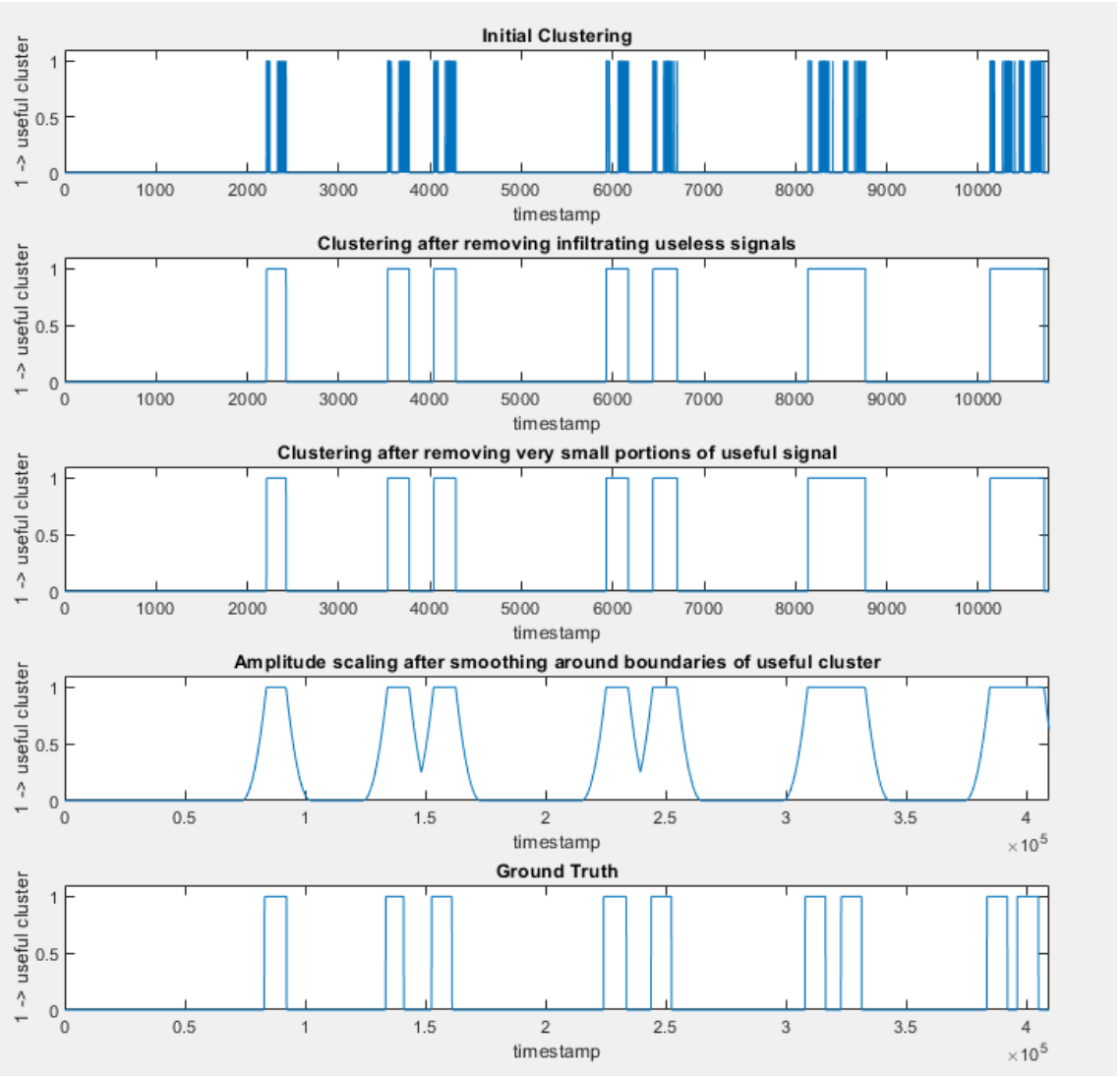


Figure 17 - Clustering results for *vaccum_cleaner_plus_bottle_throwing.wav*

Moreover, as we can see in Figure 17, the ground truth and our results closely match and we can distinguish individual portions of useful sound. The main error in the Jaccard index arrives from having to cluster two useful pulses that are very close together (in the time domain). The *smoothing* and the *merging* processes make distinguishing such pulses hard. Nevertheless, the results are quite satisfactory, and after tests on a third signal *loud_vacuum_with_muffled_claps.wav,* we can be confident in the effectiveness of our procedure.

# Conclusion

This project had the objective of analyzing and clustering a signal with two different major parts which are a useful signal and a constant noise. It used a windowed Fast Fourier Transformation to construct a spectrogram so that it could be possible to analyze and cluster the signal. This way it was possible to divide the signal into two parts. To know how good the clustering was we did our own manual clustering by manually selecting the useful parts of the signal and then compared both outputs using the Jaccard similarity index.

With the help of the Jaccard similarity index we changed the parameters of the fast fourier transformation window size, the overlap ratio and the length of the smoothing in order to make it as high as possible. As the result output can be listened to after converting it to a .wav file we could also confirm by listening if the results are as expected. We ran the code several times with different parameters and concluded the best size for the window, overlay and length of the smoothing. Which were: Window size: 128, Overlay: 75% and Smoothing: 2.50% . We can conclude that it is possible to perform clustering to a signal by using its frequency and timestamp. It is also concluded that the Post-Processing of the initial clustering is crucial to obtain good results. Without it the result output would contain a lot of cuts in between the useful signal.

The future work for this project could be the use of a better performance index than Jaccard (the reasons discussed in relevant section), and do a better parameter search using different audios as input. This way we can improve the parameters in a much more refined way.

# References

[1] “Understanding FFTs and Windowing” - National Instruments.
https://download.ni.com/evaluation/pxi/Understanding%20FFTs%20and%20Windowing.pdf