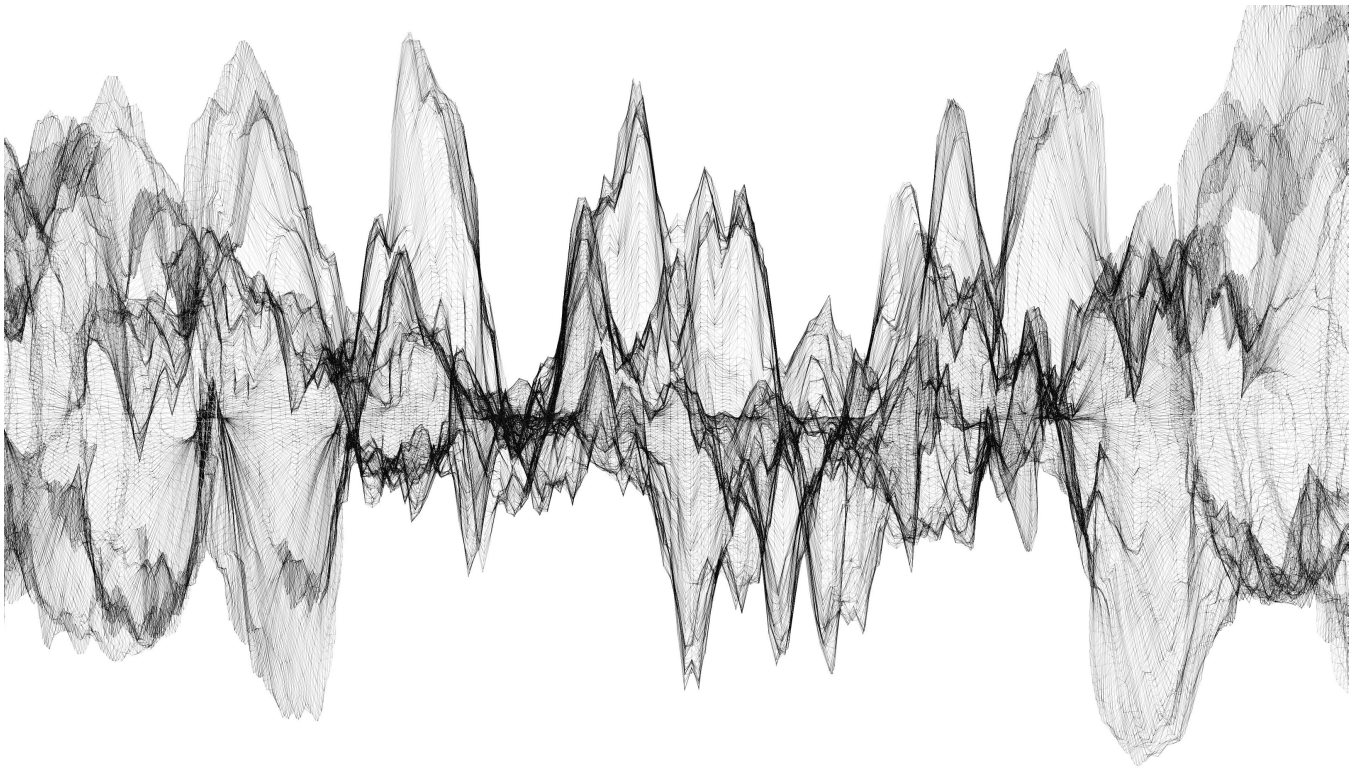


https://www.pinterest.com/pin/402368547950165147/

Machine Learning on Voice: a gentle introduction with Snips Personal Wake Word Detector

Thibault Gisselbrecht Follow
May 2, 2018 · 13 min read

Written by Thibault Gisselbrecht and Joseph Dureau

The wake word is the word you use to start a conversation with a

voice assistant. Inside every device in which a voice assistant is embedded, a tiny process keeps listening, waiting to detect the wake word out of a continuous stream of audio. The process at stake is both typical of how voice is generally transformed to feed a machine learning model, and quite simple from a Machine Learning perspective. We are going to take this example as an illustration of how Machine Learning is done on voice.

The personal wake word detector is a new feature of the Snips Voice Platform, that we release in response to strong demand from the community. It makes it possible for anyone to pick any wake word they want to use to call their voice assistant. Go straight to the tutorial if you want to start playing with it. To understand how this detector works, and how voice is handled in many other Machine Learning applications, carry on reading!

Personal and Universal Wake words

There are two kinds of wake word detectors: *Universal* and *Personal* ones.

The universal wake word detectors are trained on a large variety of voices. The underlying model is generally a Deep Learning algorithm, that is trained to identify when anyone says the wake word.

On the other hand, the personal wake word detector is trained locally, on your device, with a small number of voice samples that you provided. This alternative is much more versatile compared to the universal one, since it allows you to use any arbitrary wake word. The

difference is just that it's not meant to work when someone else says the wake word.

What's common to both type of detectors is the way voice is transformed before it's fed into the Machine Learning model. This pre-processing is actually shared with many other applications of Machine Learning with voice, like Speech Recognition, or Speaker Identification. It just turns out that results obtained with this pre-processing step are often better than without, although exceptions are starting to arise.

Let's see how this personal wake word detector works in terms of training, inference, and performances.

Training

Audio trimming

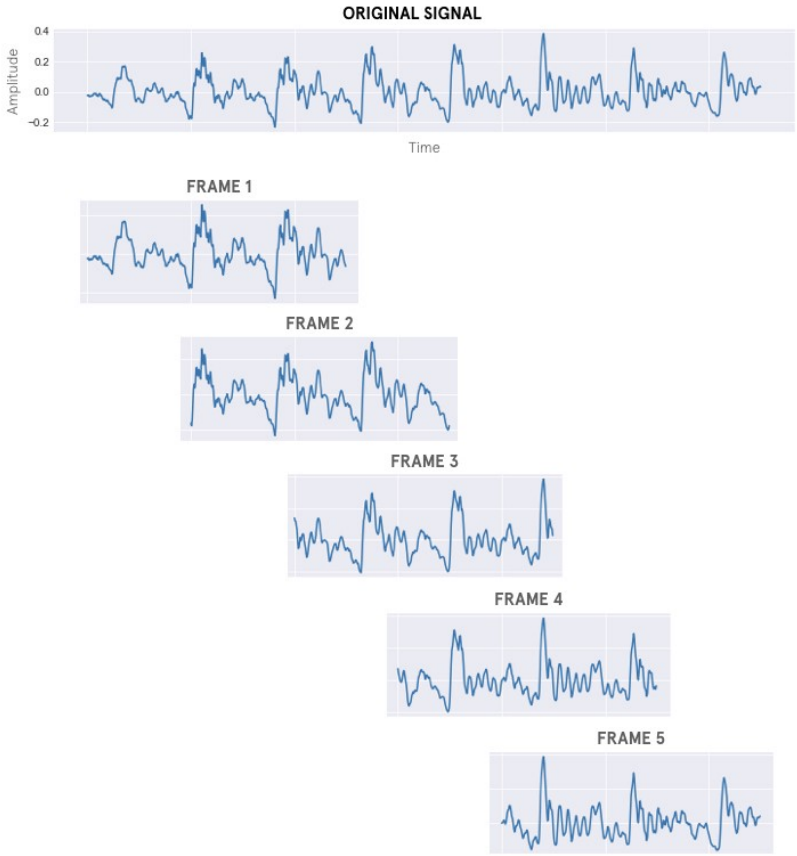
The principle of the personal wake word detector is to compare an incoming stream of audio to a set templates of the wake word recorded by a user. It is a nearest neighbor logic. Hence, the first step is to acquire those templates. In practice, since we have no idea of how long the wake word will last, a margin is taken to record them. In our case, we give the user 2 seconds to record each template (see the documentation here). The sample rate used to acquire the sound is set to 16000 samples per seconds. Each record is then initially composed of 32000 samples.

Naturally, what comes before and after the wake word in the

recording is not useful. If the recording takes place in a quiet environment, which we heavily recommend, what comes before and after are silences. In order to remove those silences and keep only the meaningful part of each template, a process called trimming is applied. This process consists in:

- Dividing the signal into small chunks (framing)
- Computing the energy of each frame
- Removing every frame which energy is lower than a predefined threshold.

For example, a 65 milliseconds signal might be sliced into 5 frames of 0.025 seconds with an overlap of 0.01 seconds between each frames. The figure below illustrates the framing process on a short audio signal.

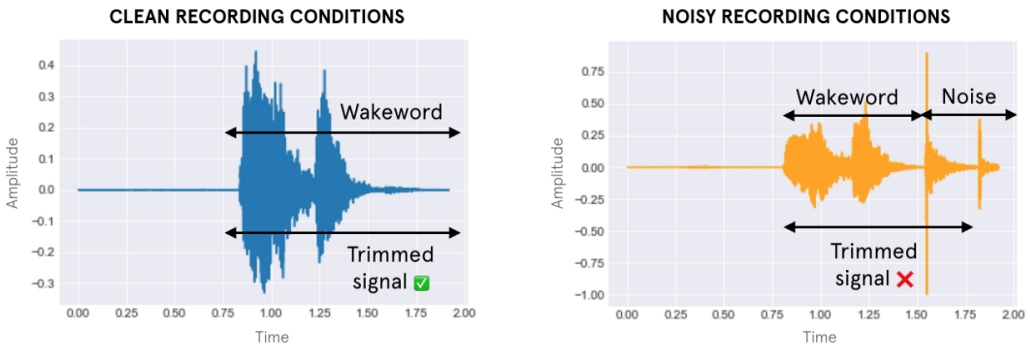


5 frames of 0.025 seconds with 0.01 second overlap

The energy of a frame is the mean of the squares of the signal. A classic approach is to compute the energy of each frame, and compare it to a predefined threshold to classify the frame as silence or not.

To increase robustness, we take a slightly different approach. We set a threshold on the ratio of energies between the energy of each frame, and the energy of the frame with highest energy. If this ratio is below 20 decibels, the frame is classified as silence. The threshold can be manually configured in the Snips platform.

The trimming process is illustrated below. On the left, the signal was recorded in clean conditions (no noise), and the trimmed signal strictly captures the wake word. On the right, an example with background noise recorded after the wake word. This example shows how noise can disrupt the trimming process.



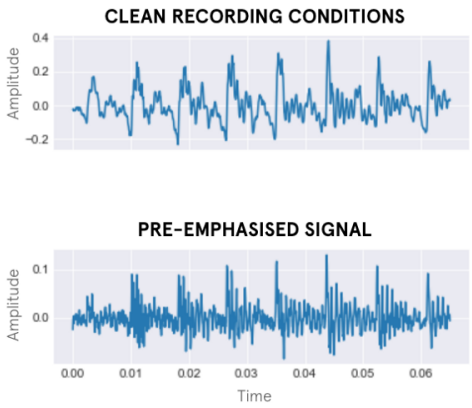
Trimming process in various recording conditions

Feature extraction

The second step of the training process is to extract a concise and meaningful representation of each template, to feed a simple machine learning model. This process is called *feature extraction*.

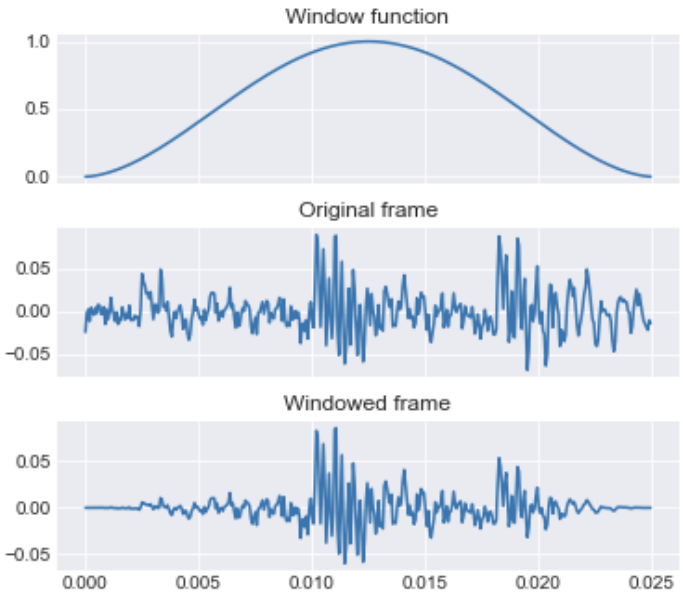
Mel Frequency Cepstral Coefficients (MFCCs) are widely used for Automatic Speech Recognition applications. This transformation tries to mimic some parts of the human speech perception by reproducing the logarithmic perception of loudness by the human ear. For a full tutorial on how MFFCs work, follow this link. The process to obtain the MFCCs from an audio signal is the following:

a. *Pre-emphasis (optional).* This step is aimed at amplifying the high frequencies, in order to balance the signal and improve the overall signal to noise ratio. In practice, we use a pre-emphasis coefficient if 0.97.



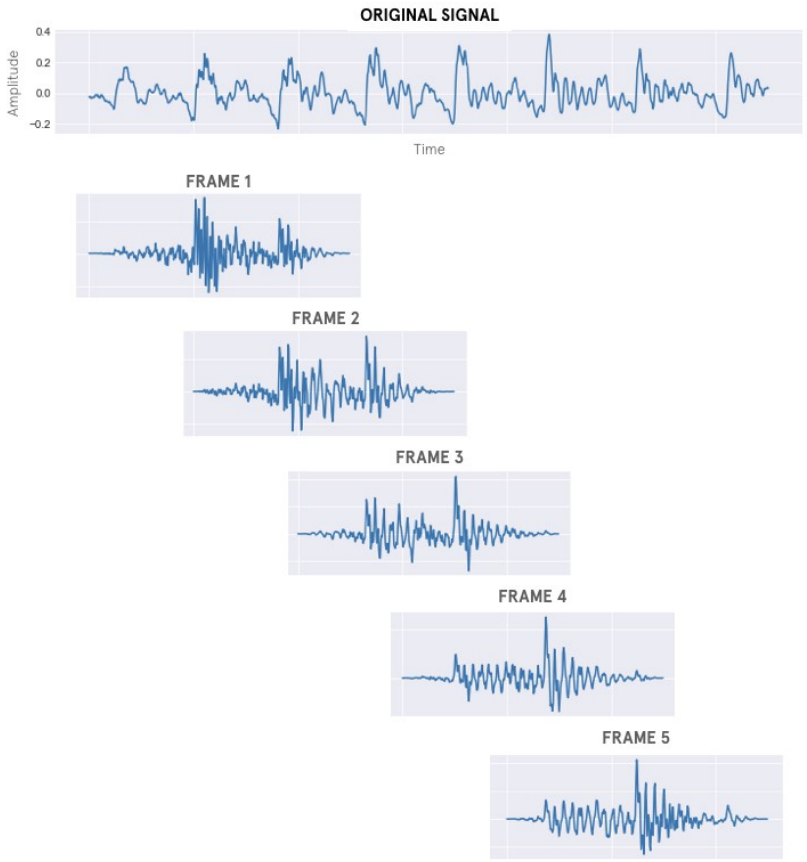
b. *Framing*. The signal is splitted into fixed-sized slices (usually between 20 ms to 40 ms long), with a predefined amount of overlap between each slice. This step is the same than the one described in the initial Trimming step, but the window sizes and the overlaps might be different. Let us denote *nFrames* the number of frames obtained from the original signal.

c. *Windowing*. A *Povey* window function is applied to each frame to reduce side lobes. The *Povey* window function is similar to the *Hamming* window, but equals zero at the edges.



Frame windowing.

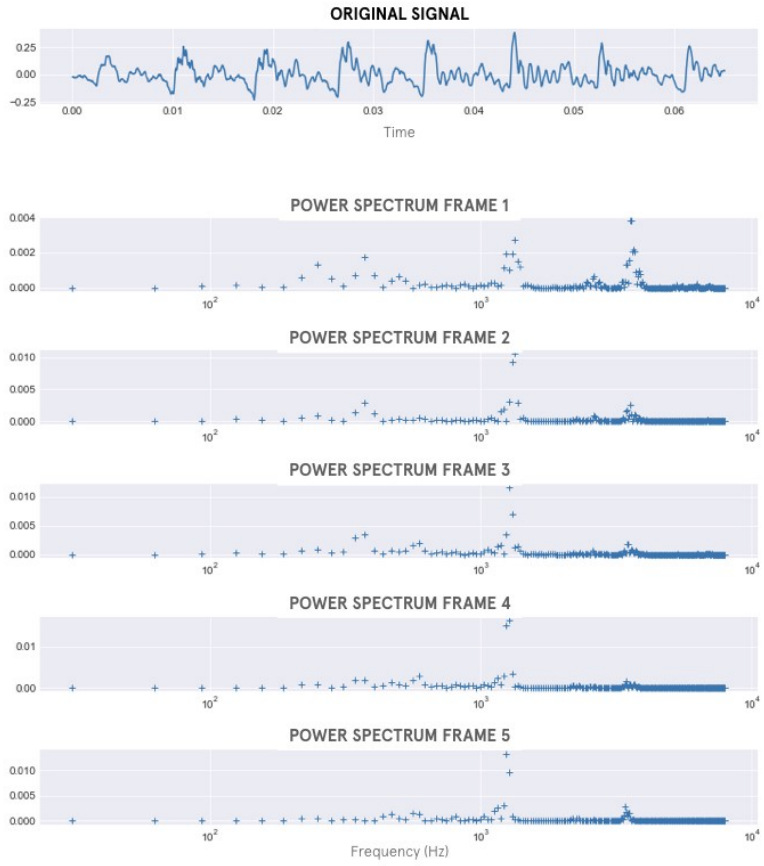
At this point the original has been pre emphasised, framed and windowed, which we illustrate in the figure below.



Emphasised, framed and windowed signal.

d. *Transformation.* We first compute the Discrete Fourier Transform for each frame, with a predefined number of components (512 here). Then, we compute the square of the module of each coefficient to obtain the energy distribution of the signal across frequencies. At this stage, each frame is represented by 512 values, which represent the energy of the signal at different frequencies. The *i-th* value of this signal corresponds to a frequency of *i*16000/(2*512)*.

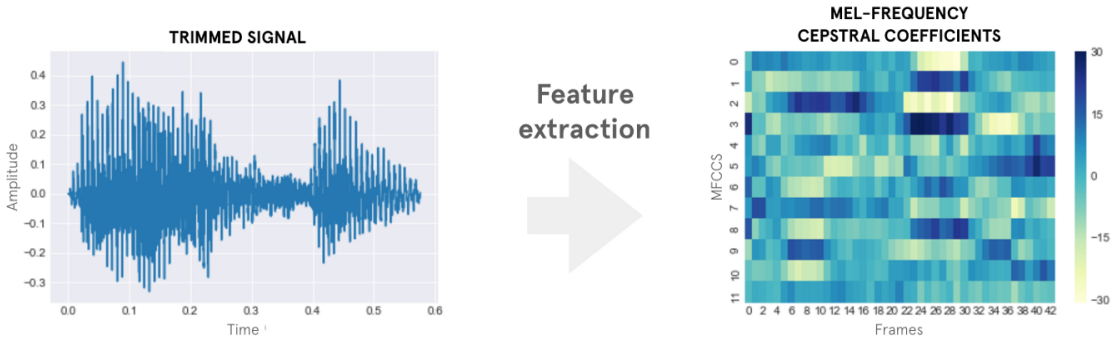
We illustrate this on our toy examples below.



e. *MEL scale mapping*. We map each component onto the mel scale using triangular filters: this transformation is aimed at reflecting the way the human ear perceive sounds. The number of filters, noted *nFilters,* is a parameter of the algorithm. At that point, each frame comes with a vector of size *nFilters*. We then take the log of each component to obtain the log-filter banks.

f. *Normalisation*. Finally, we apply a discrete cosine transform (DCT) and normalize each component by removing their mean. This last step is meant to decorrelate the log-filter bank.

At this point, our audio sample has been transformed into a feature matrix of size *nFrames* ×*nFilters* where *nFrames* is the number of frames resulting from the second step above and *nFilters* is selected the number of mel coefficients at the last step. The figure below illustrates the feature matrix of the trimmed audio signal from the previous section.



Features extraction process

In the next section, we will explain how we use those transformed templates in our wake word detection algorithm.

Inference

Dynamic Time Warping

Now that we have a clean representation of the audio templates, our goal is to compare an incoming stream of audios to these templates, in order to decide whether a wake word is present or not is the audio stream.

To make things simpler, let us consider for now that only one audio template is available. Concretely, the live audio stream is sliced into windows with a size equal to the template and a defined overlapping shift. Each window is then processed with the feature extractor, and finally compared to the template. To perform this comparison and output a decision, we use Dynamic Time Warping, which measures the similarity between two time series (see this link for a full introduction to DTW techniques).

Let us consider two times series of respective sizes $N \times K$ and $M \times K$, where N and M are time dimensions, and K is the dimension of the features space. In our application *N=nFrames[Template]*, *M=nFrames[Stream]* and *K=nFilters*.
The first step to compute the DTW between those series is to evaluate the related pairwise element cost matrix. The metric used to compute this matrix can change depending on the need, for our algorithm we chose the *cosine similarity*. Each element of this matrix is defined as:

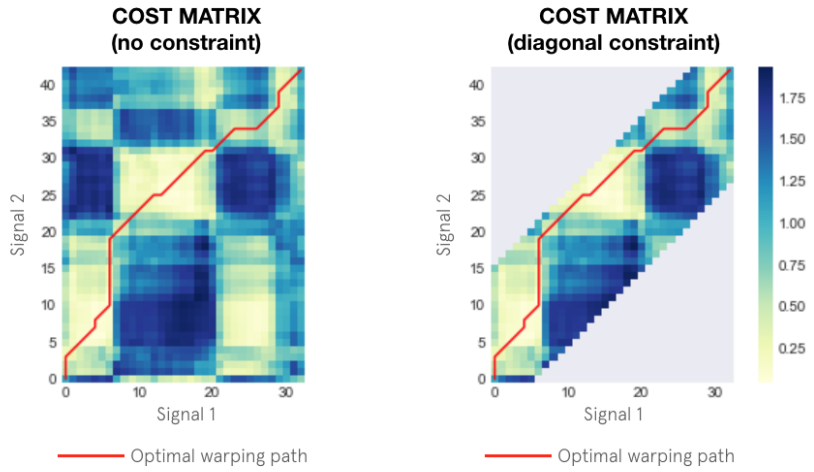
$$c(i,j) = 1 - \frac{x_i \cdot x_j}{||x_i||_2 ||x_j||_2}$$

$$\text{where } i \in \{1..N\}, j \in \{1..M\}, x_i \in R^K, x_j \in R^K$$

After this step we end with a matrix of size *(N,M)* called *cost matrix*. The aim of DTW is to find a path going from the first element (with coordinates *(1,1)*) to the last element (with coordinates *(N,M)*) of the *cost matrix* with a minimal cumulated cost. This path is called an

optimal warping path and its total cost defines the DTW distance between both signals.

The figure below on the left represent the cost matrix between two series together with the related optimal warping path. Intuitively, the warping path tends to align the points of the two time series together with the constraint that every point from sequence 1 (respectively sequence 2) must be mapped to at least one point of sequence 2 (respectively sequence 1). In our specific case, since a wake word is always pronounced at a relatively constant tempo and since all its syllables are always pronounced in the same order, some warping path can be discarded. This constraint can be encoded directly encoded in the *cost matrix* by setting infinite cost to each element in the area that we want to discard. For our wake word detection algorithm we use a diagonal constraint (see figure below on the left).



Dynamic Time Warping

Reference distance and prediction

At this point, we are able to compare a live audio stream with each template by computing the corresponding DTWs. Formally, if we have 3 templates and consider a window from the audio stream, we can compute the 3 related DTWs, denoted DTW_1, DTW_2 and DTW_3. In practice, the duration of the audio stream window is set to the average duration of the 3 templates.

Our objective is to classify the audio stream window as containing a wake word if its DTW with respect to *at least one* the templates is less than a predefined threshold. An optimal value for this threshold can be found empirically for each specific wake word. Yet, this value will heavily depend on the size of the templates. The higher the number of frames, the longer the path, and hence the highest the DTW. To counter this effect, and to be able to define a universal threshold (as in not wake word dependent), each DTW is normalized by the sum of the temporal dimension of both inputs.

Finally, if we call the decision threshold τ, and consider an input sequence from the audio stream, the detector will trigger if either of the normalised versions of DTW_1, DTW_2 and DTW_3 is lower than τ.

Let's now see how we determined a good default value for τ.

Confidence

The wake word detection problem can be seen as a binary classification problem. The audio stream window either contains, or doesn't contain the wake word. To tune this kind of problems, it is particularly useful to be able to output a probability to belong to each

class (in our case to be or not to be a wake word). To this aim, we artificially define 3 confidences related to each input sample:

$$p_i = \frac{1}{1+e^{-\frac{DTW_i - \tau}{\tau}}}$$

where i is template indice. Note that if DTW_i is less than τ, then the probability will be greater that 0.5, and will increase as DTW_i becomes smaller, as expected. Finally, the probability threshold above which the detector will trigger is the parameter that we expose to the user (see the tutorial here). We call it the sensitivity. The higher sensitivity, the higher the number of false alarm and the lower the number of missed wake words.

The next section is dedicated to an analysis of the performances of our algorithm, and its limitations.

Performances and limits

Finding an acceptable reference distance τ

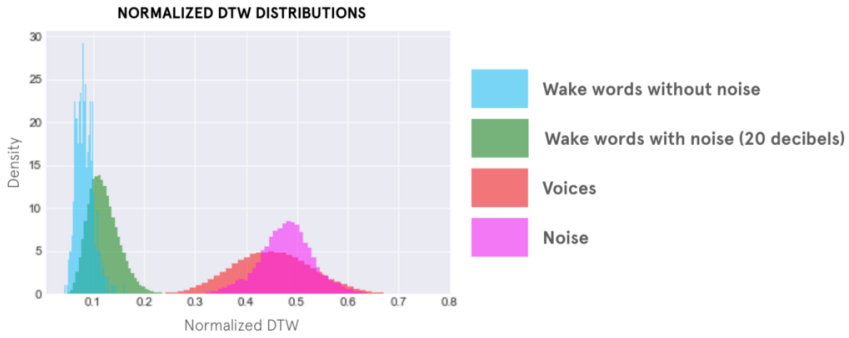
We looked for an acceptable universal value for τ. To do so, we empirically recorded different wake word templates, for different people, languages, and lengths of the wake word. For each of them, we computed the normalized DTWs with respect to the following set of audio samples:

1. *Other templates, not noisy:* for each wake word, and each

template of this wake word, this is the set of all the other wake word templates of the same wake word, except the one in question. Ideally, all DTWs with respect to the studied template should be lower than τ.

2. *Other templates, noisy (20db):* similarly, for each template of each wake word, this is the set of templates of the same wake word, except the one in question, augmented with background noise, with a signal to noise ratio of 20 decibels. Ideally, all DTWs with respect to the studied template should be lower than τ.
3. *Voices:* this is a set of audio recordings composed of people pronouncing random text. This set is around 10 hours long in total. Ideally, all DTWs with respect to the wake word templates should be greater than τ.
4. *Noises:* this is a set of recordings of background noise . Ideally, all DTWs with respect to the wake word templates should be greater than τ.

We repeated this process for each wake word template, for different wake words, and aggregated the results. The figure below represents the normalized DTW distribution for each of the sets defined above. Looking at this figures gives a first intuition about the distance threshold that could be used. It appears that setting it around 0.22 will lead to a good separation between wake word and non-wake word recordings, both with and without background noise.



To confirm this intuition we computed the *False Alarm rate per Hour (FAH) on* the *Voices* dataset, that is the trickiest one, and the *False Rejection rate (FRR)* for actual templates, both in noisy and non noisy conditions, with a distance threshold of 0.22. The FAH quantifies the number of times the detector will accidentally trigger each hour while the seconds quantifies the rate at which a wake word will be missed. Both measures are the most commonly used in the literature for this task.

On voices, the *False Alarm rate per Hour* is 2.06, which means that with people constantly talking next to it, the wake word Detector will trigger about twice per hour with default settings. Improved performances can be obtained by tuning decision parameters on a case by case setting. The *False Rejection rate* is 0% without background noise, and 0.3% in noisy conditions.

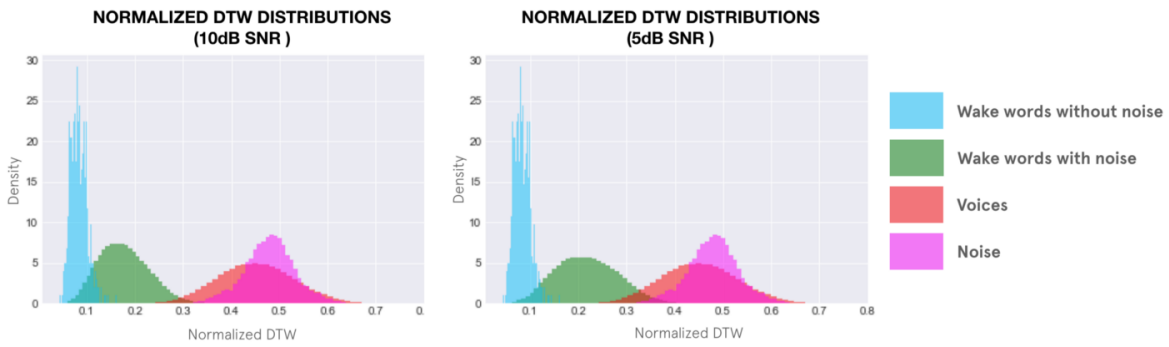
Limits of the approach

To push the exploration further, we increase the level of noise in the test datasets. We created a first dataset with 10 decibels signal to noise ratio, which is lower than the 20db used initially, and a second

one at 5 decibels signal to noise ratio. Intuitively, the performance of the algorithm should decrease when the signal to noise ratio decreases, since the Wake word becomes more difficult to detect.

The same plots as in the previous section are shown with the 10 decibels set on the left and the 5 decibels on the right. As expected, in both cases the frontier between positive and negative samples

Thanks to Joseph Dureau, Alice Coucke, Rand Hindi.



Keeping the distance reference to a value of 0.22 the *False Rejection rate* respectively increase to 2.8% and 20.4% with 10 and 5 decibels signal to noise ratios.

Of course, as mentioned before, those are average performances obtained with default sensitivity threshold. Improved performances can be obtained by adapting the sensitivity for each Wake word.

In order to improve the robustness of the system, we are currently working on two approaches:

- Adding an audio frontend in order to artificially reduce the