Lab Work 2

André Pedrosa [85098], Filipe Pires [85122], João Alegria [85048]

Algorithmic Information Theory
Department of Electronics, Telecommunications and Informatics
University of Aveiro

November 23, 2019

Introduction

This report aims to describe the work developed for the second assignment of the course of 'Algorithmic Information Theory', explaining all programs developed by us, and presenting the results we considered most relevant regarding the quality of the solutions.

The programs implemented in C++ have the purpose of analysing and encoding audio files and ultimately being capable of, from a small audio segment, identifying the music that it most likely belongs to.

Along with the description of the solution, we also discuss the effects of the variation of the programs' parameters and how accurate are the results. All code developed is publicly accessible in our GitHub repository: `https://github.com/joao-alegria/TAI`.

1. Data Visualization

In this chapter we present a description of the dataset used, the small script we developed to convert audio segments from stereo to mono and the histograms we are capable of plotting by adapting one of the scripts given along with the assignment's description (*1*).

1.1. Dataset

We were given the access to a small dataset containing 7 audio files from different musics. It was these music fragments we used to test our code during development. Each audio file is in `.wav` format and has two signal channels (stereo). They vary between 13 and 29 seconds of audio and, when played, none seems to contain significant noise.

For testing the performance of the programs once the development phase was completed, we came up with our own dataset of musics of different genres. These music files vary between 3:07 and 8:09 minutes and have the same format and number of channels as the original dataset. As they were downloaded from the original sources, their quality is close to ideal.

1.2. Mono Conversion

One of the tasks proposed was to create a script that converts stereo audio files into mono. This was fairly straightforward to do, as it only required for us to read the v alues from all signal channels and calculate the average of each. The script is executed in the format presented below, once built:

```
$ ./wavquant inputFile outputFile [-q quantSize] [-r reductFactor]
```

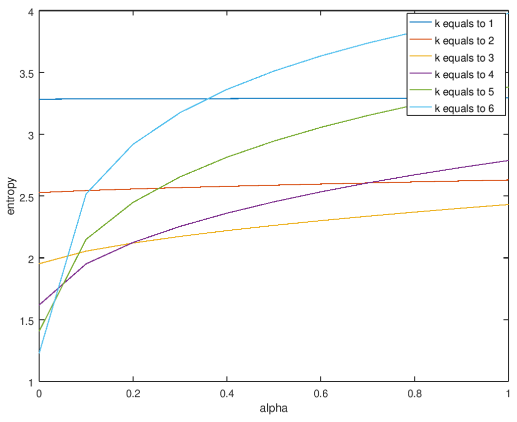
This script, `wavquant.cpp`, is also used for other purposes, in which the `quantSize` and `reductFactor` parameters are useful. For this reason, we made the parameters optional, so that a user can run `wavquant` to simply convert stereo files into mono, with a default number of bits used to encode each value of the segment (each sample) of 16 and no frequency reduction factor. This script is mentioned further ahead in greater detail.

1.3. Histograms

We were also given a script called `wavhist.cpp` that outputted to the terminal the histogram of an audio file. We adapted this script so that it accepts both stereo and mono audio files and plots in a figure the histogram of one of the audio channels. The script uses `gnuplot` (*2*), a portable command-line driven graphing utility, and has the following format:

```
$ ./wavhist inputFile channel
```

Figures 1 and 2 contain the histograms plotted from the same music in the original format (stereo) and after its conversion to mono. The x axis represents the frequency of the values and the y axis the number of occurences in the music of each frequency.



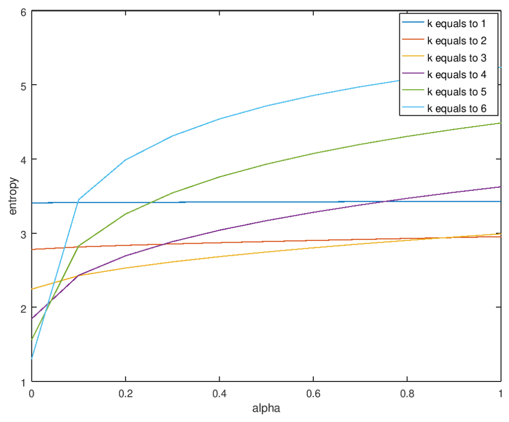


Figure 1: Histogram of music_name in the original format - channels 0 and 1.

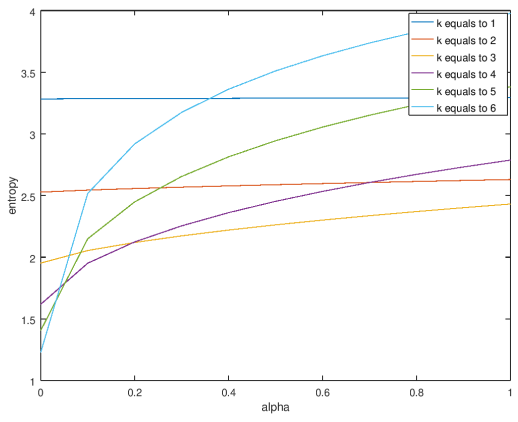


Figure 2: Histogram of music_name after its conversion to mono (1 channel only).

....

Mention something about why do the shapes are as they are and why is mono so different

....

The C++ scripts mentioned in this chapter all use `libsndfile`, a C library used for reading and writing files containing sampled sound (*3*). This was proposed on the assignment and allowed us to read and manipulate the audio files for more complex tasks.

2. Data Processing

Once we were capable of visualizing the data, we proceeded to actually doing something useful with it. In this chapter we explain the implementation of the program `wavquant.cpp`, responsible for reducing the number of bits used to represent each audio sample. The implementation of the formulas presented on the assignment's description for the signal-to-noise ratio and the energies of the signals and noises is described as well, along with the computation of vector quantization codebooks of audio files.

2.1. Uniform Scalar Quantization

The idea behind a uniform scalar quantization (USQ) is the reduction of bits used to represent a signal. Its usage has an instrinsic tradeoff between signal quality and memory space required to store the information. We do not get into much detail regarding the mathematics behind this process, but we make available a figure taken from a presentation from the Stanford University (*4*) that helps visualizing the outcome of applying the USQ to a signal. Figure 3 contains a signal presented in blue and the outcome of the signal after it is quantized is presented in red. The figure also contains the quantization error variation on the second plot.

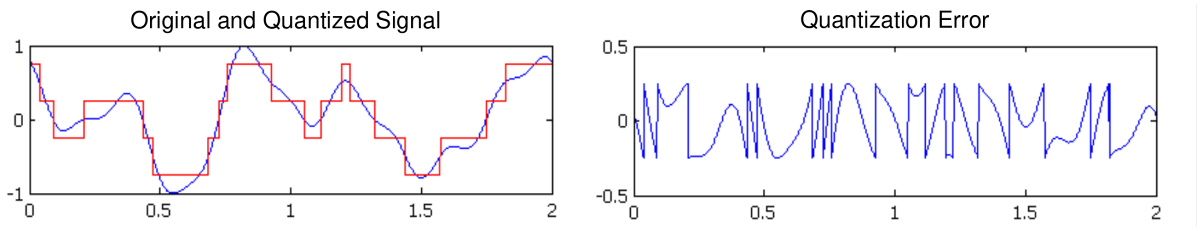


Figure 3: Example of a quantized waveform.

It is `wavquant.cpp` that is responsible for this process. As we have seen in Section 1.2, the script accepts two optional parameters: `quantSize` defines the number of bits to be used to represent the audio fragment given as input (ideally, this value should be less than of the original source); `reductFactor` defines the number of times the user would like to reduce the total number of values of the audio signal. This reduction factor works by calculating the average between `n` values, where `n = reductFactor`, and doing this for all values from the segment. The result is a signal with `n` times less values (samples).

2.2. Error Calculation

A signal-to-noise (SNR) ratio compares a level of signal power to a level of noise power. Higher numbers generally mean a better specification, since there is more useful information (the signal) than there is unwanted data (the noise).

In `wavcb.cpp` we calculate this ratio, along with the maximum absolute error per sample. The SNR is defined as stated in equation 1, taken from the assignment's description.

$$SNR = 10log_{10}\frac{E_s}{E_n}(dB) \tag{1}$$

Here, E_s is the energy of the signal, given by $E_s = \sum_k x_k^2$, and E_n is the energy of the noise, given by $E_n = \sum_k(x_k - \bar{x}_k)^2$, where x is the values from the audio segment. The maximum error is defined in equation 2, derived from the noise energy equation.

$$error = |x_k - \bar{x}_k| \Leftrightarrow max(error) = max(|x_k - \bar{x}_k|) \tag{2}$$

In practice, the error of a quantization tells us how distant from the original signal is the quantized one. Figure 3 shows an example of how this error looks like.

However, for the following assignment tasks, we determined that the energy of the noise would be more useful to us. One way to compare audio fragments would be through the signal-to-noise ratios; But we found that the only calculations actually required are of the noise energies as they work as the distances between two samples. By focusing on these values, not only do we save processing time, but we are also able to compare input audio segments to quantization codebooks already present in a program to determine the similarity between samples (and consequently between segments). These codebooks are explained in this next section.

2.3. Vector Quantization Codebook

Each song is represented as a sequence of samples, where each sample is a singular frequency if the file is in mono or the frequency of each channel if the file is in stereo that was registered in that exact second. A codebook is an abstract representation of the overall music, and in this context we used the concept of clusters to represent a given number of samples. The base idea for clustering is the fact that similar data entities should have similar properties and can be aggregated in a group, where data entries that differ significantly from each other, the properties that represent them should be considerably different, implying that they should belong to different groups (*6*). Dividing the totality of samples present in each song into vectors with X samples and representing them as points in a multidimensional vectorial space, as represented in Figure 4, enables the better comprehension of a cluster. Geometrically, clusters can be directly identified by the close grouping of the multidimensional points, since each a axis represent a data property, similar points with similar properties will stay closer to each other. Taking advantage of this natural grouping, clustering algorithms try to find points that represent the several data groups in a viable way, most of the time a middle point of the given group, called **centroids**.

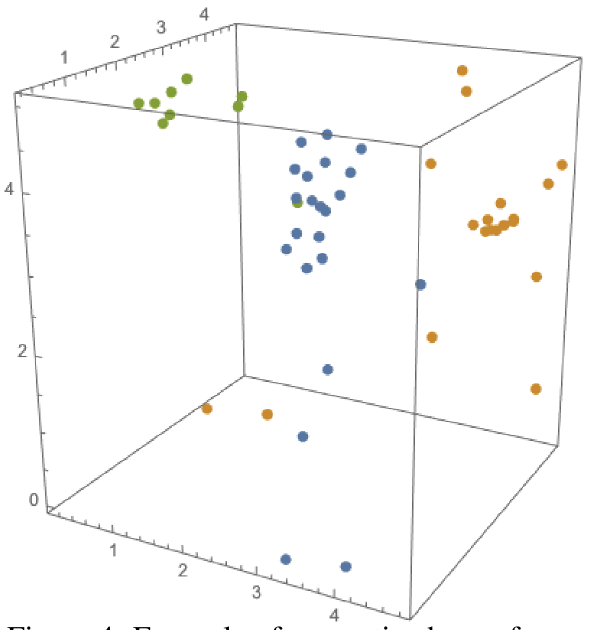


Figure 4: Example of a quantized waveform.

For our problem, we used the KMeans Clustering algorithm, probably the most well-known clustering algorithm (*6*). KMeans is a good approach because it's fast to implement and to execute, since it's only necessary to calculate the distance between a given point and the centroids. Our KMeans implementation starts by doing the already mentioned music partition in different blocks/vectors of a given size provided by the user, then the algorithm chooses in a random fashion an a priori given number of centroids. After the pre-processing is done, the main algorithm starts by calculating the closest centroid for each vector, and then updates each centroid by calculating the middle point of the points assigned to that specific centroid, this is repeated as many times as necessary for the error delta, i.e., the error between each iteration to be lower than a given threshold specified by the user. The script `wavcb` is the implementation of this algorithm and enables the creation of a codebook for a given song, being its interface the following:

```
wavcb <inputFile> <blockSize> <overlapFactor> <errorThreshold>
<numRuns> <outputFile>
```

Giving a brief explanation of what each parameter means, *inputFile* is the path to the song the user intends to serve as base for the codebook, *blockSize* is the number of samples to be used in the blocks/vectors division, *overlapFactor* is the factor(>1) that each block will overlap with the previous one, the bigger this percentage is the better the song is represented in the multi-

dimensional space. *errorThreshold* is the maximum error that allowed to exist in last iteration, meaning that the centroids are no longer being adjusted in a significant way, *numRuns* is the number of times the KMeans algorithm will run to find the best local minimum and finally the *outputFile* is the path to the file were the codebook should be stored.
Although quite flexible, KMeans has some disadvantages such as the fact that it is necessary to insert the number of centroids to take in consideration, this in many situation is not the best option since the main purpose of using clustering is to get some insights about the data, preferring that the algorithm finds the number of centroids by itself. Another disadvantage is that the KMeans centroids start from a random initial starting position, each means that each run could converge to a local minimum, making each run different from each other.
In our implementation we tried to combat the last error by running several times the KMeans algorithm to gain access to several local minimums, being the possible to choose the best one. Even though KMeans is considered a fast clustering algorithm, if given a considerable number of data entries, it's normal it starts to became cumbersome. To combat this aspect we implemented parallelism into the process in two levels, both internal, which in a practical sense means that we used threads to calculate the closest centroid to a given point, and external which translates in using different threads to different KMeans runs.

2.4. Codebook Parallel Processing

As audio files get longer, the number of frames per file increases. This will lead to the growth of the number of blocks that will be extracted from it. Furthermore this blocks can be extracted with overlap among them which increases even more the total number of blocks. This high number of blocks, brings computational time implications since on the step of classification of blocks to the closest centroids on the K-Means algorithm has to go over all the block and compare each one to all centroids and get the closest for each block. To reduce this impact, this step can be parallelized by dividing the blocks into groups and assigning each group to one thread. To prevent having to create complex synchronization mechanisms, each thread stores the association between blocks and the closest centroids on different data structures and then the main thread is in charge of reading from those to recalculate the centroids.

As mentioned in section 2.3. the K-Means algorithm give us a local solution for a given centroids' initial position, which means other and better solutions might exist. In order to find them, different centroids initializations must be tested, yet since a K-Means run has high computational cost, several of them even more computational cost has. To ease this process, each K-Means run can be assigned to a thread allowing to execute several runs at the same time. In terms of implementation it implies some synchronization mechanisms so the main thread knows when a run was completed and to manipulate the structure to store the centroids plus associated error calculated on each run.

3. Automatic Music Identification

The program `wavfind.cpp` is the application of the previous scripts on a program with a specific purpose. ¡¡¡¡¡¡¡ HEAD WAVFind is ment to interpret an audio fragment and attempt to identify which music ======= WAVFind is meant to interpret an audio sample and attempt to identify which music ¿¿¿¿¿¿¿ 3bb061aa9116cf34b76b6ed8e40679589ad52aa2 from a database it belongs to. In this chapter we discuss our solution, the consequences of varying the parameters passed to it and the quality of the results.

3.1. Most Probable Music

The idea of WAVFind is very similar to the well known mobile app Shazam (*5*) - a user plays a sound, the app listens to it, processes it and determines the most probable music that sound belongs to. Shazam has a large infrastructure, counting on online real-time music detection, a nice user interface with access to the smartphone's microphone, and a variety of other features. We, on the other hand, focus on the functional end of this idea, implementing it as a command line program with a limited dataset of known musics and with the capacity to receive audio segments as input through their respective paths on the computer running the command.

To execute this command, one needs to use the following format:

```
$ ./wavfind inputFile blockSize overlapFactor codebookDir
```

All arguments are mandatory. *InputFile* is the name of the .wav audio whose origin music is to be identified. *BlockSize* is the value used as the size of the blocks in which the audio segment is split; this value must be the same size used when the codebooks of the music dataset were built. *OverlapFactor* corresponds to the percentage of each block that is overlapped with the following block; this is done to make the program more robust in particular situations where, for example, the input audio fragment corresponds to the final half of one block and the initial half of the next of a codebook. At last, *codebookDir* is the path to the directory containing the preprocessed codebooks of the music dataset.

The dataflow of the program is as follows: first, the *inputFile* is validated and its information is extracted; then, it is split into blocks with size according to the parameter *blockSize*; now, for each codebook present in the *codebookDir*, each segment block is compared to each codebook block by calculating the E_n (noise energy) between the two; here, the minimum E_n of each segment block is summed to a cumulative total that represents the error between the segment and the codebook; as this cumulative error is calculated for all codebooks, we then check which codebook returned the smallest error and assign it as the most similar to the given audio segment. This smallest error is determined iteratively - as each codebook is processed, we compare its cumulative error to the previous one and keep the smallest. The identified codebook's is finally printed onto the console (as its name is supposed to identify the music it belongs to).

Now, for the program to work properly, there are a few prerequisites. First of all, the program must have access to the codebook dataset. The process of creating codebooks is neither trivial nor short-lasting, so to have access to the audio dataset is not enough. Next, the audio segments passed as input files must contain only one signal channel (mono). This restriction reduces the complexity of the comparison process, with the obvious tradeoff of reducing the quality of the audio. On the other hand, in a real case scenario (like with Shazam), the user would usually record audio fragments from external sources such as speakers; so, to consider characteristics related to stereo files in the music identification process could lead to deceiving the program as the recording could fail in distributing the collected information to the proper channels.

3.2. Parameters Variation

In this section we present some results of executing WAVFind for all datasets. We also explain some experiences done with variations to the parameters passed to the command and the respective consequences.

...

3.3. Results Discussion

...

Conclusions

After completing the assignment, we drew a few conclusions regarding our solutions and the applicability of algorithms such as the K-Means to solving problems such as music identification.

First of all, ...

Regarding our satisfaction with the delivered code,...

In terms of code organization and readability, we made sure our repository was as well structured as possible and our code properly commented and documented. The base folder contains a *README* file for basic instructions and a *Makefile* to make the compilation process easier. All code is in the *src* folder and its documentation is accessible, with the help of the *Makefile* and the command ”make docs”, through the automatically generated *index.html* file in the *docs* directory.

References

1. Armando J. Pinho, *AIT: Lab Work no.2*, University of Aveiro, 2019/20.

2. H.B. Broeker, G. Clark, L. Hecking and E. Merritt, *Gnuplot: graphing utility*, `http://www.gnuplot.info/` , May 2019, [accessed in: November 2019].

3. Free Software Foundation, *GLibsndfile API*, `http://www.mega-nerd.com/libsndfile/api.html` , April 2013, [accessed in: November 2019].

4. Bernd Girod, *Image and Video Compression: Quantization*, `https://web.stanford.edu/class/ee398a/handouts/lectures/05-Quantization.pdf` , [accessed in: November 2019].

¡¡¡¡¡¡¡ HEAD

5. Apple Inc., *Shazam for iOS & Android*, `https://www.shazam.com/apps` , [accessed in: November 2019]. =======

6. George Seif, *The 5 Clustering Algorithms Data Scientists Need to Know*, `https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68` , [accessed in: February 2018].

¿¿¿¿¿¿¿ 3bb061aa9116cf34b76b6ed8e40679589ad52aa2