Lab Work 2

Andr Pedrosa [85098], Filipe Pires [85122], Joo 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 sample, 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 samples 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 samples from different musics. It was these samples we used to test all our code. 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.

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 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 audio files (either stereo or mono) 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 yaxis the number of occurences in the music of each frequency.

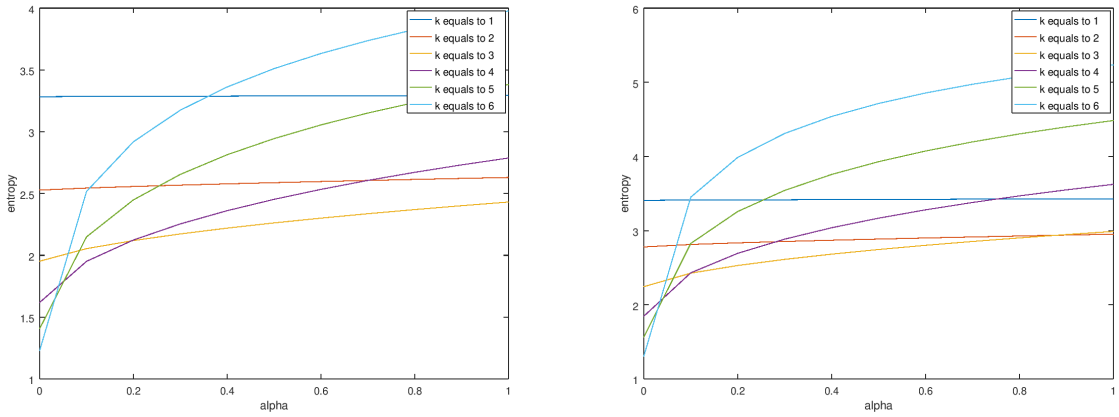

Figure 1: Histogram of music_name in the original format - channels 0 and 1.

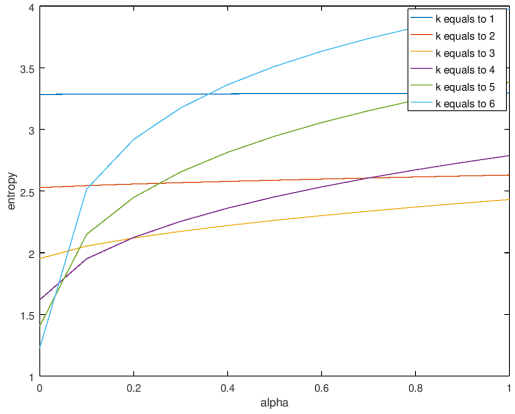

Figure 2: Histogram of music_name after its conversion to mono (1 channel only).

.....

Try to reduce the space between figure and caption

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.

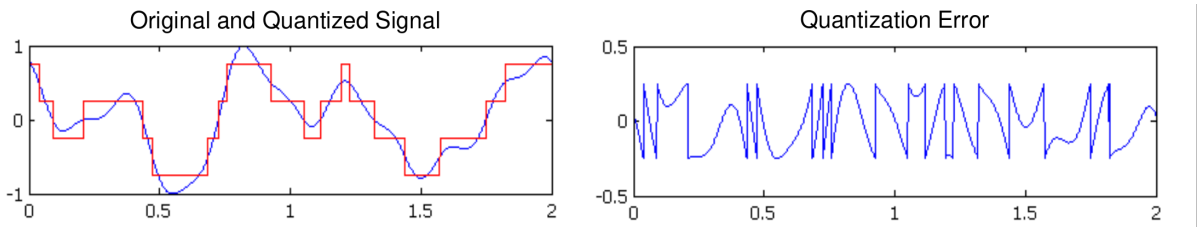

Figure 3: Example of a quantized waveform.

It is the `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 sample given as input (ideally, this value should be less than of the original sample); `reductFactor` defines the number of times the user would like to reduce the total number of values of the sample. This reduction factor works by calculating the average between `n` values, where `n = reductFactor`, and doing this for all values from the sample. The result is a sample with `n` times less values.

2.2. Error Calculation

...

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 (*5*). 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**.

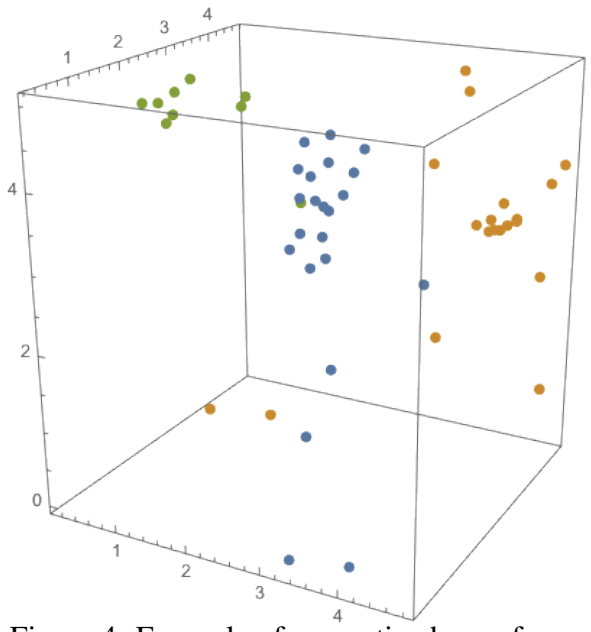

Figure 4: Example of a quantized waveform.

For our problem, we used the KMeans Clustering algorithm, probably the most well-known clustering algorithm (*5*). 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

...

3. Automatic Music Identification

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

3.1. Most Probable Music

...

3.2. Parameters Variation

...

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,...

Finally, 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].

5. 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].