uc3m | Universidad Carlos III de Madrid

Grado Universitario en Ingeniería de Sistemas Audiovisuales
2017-2018

Trabajo Fin de Grado

"Recommendation of songs through deep learning techniques"

Flavia García Vázquez

Tutor

Fernando de la Calle Silos

Leganés, 2018

ABSTRACT

The present project exposes the investigation and implementation of song recommender systems based on collaborative filtering (CF) and deep learning based techniques. These song recommender systems will automatically create personalized lists of songs depending on the tastes of each user. Recommender systems have become nowadays a very popular and important field of study in machine learning because of the evolution of music industry.

To develop the recommender systems, the Million Song Dataset will be used. This dataset will be analyzed thoroughly to conclude if it is valid for recommendation tasks. If these results are valid, a subset of this dataset will be taken to input the recommender system model. First, a Collaborative Filtering recommender will be developed, having as input the number of times each user has listened to a particular song (implicit feedback). This recommender will be trained, validated and tested to be aware of its performance. Consequently, an artist classifier having as a model a convolutional neural network (CNN) and as input a song audio signal will be developed. This is done in order to have a prepared neural network in order to implement deep content-based technique in future steps. The inputs of the CNN will be MFCC of the songs audio signals. Different procedures to extract the MFCC and will be done and compared based on the CNN results. Different CNN architectures will be studied as well.

Finally, an approach of a hybrid recommender system (called *novelty detection* in this project) will be made. This hybrid recommender system will combine collaborative filtering and deep learning based techniques. As a result, a system able to recommend popular and unpopular songs will be obtained (thanks to deep learning based technique).

Keywords: Recommender systems, collaborative filtering, convolutional neural network, content-based, MFCC (Mel Frequency Cepstrum Coefficients).

RESUMEN

El presente proyecto expone la investigación e implementación de un sistema de recomendación de canciones, basado en las técnicas de filtrado colaborativo y *deep learning*. Este sistema de recomendación de canciones creará de forma automática listas de canciones personalizadas en función de los gustos de cada usuario. Actualmente, los sistemas de recomendación son bastante famosos y se han convertido en un campo de estudio muy importante en aprendizaje automático, debido a la evolución de la industria de la música.

Para desarrollar el sistema de recomendación se ha utilizado el conjunto de datos *Million Song Dataset*. Este conjunto de datos será analizado minuciosamente para concluir en su es válido o no para desarrollar el recomendador. Si resulta ser válido, un subconjunto de datos de este conjunto de datos será la entrada del model del sistema de recomendación. Primero, un recomendador basado en filtrado colaborativo será desarrollado, teniendo como entrada el número de veces que cada usuario ha escuchado cierta canción (*feedback* implícito). Este recomendador será entrenado, validado y probado para ser conscientes de su funcionamiento. Posteriormente se desarollará un clasificador de artistas que tendrá como modelo una red convolucional, y como entrada la señal de audio de una canción. Esto se hará para tener una red neuronal preparada para implementar la técnica *deep content-based* en un futuro. Las entradas de la red convolucional serán los coeficientes de Mel (MFCC) de la señal de audio de las canciones. Se realizarán y comparán diferentes procedimientos para extraer estos coeficientes be done and compared based on the CNN results. También se estudiarán diferentes arquitecturas de la red convolucional.

Finalmente, se realizará un acercamiento a un sistema de recomendación (llamado en este proyecto: *novelty detection*). Este sistema de recomendación híbrido combinará las tecnicas de filtrado colaborativo y deep learning. Como resultado se tendrá un sistema capaz de recomendar canciones populares y no populares (gracias a la técnica *deep learning based*).

Palabras clave: Sistema de recomendación, filtrado colaborativo, redes neuronales convolucionales, *content-based*, MFCC (Mel Frequency Cepstrum Coefficients).

ACKNOWLEDGEMENTS

Thanks to all those who have helped me on this path. Especially my tutor for having accepted and helped me with a topic that motivated me, to my colleagues at Deloitte Digital for having solved my doubts and supported and, above all, my friends, my family and my boyfriend who have encouraged and stood me in this period.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The way of listening to music has changed. Streaming music platforms are nowadays the most accessible ways of listening to music. The reasons behind this change will be explained afterwards, but the fact of this platforms being vital to the music industry remains there, and it would not have been possible without the adequate technology.

The appeal of a streaming music platform builds over one simple concept: for it to be good enough, it needs to have a really wide catalog. More than "really wide"; it needs to be huge. The success of this programs depends on the user feeling that almost every song or music composition that he or she will want to listen, is going to be there for him/her to find quickly and easily.

This leads to a major milestone to overcome: people like to consume products that feel designed for them. In the same way that physical stores are designed in order to target certain types of clients, web pages and computer programs need to appeal their consumers. This personalization seems difficult when combined with the fact that every music genre needs to be there for its success.

Nevertheless, the industry has realized this and addressed the problem upfront. And this has led to a solution heavily dependant on technology: recommender systems. A streaming music platform heavily increases its value just by offering each user personalized music recommendations. In an "ocean" of music, anyone will be more likely to be comfortable with the program if the by default offering fits his or her taste. If the recommendations are accurate, not only will a given user consume more from a certain platform, but also, he/she will be more likely to prefer the concrete system over the rest of them.

The proposal of this project is, then, to develop a **music recommender system** that creates personalized content depending on the tastes of each user. This will be done from scratch, using as input a real dataset with a lot of different characteristics of songs, and also with users' behaviour information. The proposed recommender is based in the combination of two types of recommendation techniques: collaborative filtering and deep learning content based recommender systems.

1.1. Motivation

Recommender systems should be capable of understanding perfectly each user and each item (in this case, songs). This could divide the problem in two: analysis of each song, and analysis of each user.

Regarding the problem of the song analysis, several questions could be asked: 'are there any specific melody, harmony, rhythm or combination among them that produce the maximum pleasure to its consumer? 'Which are the variables that make enjoyable a music

composition? ‘Can they be predicted? ‘Is it possible to categorize a composition into a certain genre by analysing the sound it produces? Some time ago it would not have seen plausible, but over the last years computational capacity has dramatically increased. This has led to the implementation of complex models such as deep neural networks, leading to possible and affordable classification of music by the analysis of music itself.

Despite this classification being useful, it is far from being the complete solution to the mentioned problem. When talking about recommender systems, physical variables within music are far from being the only variables that produce satisfaction to a consumer. As almost everything else, listening to music is usually a social activity. People tend to enjoy what their relatives do, what the publicity invests on, what trends at the moment… Whatever the sociological motive, musical taste depends as highly on the customer than on the song or musical composition. The solution to this problem, although maybe seemed as ‘intuitive’, is pretty resourceful.

People tend to enjoy similar music genres and styles. And music genres and styles are liked by different people. The best recommendations would come, then, from those who listen to similar music to the one that the user enjoys. The thing is not to depend on actual recommendations done personally by these people with similar tastes, but fortunately this is not needed. As every reproduction is stored in current streaming platforms, they have begun to use this information to obtain the best “candidates” of being liked to each user by using a technology called collaborative filtering. This allows the program to automatically give high quality recommendations just basing on what similar users play.

The idea is, as stated, to obtain high quality automatic recommendations by using information that was recorded by service companies long before it was used for this purpose. In this particular problem, it will be applied to music, but the tools used for this task may be applied without much effort to other kind of products.

1.2. Project objectives

The objective of this project is to address the problem of music recommendation in real datasets, generating recommendation playlists personalized for each user. Two main research lines are followed. On one hand the use of traditional collaborative filtering techniques and in the other the combination of these techniques with novel deep learning methods.

Specifically, the main approaches of this project are:

- To achieve interesting insights into the data set, with real data obtained from a music streaming platform. Exploring the data set it will be possible to discover errors in it or maybe features that will be important to take into account when developing the recommender system.
- To take important and correct decisions about data separation that will greatly impact the result of the recommender system.

- To get an accurate recommender system just based on collaborative filtering techniques, using the correct metrics for its evaluation.
- If a good recommender system is not achieved, to have the knowledge to understand perfectly why is it not making accurate recommendations and how it would be possible to improve it.
- To extract meaningful features from the audio signal of songs and construct a deep learning architecture suitable for detecting the artist of these songs. To evaluate this classifier system is also very important to use the most suitable metrics.
- To be based in the network architecture of the classifier to develop the deep content based recommender. With this done, to combine successfully both complex techniques (collaborative filtering and deep learning) in order to recommend also new songs that have never been listened.

1.3. Socio-economic environment

Before XXI century the music industry was led by the record companies. This companies owned the music and their income model was based on the sale of records (CDs). They had a large amount of income and it was not predicted that it would decrease over the years.

The decline of this industry began in 1995 with the creation of the MP3 format [1] (MPEG-1 Audio Layer III or MPEG-2 Audio Layer III), developed mainly by Karlheinz Brandenburg. This format allowed to reduce the size of the original file between 12 and 15 times, depending on the bitrate used, and there was hardly any sound quality loss compared to CDs. Due to these characteristics, the MP3 was accepted by the public as a compressed audio standard. This new format facilitated the possibility of piracy.

In 1997 pirate exchanges began with whole albums instead of single songs, and two years later, in 1999 piracy started as is currently known thanks to the creation of the Napster [2] platform, a Peer to Peer application, by Sean Parker and Shawn Fanning. Soon after, many other Peer to Peer programs appeared and ended up being used by widely, producing that in 2001 the sales of the music industry dropped by 5.1%.

Fighting against piracy, at the beginning of the century, Apple came with his Ipod and with it the possibility of buying a song by one euro instead of buying the whole physical album. The rules of the game changed. In that moment, the people that bought just a song were a blessing for the record companies, but this hardly happened, the sale of the music barely reported income anymore.

Spotify [3] was born at the end of 2008, and its great success implied that people was willing to pay for music again. Spotify emerged as a platform where you could listen to streaming music without having to download it and, consequently, without owning it, which was something revolutionary. Spotify gives its users two options, premium account and free account, in exchange for advertising. This platform has managed to reduce a very

high percentage of piracy and is currently the world leader in streaming music with more than 140 million customers, being half of them premium subscribers.

In 2016, thanks to Spotify, the music industry grew 6%, the largest increase since 2009, and its own business increased its revenues by 60%.

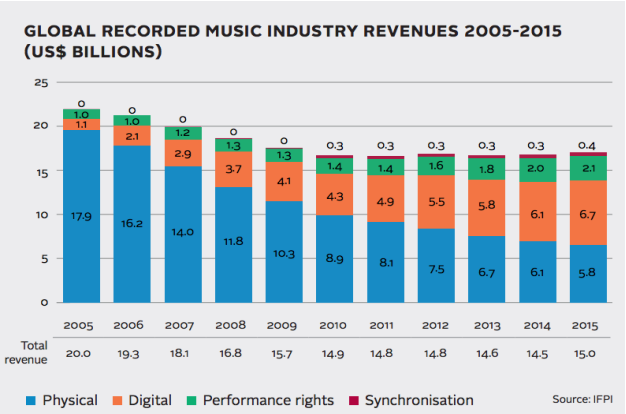


Figure 1.1. Evolution of music digitalization [3]

Due to the high success of streaming music, other digital multinationals have started to build their own platforms (Apple, Google and Amazon). Spotify faces the challenge of competing against them (*Figure 2*) in a business where it is difficult to be profitable (*Figure 3*).

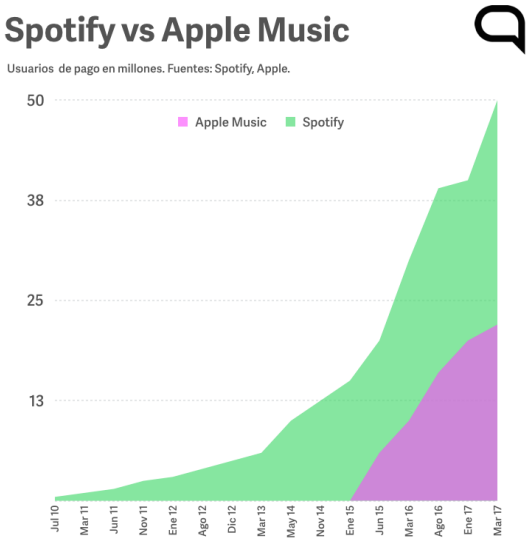


Figure 1.2. Number of Spotify premium users vs Apples' [5]

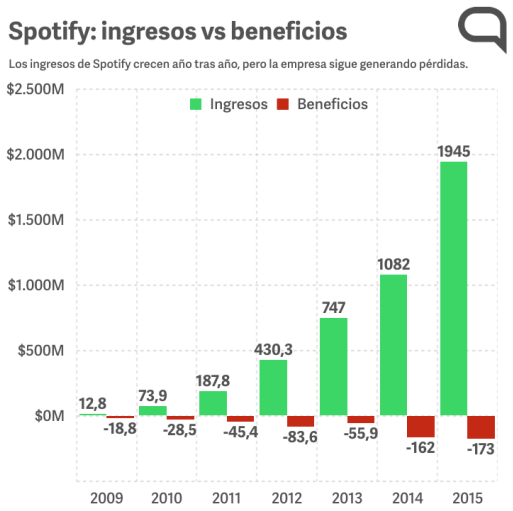


Figure 1.3. Income versus benefits [5]

With the increase in the use of e-commerce, companies see an opportunity of getting more money and making its platform to have more value for users. They realised that having such a wide catalog it was necessary to make the user feels unique. Because of this idea**, recommender systems** appeared. From the technical point of view, a recommender system is a machine learning algorithm that utilizes Big Data to help users discover items they may like. These recommendations are based on past purchases, product ratings, demographic information, or their search history. Providing good suggestions (movies, products, news...) help to increase sales, encourages the user to stay engaged and create brand loyalty through relevant personalization. Recommender systems have made a really big impact in companies [6] such as Amazon (35% of its revenue is generated by its recommendation engine) and Netflix (75% of what consumers watch comes from the company's recommender system).

The biggest streaming music platform has not been left behind in this aspect. **Spotify** has implemented recommender systems by creating personalized recommendations playlist**.** Their most innovative uses of Artificial Intelligence (AI) and recommendation systems is their popular **Discover Weekly playlist**, also known as Release Radar. This algorithmically powered tool updates personal playlists every week so that users won't miss newly released music by artists they like or will like. Spotify's recommendation playlists have increased its number of monthly users from 75 million to 100 million at a time, in spite of competition from rival streaming service Apple Music (*Figure 2*).

Due to the socioeconomic environment explained, it is necessary to investigate and implement novel recommender solutions. The more these models are improved, the more engagement will be possible to have between the company and the user.

1.4. Outline

The material presented in this project is organized as follows:

- *Chapter 1* gives an introduction of the project and describes the socio-economic environment to understand why recommender systems appear and why are they so important nowadays.
- *Chapter 2* presents the state-of-art in recommender systems and explains the main types of recommender systems.
- *Chapter 3* explains what is a deep neural network and how are they trained. Also, convolutional neural networks are explained. This type of neural networks are the ones used to implement deep content based technique.
- *Chapter 4* depicts and analyses the different datasets employed in this project.
- *Chapter 5* explains all the theory needed to implement and evaluate collaborative filtering technique. There is also a part explaining all the experiments done to improve the recommender, with their results.
- *Chapter 6* describes the application of novel deep learning architectures to classify the artist of a given audio of a song (30 seconds).
- *Chapter 7* is where the incorporation of both recommender techniques (collaborative filtering and deep content based) is explained.
- *Chapter 8* draws some conclusions and further lines of research.
- Finally, in *Chapter 9* the budget of the project is presented

2. RECOMMENDER SYSTEMS

In this chapter, the main types of recommender systems and their state-of-art will be explained.

2.1. Introduction

It is possible to recommend items based on different aspects. This differents aspects could be: demographic information, items popularity information, user metadata information (age, sex…), user behaviour information, item content… In fact, recommender systems have been around for a long time, but these gave very obvious results (for example if you were looking shoes, the platform recommended more shoes). The current recommender systems have become more complex, providing surprisingly good results. The state-of-art technologies regarding this topic are the **deep learning based recommender systems**. These consist on developing a DNN to make recommendations. Due to technological advances, it is already possible to carry out this type of recommender system. This type of recommender will be explained more deeply in the next chapter.

2.2. Types of recommender systems used today

In this section, the different types of recommender systems used today by enterprises with an e-commerce platform will be presented. These recommenders will be explained and their advantages and disadvantages mentioned. The most used types of recommender systems are: *content-based*, *collaborative filtering*, *hybrid* and *deep learning based*.

2.2.1. Content-based recommender

Content-based recommender [7] [8] has the item as the basis of its prediction instead of having the user.

It predicts what users like based on what they have liked in the past. The type of data that the recommender will use, could be:

- Implicit data: clicks, visualizations, purchases…
- Explicit data: ratings that the user has gave to each item

These systems are very focused on textual sources, since these can be analyzed using classical techniques of Data Mining and Natural Language Processing, which allow to extract user profiles and items in a simple way.

Advantages

- Unlike the Collaborative Filtering (CF), if the items have enough information in its description, the "cold-start" problem is avoided. This problem consists on the lack of possibility of making recommendations for new items and users because of not having data about them.
- It is easier to explain and understand the recommendations done (the item content itself serves as an explanation).
- The representations of the content are varied, therefore different techniques of text processing can be used.

Disadvantages

- Over-specialization problem: the system recommends overmuch similar to those consumed.
- It has the so called "new-user problem". This problem would occur when a new user needs a recommendation, but it can not be done because information of the items that the user likes is needed.

2.2.2. Collaborative filtering recommender

Collaborative filtering recommender [8] predicts what a particular user likes based on what other similar users like. Therefore, this system first identifies similar users and then recommends items liked by them to the initial user.

CF is able to detect similar users (user-user CF), similar items (item-item CF), and items that a particular user would like.

This type orecommender is implemented and explained deeply in chapter 5.

Advantages

- It extracts data characteristics that are difficult to achieve with content-based techniques.
- Empirical experiments have demonstrated that CF recommenders make more accurate recommendations that content-based recommenders.

Disadvantages

- These recommenders suffer from the known "cold-start" problems.

2.2.3. Hybrid recommender system

A hybrid recommender system is any one that combines different techniques to have their advantages and eliminate their weaknesses.

The most famous hybrid recommender system[7] is the one that combines collaborative filtering technique and content-based technique. This recommender takes the advantages of the content of the items as well as the use of the similarities among users. Several empirical studies have demonstrated that hybrid recommender give more accurate recommendations that the techniques individually.

There are several ways to combine the CF and content-based techniques: to make content-based and collaborative-based predictions separately and then combine them or to add content-based capabilities to a collaborative-based approach.

This is the recommender system most appreciated by companies. As matter of fact, Netflix has implemented this type of RS [9]. Netflix website makes recommendations by comparing the watching and searching habits of similar users (CF) as getting films which share characteristics with films that users have rated high (content-based).

Advantages

- It does not have the "cold start" problem and the sparsity problem (users have consumed only a few items of the entire catalog).
- It makes more accurate recommendations that the techniques that combined, individually.

Disadvantages

- It can not make recommendations to users who do not have registered data.

2.2.4. Deep learning based recommender systems

Deep learning based recommender systems [10] are the state-of-art in recommender systems. This type of RS is based on implementing deep learning in order to make recommendations. During the past few years deep learning has shown amazing results in many fields. They are the order of the day and RS have not stayed behind on this topic.

To enhance the power of recommendations the following deep learning models have been implemented:

- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks (RNNs)
- Deep Neural Networks (DNNs)
- Auto Encoders (AE)

Advantages

- It does not have the "cold start" problem.
- It makes possible to make recommendations based on the content of digital images or audio signals.

Disadvantages

- It needs a big amount of data to make accurate recommendations.
- Because of the great complexity and little transparency of the model, if the recommendations are not accurate, it is quite difficult to discover the reason.

Due to these new researches, hybrid recommender systems that combine CF and deep learning have been developed (deep explanation in *chapter 7*). A good example of this type of recommender is the one developed by Spotify. This system uses CNNs to extract audio features from music tracks. It is subsequently combined with CF. A CNN that extracts audio features has been developed in *chapter 6*.

3. OVERVIEW OF DEEP NEURAL NETWORKS

The first machine learning algorithms perform well when solving linear problems. Nevertheless, real life problems were very likely to be non-linear. For example, in a classification problem, as depicted in *Figure 3.1* a linear function can not give an accurate solution. On of the methods that can fit non-linear boundaries to data are **neural networks.**

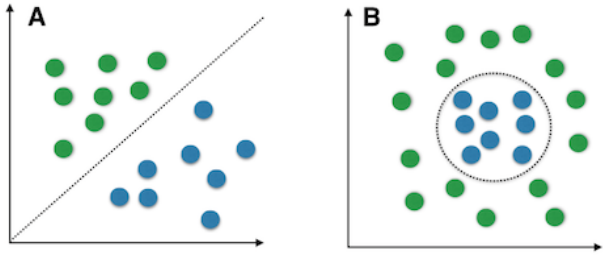


Figure 3.1. Linear vs. nonlinear problems [11]

3.1. First Artificial Neural Networks (ANN) architectures

The development of Artificial Neural Networks began with the invention of the ***Perceptron*** in **1958** by **Frank Rosenblatt** [12]. For this creation, he was inspired by brain architecture *(Figure 3.2)* due to the fact that it is the most intelligent "machine" known.

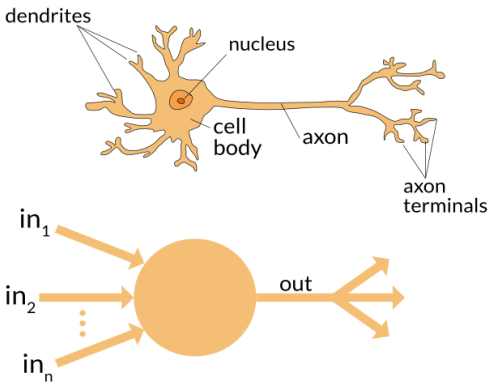


Figure 3.2. Perceptron biological analogy [13]

The *Perceptron* is the simplest ANN architecture *(Figure 3.3)*. It is composed of one artificial neuron called ***Linear Threshold Unit*** (**LTU**). This artificial neuron has multiple input connections (numbers with an associated weight) and one output connection (number). LTU computes a weighted sum of its inputs *(3.1)* like a linear regression, and

then applies a *step function* to that sum. This step function outputs a probability (number between 0 and 1). The step function, also called activation function, is the element that applies the nonlinearity to the algorithm.

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = \overline{w^T} * \bar{x} \quad (3.1)$$

Where: $x_1, x_2, \ldots, x_n \rightarrow$ LTU inputs, $w_1, w_2, \ldots, w_n \rightarrow$ each input associated weight.

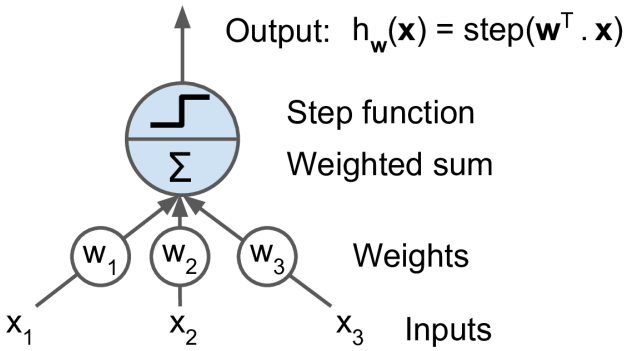


Figure 3.3. Linear threshold unit diagram [14]

The basic idea of training an LTU consists in finding the right values of the different weights.

A **binary classifier** can be build with one single LTU and with a fix threshold at the output. For achieving a classification with more than two classes (**multi output classifier**) a *Perceptron* with more than one LTU is necessary *(Figure 3.4)*.

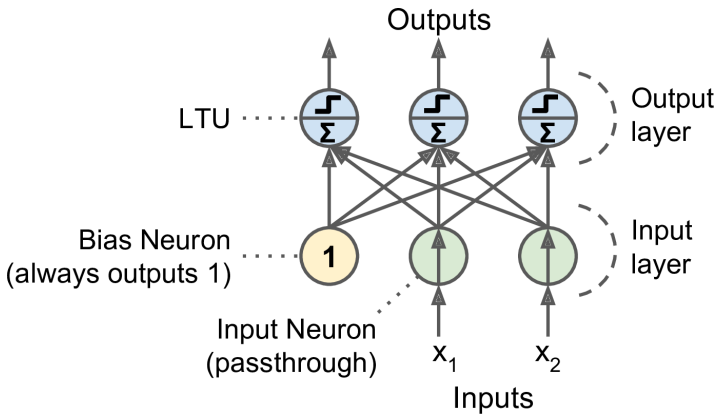


Figure 3.4. Perceptron diagram [14]

When the new perceptron architecture was proposed *(Figure 3.4)*, the inputs were represented by the input neurons, and a bias neuron was included. The bias neuron always has value one, so the weighted sum operation of each LTU in this perceptron architecture is: (3.2)

$$z = 1 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = 1 + \overline{w^T} * \bar{x}$$

Where: $x_1, x_2, \ldots, x_n \rightarrow$ perceptron inputs, $w_1, w_2, \ldots, w_n \rightarrow$ each input associated weight.

Eventually *Perceptrons* were demonstrated that they had a lot of limitations [15]. It was discovered that these limitations could be eliminated by stacking multiple perceptrons (more than one LTU layer). This new architecture was called **Multi-Layer Perceptron (MLP)**.

3.2. Multi-Layer Perceptron (MLP)

An MLP *(Figure 3.5)* is composed by:

- **Input layer**: input neurons and the bias neuron
- **Hidden layers**: LTU and a bias neuron in each layer
- **Output layer**: last LTU layer (no bias neuron)

MLP with more than one hidden layer, are the so called **Deep Neural Networks (DNN)** [16]. DNN are widely used in industry nowadays due to two main factors: high computational capacity able to train DNN with many hidden layers and huge amounts of data to train it.

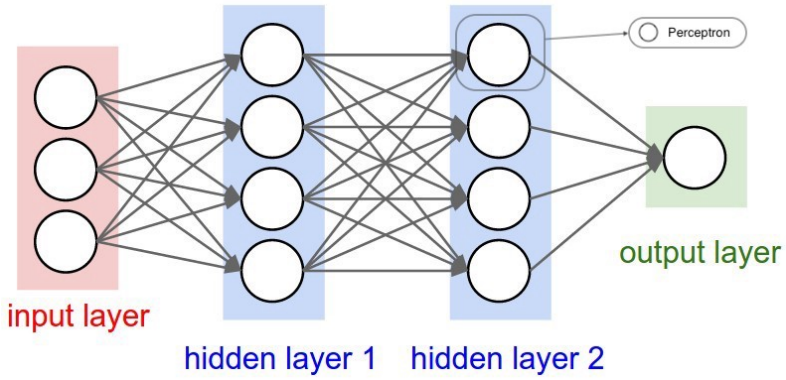


Figure 3.5. Multi-Layer Perceptron [17]

The number of neurons in each layer and the number of hidden layers, is something that has to be chosen by the developer and will vary depending on the problem (this

hyperparameters will need to be tuned). Nowadays there is not any rule or algorithm for determining this numbers. Empirical studies have demonstrated that DNN works better with number of neurons that are powers of two.

3.2.1. Training MLP: Backpropagation

DNN have a lot of connections and with it a lot of weights that need to learn about data.

Developing a good algorithm for training MLP was not an easy task. A lot of researchers tried to do it with unsuccessful results, but finally in **1986, D.E.Rumelhart** introduced **backpropagation training algorithm** [18] that worked for MLP. Backpropagation is still used for training DNN.

How does backpropagation work?

Backpropagation goal is minimizing the output error of the network (difference between the expected output and the actual output of the network).

Each training instance will be fed to the network. Each neuron will output a probability value that will be fed to the next layer. This will be done, one time and another, through all the layers of the network, until the output layer is reached. This process is called **forward propagation.**

Backpropagation measures the output error of the network and it goes backwards (starting at the output layer) through all the different paths, calculating how much that connection contributes to the error of that path. This backward pass will stop when an input layer is reached.

In order to obtain how much that connection contribute to the error, the gradient has to be computed *(∆f)*. A NN is a massive composite function, so it will be possible to compute the gradient of each path applying the **Chain rule** *(3.3)*.

$$\frac{d}{dx}[f(g(x)] = \ f'(g(x)) \ * \ g'(x) \tag{3.3}$$

Where: $f(x), g(x) \rightarrow$ any function.

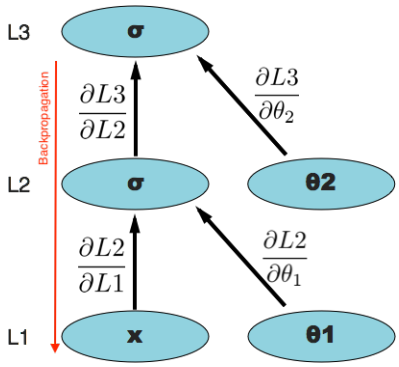


Figure 3.6. Chain rule diagram [19]

At *Figure 3.6,* L3 is the output layer, L2 the hidden layer and L1 the input layer. To get how much $x \rightarrow \boldsymbol{\sigma} \rightarrow \boldsymbol{\sigma}$ network path contributes to the output error, the following operation has to be computed:

$$\frac{\partial L3}{\partial L1} = \frac{\partial L3}{\partial L2} * \frac{\partial L2}{\partial L1} \tag{3.4}$$

Where: $L1, L2, L3 \rightarrow$ each layer of the MLP.

Last step of backpropagation is updating the weights of the connections in order to reduce the error. This is done by **Gradient Descent** [20] (GD).

Gradient Descent is a very old and generic optimization algorithm. It is based on updating *w* value for minimizing the loss function *(3.5)*. GD computes the derivative and updates *w* until the derivative at that *w* value reach its minimum (equal to zero) *(Figure 3.7)*.

A hyperparameter called *learning rate* has to be setted. The learning rate is the size of *w* update step. If this hyperparameter value is low, the algorithm will take a lot of time to converge. On the other hand, if it is high, the algorithm will never reach the global minimum (never converge). At *Figure 3.8* there is a very understandable representation of how is the loss related with the number of epochs, depending on the *learning rate* value.

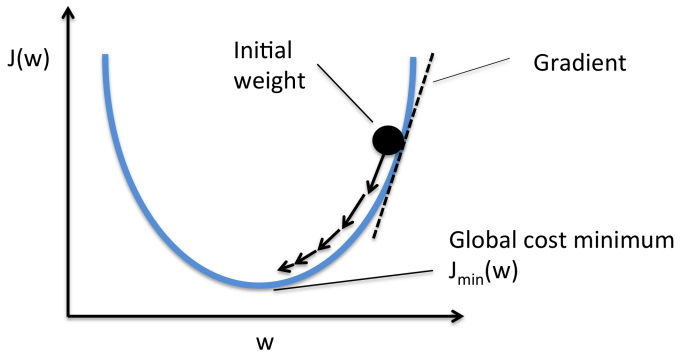


Figure 3.7. Gradient descent procedure [21]

$$w \leftarrow w - \eta \frac{\partial J(w)}{\partial w} \tag{3.5}$$

Where: $w \rightarrow$ weight value that will be updated, $J(w) \rightarrow$ loss function, $\eta \rightarrow$ learning rate (how big is the update step of w).

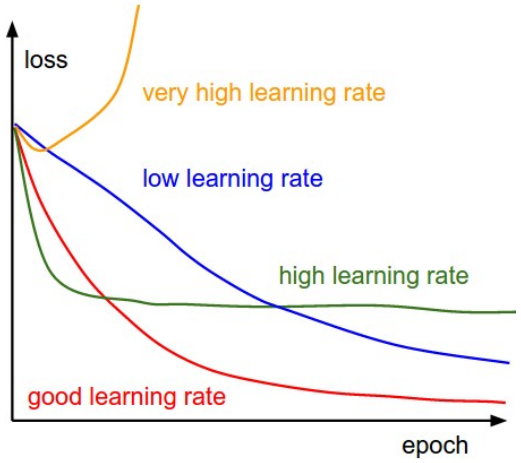
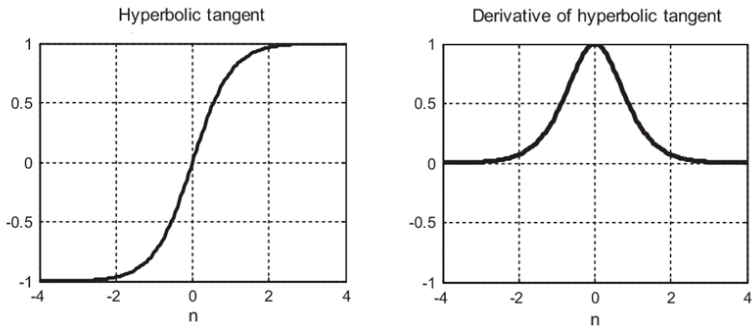


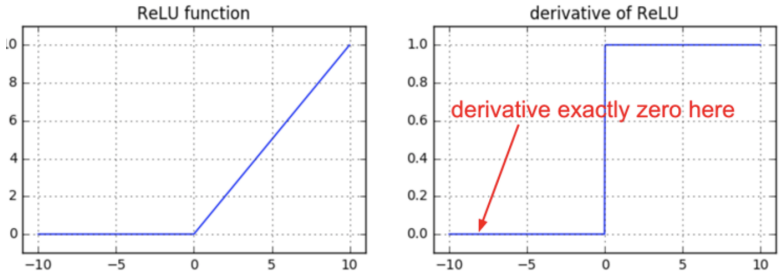
Figure 3.8. Epoch - loss graph depending on the learning rate hyperparameter value [22]

GD only works if the loss function is convex. For this reason, the developers of backpropagation made an important change to MLP's architecture. They changed step function (generates a flat loss function) to other possible activation functions: ***hyperbolic tangent function*** *(Figure 3.9)*, ***ReLU function*** *(Figure 3.10)*, ***sigmoid function*** *(Figure 3.11)*. The derivatives of this activation functions have a clear minimum so they are better for applying gradient descent.



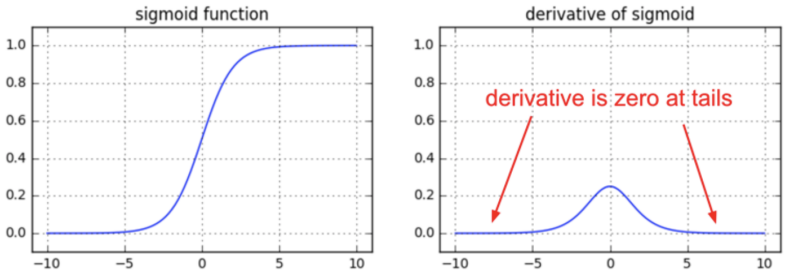
$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Figure 3.9. Hyperbolic tangent function and its derivative [23]



$$R(z) = max(0, z)$$

Figure 3.10. ReLU function and its derivative [24]



$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Figure 3.11. Sigmoid function and its derivative [24]

Depending on the activation function selected, the performance of the network will vary. The most used activation function is the ReLU function. The reasons for this selection, with respect to the other activation functions, are:

- It is faster to compute.
- GD converge faster (does not get stack in plateaus).
- Non-zero and constant derivative from positive input values, therefore there are less chances of occurring vanishing gradient. Vanishing gradient problem [25] consists in the weights do not change their value because its gradient is vanishingly small. This problem makes training take too long and accuracy suffer. Due to have constant derivative also faster learning occurs.
- Introduces sparsity because of having so many values of X derivative equal zero. Therefore, the resulting representation is sparse. Sparse representations seem to be more beneficial than dense representations (representations generated by sigmoids).

3.2.2. Applying regularization

Regularization is used to **prevent overfitting**. Overfitting is the effect of training too much the model. When a model is overfitted, it performs very good with training data, but it not generalizes well (high accuracy classifying training data, but very low classifying test data). Overfitting happens when the model is too complex or when there is not enough training data.

There exist different regularization techniques: L1 regularization, L2 regularization and dropout.

DNN are very prone to be overfitted due to the high complexity of the model. The regularization technique used for preventing overfitting at DNN is **dropout**.

Dropout was introduced in **2012** by **G.E.Hinton** [26] . His work was more detailed in a paper written by **Nitish Srivastava** [27] et al. in **2014**.

It has been proven that dropout gives very good results. State-of-the-art neural networks use dropout to achieve a not overfitted network.

The main idea of dropout is, while training, turning neurons on and off, randomly, at every network layer *(Figure 3.12)*. Each neuron will have a probability p of being temporarily turned off. p is an hyperparameter of the model called *dropout rate* (which is typically set to 0.5).

By applying dropout, the network is forced to learn new pathways that the data will flow through. This technique increases the generalization capacity of the network.

Dropout is not applied when testing.

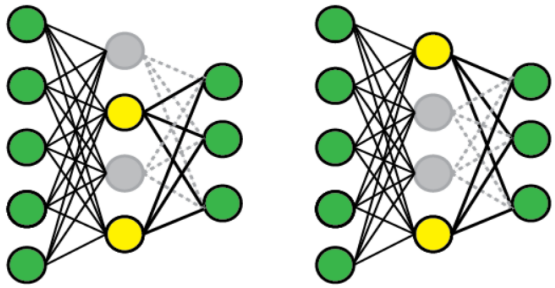


Figure 3.12. Dropout regularization [28]

3.3. Convolutional Neural Networks (CNN)

Convolutional Neural Networks are a type of NN developed to be applied to images. CNN emerged at the 1980s from the study of the brain's visual cortex. In 1968 a paper [29] was published explaining how mammals perceive the world. It explains that the way of seeing is hierarchical. Brain neurons are distributed in clusters. Each cluster is in charge of detecting a different set of features in the image. Each set of features will be more and more abstract (hierarchical structure). First, points will be detected, then lines, then curves, then more complex curves, etc. *(Figure 3.13)*.

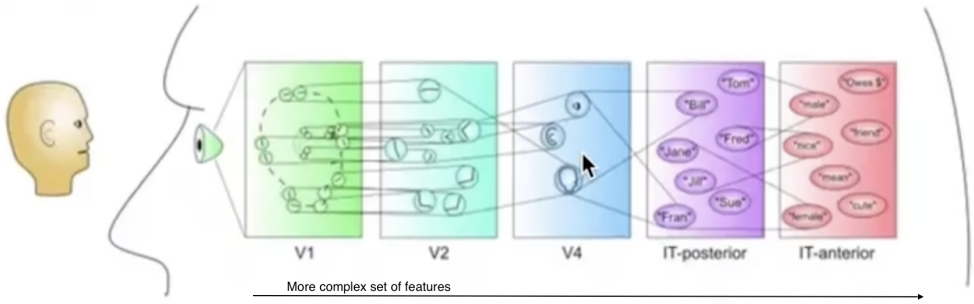


Figure 3.13. Brain's visual cortex diagram [28]

A lot of engineers were inspired by these discoveries, trying to use this knowledge to achieve visual computation.

In **1998,** it was published a paper by **Yann LeCun et al.** [30] that explains the procedure for recognising handwritten numbers. In this paper *convolutional layers*, *pooling layers* and the famous LeNet-5 architecture (*Figure 3.14)* were introduced.

Thanks to the increase of computational power and the huge amount of data available, now it is possible to implement CNN, having impressive results.

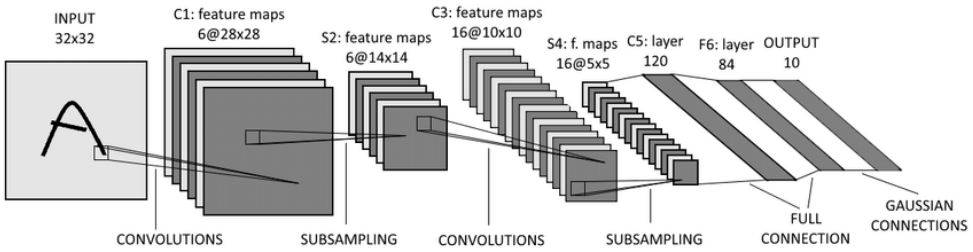


Figure 3.14. LeNet-5 architecture. Figure obtained from [31]

How does CNN works?

The input of a CNN is an image. A digital image is a three-dimensional matrix of pixeles (weight * height * depth). The depth of an image represents the channels. A color image has three channels (RGB: red, green and blue) while a grayscale image only has one channel (bidimensional matrix as input).

Input is proceeded by different layers that could be divided in two parts: *feature learning* and *classification* (*Figure 3.15*).

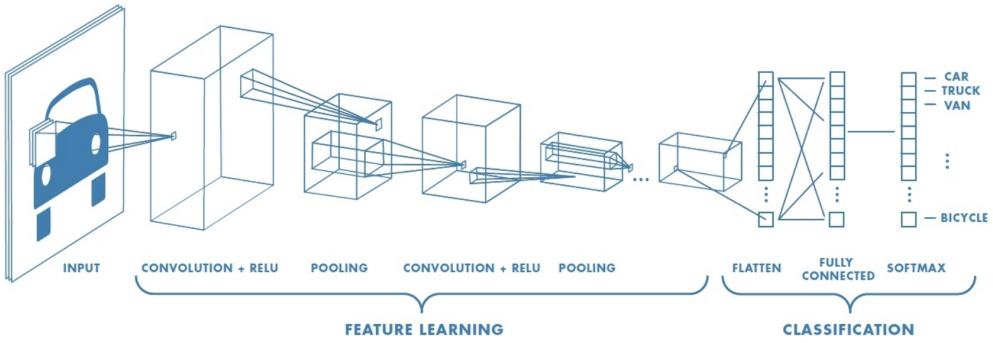


Figure 3.15. CNN architecture [28]

The CNN architecture of the *feature learning*, extract relevant features of the images in a low dimensional vector. This is achieved by doing three subsequent operations, repeated over and over again: **convolution + ReLU (or other activation function) + pooling**.

Convolutional layer computes the convolution of the whole matrix (*Figure 3.16*). The convolution computes the dot product between an input matrix and another matrix called *kernel* or *filter* (matrix K at *Figure 3.16)*. The values of this *filters* are learned through backpropagation.

Computing the convolution implies mixing different data (in *Figure 3.16* the green square has mixed information of 9 different data (red square)). *Kernel matrix size* and *stride size* (how many units in the matrix the mask moves from one convolution to another) have to

be setted in order to compute it. The resulting convolved matrix is called *feature maps* (matrix *I*K* at *Figure 3.16*).

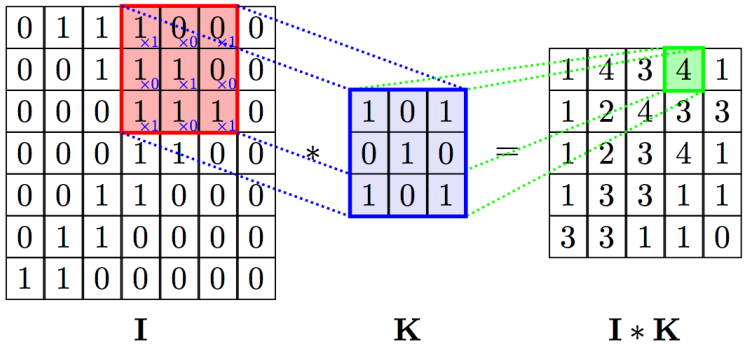


Figure 3.16. Convolution matrix operation [32]

The ***pooling layer*** have as input a matrix. Its main task is to reduce the input matrix dimensions. By doing this, pooling reduces the computational complexity of the model. There exist two types of pooling: average pooling and max pooling *(Figure 3.17)*. For CNN, the most used type of pooling is the **max pooling**. Max pooling consists on keeping the maximum value of the windowed matrix.

In order to compute pooling it is necessary to fix a *window size* and a *stride size*.

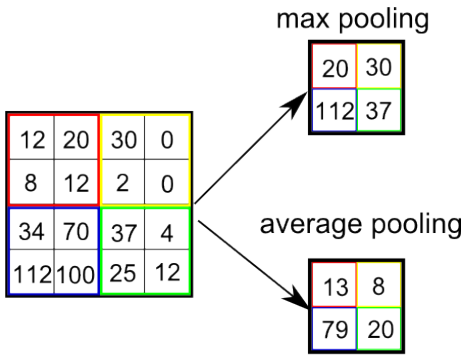


Figure 3.17. Max pooling and average pooling with *window size* = 2x2 and *stride size* = 2 [33]

When *feature learning* part has finished, *classification* part is achieved *(Figure 3.15)*. In this part, the following processes are done:

1. Flatten the matrix in a vector.
2. A NN is constructed: all the vector components (neurons) are connected to a ***fully connected layer***. This means that all the neurons of the first layer are connected to all of the next layer. This *fully connected layer* is applied to mix all the acquired

learning. There could be more than one *fully connected layer* (hidden layers). The final layer has to have the same number of neurons as classes has the classification problem. Dropout (*chapter 3.2.2*) will be always applied between all this NN layers.

3. A **softmax function** is applied to all the neuron values of the previous layer (NN output layer) in order to obtain a probability value. The highest probability value will correspond to the class that the network will give as classification result: $argmax(softmax\_outputs)$.

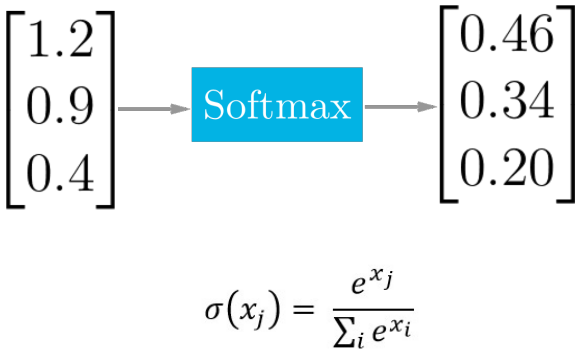


Figure 3.18. Softmax function [34]

Why CNN for music?

In this project CNN will be used for classifying music because it is possible to represent a song as an image.

The graphical representation of an audio signal is called **spectrogram**. A spectrogram represents significant information of the audio signal in the frequency domain (y axis) and time domain (x axis). Each pixel of the spectrogram represents the energy of a particular frequency component at a particular time.

It is possible to realize that the spectrogram contains a lot of information of a particular song, and each song will have a unique spectrogram, but bearing in mind that similar songs could have similar spectrograms. Depending on the rhythm, tonal relationships, instruments and voices, among other features of a song, its spectrogram will vary. For example, as explain Arijit Ghosal et al. in their paper [35], the spectrogram image of an instrumental signal (without voice) shows stable frequency peaks persisting over time. Due the presence of voice, such stability is not observed.

In *Figure 3.19* the wide differences between the spectrograms of different music genres can be observed.

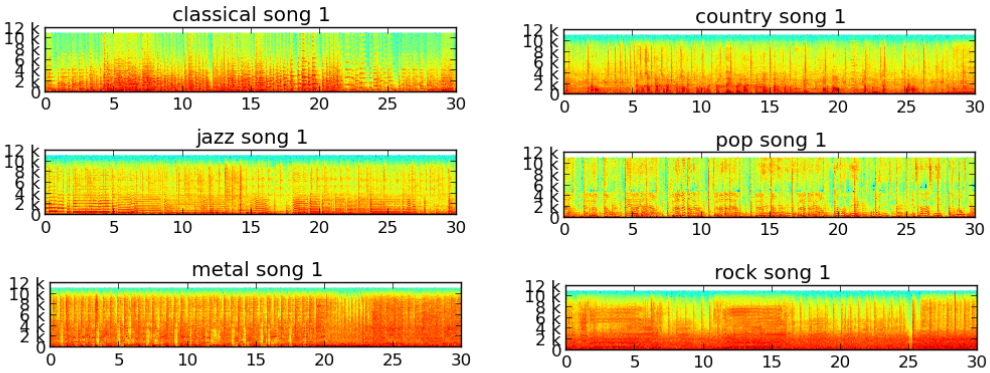


Figure 3.19. Spectrograms of different songs [36]

The spectrograms of each song will be the inputs of the network developed in this project. With more detail, this spectrogram will be composed by the coefficients of the mel frequency cepstrum (MFC). MFC encodes the power spectrum of a song by calculating the Fourier transform of the logarithm of the signal's spectrum. A deep explanation of this coefficients can be found in *chapter 6.1*.

4. DATASET DEVELOPMENT

All the data that have been used to develop the recommender system come from the **Million Song Dataset** [37] **(MSD).** It is a freely-available dataset with a collection of audio features and metadata. This dataset is provided by The Echo Nest with the collaboration of LabROSA. The Echo Nest is a platform for developers created to perform identification, recommendation, playlist creation, etc. Currently it is property of Spotify.

Mainly two well differentiated groups of datasets have been used. The datasets used to implement collaborative filtering technique have information about users behaviour, such as number of times each user has played a particular song. The datasets used to implement content based technique contain many information about each song of the full dataset, and also mp3 files with 30 seconds of each one of them.

4.1. Dataset for the traditional recommender system

In order to develop the CF recommender two different subsets from Million Song Dataset have been used: *All track Echo Nest ID* and *Triplets dataset* from *Taste Profile Dataset.*

4.1.1. All track Echo Nest ID

This dataset counts with one million of registers (all the songs of Million Song Dataset) and the following columns:

- *track ID*
- *song ID*
- *artist name*
- *song title*

It is used to translate *song ID*, at the moment of displaying the recommendations to the user, into something understandable for him.

In this dataset is possible to see that there are *artist name* - *song title* pairs that are repeated in the dataset. These repeated records have different *track ID* and *song ID*.

Having repeated songs pauperize recommender systems, because they would recommend songs that actually are the same that songs that the user has enjoyed. This makes the recommender inefficient.

Duplicated songs may be found because of covers, lives, etc. It is very important to delete this duplicates from the dataset. The distribution of the number of duplicates (greater than one) each artist-song pair has, is shown in *Figure 4.1*.

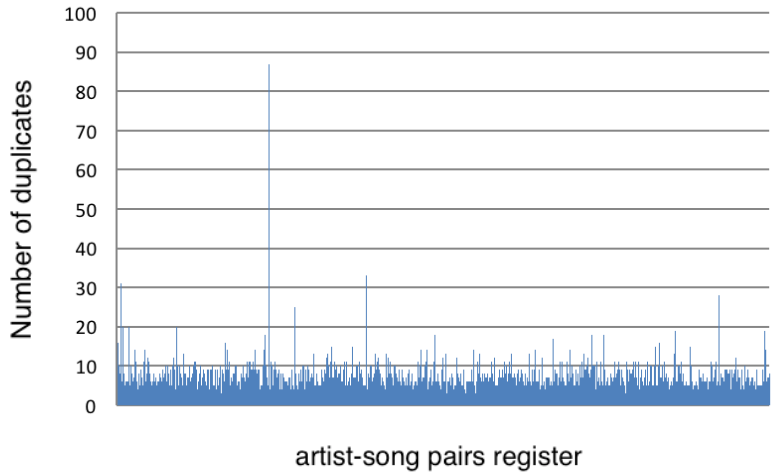


Figure 4.1. Column chart with the number of duplicates each song-artist pair has.

The maximum value of the chart in *Figure 4.1* is 87. This is a really big number of artist-song register. This case is *Der Blutchar* - *Untitled*.

Because of this song not having a name, this 87 could be the same song or not, for this reason, the songs of the dataset that have as song name "Untitled", will be deleted. This occurs in 897 songs (0.09% of the whole dataset).

The new subset with known song names still has duplicates (*Figure 4.2*). Duplicates values are still high, with a maximum of 33, but less than before (*Figure 4.1*). It is necessary to get out this duplicates from the new subset having as a result a dataset with **925,304 different songs**.

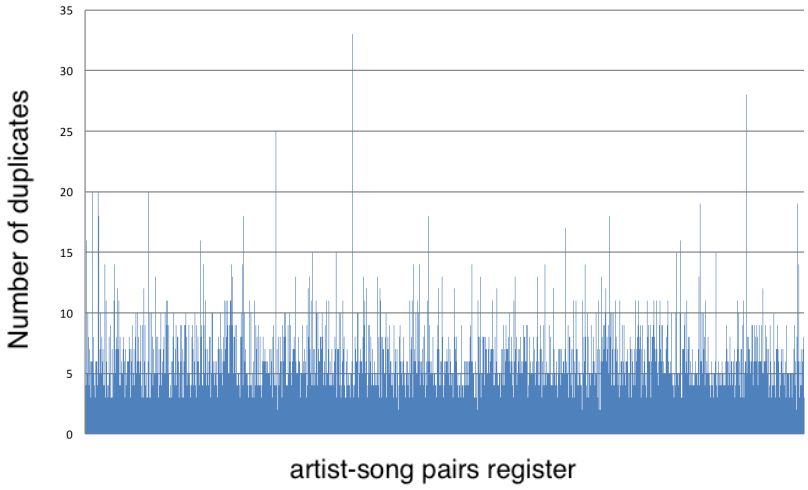


Figure 4.2. Column chart with the number of duplicates each song-artist pair has in the new dataset

4.1.2. Triplets dataset from Taste Profile Dataset

Taste Profile Dataset is the official user dataset of the Million Song Dataset.

Triplets dataset gives information of the times a user has played a particular song. It has **48,373,586 registers** and the following columns:

- *user ID*
- *song ID*
- *play count*

This dataset, a priori, gives information of 384,546 different songs listened by 1,019,318 different users. It is possible to realise that this is too much data, so the model input data will be a subset of this dataset. Before doing this it would be advisable to analyze a little bit more this dataset, it will be checked if there are repeated registers by user and song or if there is any wrong *play count* value (negative value).

Grouping the registers of the table by *song ID* and *user ID*, a table with the same number of rows as before results. This means that there are not user-song repeated registers.

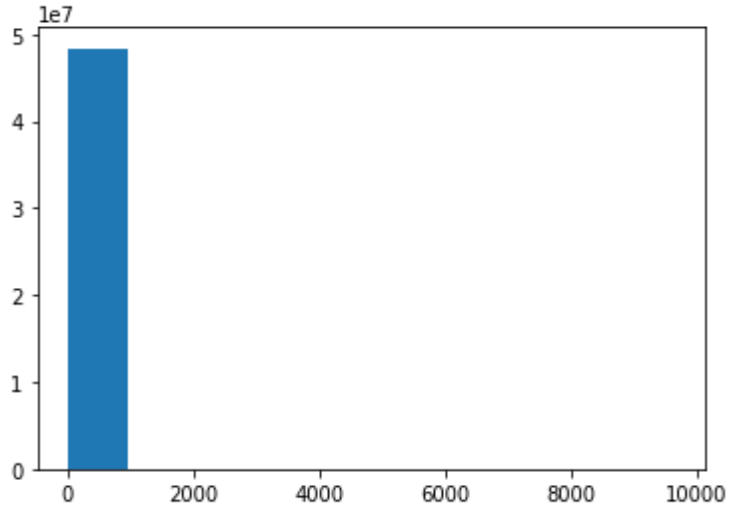


Figure 4.3. Histogram of values in the *play count* column

The histogram of *Figure 4.3* represents the distribution of the *play count* values. Because of having so many registers with a *play count* value between 0 and 1,000, it is not possible to see the histogram values with greater *play count* (>1,000). For this reason the logarithm has been applied to the *play count* column, obtaining the histogram in *Figure 4.4*. This histogram represents that there are a lot of *play count* values between 0 and 1,000 and from 1,000, the repetition of this *play count* values start to decrease. This histogram shape makes sense because listening to a song between 0 and 1,000 times is reasonable.

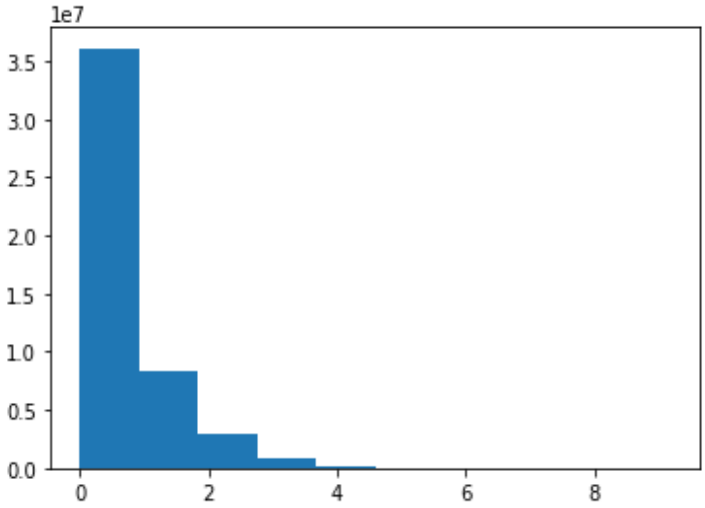


Figure 4.4. Histogram of the values obtained after applying the logarithm to the *play count* column

Looking at the x-axis interval ot the same figure, it is possible to see that there are no negative values, so it is not necessary to delete any row due to this reason, but this interval goes up to 10,000, which is a really big value for *play count*. Getting a little more insight at the big values, it results that the biggest one is 9,667 and it is followed by 3,534, 3,532, 2,948, 2,381, 2,368, 2,213, 2,165... All this values appear just once in the dataset but they seem quite abnormal, especially the 9,967 one. Highest values will make more sense if they come from a robot. For the moment, this registers will not be deleted, but it is important to take this way of proceeding into account.

From this dataset a subset of **10,000 users** and **5,000 songs** will be taken. This subset is big enough to develop the algorithm, but not huge. In order to select the registers that will compose the subset:

1. The 5,000 most listened songs have been taken.
2. The users selected have been the ones that have listened more times those songs.

The reason to be the model input by a subset instead of the by the whole dataset, is to reduce the **sparsity** to the minimum. Sparse, in mathematics, means that there are a lot of zeros in a matrix. In this problem, a zero in a register is because that user has not played that song. A subset with as much played information as possible will be needed. Because of having so many songs and users, is normal that all users (10,000) have not listen to the 5,000 songs in the subset. If this were not the case, this dataset would not be real.

Another reason to use just a subset of the full dataset, is because the computational limitations of the platform on which the project is developed.

Taking into account the dataset situation explained in *chapter 4.1.1* is necessary to apply a transformation to the subset. The *play counts* of songs that have different ID but are

actually the same song, will be summed. As a result, in the subset, there is information of **4,974 different songs** (0.52% of songs dropped out).

4.1.2.1. Data separation

Subset is splitted in **train set**, **validation set** and **test set**.

The **train set** is the data used to train the model. With this set the recommender should not be evaluated, because this evaluation would not represent its real performance.

Validation set is used to evaluate the model after its training. With this set hyperparameters tuning is done. Hyperparameters tuning consists on choosing the best value for each hyperparameter of the model.

Test set is the data used just to give the final performance results of the model. With this results it would be possible to depict if the model generalize well (have a good performance with data that it has never seen) or not.

As the model needs a lot of data to be able to learn and achieve a good performance, most of the set examples are used for training.

In machine learning problems, it is very common to have the **training set** composed of **80 %** of the whole dataset, and **validation** and **test sets** composed of **10%** each (this may vary depending on the problem).

For this particular problem is important to take into account different aspects when the data is separated. This is very important because the subset is very **sparse** (**96.7%**).

The main idea of fetching data separation is that most of the data compose the training set but at the same time enough data should be left for testing and validating, so that the results of the evaluation metrics are valid. The evaluation metrics, explained at *chapter 5.2*, are done over the top-N recommended list, being N: 10, 20 or 30. It is believed that in order to obtain evaluation metrics that make sense, evaluation and test sets should have as minimum N played songs per user as possible. If this is not taken into account, the evaluation metrics could not represent the real performance of the recommender system. It is believed that the recommender will seem worse than it actually is.

Doing data separation by fixing percentages give as result a train set with played information of 10,000 users and 4,974 songs (1,378,142 registers), a validation set with played information of 10,000 users and 4,967 songs (172,268 registers) and a test set with played information of 10,000 users and 4,964 songs (172,267 registers). As mentioned before, it is necessary to analyse if validation and test sets are valid to evaluate the model performance or not.

At the resultant validation set (*Figure 4.5)*, each user has played as minimum 11 different songs. Therefore there is data enough to be able to do a correct evaluation of the top-10

recommended list, but maybe not for top-20 and top-30. The mean of played songs in this data set is of 52.

The resultant test set *(Figure 4.6)* has lower values on the numbers of played songs per user. In it, 633 users have played less than 10 songs, and the mean of played songs per user is 16. The existence of this low values may be taken into account at the moment of evaluating the recommender system.

Therefore, to evaluate top-20 list and top-30 list over the validation set and top-10 list, top-20 list and top-30 list over the test set, the users with less than N played songs in this set will not be taken into account when calculating the evaluation metrics. Also evaluation metrics will be computed without deleting any user, in order to compare both evaluation metrics results and see if this belief is true or not. The conclusion of this is shown in *chapter 5.3.2.*

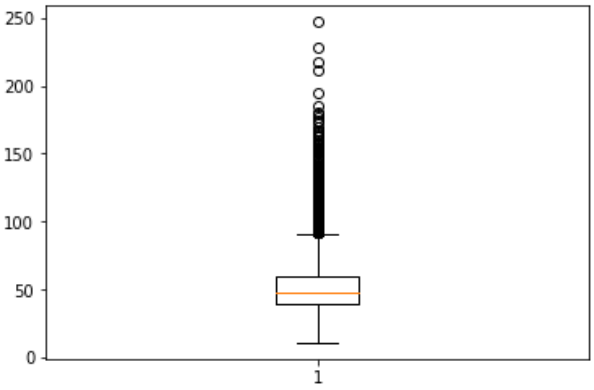


Figure 4.5. Boxplot of number of songs played/user

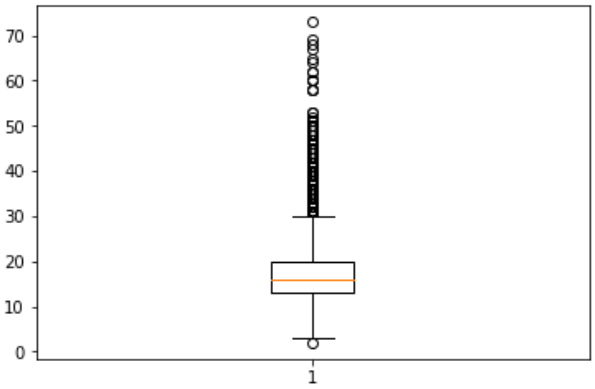


Figure 4.6. Boxplot of number of songs played/user

4.1.2.2. Understanding the training set

It is important to get a good insight of the training set, because depending of the data in it, the behavior of the model could change completely. It is possible that this data is not good enough to be the input of a RS for not having enough information, for having unbalanced information (for example a lot of data of five users and very little of the rest) or for not having information that could relate users.

It is necessary to understand users behaviour. In the set there are users more active than others. By looking at the boxplot in *Figure 4.7,* it is possible to see the number of different songs each user has played. Most users of the set have played between 109 and 151 different songs.

There are two most active users. These users have played 527 and 520 different songs, respectively. Because of this, it is very likely that these users will have a more accurate recommendation list because of the amount of information provided about their musical tastes. Their IDs, for subsequent checking, are:

- 119b7c88d58d0c6eb051365c103da5caf817bea6
- c1255748c06ee3f6440c51c439446886c7807095

The quality of the recommended lists of this two most active users can be found in *chapter 5.3.3.*

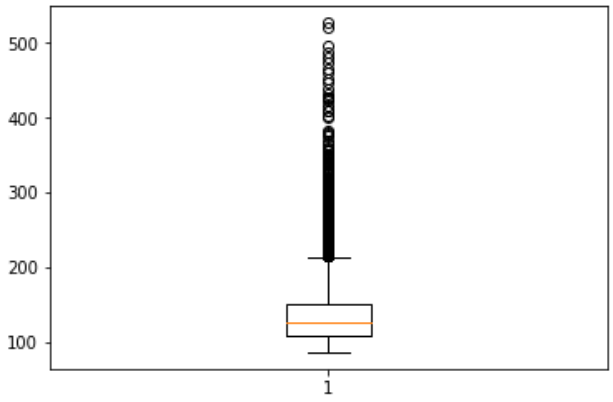


Figure 4.7. Boxplot showing the number of songs each user has played

Eventually it will be demonstrated if it is better to get out this users of the set or not (*chapter 5.3.3*), but they are going to give to the model a lot of information. It is also important to analyse if there is intersection between users. In other words, if it is possible to relate users *(Figure 4.8)*, i.e. if different users have listened to the same songs, or there is not any similarity between their behaviours. If there were no similarity between users behaviours, the recommender would not have a good performance due to the dataset.

Therefore, it would not be possible to relate users and consequently it would not be possible either to recommend songs that similar users had liked.

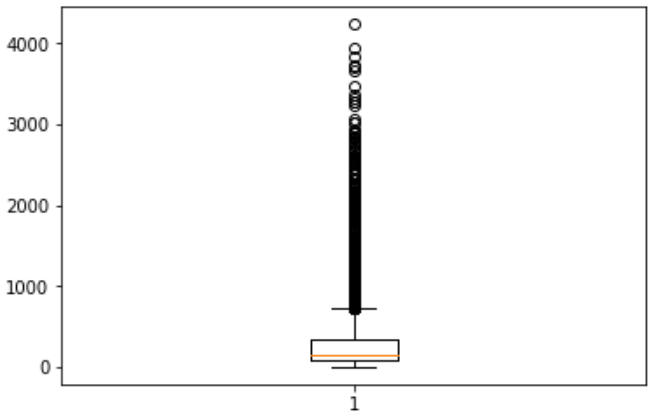


Figure 4.8. Number of users who have listened to each song (each point is a song of the dataset)

In *Figure 4.8* a boxplot is presented, where each point represents a song of the dataset and its value represents the number of users that have listened that particular song. The minimum value of this boxplot is 5, the maximum is 4220, the median is 157 and the mean is 309.84 (displaced from the median due to the great amount of big values that exist). Most of the values in *Figure 4.8* are between 10 and 300. This is good a percentage of similarity between users behaviours taking into account the number of songs that the dataset has (4,990). This situation makes possible for this dataset to be used to develop the RS while maintaining a reasonable computational complexity.

The songs with less value in *Figure 4.8* are very unlikely to appear in the recommender's results due to the low quantity of users who have listened to them. This is a famous issue that many recommenders suffer: recommend the most popular songs instead of the ones that fit better to the user. This happens because collaborative filtering uses this behaviour information for making recommendations. Therefore, if the song has not been very listened, the system will not have enough information about it.

The ten most played songs in the set are:

- *Florence + The Machine - Dog Days Are Over (Radio Edit)*
- *Kings Of Leon - Use Somebody*
- *Harmonia - Sehr Kosmisch*
- *Kings Of Leon – Revelry*
- *Coldplay – Clocks*
- *OneRepublic – Secrets*
- *Coldplay - The Scientist*
- *Charttraxx Karaoke – Fireflies*
- *Coldplay – Yellow*

- *The Killers - When You Were Young*

It is also important to look at the percentage of sparsity that the training set has. If it is greater than 99% this dataset will not have enough information to train the recommender system. The **sparsity** of this training set is of **97.24 %**.

As conclusion, this training set can be used for training the recommender system based on collaborative filtering technique.

4.2. Dataset for artist classification

To develop artist classification, which consists on detecting the artist given a 30 seconds song fragment (mp3 file), the main set of Million Song Dataset is used. This set is divided in two different groups of files: *Data* and *Additional files*.

4.2.1. Data

This group is composed by h5 files named with the track ID of each song of the dataset. This h5 files contain a lot of detailed information of each track (64 different features).

4.2.2. Additional files

Additional files is composed by 12 different files that provide additional dataset information.

The most important file is ***track_metadata.db***. This database contains one table named *songs*, which has one million of registers and 14 columns. This columns are the most important track features (found also in the h5 files) but with a more intuitive way to access them than the h5 files. This most important features are: *track_id*, *title*, *song_id*, *release*, *artist_id*, *artist_mbid*, *artist_name*, *duration*, *artist_familiarity*, *artist_hotttnesss*, *year*, *track_7digitalid*, *shs_perf*, *shs_work*.

From this huge set it will be taken a subset formed by **20 artists.** These 20 selected artists will be those with the greatest number of songs in the dataset (*track_metadata.db*). This is done in order to have the maximum possible inputs to the artist classification model.

For each track their mp3 files will be downloaded, in order to obtain their **Mel Frequency Cepstrum Coefficients** (**MFCC**) afterwards.

It is also possible to have something similar to the MFCC of each track by accessing *segments_timbre* feature of the h5 files. MSD developers define this feature as something similar to MFCC with PCA (Principal Component Analysis) applied, but there is no detailed information about this procedure.

The type of data in *segments_timbre* is a 2D array (matrix) of decimal numbers. This matrix has a fixed number of columns (12), that represents the principal components of the song segment timbre. The number of rows depends on the song duration. Each row is a segment of 0.2 - 0.4 seconds, therefore getting **120 rows** would be **24-48 seconds** of the song and no problems of different songs duration would be encouraged (all songs in the dataset have a minimum duration of 80 seconds). In 24-48 seconds of a song there is enough information to get the necessary knowledge of that song.

The disadvantage of getting "MFCC" from the h5 files, instead of computing them, is that it will not be possible to parametrize them ("MFCC" already computed by dataset developers with some values in the parameters that they had considered appropriate).

Both *segments_timbre* "MFCC" and MFCC extraction from the audio signal will be done and evaluated to get what input is better for the artist classification model. The input that gives better classification result will be selected (*chapter 6*).

Since Echo Nest has partnered 7 digital, it is possible to get the MP3 file of each track of the MSD. In MSD exists a column named *track_7digitalid* which contains the 7 digital identifier to be able to download the MP3 file of that song via **7Digital API**.

When trying to download the 30 seconds fragments of the songs of the 20 artists selected, some troubles were faced. When the identifiers were passed to the API, this seems not to exist in the 7 digital catalog. This means that the *track_7digitalid* column has corrupted data. Therefore, the MP3 files were downloaded by providing the *song title* and *artist name*. The problem of proceeding in this way is that there were more than one possible mp3 files with that *song title* and *artist name*. Listening to all the mp3s of the list, it was discovered that some of them were different versions of a song that sounded quite different than the original one. Because of not knowing which mp3 of the list were the original songs, the first was taken. This will greatly affect to the evaluation of the MFCC extracted from the audio signal because this audio signal could not correspond to the correct one.

5. TRADITIONAL RECOMMENDER SYSTEM

In this chapter, a recommender system based in historical information of the users (songs played by each user) will be developed from scratch. The dataset used for its development has been meticulously explained in *chapter 4.1*. In order to develop this recommender system, **collaborative filtering technique** would be implemented.

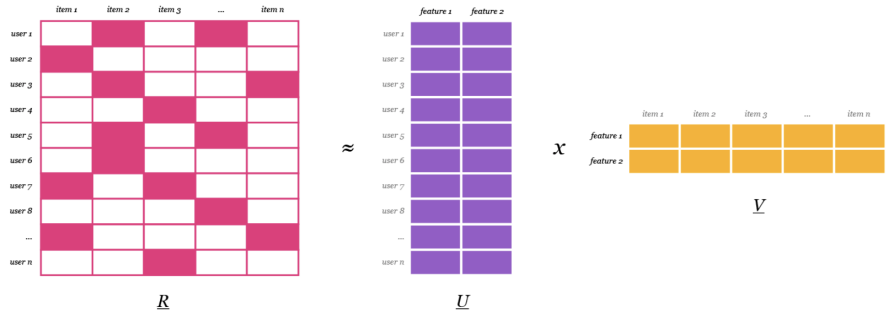
The main dataset used for this chapter (explained in *chapter 4.1.2*), contains play counts per song and per user. This is a form of **implicit feedback**. Songs played many times are assumed to be enjoyed by users. The opposite of this form of feedback is **explicit feedback**, which is, for example, ratings given by the user per song. The main difference between this two feedbacks is that when you have a zero in a record of an implicit feedback it may have been due to many causes: for example, the users might not know about their existence, or they might expect not to enjoy it. This issue makes implicit feedback datasets more challenging and not compatible with traditional matrix factorization algorithms, which have been developed to predict ratings. Otherwise, implicit feedback data is very common. There is a great amount of it (advantage), but at the same time this is very noisy and what it means is not always what it looks like (disadvantage).

5.1. Alternating least squares (ALS) algorithm

Alternating Least Squares (ALS) is a modified matrix factorization (MF) algorithm developed for implicit feedback datasets. In order to understand this algorithm is important to know the basics of matrix factorization [38].

5.1.1. Matrix factorization (MF)

The basic idea behind any matrix factorization method is to find the smallest set of **latent factors** that is representative enough to make good predictions of missing entries. To get the latent factors the whole dataset matrix (R) is factored into two matrices: users matrix (U) and items matrix (V). The product of this two matrices will equal, as much as possible, the original matrix (R).



R → original matrix (users * items).

U → users matrix (users * users latent factors).

V → items matrix (items latent factors * items).

Figure 5.1. Matrix factorization [39]

Latent factors give hidden information of each user and item. There is no knowledge of what this features really are, so it is not possible to label them as "rock", "pop", "techno"...

Let u be a user and i an item. Users and items are associated through r_{ui} which will be call ***observation***. The aim of matrix factorization is to predict this *observation*. This can be done by calculating the following:

$$\hat{r}_{u,i} = x_u^\top y_i \tag{5.1}$$

Where: x_u→ latent vector of user u, y_i→ latent vector of item i.

The target is that this prediction *(5.1)* is as close as possible to the ground truth (values of the matrix R). In order to be capable of this, this problem is framed as an optimization problem using as function to minimize, an standard squared loss (*5.2* first term) with regularization (*5.2* second term):

$$\min_{x,y} \sum_{r_{u,i}\text{is known}} (r_{u,i} - x_u^\top y_i)^2 + \lambda\left(\sum_u ||x_u||^2 + \sum_i ||y_i||^2\right) \tag{5.2}$$

Where x_u→ latent vector of user u, y_i→ latent vector of item i, $r_{u,i}$→ *observation* of user u and item i, λ→ regularization hyperparameter.

With *5.2 equation* what is wanted is to optimize the values of the latent vectors (a latent vector at *Figure 5.1* would be, for example, the first row of U). Therefore the following equations (5.3) (5.4) will iterate until a stopping criterion is reached:

$$x_u = (Y^\top Y + \lambda I)^{-1} Y^\top r_u \tag{5.3}$$

Where: $r_u \rightarrow$ *observations* of user *u*, $Y \rightarrow$ items matrix (items latent factors * items), $\lambda \rightarrow$ regularization hyperparameter, $I \rightarrow$ identity matrix.

$$y_u = (X^\top X + \lambda I)^{-1} X^\top r_i \tag{5.4}$$

Where: $r_i \rightarrow$ *observations* of item *i*, $X \rightarrow$ users matrix (users * users latent factors), $\lambda \rightarrow$ regularization hyperparameter, $I \rightarrow$ identity matrix.

X and *Y* (*U* and *V* at *Figure 5.1*) are the user and item matrices. This matrices are randomly initialized, and will be updated in each iteration.

Matrix factorization mathematically **reduces dimensionality** of the problem, making possible to express each user as a vector of their taste values, and each item as a vector of what tastes they represent. This dimensionality reduction gives better results and makes the process more computationally efficient.

There are a lot of different ways to factor a matrix. If the dataset was a representation of explicit feedback it could be used Singular Value Decomposition (SVD) or Non Negative Matrix Factorization (NMF). As mentioned before, the modelling of the data is more difficult with implicit feedback than with explicit, so it is necessary to deal differently with missing data, it is needed to *learn* from it. This is the reason to implement ALS.

5.1.2. Alternating Least Squares (ALS) algorithm

ALS was proposed by **Hu, Korenand and Volinsky** in their paper **Collaborative Filtering for Implicit Feedback** [40]. This method became very popular because of it's good results. ALS is currently implemented by Facebook and Spotify in their recommendation systems.

ALS is an **iterative optimization process** that tries to arrive closer and closer to a factorized representation, of the original data, at every iteration. This method measures the closeness to a target by calculating the sum of squared distances.

Their solution is to merge two new variables: ***preference*** (***p***) for an item and ***confidence*** (***c***) that there is about that *preference*.

Let r_{ui} be the *play count* for user u and song i. For each user-item pair, a *preference* variable p_{ui} and a *confidence* variable c_{ui}, are defined.

$$p_{ui} = \begin{cases} 1 & r_{ui} > 0 \\ 0 & r_{ui} = 0 \end{cases} \tag{5.5}$$

Where: r_{ui}→ *observation* of user *u* and item *i.*

The *preference* variable (*5.5*) is basically a binary representation of the feedback data (r). If the *preference* variable is one, it is assumed that the user has enjoyed the song.

$$c_{ui} = 1 + \alpha r_{ui} \tag{5.6}$$

Where: r_{ui}→ *observation* of user *u* and item *i,* α → hyperparameter.

The *confidence* variable (*5.6*) measures the certainty about this particular *preference*. For its calculation it is used the magnitude value of r. When r_{ui} is zero the *confidence* variable value is one. A small r_{ui} may have been caused because the song was played by mistake, but also because it was not liked by the user. For this reasons, the *confidence* variable value would also be low. *Confidence* would be greater in songs that have been played a lot of times (large r_{ui}). It is summed one in *5.6 equation*, to have a minimal *confidence* although the product of α and r is equal to zero (least informative case).

Parameter α will change the slope of *confidence*. Hu paper recommends the use of an α equal to 40, so α will be tuned by looking at the evaluation results over the validation set, in order to be sure about this (*Figure 5.6*).

As mentioned at the beginning of this chapter, latent factors will be optimized using **alternating least squares (ALS) optimization method.** Loss function is similar to *5.2 equation* but updated with the new variables:

$$\min_{x_\star, y_\star} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u ||x_u||^2 + \sum_i ||y_i||^2 \right) \tag{5.7}$$

Where: λ → regularization hyperparameter, x_u → latent factor vector of user *u*, y_i → latent factor vector of item *i,* p_{ui}→ *preference* variable of user *u* and item *i*, c_{ui}→ *confidence* variable of user *u* and item *i.*

The first term of (*5.7) equation* is a confidence-weighted mean squared error and the second term is a L2 regularization. It is possible to see that the sum in the loss function

(5.7) goes all over every song and item. In the explicit feedback case the sum will be computed only through the non zero elements.

Computing the derivative of the loss function (*5.7),* the following updated latent vectors equations are obtained. If this equations are computed: the loss of users *(5.8)* and items *(5.9)* are minimized.

$$x_u = (Y^T C^u Y + \lambda I)^{-1} Y^T C^u p(u) \tag{5.8}$$

Where: λ → regularization hyperparameter, Y → items matrix (items latent factors * items), C → *confidence* matrix, I→ identity matrix, $p(u)$→ *preference* of user *u.*

$$y_i = (X^T C^i X + \lambda I)^{-1} X^T C^i p(i) \tag{5.9}$$

Where: λ → regularization hyperparameter, X → users matrix (users * users latent factors), C → *confidence* matrix, I→ identity matrix, $p(i)$→ *preference* of item *i.*

The loss *(5.7)* is not convex in (*X, Y*). This means that each step will move toward a minimum, but not necessarily a global minimum. The result of the algorithm will depend on the initialization of *X* and *Y*, so if initialization is made in a good way, the algorithm will converge to a solution faster.

By doing λ greater, the equation will converge to the same solution each time the algorithm is executed, due to the fact that the solution space is constrained.

As a conclusion of the above explanation, ALS is slower and more difficult that optimising with Stochastic Gradient Descent (SGD). The disadvantages of SGD are: it does not work good with this dataset size, it is not so easy to parallelise as ALS (using Spark) and ALS have really good solutions just with 10 iterations (not SGD case).

5.1.3. Applications

After applying ALS to the dataset, two different actions could be made: to identify similar items and to make recommendations. This actions are based on associating close elements (users or items) in terms of cosine distance in the **latent space**. The latent space is a shared low-dimensional space obtained by projecting both items and users latent factors. An example of the latent space of a movie RS is represented in *Figure 5.2*. The position of the different users and items (movies) in the latent space is due to its latent factor vectors values. According to *Figure 5.2*, the most similar items to *Braveheart* provided by the system would be *Amadeus* and *Lethal Weapon*. The system would recommend Gus to watch, in decreasing order: *Dumb and Dumber*, *Independence Day* and *The Lion King.*

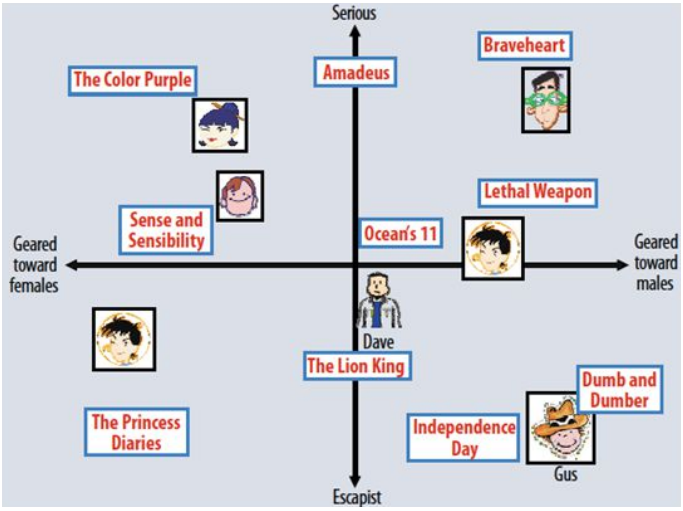


Figure 5.2. Latent space of movies recommender system [41]

5.1.3.1. Identify similar items

It is possible to find similar songs (nearby items in the latent space) by doing the following operation, giving as a result a similarity score:

$$score = V * V_i^T \tag{5.10}$$

Where: $V \rightarrow$ item matrix, $V_i \rightarrow$ latent vector of item *i*.

By changing the distribution of the sparse matrix, having as columns artists instead of songs, it will be possible to find similar artists.

5.1.3.2. Make recommendations

Recommendations can be made, per user and item, by doing the following operation, giving as a result a recommendation score:

$$score = U_i * V \tag{5.11}$$

Where: $U_i \rightarrow$ latent vector of user *i*, $V \rightarrow$ item matrix.

The items closest, in the latent space, with a particular user, will have the greater score and therefore will be recommended.

5.2. Evaluation Metrics [42]

In this chapter, the metrics used to evaluate the recommender system will be presented and deeply explained. It is important to understand that this evaluation metrics will also be used to do hyperparameters tuning. Therefore, for this task, this metrics will be calculated over the validation set. When the system is fine-tuned, the final evaluation of the RS will be obtained by calculating the evaluation metrics over the test set.

Figure 5.3 shows an understandable recommender system workflow.

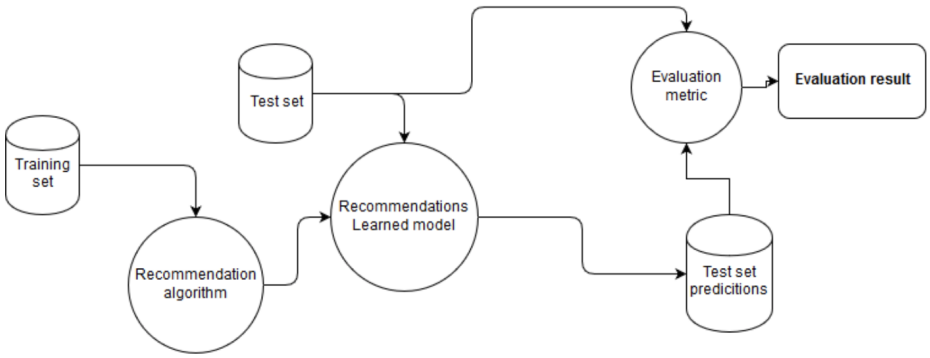


Figure 5.3. Recommendation system workflow [43]

If the RS were enriched with explicit feedback, the evaluation metrics used would be: MAE (Mean Absolute Error), MSE (Mean Squared Error) or RMSE (Root Mean Square Error). The objective of the use of this feedback would be getting as close as possible to the rating that the user gave. This is the reason for using metrics that measure the error by computing the difference between the real rating and the one predicted.

In this project the feedback is implicit, so instead of being focused on the predicted values, it is more important to evaluate the list of recommendations (top-N recommendations) based on relevancy levels. This is because the previous metrics require to know which items are disliked by the user in order to make sense. This information is not available in implicit feedback, so the previous metrics are not appropriate for this case.

The main evaluation metrics for implicit feedback are **Precision@n**, **Recall@n**, **MAP (Mean Average Precision)** and **nDCG (normalized Discounted Cumulative Gain).** All this metrics are used to show the quality of top-N recommended lists, how good they are. Before calculating this evaluation metrics, it is important to choose N. Recommendation lists offered by Spotify have 30 songs each. In this project N will be variable. It is understood that playlists have value for a user in a range of songs from 10 to 30, so N will either be 10, 20 or 30.

The detailed explanation of the metrics used to evaluate this project recommender are below.

5.2.1 Recall@n

Is a binary metric that measures the proportion of appearance of the total relevant items (song played in test set) in the top-N recommended list *(5.12)*. It is easier to understand it with an example: if computing $Recall@10$obtains a value of 0.6, this means that 60% of the relevant items are in the top-10 recommended list.

$$Recall@n = \frac{Number\ items\ in\ topN\ and\ in\ test}{Size\ test\ set} \tag{5.12}$$

Each user of the dataset will have a different recommended list, so each user will have a $Recall@n$ value. For this reason, for having an overall $Recall@n$ value of the whole recommender systems, the mean between all $Recall@n$ values of each user will have to be computed.

5.2.2 Precision@n

It is also a binary metric that measures the proportion of item in the top-N recommended list that are also in the test set. This time this proportion is measured from the number of items in the recommended list (N) *(5.13)*.

$$Precision@n = \frac{Number\ items\ in\ topN\ and\ in\ test}{N} \tag{5.13}$$

If we represent $Precision@n$ in percentage, it represents the percentage of the recommendation lists that have been relevant to the user. For example, if a $Precision@20$is obtained out of 30% this means that the 30% of the top-20 recommended list are relevant items.

There will be a $Precision@n$ value for each user of the dataset, so it is necessary to proceed in the same way as in $Recall@n$ case to obtain a $Precision@n$ of the whole recommender system.

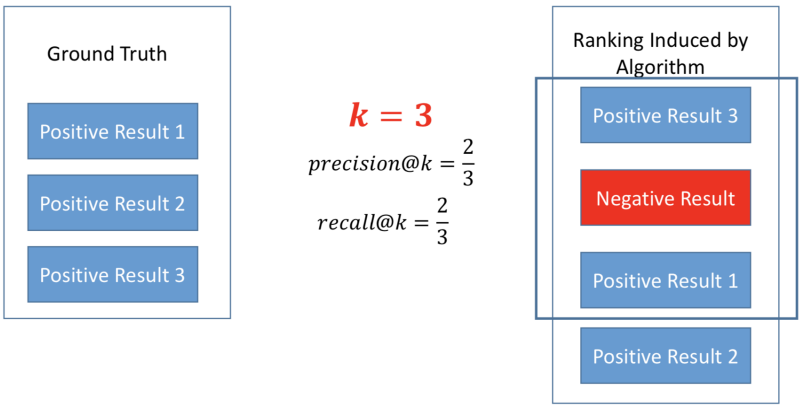


Figure 5.4. Graphic example Recall@n and Precision@n explanation [44]

5.2.3. F1 score@n

This metric gives a score of the top-N recommended list. For its calculation it takes into account $Precision@n$ and$Recall@n$ values of each user. It shows how in balance are this two metrics:

$$F1\ score@n = \frac{2 * Recall@n * Precision@n}{Recall@n + Precision@n} \tag{5.14}$$

To get the $F1\ score@n$ of the overall recommender system is also necessary to compute the mean of all $F1\ scores@n$ each user have.

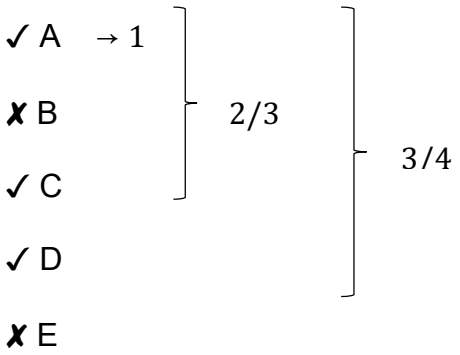
5.2.4. mAP (mean Average Precision)

mAPgives information about how good the top-N recommended list is. $mAP$is the most important metric because it is the most used when evaluating recommender systems and it was specifically designed for this type of problems.

The main difference between $mAP$and the previous metrics is that it gives different weights to relevant items in position 1 rather than in position 9. Starting from the ranking list, and going through it, mAP calculates the precision at each group of elements. When all these precisions have been calculated the average of them has to be computed (AP).

It is better to understand how $AP$is computed with an example:

Having as output of a recommender system this rank, and meaning ✓ relevant item in test set and ✘ irrelevant item in test set:



$$AP = \frac{1 + 2/3 + 3/4}{3} = 0.8$$

Example 5.1. mAP

Each user will have an APvalue associated to its top-N recommended list. For having an $AP$of the whole recommender system it is necessary to compute the mean, having as a result the recommender mAPvalue:

$$MAP(O,U) = \frac{1}{|U|}\sum_{u\in U} AP(O(u)) \tag{5.15}$$

Where: $U\rightarrow$ set of users.

5.2.5. nDCG (normalized Discounted Cumulative Gain)

$nDCG$ [45], as mAP, gives different item weights depending on the position they have at the top-N recommended list. In other words, $nDCG$ measures the utility of an item at each position in the list. This metric was also specifically designed for this type of problems, recommender systems problems.

The *gain* in explicit feedback would be the rating (r_{ui}) a user gave to a particular song. In this case (implicit feedback) the *gain* is the binary item relevance (0: irrelevant item, 1: relevant item) showed in test set.

A *discount* is then applied to the *gain*. This discount is a function that grows when its independent variable also grows as well. Most common discount (5.16) applied is a base two logarithm of the item rank position. After applying this, the discounted gains of each item in the top-N list are summed. This is why it is named Discounted **Cumulative** Gain (*5.17)*.

Each user will have a DCG value of the quality of its own recommendation list. Comparing different DCG values between users is not possible due to their different scales. For this reason DCG values need to be normalized (*5.18*). In order to normalize DCG, it is necessary to divide the DCG value of the RS by the DCG value of the same recommender if it were perfect (all items in top-N list are relevant in the test set).

$nDCG$ explains how similar is this recommender to the perfect recommender.

$$disc(i) = \begin{cases} 1 & i \leq 2 \\ log_2(i) & i > 2 \end{cases} \tag{5.16}$$

$$DCG(O,u) = \sum_{i=1}^{N} \frac{r_{ui}}{disc(i)} \tag{5.17}$$

Where: $i \rightarrow$ item ranking position, $u \rightarrow$ a particular user, $r_{ui} \rightarrow$ binary item relevance shown in test set, of that item in the position *i* of the recommended list for user *u*, $N \rightarrow$ number of items in the recommended list (length list), $disc(i) \rightarrow$ discount function (5.16).

$$nDCG = \frac{DCG}{DCG_{PERFECT}} \tag{5.18}$$

Lets see in *example 5.2* the same scenario as in *example 5.1* to understand better how $nDCG$ is computed.

Having as output of a RS this rank and meaning ✓ relevant item in test set and ✘ irrelevant item in test set:

✓ A $i = 1$

✘ B $i = 2$

✓ C $i = 3$

✓ D $i = 4$

✘ E $i = 5$

$$DCG(O,u) = \frac{1}{1} + \frac{0}{1} + \frac{1}{1.58} + \frac{1}{2} + \frac{0}{2.32} = 2.13$$

$$DCG_{PERFECT}(O,u) = \frac{1}{1} + \frac{1}{1} + \frac{1}{1.58} + \frac{1}{2} + \frac{1}{2.32} = 3.56$$

$$nDCG = \frac{2.13}{3.56} = 0.6$$

This recommender has achieved 60% of the possible DCG (perfect recommender).

Example 5.2. nDCG

Each user will have its own $nDCG$value. The mean of the different $nDCG$ values will be the $nDCG$ of the whole recommender system:

$$nDCG(O,U) = \frac{1}{|U|}\sum_{u\in U} nDCG(O(u)) \tag{5.19}$$

5.3. Implementation

As mentioned in *chapter 4.1.2,* the dataset used to implement ALS consists on three sets with play counts of 10,000 users. To train, 4,974 songs have been used. Validation has been performed with 4,967, and testing with 4,964. This sets are given in form of matrices, with songs as columns and users as rows. Therefore, this **matrices** have a **size** of **10,000 x 4,974**.

The training set matrix (***R*** matrix), as mentioned in *chapter 4.1.2.2*, has a **sparsity of 97.23 %**. This is a good dataset sparsity for a recommendation task.

To test and validate, its matrices have been converted to binary matrices. In other words, each register of the matrices that has a value (*play count*) greater than zero, will be replaced by a one. This one, in the test and validation matrices, means that the user has liked that song.

This binary conversion is done to simplify the recommendation task to a binary classifier: liked or not liked.

5.3.1. Hyperparameters tuning

Hyperparameters tuning is a very important phase of any machine learning project. Depending on the values of the hyperparameters, the result of the recommender could be completely different.

In this chapter every hyperparameter of the ALS algorithm will be tuned. This hyperparameters will be tuned by analysing the F1 score of the top-10 recommended list ($F1\ score@10$metric is explained deeply in *chapter 5.2.3*) over the validation set. It would be better to analyse mAPvalues to do hyperparameter tuning, but because of computational limitations this is not possible (mAPcomputing takes a lot of time while $F1\ score$ does not). In order to perform the hyperparameters tuning task, only the top-10 recommended list has been evaluated because the ten items that form this list are already the most important in the other two (top-20 and 30).

As explained in *chapter 5.1,* to implement ALS it is necessary to define the **number of latent factors** (***k***) of the algorithm. This is the most important hyperparameter of the model. These hyperparameter represents the amount of hidden information extracted from songs and users. If the number of latent factors is very low, extracted information will not be enough to make accurate recommendations. If this is too high, there will be too much extracted information of songs and users, and it will not be possible to make accurate recommendations either. Therefore, it is important to take this considerations into account and have a balance when tuning this hyperparameter. The number of latent factors is really dependent on the dataset. In practice people use between 20 and 200 latent factors. Therefore this interval will contain the evaluated values.

Another hyperparameter to be tuned is the **number of iterations** that the algorithm does to learn from the training data.

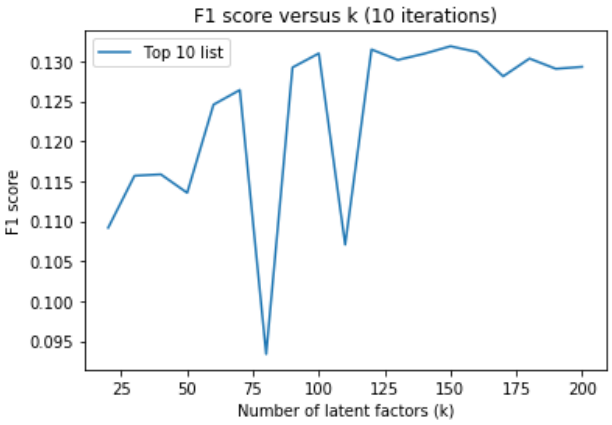
In every experiment (*Figure 5.8*) the **regularization hyperparameter** has been set to **0.01**.

The value of the $F1\ score$ has to increase while the number of latent factors (k) increases as well. This score will start decreasing when a high k value is reached (too much hidden information extracted). In *Figure 5.8* it is possible to see that just the plots trained with 20 and 30 iterations have the expected $F1\ score$ trace. This makes sense because, as explained in *chapter 5.1.2*, one of the advantages of ALS is that it gives really good results with a low number of iterations (around 10). It is also possible to see in *Figure 5.8* that when more than 30 iterations are done training, the performance of the recommender becomes completely unpredictable.

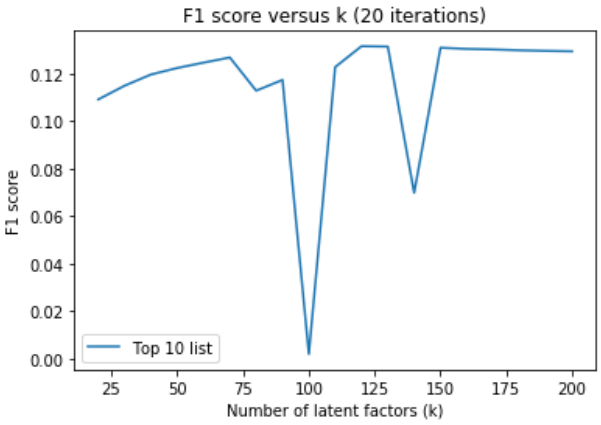
To tune the hyperparameters values, plots (b), and (c) from *Figure 5.8* are compared. The maximum value of the score in each plot is:

- Plot *b:* 0.128 at k = 75
- Plot *c:* 0.132 at k = 130

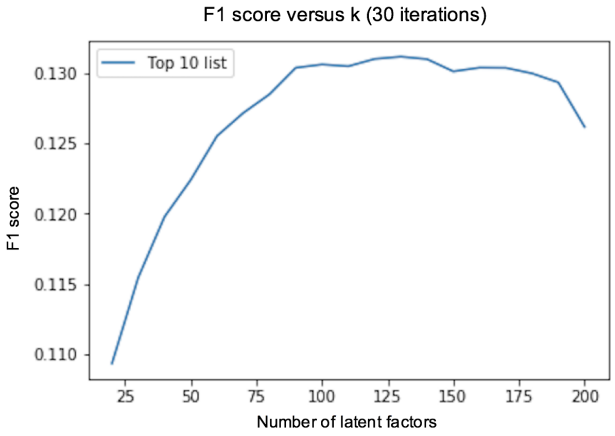
The highest $F1\ score$ between the maximum values of plots (b) and (c) is the one of plot (c). Therefore, the **tuned hyperparameters** of this collaborative filtering recommender are: **130 latent factors** and **30 iterations.**



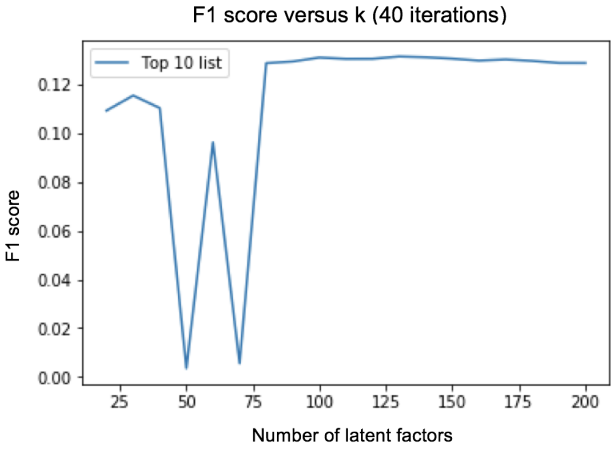
(a)



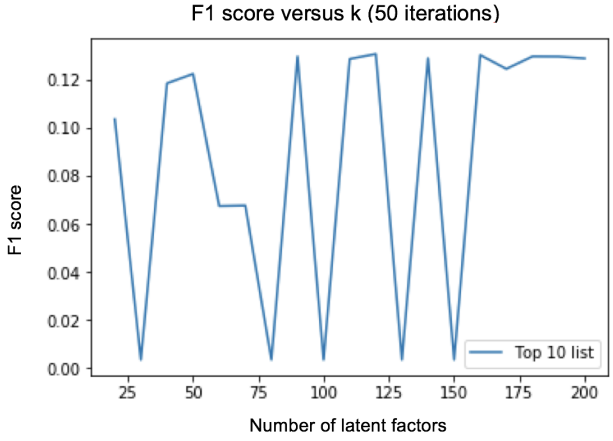
(b)



(c)



(d)



(e)

Figure 5.5. Line charts representing the F1 score versus k with different number of iterations.

The development of this traditional recommender system has been performed in Python (the code can be found in github [46]). To implement ALS and do all this hyperparameters tuning, the implicit library [47] has been used. This library is very famous because of its fast ALS training, but it has the inconvenience of not having *alpha* (*5.6*) implemented in its calculus. For this reason it is very common to multiply the whole train matrix by this *alpha*. This does not have the same effect in the recommender as the *alpha* of the *confidence* equation (5.6), but in some problems it helps. To discover what the value of *alpha* should be in this particular problem, the same technique as to get the number of latent factors is applied (*Figure 5.9*).

It is possible to see in *Figure 5.9* that the **value of *alpha* has to be 1**. Therefore, in this problem, multiplying the training matrix by an *alpha* does not improve the performance of the recommender at all.

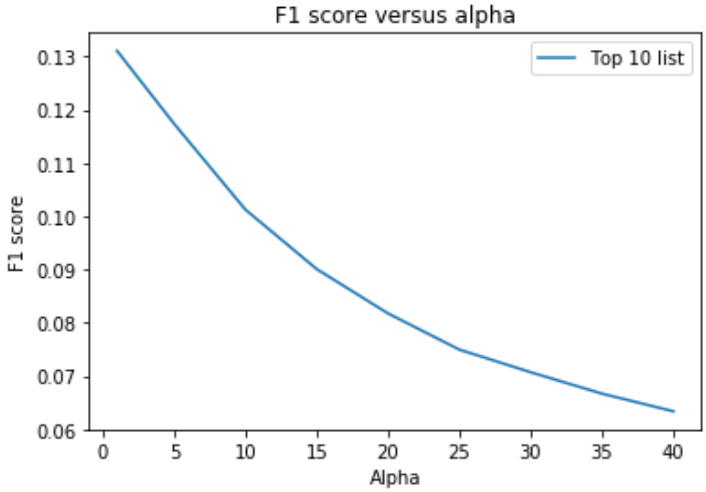


Figure 5.6. F1 score versus alpha value.

5.3.2. Experiments

In this chapter some comparisons between the results of the experiments and ascertainments are done.

All the evaluation metrics (*chapter 5.2*) of the different experiments have been calculated over the validation set. These are shown in *Table 5.1*, *Table 5.2* and *Table 5.3*. Each table shows the results for each possible top-N recommended list (10, 20 and 30).

TABLE 5.1. EVALUATION RESULTS (OVER THE VALIDATION SET) OF THE TOP-10 RECOMMENDED LIST OF THE DIFFERENT EXPERIMENTS.

Model	**Recall**	**Precision**	**F1 score**	**mAP**	**nDCG**
Random	0.002	0.003	0.002	0.0305	0.0107
***k* = 130** **30 iterations** **alpha = 1**	0.150	0.127	0.132	0.4542	0.3089
***k* = 130** **30 iterations** **alpha = 1** **without users outliers**	0.058	0.02	0.027	0.166	0.084

TABLE 5.2. EVALUATION RESULTS (OVER THE VALIDATION SET) OF THE TOP-20 RECOMMENDED LIST OF THE DIFFERENT EXPERIMENTS.

Model	**Recall**	**Precision**	**F1 score**	**F1 score_**	**mAP**	**mAP_**	**nDCG**	**nDCG_**
Random	0.002	0.004	0.003	0.003	0.037	0.037	0.011	0.011
***k* = 130** **30 iterations** **alpha = 1**	0.208	0.125	0.150	0.150	0.406	0.406	0.296	0.3
***k* = 130** **30 iterations** **alpha = 1** **without users outliers**	0.052	0.036	0.038	0.038	0.156	0.156	0.079	0.079

TABLE 5.3. EVALUATION RESULTS (OVER THE VALIDATION SET) OF THE TOP-30 RECOMMENDED LIST OF THE DIFFERENT EXPERIMENTS.

Model	**Recall**	**Precision**	**F1 score**	**F1 score_**	**mAP**	**mAP_**	**nDCG**	**nDCG_**
Random	0.006	0.003	0.003	0.004	0.036	0.036	0.010	0.010
***k* = 130** **30 iterations** **alpha = 1**	0.27	0.109	0.15	0.15	0.378	0.378	0.287	0.287
***k* = 130** **30 iterations** **alpha = 1** **without users outliers**	0.049	0.049	0.044	0.046	0.147	0.147	0.076	0.076

It is possible to observe that the results in the columns *F1 score_, mAP_* and *nDCG_* in *Table 5.2* and *Table 5.3* are almost the same as the original scores of that evaluation metrics (*F1 score, mAP, nDCG*). This "new" evaluation results were obtained not taking into account users that have less than 20 played songs in its validation set (*Table 5.2* case) or less than 30 (*Table 5.3* case). The results of this "new" evaluation metrics are completely different than expected. It was expected that they would have a higher value than its original evaluation metric results. Therefore this "new" evaluation metrics will not be calculated in the final testing of the recommender, because they do not provide more value that its original evaluation metrics. All the explanation of this belief is in *chapter 4.1.2.1*.

Due to the obtained results in the different experiments, it is possible to see that the performance of the RS over the validation set is quite acceptable for the complexity of the problem (second row of *Table 5.1*, *Table 5.2* and *Table 5.3*). There is a high difference between this results and the random RS ones ($mAP@10$of the developed RS is 10 times higher than the random RS one).

In *chapter 4.1.2.2* it was mentioned that maybe it was better to delete the users outliers (two users). Based on the results obtained from the experiments (second row and third row of the tables), it is possible to see that the RS with the most active users data in its training, is four times better that the one which does not have them. With that in mind, it is possible to affirm that eliminating the most active users greatly worsens the recommender's performance. In this chapter it has also been explained that it was very likely that the most active users have the best recommendations in comparison with the

rest of users of the dataset. In *Table 5.4* there are the evaluation metrics results of the recommendations done to this two users.

TABLE 5.4. EVALUATION METRICS RESULTS (OVER THE VALIDATION SET) OF THE MOST ACTIVE USERS OF THE DATASET.

Recommendations	**F1 score@10**	**mAP @10**	**nDCG @10**	**F1 score@20**	**mAP @20**	**nDCG @20**	**F1 score@30**	**mAP @30**	**nDCG @30**
Recommendations for active user 1	0.111	0.754	0.856	0.156	0.655	0.432	0.189	0.548	0.447
Recommendations for active user 2	0.094	0.594	0.616	0.108	0.691	0.578	0.119	0.607	0.561

Being the id of the active user 1: *119b7c88d58d0c6eb051365c103da5caf817bea6,* and the id of the active user 2: *c1255748c06ee3f6440c51c439446886c7807095*.

The evaluation results of the quality of the top-N recommended lists of the most active users of the dataset (*Table 5.4*) are much better than the evaluation results over the whole dataset (especially mAP and $nDCG$ results).

On average, active user 1 results are better that active user 2. This makes sense because user 1 is more active than user 2 in the dataset, so as a consequence the recommender "knows better" active user 1.

Focusing on the mAP and $nDCG$ results of the top-10 recommended list of the active user 1, it is possible to see that this 10 songs fit almost perfectly with this user tastes. This top-10 recommended list is an 85.6% similar to the perfect top-10 recommended list, which is a really good result.

5.3.3. Results song recommendations

After having done hyperparameters tuning and comparison between different experiments to design the best model for this traditional RS, **final performance results** are given. This performance results are the evaluation metrics calculated over the test set.

As explained in the previous chapters, the final RS based in CF technique has as model an ALS with 130 latent factors, alpha parameter equals one, the regularization hyperparameter is 0.01 and it is trained with 30 iterations. In the data to train the model, the users outliers are included.

The final results of this recommender are shown in *Table 5.5*, *Table 5.6* and *Table 5.7.*

TABLE 5.5. EVALUATION METRICS RESULTS (OVER THE TEST SET) OF THE FINAL TOP-10 RECOMMENDED LISTS OUTPUT BY THE TRADITIONAL RECOMMENDER SYSTEM.

Recall@10	**Precision@10**	**F1 score@10**	**MAP@10**	**nDCG@10**
0.127	0.21	0.154	0.089	0.035

TABLE 5.6. EVALUATION METRICS RESULTS (OVER THE TEST SET) OF THE FINAL TOP-20 RECOMMENDED LISTS OUTPUT BY THE TRADITIONAL RECOMMENDER SYSTEM.

Recall@20	**Precision@20**	**F1 score@20**	**MAP@20**	**nDCG@20**
0.208	0.174	0.183	0.095	0.037

TABLE 5.7. EVALUATION METRICS RESULTS (OVER THE TEST SET) OF THE FINAL TOP-30 RECOMMENDED LISTS OUTPUT BY THE TRADITIONAL RECOMMENDER SYSTEM.

Recall@30	**Precision@30**	**F1 score@30**	**MAP@30**	**nDCG@30**
0.27	0.152	0.189	0.095	0.039

By comparing the evaluation results over the validation set with the ones over the test set, it is possible to see that model is overfitted to the train and validation data. The mAP and $nDCG$ values over the test set are very low. The model does not generalize well. A possible solution could be to cross validate while tuning hyperparameters. Cross validation is that in each iteration the model has different train set and validation set (a deep explanation can be found at the end of chapter 6.1). This could make the hyperparameters not to be tuned according to a single validation set so that the model would not overfit.

Another possible solution could be having more data, so that the tastes of each user are more defined. Also tracking the behaviour of the users with the songs of the recommended lists will provide more data to enrich the model.

5.3.4. Results similarity songs

As mentioned in *chapter 5.1.3*, by developing ALS more applications than just recommendation of items to users can be done. It is also possible to get similar songs to

another song. By implementing the operation presented in *chapter 5.1.3.1* the results shown in *Table 5.8* have been obtained.

TABLE 5.8. SIMILAR SONGS EXTRACTED

Query	**Most similar song (ALS)**
Justin Bieber - **That should be me**	Miley Cyrus - Party in the USA Travie McCoy ft Bruno Mars - Billionaire Taylor Swift - Love Story Charttraxx Karaoke - Fireflies
Eminem - **My name is**	Usher - Yeah! Eminem/Dina Rae - Superman Linkin Park - In the end Miley Cyrus - Party in the USA
Katy Perry - **Hot N Cold**	Katy Perry - Lost Miley Cyrus - The climb Lilly Allen - Cheryl tweedy Katy Perry - I kissed a girl
Hannah Montana - Nobody's Perfect	Kelly Clarkson - Already Gone Coldplay - Shiver Coldplay - Clocks Miley Cyrus - Party in the USA

It is important to take into account (as explained in *chapter 4.1.2*) when evaluating the performance of similar songs extracted, that *Coldplay - Clocks, Miley Cyrus - Party in the USA* and *Charttraxx Karaoke - Fireflies* are among the top twenty played songs of the training set. These songs are very likely to appear as similar songs even though they are not so similar. This behaviour can be easily observed in the similar songs to *Eminem - My name is* (second row of *Table 5.8*). The rest of the similar songs extracted showed in *Table 5.8* make sense, though. Every other recommendation provides similar songs, including ones from the same artist as the analysed one.

6. ARTIST CLASSIFICATION USING CNN

This chapter consists on the implementation of a system that will detect an artist from an mp3 file. This system is known as a **multi-output classifier** (in this case 20 different artists as possible outputs will be used). The classification will be performed by extracting the **Mel Frequency Cepstral Coefficients (MFCC)** of the audio signal (mp3 file), followed by a **convolutional neural network (CNN)**.

6.1. MFCC extraction

Mel Frequency Cepstral Coefficients (MFCC) are coefficients used for the representation of speech based on human auditory perception. Davis and Mermelstein introduced them in 1980 [48]. MFCC appeared due to the necessity in automatic audio recognition of extracting relevant characteristics from the audio components to detect their content.

In order to obtain the MFCC, the audio signal will be analysed and processed through the following steps [49]:

1. Divide the signal into frames (set of samples).
2. Apply to each frame the DFT (Discrete Fourier Transform) and get the spectral power of the signal.
3. Map the spectral powers onto the mel scale (this scale is based on human hearing).
4. Take the log of these spectrums, having as a result the Mel spectrogram (*Figure 6.1*).
5. Apply the Discrete Cosine Transform (DCT) (it approximates the PCA).
6. The MFCC are the amplitudes of the resulting spectrum (each coefficient has a value for each frame of the sound).

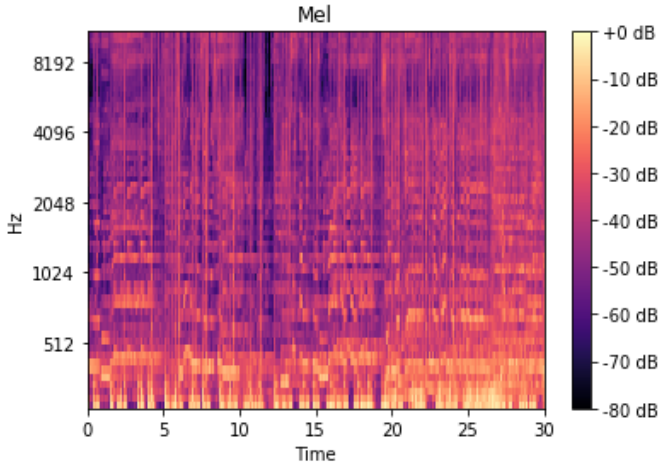


Figure 6.1. Mel spectrogram of a pop song [50]

By extracting the MFCC of an audio signal, a **time-frequency representation** will be obtained. It is important that this representation has enough information of the audio signal to input the CNN, but not too much, because that would saturate the network.

In this project problem it is possible to get the MFCC of each song by extracting it from the audio signal (mp3 file) or by accessing it via an h5 file. The h5 files have a feature with a precalculated "MFCC" by the dataset developers.

6.1.1. MFCC from mp3 files

Out of the 20 artists selected, just 72 mp3 song files of each artist, with a duration greater than 28 seconds, were possible to be downloaded (via the 7Digital API). 72 inputs for each class are not enough for the CNN to achieve good classifications. For this reason, each song will be divided in two (14 seconds each) or four (7 seconds each), having as a result 144 or 568 network inputs per class, which are still low number of inputs for a CNN. The number of divisions will be chosen depending on the result the network gives. This separations represent a tradeoff between getting more useful information from a song (greater length of song) and having a higher number of inputs in the network.

Depending on the MFCC parameters values, the MFCC calculated will vary.

Different MFCC parameter values are evaluated:

- **Type 1** → 128 MFCC, computed with a window size of 1,024 samples, corresponding to 23ms of song sampled at 22,050 Hz, and a hope size of 512 samples.
- **Type 2** → 128 MFCC, computed with a window size of 2,048 samples, corresponding to 46ms of song sampled at 22,050 Hz, and a hope size of 1,024 samples.

Extracting the MFCC of the different proposals of a 14 seconds song, the MFCC spectrograms shown in *Figure 6.2* and *Figure 6.3* are obtained.

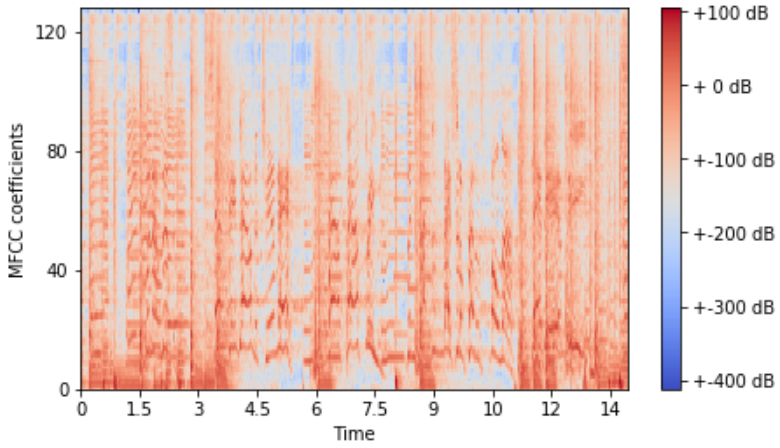


Figure 6.2. MFCC spectrogram of a 14 seconds audio fragment, having as parameters for computing 128 MFCC a window size of 1,024 samples and a hop size of 512 samples

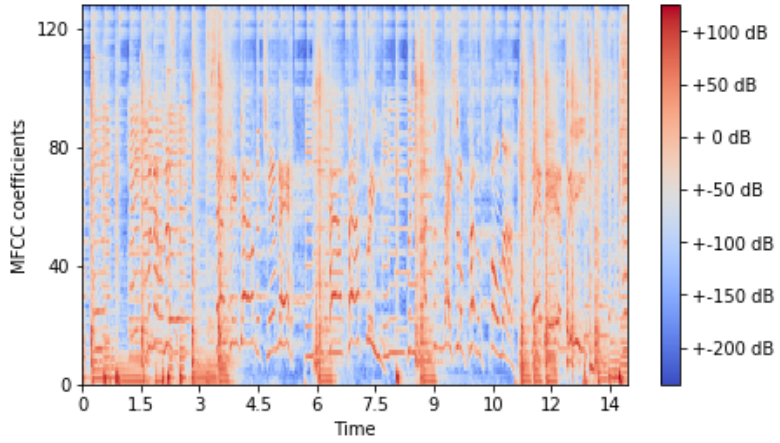


Figure 6.3. MFCC spectrogram of a 14 seconds audio fragment, having as parameters for computing 128 MFCC a window size of 2,048 samples and a hop size of 1,024 samples

The spectrogram of *Figure 6.2* is a matrix of **128 rows** (MFCC) and **600 columns** (audio frames), while the spectrogram of *Figure 6.3* is a matrix of **128 rows** (MFCC) and **300 columns** (audio frames). This spectrograms are very different, its pattern is the same but the MFCC in *Figure 6.3* have lower power values than the ones in *Figure 6.2*.

By extracting the MFCC of a 7 seconds song, the number of columns (audio frames) of the spectrogram decreases. Applying *type 1* MFCC parameter values, a matrix with **128** rows and **300** columns (*Figure 6.4*) is obtained, while applying *type 2*, a matrix with **128** rows and **150** columns (*Figure 6.5*) results instead. It is possible to see that MFCC of *Figure 6.4* are more compact and they have greater power values as average that the ones in *Figure 6.5*.

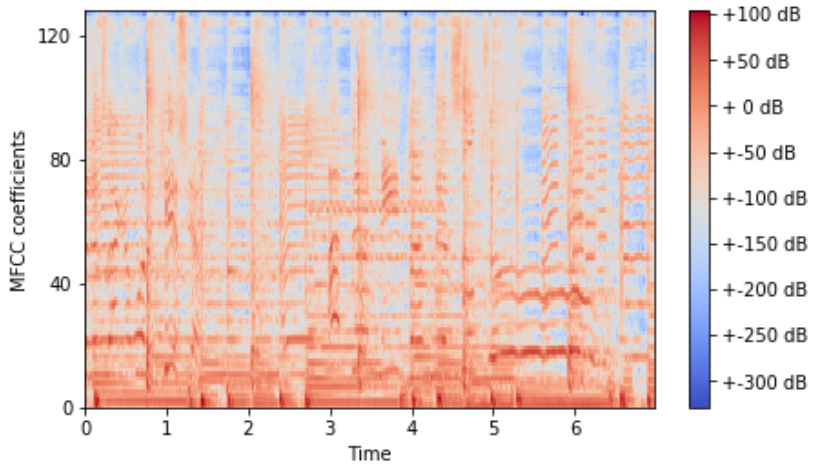


Figure 6.4. MFCC spectrogram of a 7 seconds audio fragment, having as parameters for computing 128 MFCC a window size of 1,024 samples and a hop size of 512 samples

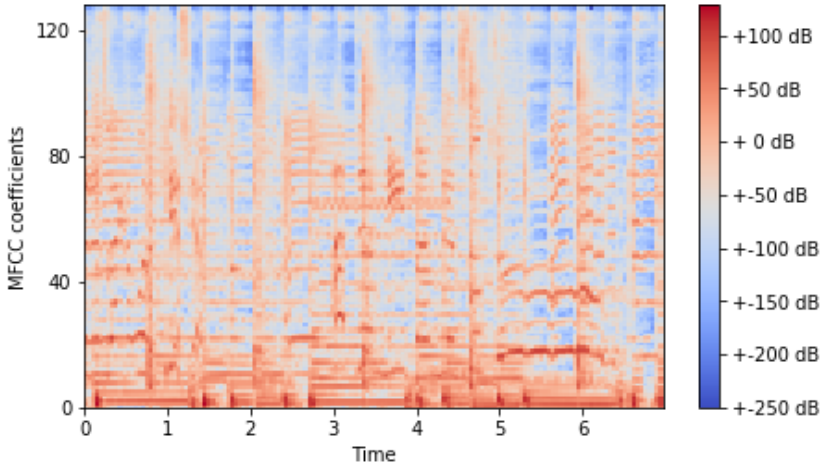


Figure 6.5. MFCC spectrogram of a 7 seconds audio fragment, having as parameters for computing 128 MFCC a window size of 2,048 samples and a hop size of 1,024 samples

For both *type 2* MFCC of 14 and of 7 seconds, the resulting matrices have less number of columns because the *type 2* window has more number of samples. This means that in each audio frame, more information was taken.

6.1.2. MFCC from h5 files

As explained in *chapter 4.2.2* the matrices with the precalculated "MFCC" found in the h5 files have 128 rows and 12 columns. Each of this matrices contains 12 MFCC of 24-48 seconds of a song.

With these "MFCC" matrices (spectrograms) are proceeded in the same way they did in the previous chapter: they are separated in smaller fragments to have more inputs of the network. As a result, there are:

- 2,358 MFCC spectrograms (128 rows and 12 columns) of 24-48 second fragments.
- 4,733 MFCC spectrograms (64 rows and 12 columns) of 12-24 second fragments.
- 9,476 MFCC spectrograms (32 rows and 12 columns) of 6-12 second fragments.

6.2. CNN architecture

A convolutional neural network is designed and trained to be capable of classifying the artist of a particular song.

6.2.1. Input of the network

The transpose of the MFCC spectrogram matrix (obtained in *chapter 6.1*) will be the input of the CNN. This matrix has to be the transpose because the columns have to represent the features (MFCC) of each observation (audio frames).

The dataset composed by all the MFCC matrices has to be splitted in train (75%) set, validation set (15%) and test set (10%). Because of not having enough data in order to partition it without losing important training (modelling) capacity, **cross validation** will be done over the train and validation set (the test set will still be a 10% of the total set).

Cross validation [51] is based on evaluating the network, ensuring that the results are independent of the separation of the data. It consists on splitting the data in two subsets, training the model with the train subset (75%) and validating it with the validation subset (15%). This will be done multiple rounds, using in each round a different partition. The validation results of each cross-validation round will be averaged, obtaining the final validation result of the network.

In this particular problem, the type of cross validation used is **K-folds cross validation** (*Figure 6.6*). This type of cross validation is characterized by never being the same data that composes the validation subset. K is the number of parts (folds) that the set is splitted into. To achieve the validation set of a 15% of the set, K will be 20 and the validation subset will be composed by 3 folds. Therefore the number of cross validation rounds will be six.

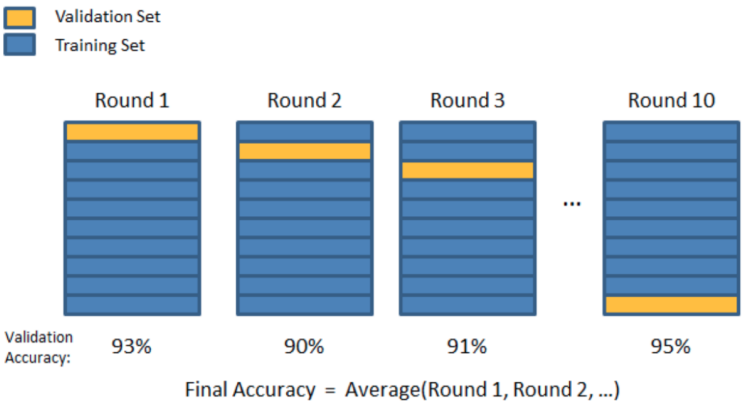


Figure 6.6. K-Folds cross validation [52]

6.2.2. CNN architecture

As explained in *chapter 5* there is no rule to determine the number of convolution + activation + pooling layers, number of hidden layers or number of neurons in each layer, that should be in the network in order to obtain the best result. This choice depends on the problem and is made by empirical experiments. Because of this, different CNN architectures will be evaluated, choosing the one that gives the best evaluation metric value.

The evaluation metric used as a criterion to select the best CNN architecture is the **accuracy** (ACC). The CNN architecture with highest accuracy value will be the selected architecture to develop the artist classifier.

While designing the architecture of the network, it is important to have in mind the balance between necessary, but not too much model complexity to be able to get insights of the MFCC, so it is possible to classify the artist of the song.

At each layer of the network a different characteristic of the song would be discovered. Some of them are: vibrato, instruments detected, voice, melody...

At *chapter 3.3* all the basic concepts to understand CNNs have been explained. In this case the CNN will be applied to sound instead of image, so some considerations are important to take into account:

- The x and y values of the sound image (spectrogram) does not represent the same, as in the case of the images (pixels). In the sound case one axis represents time and the other frequency.
- The space position of the elements in the spectrogram is not important.

The **CNN architectures evaluated** are:

1. There are two 2D convolutional layers of 128 *filters* and 256 *filters* each one, with a *kernel size* of 3x3 and a *stride size* of 1. Both *convolutional layers* are followed by a *pooling layer* with a *pool size* of 2x2 and a *stride size* of 0. A *flatten layer*

follows the second *pooling layer* to convert the matrix in a vector. After this layers, there are two *fully connected layers* with 64 neurons and 20 neurons (number of classes) each one, with a *dropout rate* of 0.2 between them.

2. Same CNN architecture as number 1 but with a *dropout rate* of 0.4 instead of 0.2.
3. There are four 2D *convolutional layers* just over the time axis (they work like 1D convolutions) of 128 *filters*, two of 256 *filters* and another one of 512 *filters*. The four *convolutional layers* have a *kernel size* and a *stride size* of 4. The first *convolutional layer* is followed by a *pooling layer* with a *pool size* of 4x4 and a *stride size* of 0, while the rest *convolutional layers* are followed by a *pooling layer* with a *pool size* of 2x2 and also a *stride size* of 0. A *flatten layer* follows the second *pooling layer* to convert the matrix in a vector. After this layers, there are three *fully connected layers* with 1,536 neurons, two with 2,048 neurons and another one with 20 neurons (number of classes). This *fully connected layers* are connected with a *dropout rate* of 0.5 between them.
4. There are two 2D *convolutional layers* just over the time axis (they work like 1D convolutions), of 128 *filters* each one, with a *kernel size* and *stride size* of 6. Both *convolutional layers* are followed by a *pooling layer* with a *pool size* of 6x6 and 5x5 each, and a *stride size* of 0. A *flatten layer* follows the second *pooling layer* to convert the matrix in a vector. After this layers, there are two *fully connected layers* with 400 neurons and 20 neurons (number of classes) each, connected with a *dropout rate* of 0.5.

All this CNN are optimized through the **Adam optimizer** [53] (extension to stochastic gradient descent) and they will be trained to minimize the categorical cross entropy between the labels and the prediction probabilities from the audio signal. The cross entropy between two probability values are calculated by the following equation:

$$\mathrm{H}(p, q) = -\sum_{x} p(x)\, \log q(x) \tag{6.1}$$

6.3. Experiments

A code in Python [54] has been developed in order to train and validate, via cross validation, each CNN architecture having as input the different extracted MFCC (*chapter 6.1*).

The CNN networks have as training parameters 100 epochs and a batch size of 64.

Due to computational limitations (not having a GPU) the training of these networks takes a great amout of time. This is an enormous limitation to compare the results of the different network architectures and inputs because of not being able to obtain all of them.

The few results obtained after a long time of execution of the code are shown in *Table 6.1*. In addition, a quite interesting plot, with the cross validation results, has been obtained (*Figure 6.7*).

TABLE 6.1. CROSS VALIDATION RESULTS OF EACH CNN ARCHITECTURE WITH EACH MFCC MATRIX, EXTRACTED FROM THE MP3 FILES, AS INPUT

CNN architecture	**MFCC**	**Train accuracy**	**Validation accuracy**
Random model	**-**	-	0.049
1	**7 seconds, type 1**	0.486	0.051
2	**7 seconds, type 1**	0.413	0.049
3	**7 seconds, type 1**	0.491	0.053
3	**7 seconds, type 2**	0.492	0.041
3	**14 seconds, type 1**	0.491	0.019

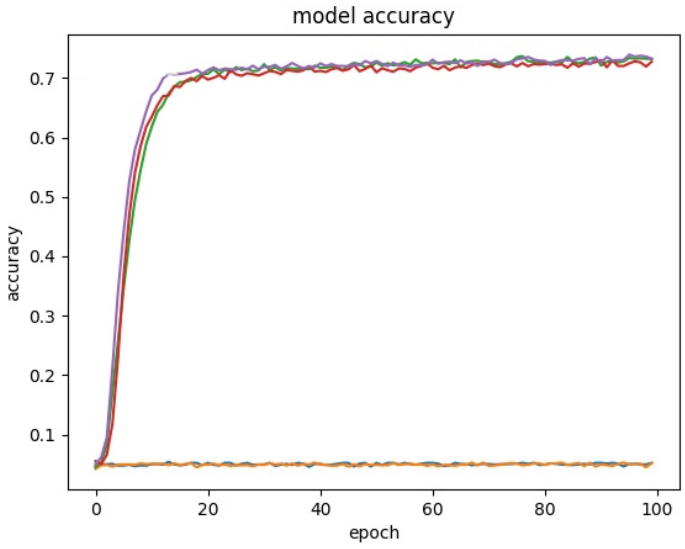


Figure 6.7. Train accuracy at each epoch of the CNN architecture number 1, having as input type 1 MFCC parameters over song fragments of 7 seconds

Figure 6.7 shows the train accuracy at each fold of the cross validation. It is possible to see that depending on the separation of the dataset in train and validation sets, the train accuracy has different values. With one type of separation, a train accuracy of 0.05 is

obtained (quite low), while with other separation, 0.7 train accuracy is obtained. This fact makes the overall train accuracy not high (*Table 6.1*), which is a problem.

In *Table 6.1* is possible to see that the validation accuracy values are quite low (almost equal to the random model ACC value). These low values are very likely to be because there are many classes as ouput (20 clases). Other reason could be that the CNN architectures are not optimal with that MFCC extracted in *Table 6.1*.

In *chapter 4.2.2* it was mentioned that the mp3 files downloaded via the 7Digital API were not neccesarily those of the original songs, therefore, it is very likely that this fact has affected the results whoen in *Table 6.1*.

7. NOVELTY DETECTION

As explained in the description of the state of the art, CF recommenders have a big limitation. This limitation is found due to the fact that CF is based on the number of times a user has listened to a song. If the song is new, for not having this information, it will never be recommended. This limitation is the so call **"cold-start"**.

This chapter presents a solution in order to be able to recommend also songs that nobody has listened (unpopular songs) but fit to the users tastes. This solution has been called ***novelty detection***. This solution is not implemented because of the lack of security about the test results of the CF recommender. Therefore, a research of the topic is presented.

Novelty detection is based on combining collaborative filtering and deep learning based techniques. Each technique is in charge of different functions:

1. **Collaborative filtering technique**: ALS is applied to the users historical data (played songs) to get the users and songs latent factors (deep explanation and development can be found in *chapter 5*). The latent factors obtained are projected into a shared low-dimensional space (latent space).
2. **Deep learning based technique**: a CNN has to be designed and trained to predict the latent factors of a new song. In other words, the network will predict the position of a song in the latent space. This CNN will have as input the MFCC spectrogram of an audio signal, and as number of output as latent factors chosen in CF phase. This CNN will be trained to minimize the **mean squared error** (**MSE**) between the latent factor vectors from the CF (ground truth) and its predictions (obtained by inputting the MFCC spectrogram). The CF latent factors must first be normalized, because all the vectors have to be on the same scale, causing it to be reduced the influence of song popularity in the network output.

If good results were achieved in the evaluation results over the test set of the CF recommender (*chapter 5.3.3*), it will be possible to implement *novelty detection*. Because this is not the case, *novelty detection* is a future step.

To implement *novelty detection* to this particular project problem, the following steps would have been carried out:

1. Extract MFCC of the audio signal songs of the CF recommender dataset.
2. Train the CNN of *chapter 6.4,* inputting the MFCC extracted in the previous step, and having as ground truth the normalized 130 latent factors that have been obtained in the CF recommender (*chapter 5.3*).
3. Evaluate the CNN performance to discover if it generalizes well or not. If not, this CNN architecture will not be the optimal and it should be changed.
4. All the latent factors of popular and unpopular songs are known (users and songs are projected into the latent space). Therefore, it would be possible to make

recommendations as for CF recommender (*chapter 5*). This are made by multiplying the users matrix by the songs matrix and recommending those songs with highest score (this would be the closest songs to the user in the latent space).

When these steps have been successfully completed, it would be possible to put this final recommender system in production. New songs will go through the CNN while the others will be known its latent factors because of doing ALS. As a result, there is in production a recommender able to recommend both, popular and unpopular songs, depending on users tastes. The workflow of this final song recommender system is shown in *Figure 7.1*.

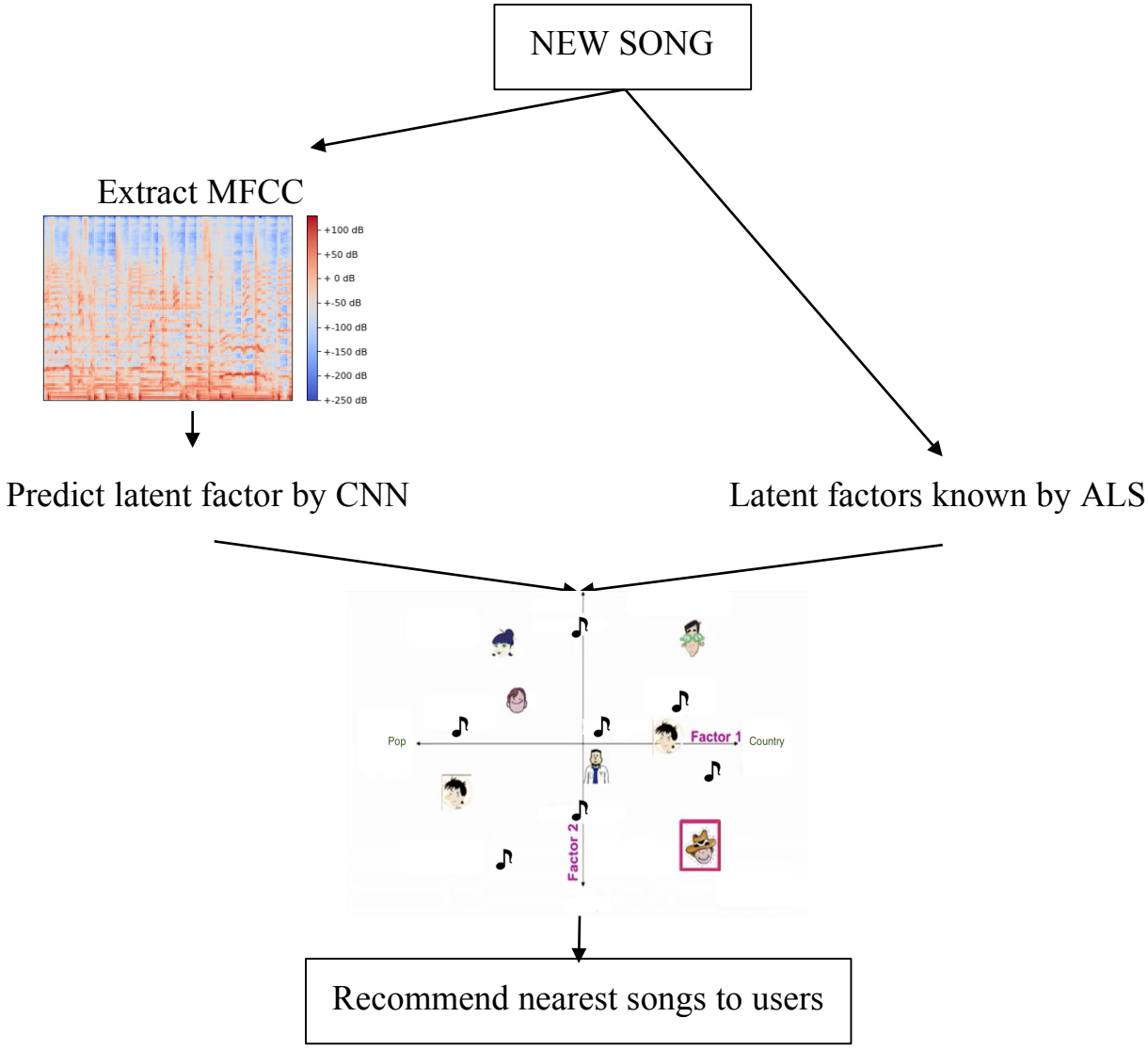


Figure 7.1. Flowchart of final recommender system in production

8. CONCLUSIONS AND FUTURE LINES

8.1. Conclusions

In this project, a song recommender system for music streaming platforms has been developed. The main idea was to develop a collaborative filtering recommender and integrate in it deep learning based technique to achieve a recommender that does not have the "cold-start" problem (the combination of these two techniques has been called *novelty detection*). Therefore, this system will recommend popular and unpopular songs that fit users tastes.

The dataset used to develop the CF recommender gives information in form of implicit feedback. As the feedback is implicit, it is not possible to rely 100% on test results. The test results obtained are not as good as expected, but this does not mean that the recommended lists do not fit users tastes. To have a real knowledge of this, it would be necessary to track users behaviour through the recommended lists. If the songs of these recommended lists have been listened a lot of times, this would mean that certainly the lists fit users tastes. If not, the obtained test results would be right. If the recommended lists actually do not fit user tastes, a possible way to fix it could be doing hyperparameters tuning by cross validation, instead of only over the validation set. If after doing this, the test results are not the optimal ones, this will mean that the dataset used does not have enough data to develop a RS, or that the dataset information itself is not valid for this task.

Due to the lack of security about the test results of the CF recommender, the novelty detection has not been implemented. This decision was made because it seemed unreasonable to develop a CNN to predict latent factors (deep learning based technique) whose ground truth (latent factors obtained by applying CF) is not correct. Despite this fact, different CNN architectures have been designed in order to compare them and select the one that gives best results. As a result, there will be a prepared network for future works, capable of learning and getting accurate results of song audio signals. The different networks have been designed for the artist classification task (*chapter 6*). In this task, also different MFCC have been extracted, in order to input the networks.

All the code of the CF recommender [46] and of the training and validation of the different CNN networks [54] is available in the public platform named Github.

To conclude, it is necessary to highlight that recommender system problems are very complex. There are too many variables to consider at the same time: did the user not listen a song because he thought he was not going to enjoy it? or because he did not know about its existence? If the song was not played a lot of times should it be assumed that the user did, or did not enjoy it?

Recommenders are not like classification or clustering problems, where just one algorithm has to be picked to solve the problem. A recommender is a whole ecosystem of algorithms working together and this have to work properly jointly.

It is believed that *novelty detection* is worth deepening in it. It is also considered technologically ahead (every enterprise wants to implement a recommender system in its e-commerce platform). I would like to continue with this project to achieve better results on CF recommender and, in this way, be able to implement novelty detection.

8.2. Future lines

During the realization of this project, possible future lines have been drawn up. These future lines should be taken into account for the improvement of what has been developed so far and for possible extensions in future projects. Some of them have been mentioned in the previous section. These are:

- To do hyperparameter tuning of the CF recommender by cross validation, to confirm if overfitting does not occur this way.
- To use a dataset from a streaming music platform. This would be more realistic that the Million Song Dataset.
- To acquire a GPU in order to execute the training of all the CNNs and in this way be able to compare them and select the best one.
- Research more about CNN architectures that best work with content-based recommenders, and test them. This should also be done with MFCC extraction.
- Implement *novelty detection* in order to have a recommender system that recommends popular and unpopular songs.
- Implement ALS using Spark to perform a distributed processing, having as a result a faster training of the CF recommender.

9. BUDGET

Finally, an estimation of the total budget that would lead to the completion of this project will be calculated. This estimation is for one year. The development of the entire project has been done by two persons:

- **Senior consultant** (Fernando Silos de la Calle): project leader in charge of guiding the project ambitions and the junior consultant, and to give technical knowledgement and solutions to the adversities found. His estimated salary is 30€ per hour.
- **Junior consultant** (Flavia García Vázquez): in charge of investigating about the topic and developing and evaluating the whole project. Her estimated salary is 15€ per hour.

To train de CNN it will be necessary a **GPU** in order to have a reasonable network training time. With a GPU, the network training takes 30 minutes while without it, it could take more than one day. The GPU selected is the NVIDIA TITAN Xp.

To sell the service of songs recommendation to entreprises that have music streaming platforms, some workshops would be done. For this year there would be 5 workshops estimated. For each workshop, there would be a catering with a price of 30 € each.

When this service is sold, it is necessary to have a **maintenance manager** in charge of supervising that the recommender is working correctly and if something goes wrong, he/she would be the one in charge of fixing it. His/her estimated salary is 5 € per hour.

In *Table 7.1* all mentioned costs are shown and the total cost has also been calculated.

TABLE 7.1. ESTIMATION OF THE PROJECT BUDGET

Concept	**Cost**	**Units**	**Time**	**Total**
Senior consultant	30 €/hour	1	24 hours	720€
Junior consultant - Investigation	15 €/hour	1	300 hours	4,500 €
Junior consultant - Development	15 €/hour	1	180 hours	2,700 €
Maintenacne manager	5 €/hour	1	1 hour/day * 251 laboral days/year	1,255 €
NVIDIA TITAN Xp	498.90 €	1	-	1,299 €
Catering	30 €	5	-	150 €
			Global total:	**10,624 €**

Therefore, with these estimations made, the **overall cost** of the project for a year would amount to **10,624 €**.

BIBLIOGRAPHY

[1] J. Ewing, «How MP3 Was Born,» *Bloomberg BusinessWeek,* 5 March 2007.

[2] M. Gowan, «Requiem for Napster,» 18 05 2002. [En línea]. Available: Pcworld.com.

[3] B. Kinsella, «IFPI data shows streaming growth, but how much?,» [En línea]. Available: https://xappmedia.com/ifpi-data-shows-streaming-growth/.

[4] G. Kreitz y F. Niemela, «Spotify--large scale, low latency, P2P music-on-demand streaming,» de *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, 2010.

[5] J. Lacort, «Las cifras de Spotify: un magnífico gigante que sueña con ser rentable,» [En línea]. Available: https://hipertextual.com/2017/03/las-cifras-de-spotify.

[6] D. Lee y K. Hosanagar, «Impact of recommender systems on sales volume and diversity,» 2014.

[7] P. Lops, M. De Gemmis y G. Semeraro, «Content-based recommender systems: State of the art and trends,» de *Recommender systems handbook*, Springer, 2011, pp. 73-105.

[8] R. Zhang, Q.-d. Liu, C. Gui, J.-X. Wei y H. Ma, «Collaborative filtering for recommender systems,» de *Advanced Cloud and Big Data (CBD), 2014 Second International Conference on*, 2014.

[9] C. A. Gomez-Uribe y N. Hunt, «The netflix recommender system: Algorithms, business value, and innovation,» *ACM Transactions on Management Information Systems (TMIS),* vol. 6, nº 4, p. 13, 2016.

[10] S. Zhang, L. Yao y A. Sun, «Deep learning based recommender system: A survey and new perspectives,» *arXiv preprint arXiv:1707.07435,* 2017.

[11] E. Davidson, «In machine learning, how can we determine whether a problem is linear/nonlinear?,» [En línea]. Available: https://www.quora.com/In-machine-learning-how-can-we-determine-whether-a-problem-is-linear-nonlinear.

[12] F. Rosenblatt, «The perceptron: a probabilistic model for information storage and organization in the brain.,» *Psychological review,* vol. 65, nº 6, p. 386, 1958.

[13] G. Yassin, «Build Simple AI .NET Library - Part 3 - Perceptron,» [En línea]. Available: https://www.quora.com/In-machine-learning-how-can-we-determine-whether-a-problem-is-linear-nonlinear.

[14] «Chapter 1. Introduction to Artificial Neural Networks,» O'reilly.

[15] M. Minsky y S. A. Papert, «Perceptrons: An Introduction to Computational Geometry,» 1969.

[16] J. Schmidhuber, «Deep learning in neural networks: An overview,» *Neural Networks,* vol. 61, pp. 85-117, 2015.

[17] G. Zaccone, Getting Started with TensorFlow, O'reilly.

[18] D. E. Rumelhart, G. E. Hinton y R. J. Williams, «Learning internal representations by error propagation,» 1985.

[19] B. Brown, «On chain rule, computational graphs, and backpropagation,» [En línea]. Available: http://outlace.com/on-chain-rule-computational-graphs-and-backpropagation.html.

[20] A. Cauchy, «Méthode générale pour la résolution des systemes d'équations simultanées».

[21] A. Chavan, «What is Stochastic Gradient Descent?,» [En línea]. Available: https://www.quora.com/What-is-Stochastic-Gradient-Descent.

[22] H. Zulkifli, «Understanding Learning Rates and How It Improves Performance in Deep Learning,» [En línea]. Available: https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10.

[23] V. Plevris y P. Asteris, «Modeling of Masonry Failure Surface under Biaxial Compressive Stress using Neural Networks. Construction and Building Materials.,» 2014, p. 447–461.

[24] A. Karpathy, «Yes you should understand backprop,» [En línea]. Available: https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b.

[25] S. Hochreiter, «Untersuchungen zu dynamischen neuronalen Netzen,» *Diploma, Technische Universität München,* p. 1, 1991.

[26] G. H. a. al., «Improving neural networks by preventing co-adaptation of feature detectors,» 2012.

[27] N. S. e. al., «Dropout: A Simple Way to Prevent Neural Networks from Overfitting,» 2014.

[28] S. Raval, «Convolutional neural network,» [En línea]. Available: https://github.com/llSourcell/Convolutional_neural_network/blob/master/convolutional_network_tutorial.ipynb.

[29] D. H. Hubel y T. N. Wiesel, «Receptive fields and functional architecture of monkey striate cortex,» *The Journal of physiology,* vol. 195, nº 1, pp. 215-243, 1968.

[30] Y. LeCun, Y. Bengio, Y. Bengio, P. Haffner y L. Bottou, «Gradient-based learning applied to document recognition,» *Proceedings of the IEEE,* vol. 86, nº 11, pp. 2278-2324, 1998.

[31] V. V. Krasnoproshin y S. V. Ablameyko, Pattern Recognition and Information Processing: 13th International Conference, Prip 2016, Minsk, Belarus, October 3-5, 2016, Revised Selected Papers, vol. 673, Springer, 2017.

[32] S. C. «Deep learning for complete beginners: convolutional neural networks with keras,» [En línea]. Available: https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html.

[33] J. Salomon y B. Schoen Phelan, «Lung Cancer Detection using Deep Learning,» 2018.

[34] M. Chablani, «Deep learning concepts. PART 1,» [En línea]. Available: https://towardsdatascience.com/deep-learning-concepts-part-1-ea0b14b234c8.

[35] A. Ghosal, R. Chakraborty, B. C. Dhara y S. K. Saha, «Song/instrumental classification using spectrogram based contextual features,» de *Proceedings of the CUBE International Information Technology Conference*, 2012.

[36] J. Singh, «genreXpose github project,» [En línea]. Available: https://github.com/jazdev/genreXpose.

[37] T. Bertin-Mahieux, D. P. Ellis, B. Whitman y P. Lamere, «The Million Song Dataset,» de *Ismir*, 2011.

[38] Y. Koren, R. Bell y C. Volinsky, «Matrix factorization techniques for recommender systems,» *Computer,* vol. 42, nº 8, 2009.

[39] I. Dewancker, «SigOpt for ML: Bayesian Optimization for Collaborative Filtering with MLlib,» [En línea]. Available: http://blog2.sigopt.com/post/148703071378/sigopt-for-ml-bayesian-optimization-for.

[40] Y. Hu, Y. Koren y C. Volinsky, «Collaborative filtering for implicit feedback datasets,» de *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, 2008.

[41] J. Bacardit, « Data mining techniques and applications,» [En línea]. Available: http://slideplayer.com/slide/4530010/.

[42] C. D. Manning, P. Raghavan y H. Schaitze, «Evaluation in information retrieval,» de *Introduction to Information Retrieval*, Cambridge, Cambridge University Press, 2008, pp. 139-161.

[43] M. Malaeb, «Recall and Precision at k for Recommender Systems,» [En línea]. Available: https://medium.com/@m_n_malaeb/recall-and-precision-at-k-for-recommender-systems-618483226c54.

[44] S. S. «Rank-Based Measures,» [En línea]. Available: https://blog.semanticscholar.org.

[45] K. Järvelin y J. Kekäläinen, «Cumulated gain-based evaluation of IR techniques,» *ACM Transactions on Information Systems (TOIS),* vol. 20, nº 4, pp. 422-446.

[46] F. Garcia, «Collaborative filtering recommender,» [En línea]. Available: https://github.com/FlaviaGarcia/Collaborative-Filtering-Recommender-System.

[47] Benfred, «Implicit library,» [En línea]. Available: https://github.com/benfred/implici.

[48] S. B. Davis y P. Mermelstein, «Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences,» de *Readings in speech recognition*, Elsevier, 1980.

[49] B. Logan, «Mel Frequency Cepstral Coefficients for Music Modeling,» de *ISMIR*, 2000.

[50] K. Song, M. Griffin y X. Liang, «Shazam but Magic,» 2017. [En línea]. Available: https://machineramblings.wordpress.com/2017/05/09/shazam-but-magic/.

[51] P. A. Devijver y J. Kittler, «Pattern Recognition: A Statistical Approach,» *Int.J.Remote Sens.,* nº 2, 1982.

[52] C. McCormick, «K-Fold Cross-Validation, With MATLAB Code,» [En línea]. Available: http://mccormickml.com/2013/07/31/k-fold-cross-validation-with-matlab-code/.

[53] D. P. Kingma y J. Ba, «Adam: A method for stochastic optimization,» 2014.

APPENDIX A. ACRONYMS

MSD Million Song Dataset

NN Neural Network

DNN Deep Neural Network

CNN Convolutional Neural Network

mAP mean Average Precision

AUC Area Under the Curve

ROC Receiver Operating Characteristic

DCG Discounted Cumulative Gain

nDCG normalized Discounted Cumulative Gain

MF Matrix Factorization

GD Gradient Descent

MLP Multi-Layer Perceptron

PCA Principal Component Analysis

MFCC Mel Frequency Cepstral Coefficient

ALS Alternative Least Squares

CF Collaborative Filtering

MAE Mean Absolute error

MSE Mean Square error

RMSE Root Mean Square error

FPR False Positive Rate

TPR True Positive Rate

FNR False Negative Rate

TNR True Negative Rate

ReLU Rectified Linear Unit

LTU Linear Threshold Unit

ACC Accuracy

AWS Amazon Web Services

API Application Programming Interface

RS Recommender System

2D Two-Dimensional

1D One-Dimensional