UNIVERSITÀ DI PISA

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica per l'economia e per l'azienda

(Business Informatics)

TESI DI LAUREA

Predicting and explaining the popularity of songs with data mining

RELATORE:

Prof.ssa Anna MONREALE

CORRELATORE:

Dott. Luca PAPPALARDO

CANDIDATO:

Massimiliano CAMPAGNA

ANNO ACCADEMICO 2016-17

Alla mia famiglia
e a Fabiana
per avermi sostenuto
in questi anni

Contents

Abstract

Data Mining techniques are widely used in the analysis of musical items. In this thesis we study the factors affecting the popularity of songs and given an answer to the so-called "four chord" myth, according to which all popular songs consists of the same four chords. The work illustrated in the thesis covers all the stages of the Knowledge Discovery in Databases process. In particular, data on music songs are collected and used to create several predictive models using data mining algorithms. The best model is interpreted and tested with specific instances to better understand the origina of popularity of music songs.

Introduction

A rock guitarist plays
three chords to a million people
and a jazz guitarist plays
a million chords to three people

Today the amount of data that is being generated is far more than can be handled, almost every single activity or interaction leaves a trail that somebody somewhere captures, stores and analyses. The size of this data has gone beyond human-sense capabilities, in fact it is quite impossible for a person to detect patterns just by looking at the data. At this point *Data mining* comes into play.

Data mining is the computing process of discovering patterns in large data sets involving methods at the intersection of machine learning, statistics, and database systems. The overall goal of the data mining process is to extract information from a data set and transform it into an understandable structure for further use. Additionally, this process is only a part of the so-called *Knowledge discovery in databases* process also known as *KDD*. There are several formal definitions of KDD, all agree that the intent is to harvest information by recognizing patterns in raw data.

The research area of *Music Information Retrieval* has gradually evolved to address the challenges of effectively accessing and interacting large collections of music and associated data, such as styles, artists, lyrics, and reviews. Several music researches can be found on Internet based on the data mining techniques. Tao Li, professor in Computer Science at *Florida International University*, formally defines the concept of *Music Data Mining* also called *MDM*. He shows how to use data mining techniques in order to solve several problems and he presents some social aspects. In particular, some of these studies use classification algorithms to create models able to detect mood or the genre of songs. Additionally, there are also projects that focus on classification and recognition of musical instruments. Therefore, all these studies feed on the importance

of the of music in our life and they are aimed to improve music marketing and music creation by artists, as well as to suggest songs to users based on their mood and tastes.

According to marsbands.com, there are at least 97 million songs in the world. Of course, these are only the songs officially released. This number could reach 200 million songs, if the songs which everyone knows or if the old Celtic songs are considered. Generally speaking, the music world has its roots in the *music theory*. The music theory can be defined as *«the study of the practices and possibilities of music»*. According to Fallows and David from *The Oxford Companion to Music*, this term can be used in three main ways. The first is called *rudiments*, currently taught as the elements of notation, of key signatures, of time signatures, of rhythmic notation, and so on. The second is the study of writings about music from ancient times onwards. The third is an area of current musicological study that seeks to define processes and general principles in music. In any case, the music theory is frequently concerned with describing how musicians and composers make music.

The *myth of the 4 chords* is well known on Internet, particularly, by those who play an instrument. This myth asserts that all popular songs can be played by only 4 chords. A chord, in music, is any harmonic set of three or more notes that is heard as if sounding simultaneously. These chords do not need actually be played together: arpeggios and broken chords may, for many practical and theoretical purposes, constitute chords. The most frequently encountered chords are *triads*, so called because they consist of three distinct notes and a series of chords is called a *chord progression*. Although any chord may in principle be followed by any other chord, certain patterns of chords have been accepted as establishing key in common-practice harmony. For simplicity, the chords are commonly numbered by means of Roman numerals.

At first glance, it seems strange that many popular songs should have the same chords. After all, with 12 notes to choose from, and a choice of *major* mode or *minor* mode, there should be thousands of chord progressions that could be played. Consequently, it is important to understand what is the real impact of chord progressions and if they could influence the popularity of songs.

This thesis focuses on songs popularity in order to find out unknown patterns that could describe the relationship among songs popularity, chord progressions and other music features. The whole project covers all the stages of the KDD process since several regression and classification algorithms were used in order to achieve the goal. Finally,

a data driven software was created in order to give the users the opportunity to employ our models.

This project can be subdivided into 4 stages and each one is described in the following chapters:

- The first chapter concerns the *Data collection* stage. This chapter covers and describes the whole process performed in order to create the dataset which will be used in the analysis.
- The second chapter concerns the *Data understanding* stage. The dataset already obtained was analyzed under the statistical point of view in order to better understand the meaning of the data collected.
- The third chapter focus the attention on the Data mining techniques.This chapter describes the creation of regression and classification models
- The fourth chapter concerns the interpretation and the simulation of the models. In particular, this chapter describes the behavior of the models created in terms of songs popularity prediction.

Chapter 1

DATA COLLECTION

1 Summary

This chapter explains how the data collection process was performed. In particular, this process is made up of two stages. In the first stage the data are retrieved using the *Hooktheory's API*, then the outcomes are used as input in the second phase to collect data from *Spotify* using its Web API[1]. The Hooktheory's API allows to retrieve basic information about songs that use a specific chord progressions such as song title and artist's name. These information are used to collect data from Spotify such as for example songs popularity, artist popularity, mode, key and so on. The final outcome is a tab separated text file or *.tsv*[2] and each row in this file is a *Tuple*[3] that represents a unique track.

2 Data collection process

Nowadays, there are several databases on Internet that contain information about music. However, these databases often contain only a subset of all the necessary information to achieve the goal. This is the reason why the data collection process involves two different sources of data. Therefore, the data collection process can be divided into

[1] Web API is an API over the web which can be accessed using HTTP protocol. An application programming interface (API) is a set of subroutine definitions, protocols, and tools for building software and applications.

[2] TSV file is a simple text format for storing data in a tabular structure, e.g., database table or spreadsheet data, and a way of exchanging information between databases

[3] A tuple is a finite sequence of attributes, which are ordered pairs of domains and values.

2 stages: the first stage concerns the data collection from the Hooktheory website and the second one concerns data from Spotify.

The process begins to retrieve data from Hooktheory, particularly, the famous chord progressions were used to detect and collect songs that use some particular harmonic structures. Once the requests were performed, the data collected were stored in a file which will be used to retrieve data from Spotify. This last file is made up of three information: *artist's name*, *song title* and *Chord progressions list*. Each row in this file represents a unique song with its harmonic structures.

The second stage uses the pairs made up of artist's name and song title to collect data from Spotify. In particular, the Spotify Web API were used to retrieve information about audio features, popularity, tonality and mode of songs. This last stage allows to get the final dataset which will be used for the analysis. The following image summarizes the whole data collection process.

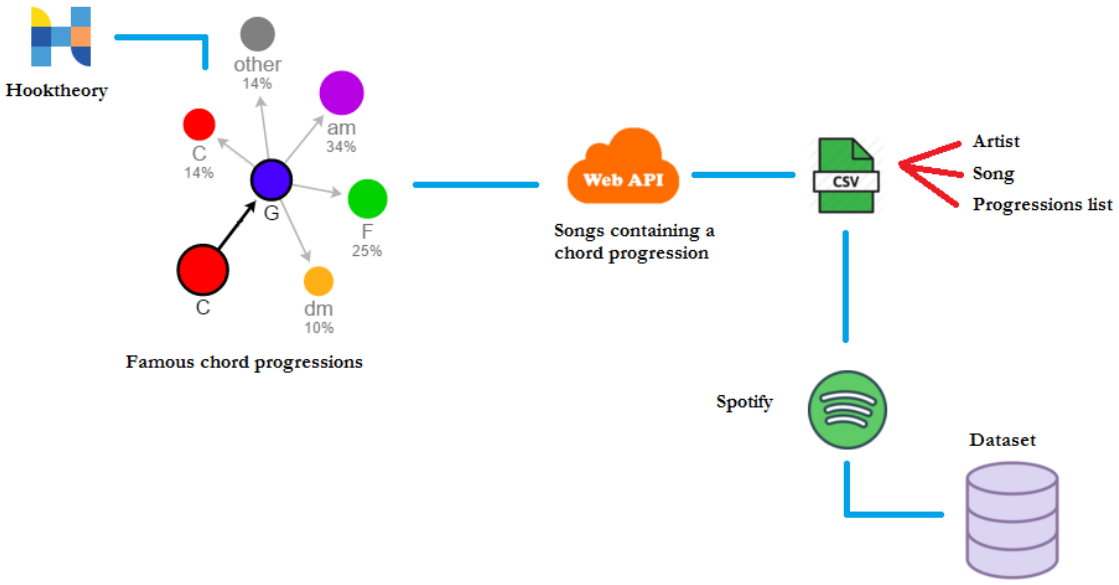

Figure 1.1: Data collection process

3 Hooktheory case study

The Hooktheory website provides users with a platform to learn how to write the music. The users can also select a song and look at its tab. In *progressions* section there is a page dedicated to *famous chord progressions* and one dedicated to the *progression builder*. The builder assumes a central role in the site, in fact, its purpose is to provide the list of songs which have the same chord or progressions sequence in a part or in the whole song. In addition, the authors also provide a Web API which implements different features in order to interact with their *Database*. As mentioned before, famous chord progressions are used to collect data by means of the Web API. This API exposes the chord probability data and it contains two different endpoints: one for *next chord probabilities* and one for *songs containing a chord progression*. It is necessary remember that each API request must be made over an *HTTPS* connection and each one also must begin with the url `https://api.hooktheory.com/v1/` or otherwise the request is rejected. For more details on the API, the link `https://www.hooktheory.com/api/trends/docs/` refers to the official documentation.

The sources codes used to collect data from Hooktheory are stored at the following link HookTheory on Github. This folder contains two different files: the first one is called *1_ request.py* and it is used to send requests and to save the respective results whereas the other one is called *2_ extract.py* which is used to process the outcomes in order to remove redundancy. Consequently, the collection process can be divided into two phases.

The first phase consists in sending different requests to the Hooktheory server. In particular, the *1_ request.py* takes as input the file *note.txt* that contains the list of famous chord progressions and the file *login.json*. This last file stores the credentials which are used to perform the login procedure. Once the login procedure was performed, the Hooktheory server send a response in *JSON* format. This message contains the *activkey* property which is the *HTTP Bearer Token* and its value will be used in order to authenticate future requests.

As mentioned before, the API provides two different endpoints: the first one is called *next chord probabilities* and it provides a list of chords and their probability whereas the second one is called *songs containing a chord progression*. The data collection process is focused on this last powerful endpoint. In particular, this endpoint provides a list of songs related to a chord progression. The entry point for songs containing a

chord progression is the following:

Hooktheory API songs containing a chord progression

```
GET trends/songs?cp=4,1
```

The *cp* parameter stand for *child path* and it is essentially a chord progression. In general, the response is divided into *pages*. Each page can contain a list of 20 songs and it is necessary to specify the page number in the request in order to retrieve the next page of results. Finally, the response is conform to the *JSON* format and so each song can be easily processed and stored.

Hooktheory API songs containing a chord progression response

```
[
    {
            "artist":"Adele",
            "song":"Someone Like You",
            "section":"Chorus",
            "url":"http://local.www.hooktheory.com/.."
    },
    ...
]
```

The collection process proceeds with the request phase. The chord progressions, stored in the file *note.txt*, can be considered as single requests. There are 21 chord progressions and each one could have n-pages of results. In addition, the Hooktheory API limits requests to 10 every 10 seconds and therefore each request was delayed before to be sent to the server. The limitations on requests, the pagination of the outcomes, the connection speed and the server response speed can be considered as the bottleneck of the whole process, even if each request was parallelized to speed the process up. This is the reason why the entire process lasts about 3 hours.

Finally, the outcomes of the first phase was stored in a file called *requested.txt* which will be used as input in the second phase.

Once the data were collected from Hooktheory, the second phase concerns the creation of the file which will be used to retrieve data from Spotify. As seen before, the *2_ extract.py* file takes the *requested.txt* file as input in order to extract informations about songs. In particular, this last file is a collection of objects and each one is made

up of 4 properties:

- artist, the artist's name;
- song, the song title;
- section, the part of the song in which the chord progression appears;
- pattern, the chord progression.

Once the input was loaded, the software scans the outcomes in order to collect the pairs made up of artist's name and song title from each chord progression and from each page. Next, these pairs were inserted in a set in order to remove redundancy. Finally, the software associates each pair with the list of the chord progressions in which pairs appear and the *input.csv* file is produced. This last file will be used as input to retrieve data from Spotify and the following table shows the outcomes just obtained.

Artist's name	Song title	Chord progressions list
Big Dope P	Hit Da Blokk	['1,5,6,4']
ClariS	Connect	['6,5,4,5']
Daughtry	Home	['1,4,6,5']
Alex Lloyd	Amazing	['1,5,6,4']
Klingande	Punga	['1,5,6,4']
Jason Derulo	Trumpets	['1,4,6,5']
Blink 182	Bored To Death	['1,5,6,4']
Taylor Swift	22	['6,5,4,5']
Teen Beach Movie	Can't Stop Singing	['4,b4,1', '1,6,4,5']
Jimi Hendrix	All Along The Watchtower	['6,5,4,5']
Shakira	Waka Waka	['1,5,6,4']
Britney Spears	Lucky	['1,6,4,5']
Oasis	Don't Go Away	['6,5,4,5']
Duran Duran	Come Undone	['6,5,4,5']
Avicii	Father Told Me	['1,5,6,4']
Radiohead	Karma Police	['4,16,2']
Danny McCarthy	Silver Scrapes	['1,5,6,4']
The Gaslight Anthem	Rollin' And Tumblin'	['1,5,6,4']
Meaghan Smith	It Snowed	['4,b4,1']

The chord progressions list field has a special syntax which is explained at the following links: trends-api-chord-input and vizualitation-of-all-chord-progressions.

4 Spotify case study

Nowadays Spotify is the most popular software for music in the world, in fact, it has a huge songs database. This database contains many information on artists, tracks, albums and also audio features, thus the data collection process uses this database in order to create the final dataset of this project. The data were retrieved from Spotify by means of its Web API which are implemented in different programming languages. In particular, the *spotipy* interface was used. This implementation is a lightweight Python library that provides different features in order to interact with the Spotify server. For more details on the library, the link `https://spotipy.readthedocs.io/en/latest/` refers to the official documentation.

As mentioned before, this phase begins once the collection process from Hooktheory was completed. The sources codes used to collect data from Spotify are stored at the following link `https://github.com/jonpappalord/music_analysis/tree/master/Spotify/` on Github. This folder contains many files, some of these are the outcomes and others are the sources codes implemented in order to collect data from Spotify. In particular, the python project is divided in the following files:

- *work.py*, the main file;
- *spotify_request.py*, the file used to send each request to the Spotify server;
- *utils.py*, the file used to process and to extract data from each request;
- *db_spotify.py* and *tp_spotify.py*, the files used to locally store information at runtime.

The data collection process begin with the login procedure but this task requires the user credentials and the key released by Spotify once the application was registered on its domain. The key obtained from Spotify is necessary to send requests to its Server. The file *login.json* stores these information which will be used to perform the login procedure.

The Web API provides several endpoints in order to retrieve information about songs. These can be classified under two main classes: the first class concerns the so-called *look up functions* whereas the other one concerns the *search functions*. In general, the *look up functions* are quite faster than the *search functions* because they use *element's IDs* to speed the data collection process up. However, Spotify limits the number of elements that can be retrieved simultaneously by using the multiple look

up function. In addition, it is necessary to know a priori which is the element's ID in order to obtain its information. On the other hand, the *search functions* overcome this problem, in fact, it provides information about artists, albums, tracks or playlists that match a keyword string. The following example shows how to sent a request by means of a query.

Spootify API: Track search

```
1 GET       q=track:Mamma%20Mia%20artist:abba&type=track
```

The Spotify Server returns a list of objects in the JSON format and each one has several information about the element of the query. The `https://developer.spotify.com/web-api/search-item/` provides more details about filtering options and information retrieved.

Once the login procedure was performed, the information retrieved from Hooktheory were used to perform the *tracks search function*. Therefore, the pairs artist's name and song title were used to collect tracks information such as the *song popularity*, the *track id*, the *artist id* and the *album id*. These outcomes were stored in a file called *1_tracks.json* whereas the file *missed.txt* contains the songs which were not found on Spotify.

As seen before, the element's id can be used to retrieve data by means of the *look up function*. Consequently, the *track id*, the *artist id* and the *album id* were used to collect their information and these outcomes were stored in different files. The last task concerns the creation of the dataset which will be used for the analysis process. The file called *utils.py* processes each response and then it creates the so-called tuples of songs. Finally, the dataset were stored in the file called *SPOTIFY_DB.tsv* and the following table shows five instances of this dataset.

track_id	title	artist_name	popularity	artist_popularity	release_date	key	acousticness	danceability
4ye3W4xN	dance in the dark	lady gaga	46	86	2009	8	2.99e-05	0.645
2raJLzvN	animals	muse	52	78	2012	3	0.00571	0.446
6OmApaLQ	doctor worm	they might be giants	33	55	1998	6	0.0826	0.537
5xEM5hIg	complicated	avril lavigne	71	75	2002	5	0.0572	0.585
64yrDBpc	21 guns	green day	72	82	2009	5	0.0518	0.268

duration_ms	energy	instrumentalness	liveness	loudness	mode	speechiness	valence	patterns_list
289013	0.768	4.49e-05	0.276	-6.211	1	0.036	0.102	['1,5,6,4']
262813	0.804	0.796	0.696	-6.906	0	0.0307	0.596	['6,642,4']
181533	0.633	0	0.0755	-5.903	1	0.0306	0.719	['1,6,4,5']
244507	0.776	7.74e-06	0.3	-5.898	1	0.0459	0.43	['1,5,6,4', '1,6,4,5']
321093	0.742	0	0.626	-4.939	1	0.0355	0.413	['1,5,6,4', '1,5,4,5', '1,56,6,5']

Chapter 2

DATA UNDERSTANDING

1 Summary

This chapter concerns the data understanding process. In particular, the data collected are analyzed in order to better understand their meaning. First of all, the attributes were explained by defining their domain and providing a brief description. Then, two different tasks were performed: the first one concerns statistical analysis over the attributes such as distribution and frequency; the second one concerns queries ad-hoc performed in order to better understand the behavioir of songs popularity.

2 Data exploration process

Once the Data collection process was performed, the next phase concerns the Data understanding process. The dataset seen before contains over 900 instances and 20 attributes. In particular, each row represents a unique song which is made up of several features. These information could be divided in two classes: the first class concerns descriptive information and the second one refers to some technical details on songs. In general, the data understanding process firstly performs some statical analysis on the attributes and then it proceeds to perform ad-hoc analysis in order to better understand the behavior of the songs popularity. This process was performed only on a subset of the available attributes, hence the attributes selected could be those chosen in the models creation process. These analysis were performed using the *Jupyter notebook* software and they can be found at the following link Analysis/DataUnderstanding/-data_understanding_1. Finally, the following table summarizes the information of

each attribute, in particular, it shows the attributes names, their domain and it gives a brief description.

No.	Attribute name	Description
1	track_id	The Spotify ID of a song.
2	title	Song's title.
3	artist_name	The artists who performed the track.
4	popularity	The popularity of the track (0 - 100).
5	artist_popularity	The popularity of the artist (0 - 100). It is calculated using the popularity of all the artist's tracks.
6	release_date	The date the album was first released.
7	genres	The list of the genres used to classify the album. (could be empty)
8	key	The key is an integer (0 - 11) and it is conform to the Pitch Class notation. (0 = C, 1 = $C_{\#}$ ...)
9	acousticness	A confidence measure whether the track is acoustic. (0.0 - 1.0) 1.0 represents high confidence the track is acoustic.
10	danceability	Danceability describes how suitable a track is for dancing based on a combination of musical elements. (0.0 - 1.0) A value of 0.0 is least danceable.
11	duration_ms	The duration of the track in milliseconds.
12	energy	Energy represents a perceptual measure of intensity and activity. (0.0 - 1.0) Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale
13	instrumentalness	Predicts whether a track contains no vocals. (0.0 1.0) "Ooh" and "aah" sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly "vocal". The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content.
14	liveness	Detects the presence of an audience in the recording. (0.0 - 1.0) Higher liveness values represent an increased probability that the track was performed live.

No.	Attribute name	Description
15	loudness	The overall loudness of a track in decibels (-60 - 0 dB). Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude).
16	mode	Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.
17	speechiness	Speechiness detects the presence of spoken words in a track. (0.0 - 1.0) The more exclusively speech-like the recording the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech. Values below 0.33 most likely represent music and other non-speech-like tracks.
18	tempo	The overall estimated tempo of a track in beats per minute (BPM).
19	valence	A measure describing the musical positiveness conveyed by a track. (0.0 - 1.0) Tracks with high valence sound more positive (e.g. happy), while tracks with low valence sound more negative (e.g. sad)
20	patterns_list	List of the famous chord progressions which are in a song.

3 Statistical analysis

As mentioned before, the data understanding process begins with the statistical analysis. In particular, these analysis focus the attention on distribution and frequency of some attributes. The *key* attribute was firstly analyzed because this attribute concerns the tonality of a song. In general, tonality is a musical system that arranges pitches or chords to induce a hierarchy of perceived relations, stabilities, and attractions. In this hierarchy, the individual pitch or triadic chord with the greatest stability is called the tonic. The root of the tonic chord is considered to be the key of a piece or song. Thus a piece in which the tonic chord is C major is said to be in the key of C. This attribute can have a value between 0 and 11, in fact, each value represents a unique key. The first question concerns the frequency of each key in order to show which is the most frequently used. Each value of the *key* attribute was replaced with the relative letter in the English notation and the following image shows their frequency.

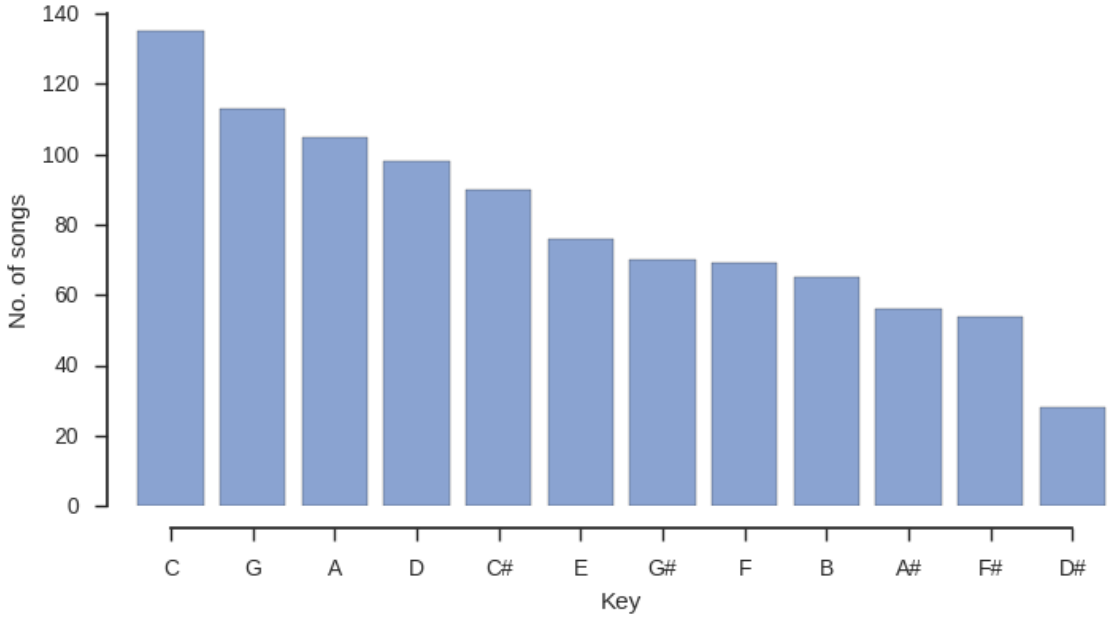

Figure 2.1: Frequency of each key in descending order

These outcomes show that the tonality most frequently used is the C followed by G, A and D. In general, the choice among several tonalities frequently depends on the ability of the musician. A song played in the tonality of C is simpler than one played on $D_{\#}$.

The tonality can be divided in two main mode, *major* and *minor*. The mode determines the mood of a song which will be, for example, cheerful or sad. The *mode*

attribute denotes with 0 the minor mode whereas with 1 the major mode. The following image shows that the majority of the songs in the dataset are in major mode.

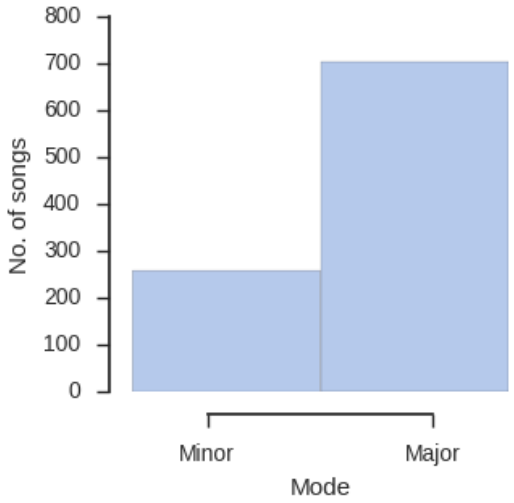

Figure 2.2: Mode distribution

The keys and the mode can be combined together in order to get the frequency of each key in both modes. In particular, the data were grouped on the *key* attribute and on the *mode* attribute, then the frequencies were calculated. The following image shows these frequencies in descending order.

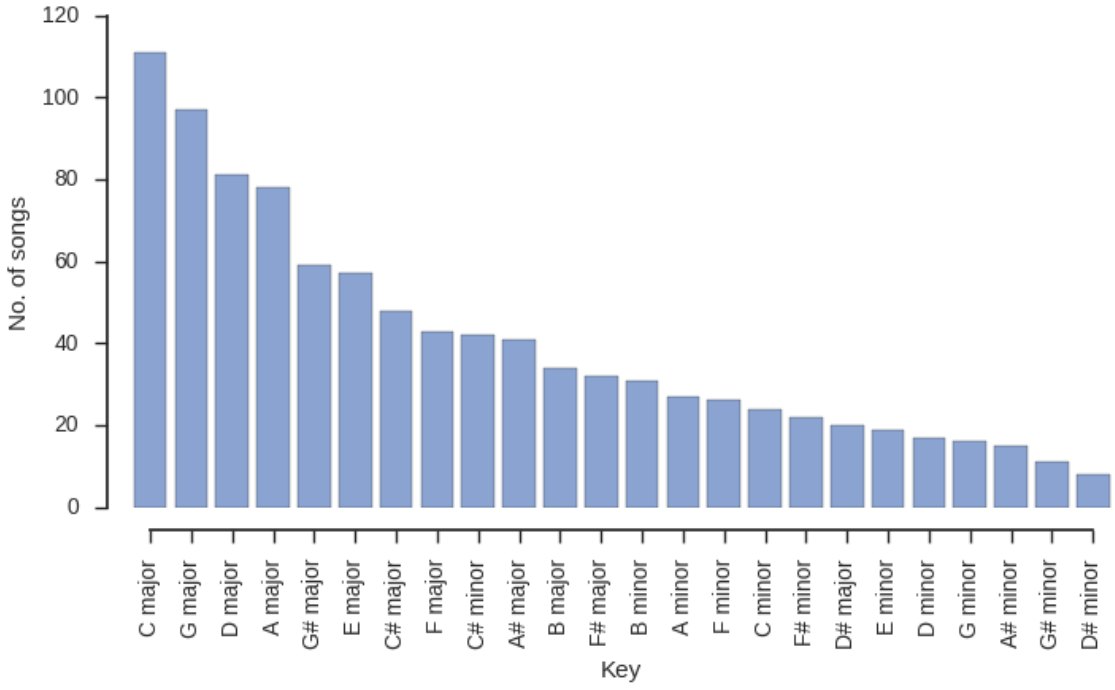

Figure 2.3: The frequency of each key on both modes

As seen before, the keys C, G, A and D are still in the top positions whereas the

minor keys are not so frequently used.

The analysis proceeds with the *duration* attribute. This attribute represents the duration of a track in milliseconds. It is important to determine if a track has a long duration or a short duration in order to distinguish songs from simple tracks. For simplicity, the attribute were transformed in seconds and the following image shows its distribution.

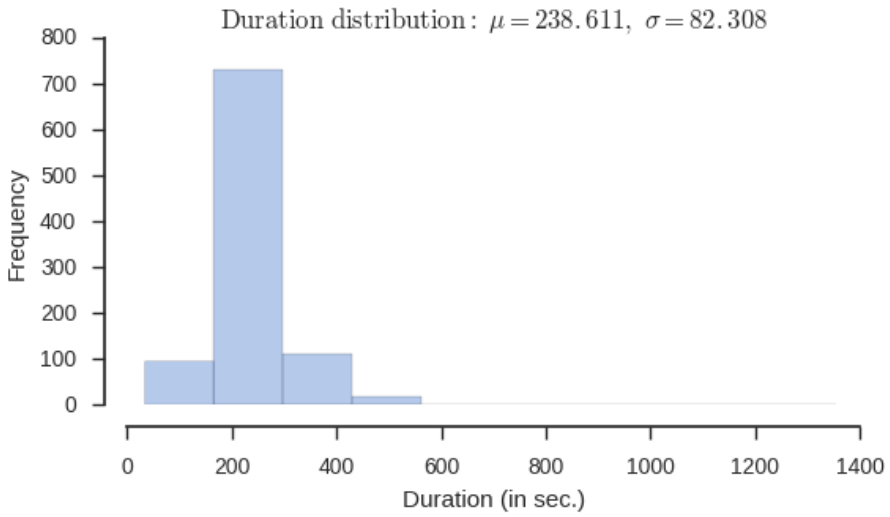

Figure 2.4: Distribution of the duration

The majority of the songs has a duration of 200 seconds and so they are not simple tracks. The next question concerns the release date distribution of these songs. The *release_date* attribute refers to the date of a album was first published. In particular, a song could be contained in an album which will be published in a date. For simplicity, the *release_date* attribute stores only the year in which the album was released and the following image shows the distribution of the release date of the songs.

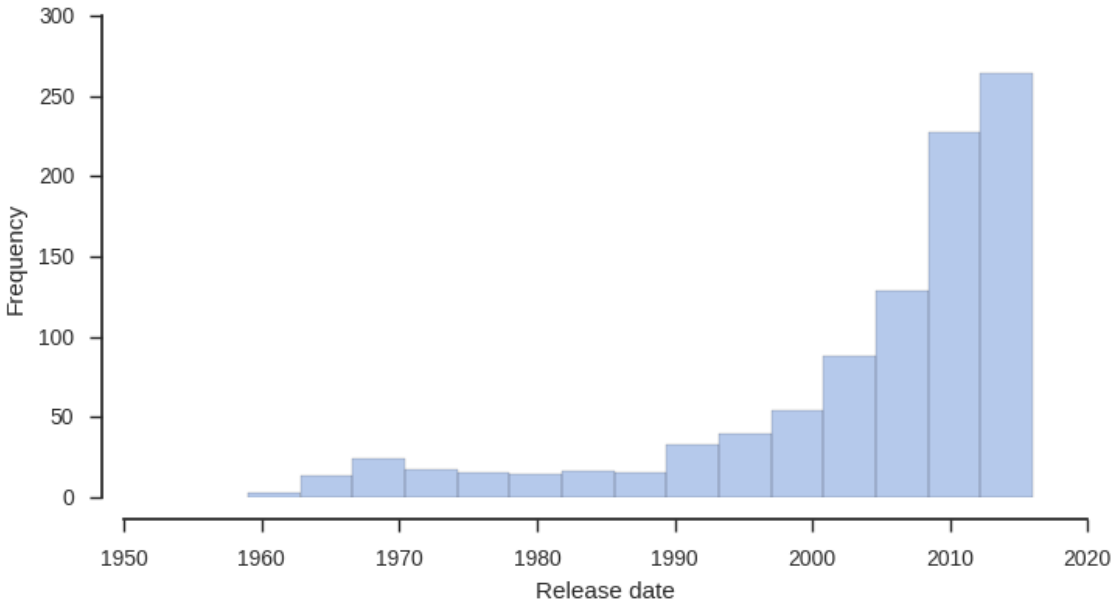

Figure 2.5: Distribution of the release date of the songs

These outcomes show that the majority of songs was recently released, in fact, the years after the 2008 are the most populated. Others analysis focus the attention on the quality of songs, in particular, the *valence* , the *energy*, the *instrumentalness* and *liveness* attributes were analyzed. As mentioned before, these attribute can have a value between 0.0 and 1.0. The *valence* is a measure of the musical positiveness conveyed by a track, hence a value near to 1.0 means that a track sound more positive than a track with low valence. The *energy* attribute denotes the intensity and the activity of a track, in particular, the more a track is energetic the more is the intensity. For example, death metal has high energy, while a Bach prelude scores low on the scale. On the other hand, *instrumentalness* attribute measures if a track is based only on musical instruments or not. The closer the *instrumentalness* value is to 1.0, the greater likelihood the track contains no vocal content. Finally, the *liveness* attribute denotes the probability of a track which was performed live or not. Higher liveness values represent an increased probability that the track was performed live. The following images show the distribution of these attributes and for simplicity, the probability intervals were set with the 33th percentile.

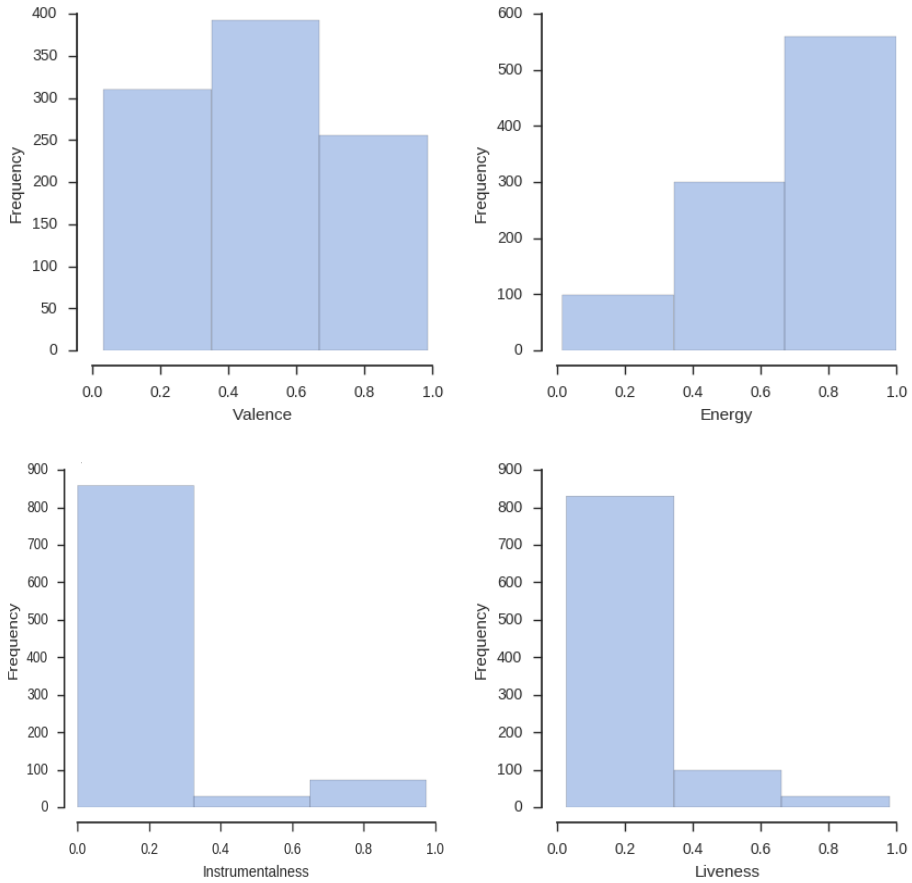

Figure 2.6: Distribution of the valence, the energy, the instrumentalness and the liveness attributes

In general, songs tend to have low levels of *instrumentalness* and *liveness*, hence they are not live songs and they are not only played but are also sung. In addition, these songs have a high level of *energy* and the *valence* attribute denotes that songs are both positive and sad.

4 Ad-Hoc queries

The second phase of the analysis concerns the *popularity* attribute in order to better understand the relationships with the others attributes. The popularity of a track is a value between 0 and 100, with 100 being the most popular. The popularity is based on the total number of plays the track has had and how recent those plays are. Generally speaking, songs that are being played a lot now will have a higher popularity than songs that were played a lot in the past. The first question on songs popularity concerns the frequency of each level of popularity. The following image shows the number of songs on each level of popularity.

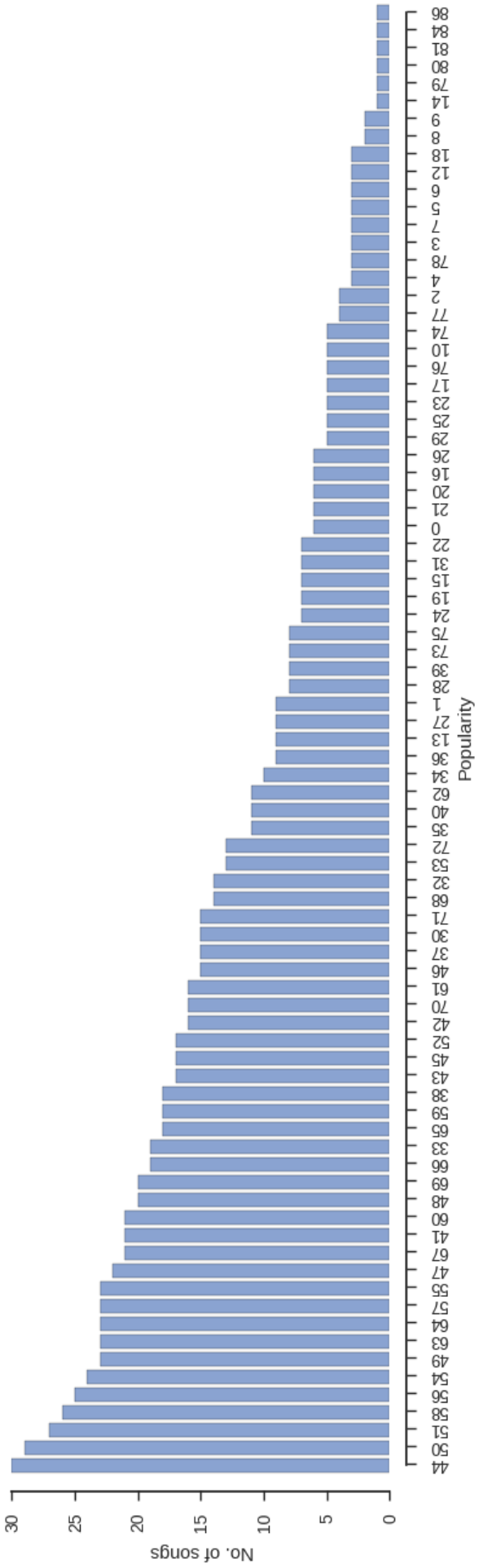

Figure 2.7: Frequency of songs popularity

There are few songs with high popularity because the majority of songs tends to have a medium level of popularity. Additionally, the *popularity* attribute was analyzed under several aspects, in particular the distribution of popularity was calculated on each tonality. This analysis allows to determine which is the key that better represents the highest or the lowest values of popularity. Therefore, the following image shows the songs popularity distribution for each key.

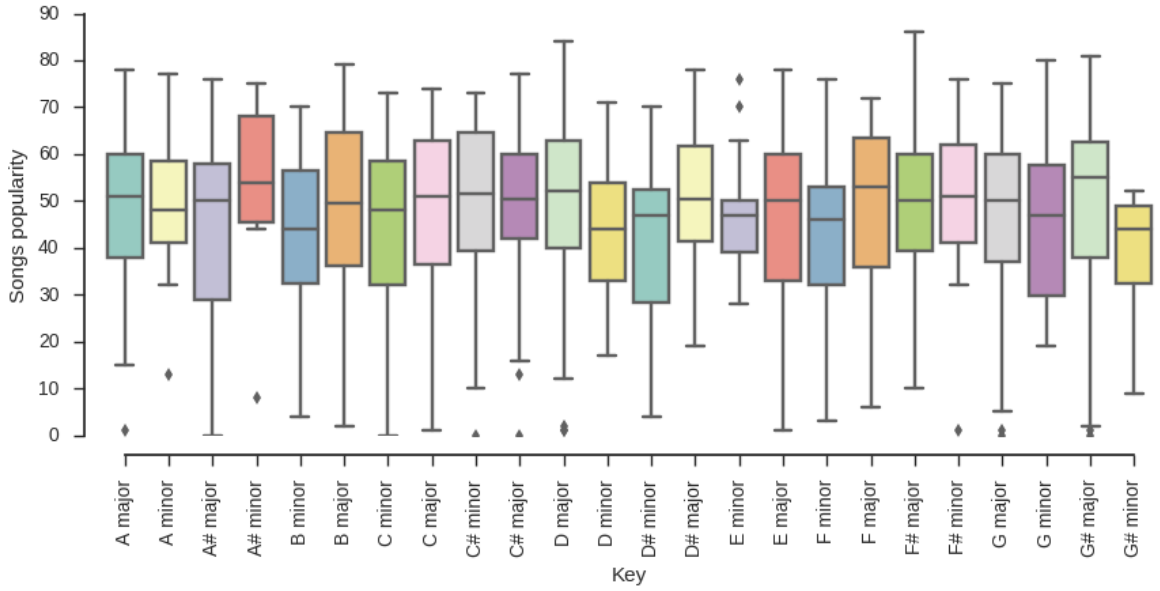

Figure 2.8: Songs popularity distribution for each key

These outcomes shows that the tonality attribute tends to have a popularity distribution similar in each key. It seems that there is not a connection between the *popularity* attribute and the *key* attribute. Hence, these attributes can be considered independent to each other. The next step concerns the *pattern* attribute in order to show which is the connection with the songs popularity. As previously stated, the *pattern* attribute represents the famous chord progressions which are contained in a song. Therefore, there is a one-to-many relationship between songs and the chord progressions. This is the reason why a new table that has as columns *track_ id*, *popularity*, *pattern* and *len* was created and each row still represents a song. However, now it refers to a specific progression. Hence, a song could be duplicated because this depends on the number of chord progressions associated to a song. Once the table was created, the first analysis concerns the chord progressions frequencies. The following image shows these frequencies in descending order.

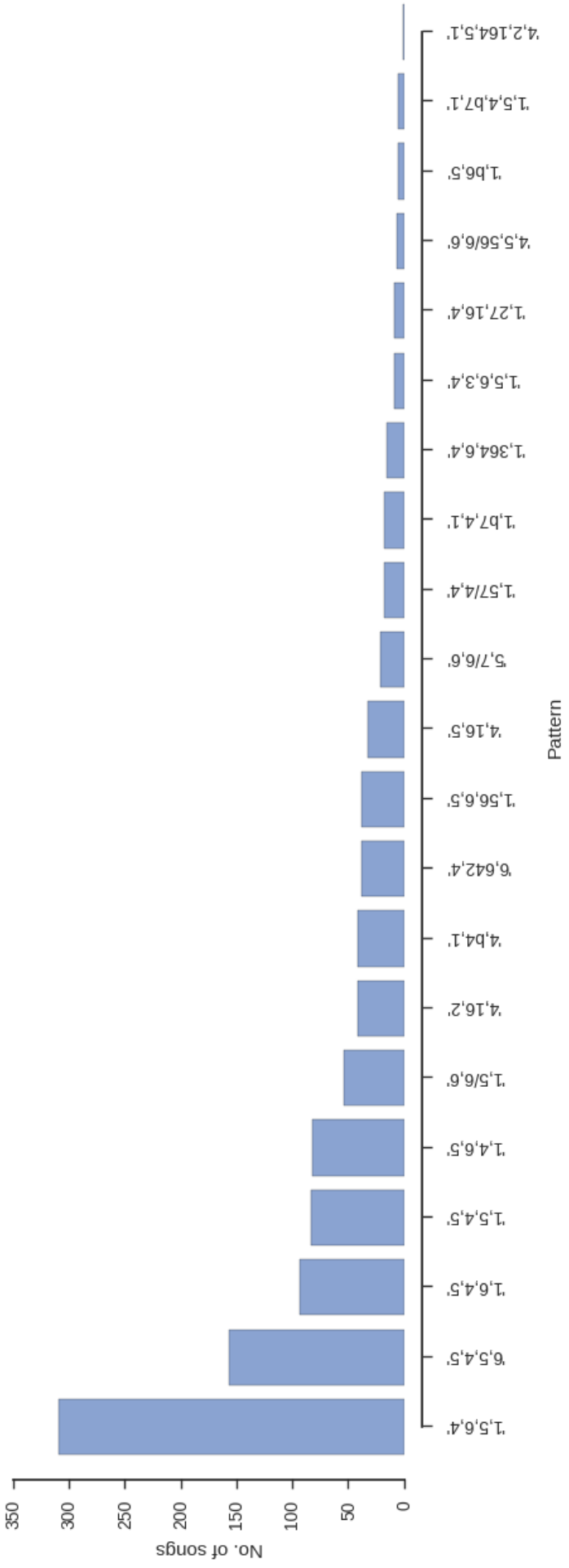

Figure 2.9: Famous chord progression frequencies

As expected, the progression *1, 5, 6, 4* is the most frequent. This progression in the *C* tonality represents the progression *C, G, Am, F*. This is the most common and the most popular chord progression across several genres of music. It is also known as the chord progression used to play several songs. For example, there are many videos on `www.youtube.com` that teach how to play hundred of songs with these 4 chords. The next analysis compares the popularity distribution of songs in each chord progression. This analysis allows to understand which is the relationship between songs popularity and the famous chord progressions.

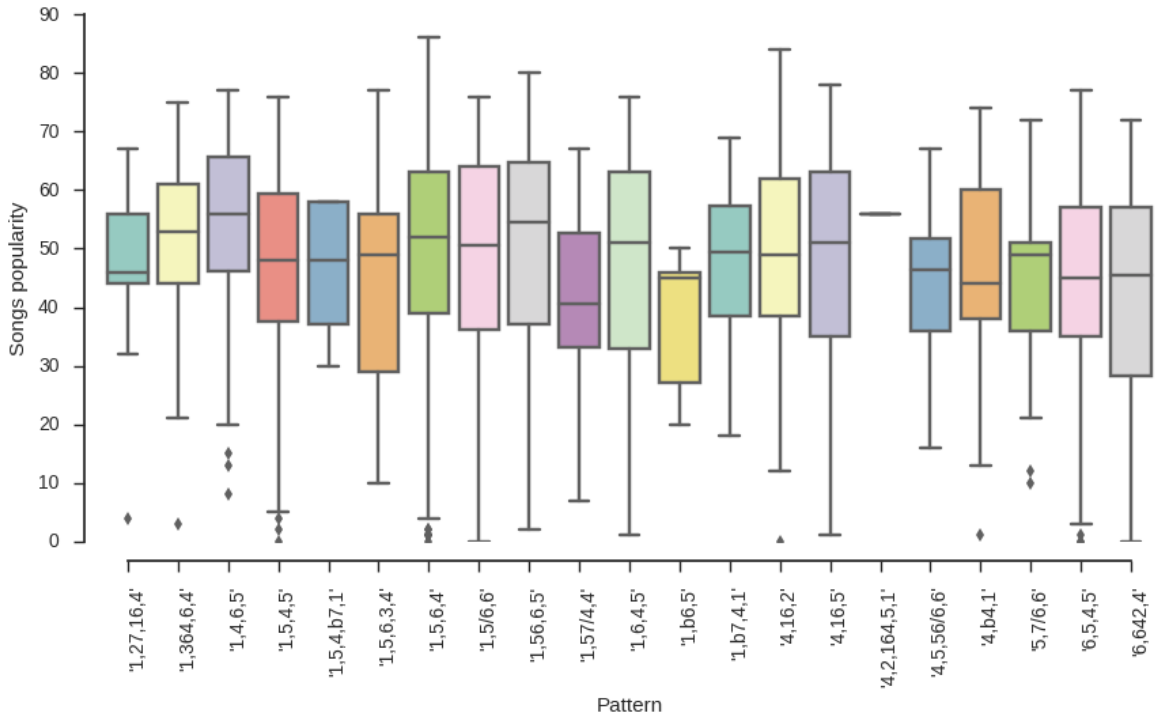

Figure 2.10: Songs popularity distribution for each chord progression

In general, these outcomes show that there is not a chord progression that represents only high or only low values of songs popularity. Therefore, these attributes can be considered independent to each other because there is not a pattern able to create always famous songs and vice versa. Finally, the last query determines the top 10 most popular chord progressions. The data were grouped on chord progressions and on songs popularity in order to get the 10 most popular.

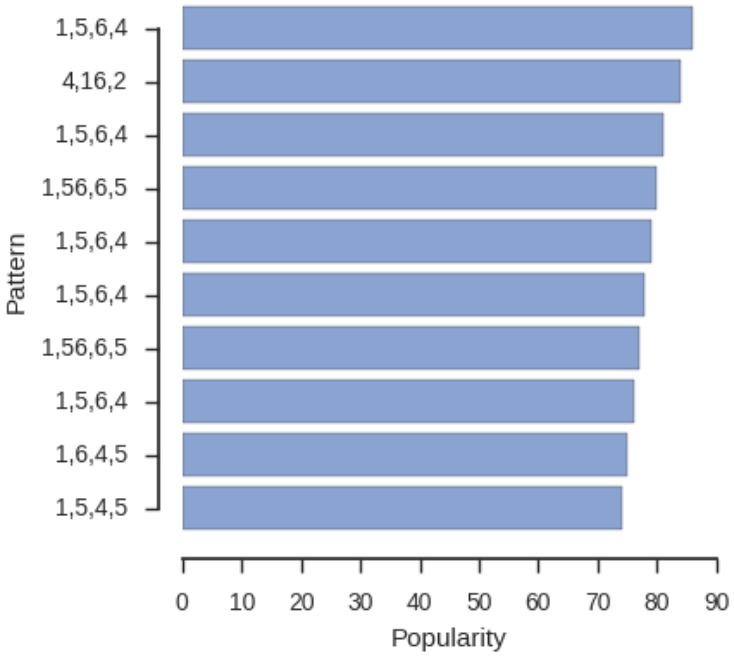

Figure 2.11: Top 10 popular chord progressions

Chapter 3

PREDICTION ON POPULARITY FROM MUSICAL STRUCTURES

1 Summary

This chapter describes the process performed in order to find out the elements that determine songs popularity. In particular, several models were created by using both regression and classification algorithms. First of all, the model creation process is explained and the tools used were indicated. Then, the models obtained and their outcomes are show for each algorithm considered. The objective is to find a model that better explains the relationships among songs attributes and songs popularity. In particular, the data mining techniques are used to predict and explain songs popularity.

2 Models creation process

This stage concerns the experiments performed, with a detailed description of the models creation process. The phases performed are the same in both regression and classification problems. Obviously, the main differences concern the algorithms used and some small changes in the dataset.

The dataset version for regression is called *data_for_regression.tsv* and it is stored at the following link Analysis/Regression/ on Github. The song popularity was not modified, since it is a numerical variable and in fact it does not need any modification to be used in the regression problem. The changes concern the attribute *patterns_list*[1],

[1]Remember that this last is the list of the chord progressions associated with each song.

in fact it was subdivided in multiple rows and the others information were duplicated. In addition, some attributes were removed such as *track_id*, *title*, *artist_name*, *release_date* and *genres* because they were useless. On the other hand, the models creation process in the Classification approach is the same used to solve the Regression problem. The main difference concerns the response value. The *song_pop* attribute was transformed into a categorical variable by means of the 33th percentile. In particular, the domain of the attribute was divided into 3 classes. The first class was set within an interval ranging from 0 and 42 of songs popularity and it is the low popularity class or *LP*. The medium popularity was set within an interval ranging from 43 and 57 and it is called *MP* whereas the others values represent the high popularity class or *HP*. This version of the dataset can be found at the following link DATA_FOR_CLASSIFICATION.tsv and it was used to solve the classification problem.

Each single model was created by means of the jupyter notebook software. The models creation process begins with the data preparation phase. In this phase, the X matrix of predictors and the y response vector (song popularity) were created. Next, the categorical attributes were transformed in numerical ones by means of the *LabelEncoder* function. In addition, the data were normalized by means of the *StandardScaler* function. This last task in the Classification approach was not performed in order to maintain models interpretability.

The second phase concerns detection and selection of the attributes which will be used in the creation models phase. This task was performed by means of the *Recursive Features Elimination Cross Validation* or *RFECV* function. Once the attributes were selected, the process proceeds to the *Train Test split* approach in order to get two subsets of the dataset. The 20% of the dataset is the *Train set* and it was used to create the models while the 80% is the *Test set* and it was used to evaluate these models.

The next phase is the creation models one in which the estimators were performed. In particular, different models were created for each regression and classification estimator by means of the *GridSearchCV* function. This last function allowed to get the so-called hyper-parameters. Next, the validation phase was performed by means of the *k-fold cross validation* approach and finally the outcomes will be shown for each class of problem. In conclusion, the following image summarizes the whole process.

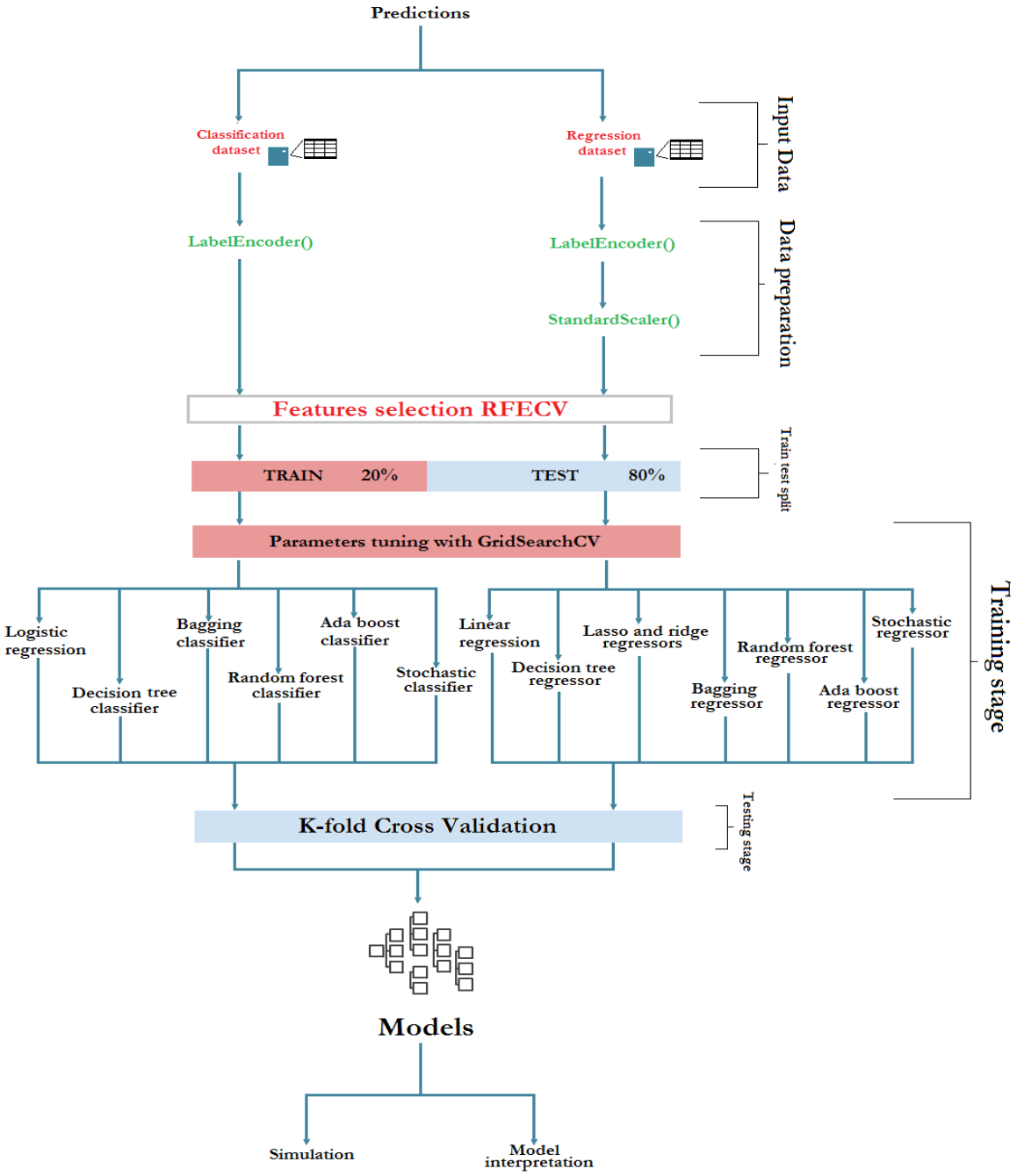

Figure 3.1: Models creation process

3 Regression problem

In statistical modeling, regression analysis is a statistical process for estimating the relationships among variables. It includes many techniques for modeling and analyzing several variables, when the focus is on the relationship between a dependent variable and one or more independent variables (or *predictors*). More specifically, regression analysis helps to understand how the typical value of the dependent variable changes when any one of the independent variables is varied, while the other independent variables are held fixed. Most commonly, regression analysis estimates the conditional expectation of the dependent variable given the independent variables - that is, the average value of the dependent variable when the independent variables are fixed. The estimation target is a function of the independent variables called the regression function.

3.1 Linear regression and Decision tree regressor

The regression analysis begins with the linear regression model and the decision tree regressor model. These models and their outcomes are stored in a file called *linear_and_tree_regressor.ipynb* at the following link /linear_and_tree_regressor on Github. Simple linear regression is a useful approach for predicting a response on the basis of a single predictor variable. However, in practice there may be more than one predictor. This last is the case of *Multiple linear regression* which is used to explain the relationship between one continuous dependent variable and two or more independent variables. On the other hand, the decision tree learning expresses this relationship by means of the trees. The prediction space is divided into a number of simple regions which are used to predict each observation.

As previously mentioned, the RFECV was performed both on the linear regression and the decision tree regressor once the dataset was normalized. The attributes previously selected will be included in the models creation phase and these are the following:

RFECV: Selected attributes for regression problem

```
{
 seq, artist_pop, danceability, energy,
 instrumentalness, liveness, loudness, tempo
}
```

The above list shows 8 attributes which will be the same obtained from the others regression analysis. The next step concerns the GridSearchCV approach. This last technique was performed on both estimators in two different versions: one that sets the cross validation to 5 and another that sets the cross validation to 10. In addition, the r-squared was used as evaluation metric for each model.

Since the linear regression estimator has few parameters to set, the GridSearch was performed quickly. The following outcomes show two different Linear models: the first one concerns the best model trained with cross validation set to 5 whereas the second one is the best model obtained with cross validation set to 10.

GridSearchCV: Linear regression outcomes

```
# set of parameters to test
param_linear = {"normalize": ["False", "True", ],
               }
-- Grid Parameter Search via 5-fold CV
GridSearchCV took 0.04 seconds
for 2 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.506 (std: 0.087)
Parameters: {'normalize': 'False'}
-- Grid Parameter Search via 10-fold CV
GridSearchCV took 0.07 seconds
for 2 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.494 (std: 0.149)
Parameters: {'normalize': 'False'}
```

In general, results tend to be similar to each other, therefore the model chosen and its parameters are those that maximize the metric considered. Both models have the r-squared near to 50% and the GridSearchCV took few milliseconds to complete the

training stage. On the other hand, the decision tree regressor allows to set several parameters. These parameters concern the depth of the tree or the number of leaves that the tree could have and so on. The set of parameters used influences the execution time of the GridSearchCV. In fact, the training phase was completed in one minute for the first version of the GridSearch whereas the second version took over two minutes to find the best model.

GridSearchCV: Decision tree regressor outcomes

```
# set of parameters to test
param_TR = {"criterion": ["mse", ],
            "min_samples_split": [2, 5, 10, 15, 20],
            "max_depth": [2, 5, 10, 15, 20],
            "min_samples_leaf": [1, 5, 10, 15, 20],
            "max_leaf_nodes": [None, 5, 10, 15, 20],
            "max_features": [None, 3, 6, 8],
            }
-- Grid Parameter Search via 5-fold CV

GridSearchCV took 72.31 seconds
for 2500 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.529 (std: 0.064)
Parameters: {'max_leaf_nodes': 10, 'min_samples_leaf': 10,
             'min_samples_split': 10, 'criterion': 'mse',
             'max_features': 6, 'max_depth': 15}

-- Grid Parameter Search via 10-fold CV

GridSearchCV took 145.05 seconds
for 2500 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.492 (std: 0.124)
Parameters: {'max_leaf_nodes': 15, 'min_samples_leaf': 10,
             'min_samples_split': 15, 'criterion': 'mse',
             'max_features': 6, 'max_depth': 10}
```

Similar to the Linear models, results show a mean validation score near to 50%, therefore, there are no improvements. Once the hyper-parameters were obtained, the next phase consists to evaluate the models just created. As mentioned above, this task was performed on the Test set by means of the K-fold Cross validation approach. The number of folds was set within an interval ranging from 2 and 10 in order to see how the r-squared vary based on the number of folds. The following image shows this relationship on both models.

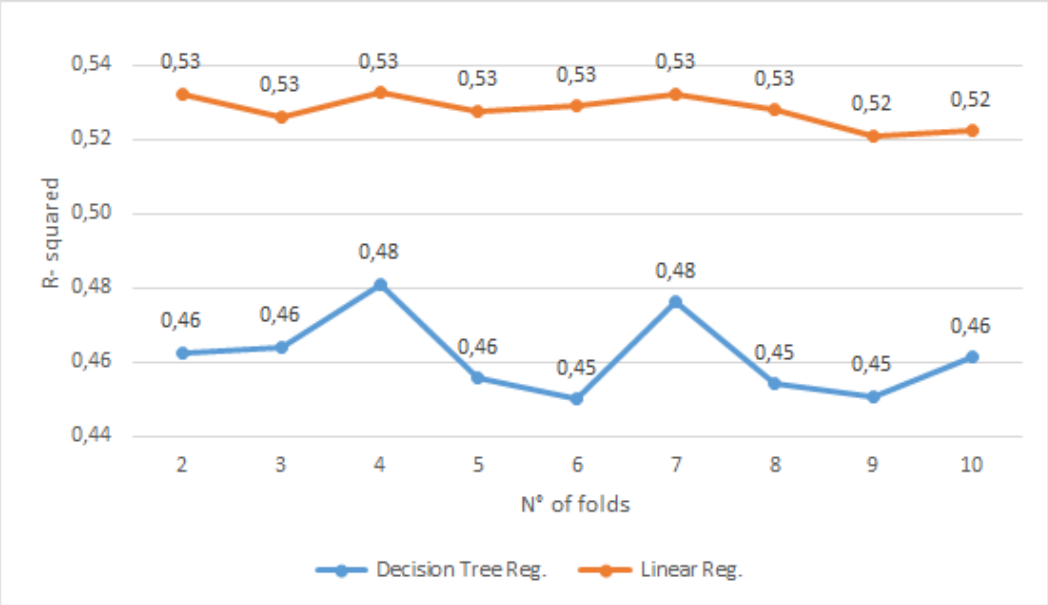

Figure 3.2: CV on Linear regression and Decision tree regressor

The above results, obtained in the evaluation phase, show that the Linear regression model generally has the r-squared greater than that of the decision tree regressor. This depends on the fact that Linear regression models have a low variance but a high bias whereas Decision trees have a high variance. The following linear equation represents the relationship among the attributes and songs popularity of the linear model.

$$Song\_pop \approx 47.95 + 13.25_{artist\_pop} + 1.27_{loudness} + 0.92_{danceability} - 0.79_{tempo} - 0.88_{liveness} - 1.17_{seq} - 1.29_{energy} - 2.73_{instrumentalness}$$

Although the data were normalized, it is possible to understand which is the real impact of each attributes respect to the song popularity in both models. In particular, the artist popularity attribute is crucial to determine which will be the response value. On the other hand, the model obtained by means of the Decision tree regressor is the following:

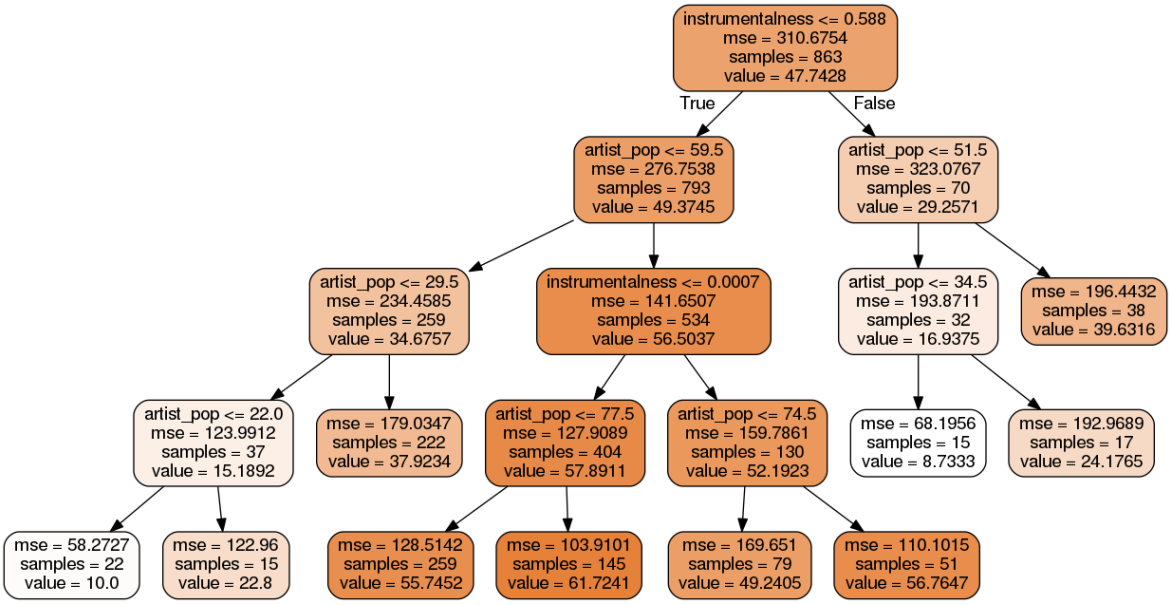

Figure 3.3: Decision tree regressor

The above model splits the predictors space into 10 regions, each region represents the songs popularity value which will predicted. Additionally, the model uses only 2 attributes over 8 as split nodes: the *artist_pop* and the *instrumentalness*. The first one is quite crucial to determine which will be the value of songs popularity, in fact, the majority of splits are based on this attribute. Generally, the less is the artist's popularity the less is the song popularity and viceversa. Finally, the following image shows the features importance of the Decision regressor model.

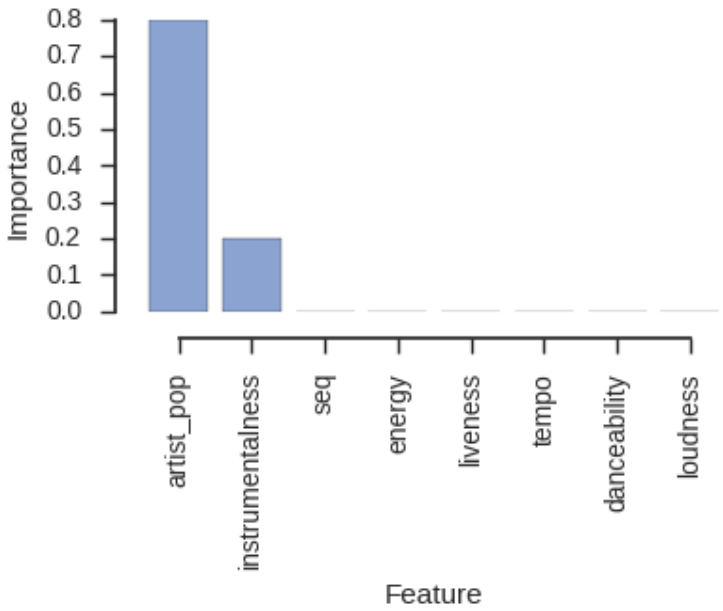

Figure 3.4: Features importance of the decision tree regressor

3.2 Lasso and Ridge regressor

The regression analysis proceeds with the Ridge and the Lasso estimators in order to check if they could be better than the previous ones. These analysis are stored in a file called *lasso_and_ridge.ipynb* at the following link /lasso_and_ridge on Github.

The simple linear model can be improved, by replacing plain least squares fitting with some alternative fitting procedures. These alternative fitting procedures can yield better prediction accuracy and model interpretability. The Ridge and the Lasso estimators are also known as the Shrinkage methods. In general, a Shrinkage method involves fitting a model involving all p-predictors. However, the estimated coefficients are shrunken towards zero relative to the least squares estimates. This shrinkage (also known as *regularization*) has the effect of reducing variance. Depending on what type of shrinkage is performed, some of the coefficients may be estimated to be exactly zero. Hence, shrinkage methods can also perform variable selection.

As mentioned before, the RFECV approach was performed on both estimators and it has selected the same 8 attributes of the previous analysis. Furthermore, the GridSearchCV was also performed in the two versions already seen before. The analysis begin with the Ridge estimator which seems to be faster than the other one. In fact, this estimator has few parameters to be set and the GridSearch was completed in few seconds.

GridSearchCV: Ridge regressor outcomes

```
# set of parameters to test
param_ridge = {"alpha": np.arange(1, 10,0.5),
                  "normalize": [True, False],
                  "solver": ["auto", "svd", "cholesky",

               }
-- Grid Parameter Search via 5-fold CV
GridSearchCV took 4.25 seconds
for 216 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.512 (std: 0.082)
Parameters: {'normalize': False,
              'alpha': 9.5, 'solver': 'lsqr'}
```

```
-- Grid Parameter Search via 10-fold CV
GridSearchCV took 9.98 seconds
for 216 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.499 (std: 0.142)
Parameters: {'normalize': False,
             'alpha': 9.5, 'solver': 'sparse_cg'}
```

As mentioned before, the first model concerns the best model obtained with the cross validation set to 5 whereas the second model is the best obtained with the cross validation set to 10. The outcomes obtained are quite similar to those of the Linear regression but with small improvements. Both models have a mean validation score near to 50% but they use two different solvers. On the other hand, the Lasso estimator can be set with several parameters. The training phase was completed in 30 seconds for the first version of the GridSearch and the second version took over one minute.

GridSearchCV: Lasso regressor outcomes

```
# set of parameters to test
param_lasso = {"alpha": np.arange(1, 10,0.5),
               "normalize": [True, False],
               "precompute": [True, False],
               "tol": [200.0, 300.0, 500.0],
               "warm_start": [True, False],
               "positive": [True, False],
               "selection": ["cyclic", "random"],
              }

-- Grid Parameter Search via 5-fold CV

GridSearchCV took 36.48 seconds
for 1728 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.548 (std: 0.029)
Parameters: {'normalize': False, 'warm_start': True,
             'selection': 'random', 'positive': False,
             'precompute': True, 'tol': 200.0, 'alpha': 3.0}
```

```
-- Grid Parameter Search via 10-fold CV

GridSearchCV took 80.25 seconds
for 1728 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.517 (std: 0.133)
Parameters: {'normalize': False, 'warm_start': True,
             'selection': 'cyclic', 'positive': False,
             'precompute': True, 'tol': 200.0, 'alpha': 1.5}
```

The above results show that the lasso estimator generates better models than the ridge estimator. In fact, the first version of the GridSearch select a lasso model with the mean validation score near to 55%. In the evaluation phase, it is possible to notice which are the main differences between these models. As mentioned above, the focus of the Shrinkage methods is on reducing variance at the expense of the bias. This situation is reached by introducing the so-called *shrinkage penalty*. In particular, this factor serves as tuning parameter on the regression coefficients estimates. The following image shows the outcomes of the evaluation phase.

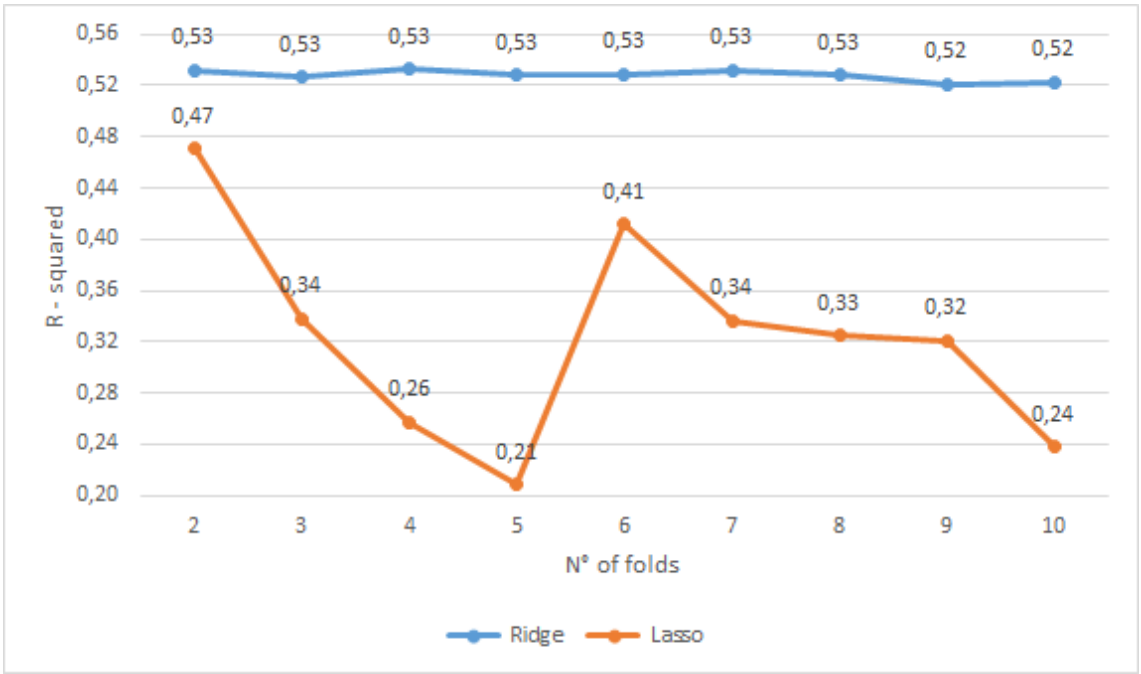

Figure 3.5: CV on Ridge and Lasso regression

These outcomes show that the Ridge model has a better average r-squared than that of the Lasso, even if this last has better results in the training stage. In addition,

the regression coefficients estimates of the Ridge model are quite similar to those of the Linear regression, in fact, their evaluation curves are the same. Consequently, it is easy to suppose that these two models are interchangeable, so it does not matter which one is used. The Ridge equation is the following:

$$Song\_pop \approx 47.95 + 13.07_{artist\_pop} + 1.25_{loudness} + 0.90_{danceability} - 0.78_{tempo} - 0.89_{liveness} - 1.17_{seq} - 1.24_{energy} - 2.74_{instrumentalness}$$

As mentioned above, this equation is similar to that of the Linear model, so the Ridge estimator expresses the relationship among the predictors and the song popularity in the same way of the Linear regression estimator. Hence, *the artist_pop* is crucial to determine the response value. On the other hand, the Lasso equation is very different from the one above.

$$Song\_pop \approx 47.88 + 10.21_{artist\_pop} + 0.0_{loudness} + 0.0_{danceability} + 0.0_{tempo} + 0.0_{liveness} + 0.0_{seq} + 0.0_{energy} + 0.0_{instrumentalness}$$

Only 1 attributes over 8 was used in the model whereas the others have a null value because the Lasso estimator can shrunk the value of each coefficient in order to be exactly zero. Therefore, the estimator performs a variables selection.

3.3 Bagging regressor and Random forest regressor

The outcomes already obtained could be improved by using the Bagging algorithms. These are respectively the *Bagging regressor estimator* and the *Random forest regressor estimator*. Their outcomes are stored in two different files: one called *bagging_regressor.ipynb* and the other called *random_forest_regressor.ipynb*; they are both stored at the following link Analysis/Regression/bagging/ on Github.

A Bagging regressor is an ensemble meta-estimator that fits base regressors each on random subsets of the original dataset and then aggregate their individual predictions, either by voting or by averaging, to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it. On the other hand, Random forest provide an improvement over bagged trees. As in bagging, a number of decision trees are built

on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors. The split is allowed to use only one of those m predictors. A fresh sample of m predictors is taken at each split. This procedure makes the trees not strongly correlated to each other.

The Bagging regressor estimator allows to set which will be the so-called base estimator to fit on random subsets of the dataset. This parameter was set with the list of the estimators and their best parameters already seen before. The model creation phase was completed in one minute for the first version of the GridSearch whereas the second version took over two minutes to complete the training stage. The best models selected by the GridSearch use the Lasso estimator as base estimator. The following outcomes show that the first model has a mean validation score of 52% whereas the second one has a mean r-squared of 42%.

GridSearchCV: Bagging regressor outcomes

```
# set of parameters to test
DTR = DecisionTreeRegressor(**best_param_DTR)
LR = LinearRegression(**best_param_LR)
lasso = Lasso(**best_param_lasso)
ridge = Ridge(**best_param_ridge)

param_BR = {"base_estimator": [DTR, LR, lasso, ridge],
              "n_estimators": [5, 10, 20, 50],
              "max_samples": [2, 5, 8, 10, 15],
              "random_state": [99],
              "bootstrap": [True, False],
              "bootstrap_features": [True, False],
             }

-- Grid Parameter Search via 5-fold CV

GridSearchCV took 93.87 seconds
for 320 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.522 (std: 0.021)
```

```
Parameters: {'max_samples': 15, 'base_estimator':
             Lasso(alpha=3.0, copy_X=True, fit_intercept=True,
             max_iter=1000,
             normalize=False, positive=False,
             precompute=True, random_state=None,
             selection=u'random', tol=200.0, warm_start=True),
             'bootstrap': False, 'n_estimators': 5,
             'random_state': 99, 'bootstrap_features': False}

-- Grid Parameter Search via 10-fold CV

GridSearchCV took 197.82 seconds
for 320 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.463 (std: 0.119)
Parameters: {'max_samples': 15, 'base_estimator':
             Lasso(alpha=3.0, copy_X=True, fit_intercept=True,
             max_iter=1000,
             normalize=False, positive=False,
             precompute=True, random_state=None,
             selection=u'random', tol=200.0, warm_start=True),
             'bootstrap': False, 'n_estimators': 10,
             'random_state': 99, 'bootstrap_features': False}
```

On the other hand, Random forest regressor estimator was performed only in the first version of the GridSearchCV. The second version was aborted because this training stage seemed not to stop even after ten minutes. Similar to the Decision tree regressor, the Random forest regressor has several parameters to be set, for example the depth of the trees or the number of the leaves and so on.

GridSearchCV: Random forest regressor outcomes

```
# set of parameters to test
param_RFR = {"n_estimators": [10],
             "criterion": ["mse"],
             "max_depth": [5, 10, 15, 20],
             "min_samples_split": [5, 10, 15, 20],
```

```
                "min_samples_leaf": [1, 5, 10, 15, 20],
                "max_leaf_nodes": [None, 5, 10, 15],
               }

-- Grid Parameter Search via 5-fold CV

GridSearchCV took 60.53 seconds
for 320 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.549 (std: 0.050)
Parameters: {'max_leaf_nodes': 15, 'min_samples_leaf': 5,
             'n_estimators': 10, 'criterion': 'mse',
             'min_samples_split': 20, 'max_depth': 15}
```

In general, the above result shows a Random forest model which has a mean validation score of 55%. This last model, in fact, seems to be more accurate than the Bagging regressor models. It is necessary to remember that the bagging models use the Lasso estimator as base estimator and probably these models could show some troubles in the validation stage. The following image shows the evaluation curves of both estimators in order to point out which one is the best.

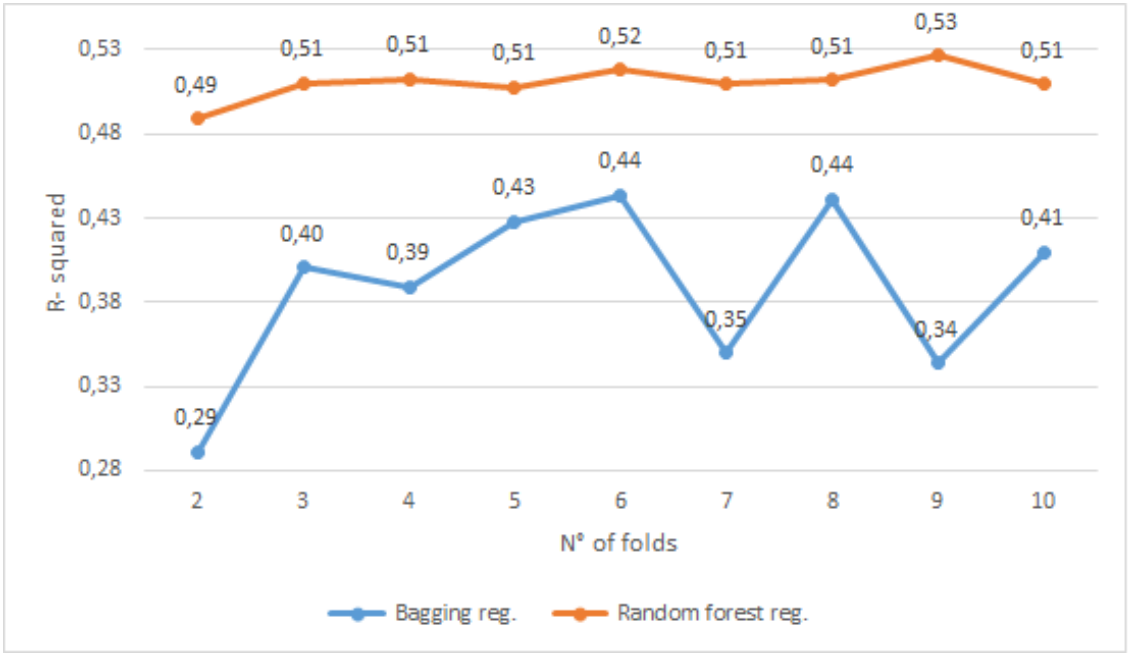

Figure 3.6: CV on Bagging and Random forest regression

The evaluation stage shows that the Bagging regressor improves the performance of the Lasso estimator, but the Random forest model has better results. Regardless of the number of folds, this last model has a mean r-squared of 50%. The following image shows the model obtained by means of the Random forest estimator.

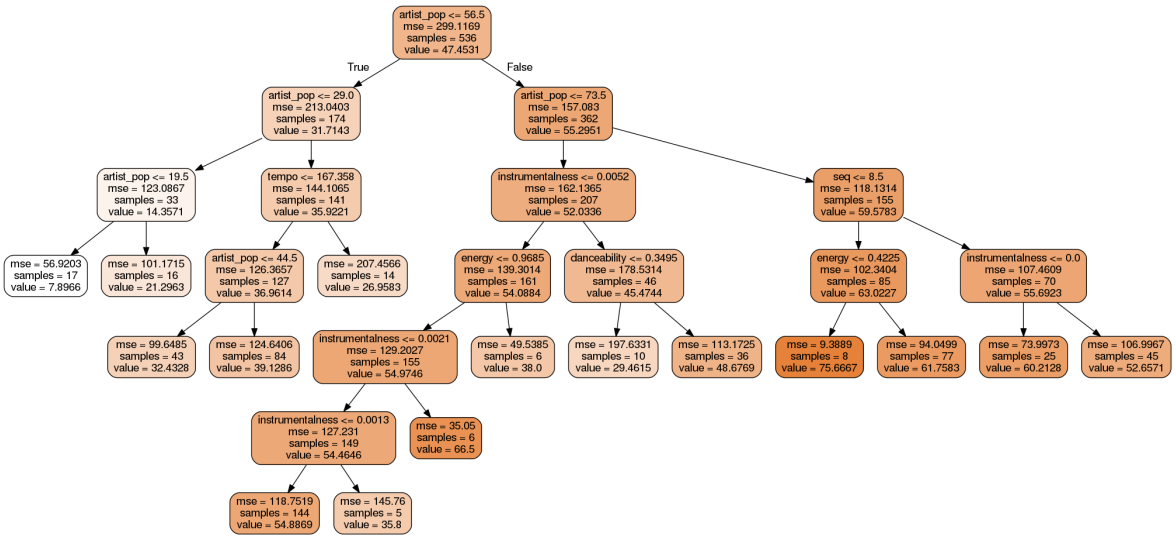

Figure 3.7: Bagging Random forest regression tree

The predictors space was divided into 15 regions and the model seems to be more complex than that of the Decision tree regressor. In fact, the Random forest model uses all the 8 attributes in order to predict which will be the songs popularity. Similar to the Decision tree, the majority of splits are based of the *artist_pop* attribute. This model confirms that the artist's popularity is crucial to determine the songs popularity. Additionally, the others attributes are also quite important, for example the *instrumetalness* attribute is second as importance. The following image shows the features importance of the model.

3.4 Ada boost regressor and Stochastic gradient boost regressor

The Regression analysis end with the Boosting algorithms. Like bagging, boosting is a general approach that can be applied to many statistical learning methods for regression or classification. The idea of boosting came out of the idea of whether a weak learner can be modified to become better.

An AdaBoost regressor is a meta-estimator that begins by fitting a regressor on the original dataset and then fits additional copies of the regressor on the same dataset

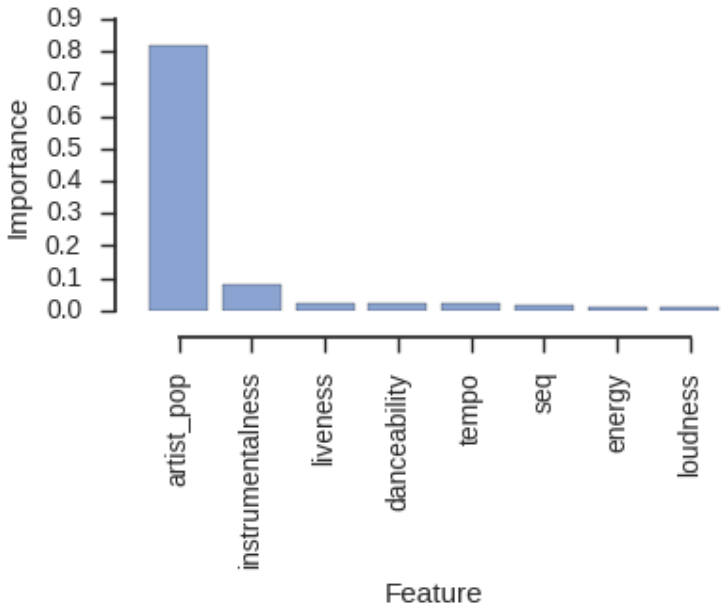

Figure 3.8: Features importace of the Random forest regression

but where the weights of instances are adjusted according to the error of the current prediction. As such, subsequent regressors focus more on difficult cases.

Like other boosting methods, gradient boosting combines weak learners into a single strong learner, in an iterative fashion. This framework was further developed by Friedman and called Gradient Boosting Machines. The statistical framework cast boosting as a numerical optimization problem where the objective is to minimize the loss of the model by adding weak learners using a gradient descent like procedure. This class of algorithms were described as a stage-wise additive model. This is because one new weak learner is added at a time and existing weak learners in the model are frozen and left unchanged. The generalization allowed arbitrary differentiable loss functions to be used, expanding the technique beyond binary classification problems to support regression, multi-class classification and more.

These analysis are stored in two different files: the first one is called *ada_regressor* and the other one is called *stochastic_gradient_boosting_regressor*. They are both stored at the following link Analysis/Regression/boosting on Github. As the Bagging regressor, the Ada regressor allows to set which will be the base estimator and this parameter was set with the list of the estimators and their best parameters already obtained before. In general, the models creation phase for the Stochastic estimator is faster than the Ada estimator and the outcomes are quite similar to each other. The first Ada model has a mean r-squared of 52% and the training stage was completed

in one minutes. The second model obtainend by means of the GridSearch has a mean validation score of 50%.

GridSearchCV: Ada boosting regressor

```
# set of parameters to test
DTR = DecisionTreeRegressor(**best_param_DTR)
LR = LinearRegression(**best_param_LR)
lasso = Lasso(**best_param_lasso)
ridge = Ridge(**best_param_ridge)

param_ADA = {"base_estimator": [DTR, LR, lasso, ridge],
             "n_estimators": [5],
             "loss": ['linear', 'square', 'exponential'],
             "random_state": [99],
            }

-- Grid Parameter Search via 5-fold CV

GridSearchCV took 0.81 seconds
for 12 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.521 (std: 0.081)
Parameters: {'n_estimators': 5, 'loss': 'exponential',
'base_estimator': LinearRegression(copy_X=True,
fit_intercept=True, n_jobs=1,
normalize=u'False'),
'random_state': 99}

-- Grid Parameter Search via 10-fold CV

GridSearchCV took 1.52 seconds
for 12 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.504 (std: 0.123)
Parameters: {'n_estimators': 5, 'loss': 'exponential',
             'base_estimator': Lasso(alpha=3.0, copy_X=True,
```

```
                fit_intercept=True, max_iter=1000,
                normalize=False, positive=False,
                precompute=True, random_state=None,
                selection=u'random', tol=200.0,
                warm_start=True),
                'random_state': 99}
```

The above results show that the first model uses the Linear estimator as base estimator whereas the second one uses the Lasso estimator. Generally, it looks like that the first model is more accurate than the second one. On the other hand, the Gradient estimator produces 100 sequentially trees in order to improve the models created at run time. However, these models are less accurate than the previous ones.

GridSearchCV: Stochastic gradient boosting regressor

```
# set of parameters to test
param_SGBR = {"loss": ['ls'],
              "n_estimators": [100],
              "max_depth": [2, 5, 10, 15],
              "min_samples_split": [2, 5, 10, 15],
              "min_samples_leaf": [ 5, 10, 15],
              "max_leaf_nodes": [None, 5, 10, 15],
              "random_state": [99],    }

-- Grid Parameter Search via 5-fold CV

GridSearchCV took 30.22 seconds
for 192 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.502 (std: 0.076)
Parameters: {'loss': 'ls', 'max_leaf_nodes': 5,
             'min_samples_leaf': 10, 'n_estimators': 100,
             'random_state': 99, 'min_samples_split': 2,
             'max_depth': 5}

-- Grid Parameter Search via 10-fold CV

```

```
GridSearchCV took 87.83 seconds
for 192 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.503 (std: 0.111)
Parameters: {'loss': 'ls', 'max_leaf_nodes': 5,
             'min_samples_leaf': 10, 'n_estimators': 100,
             'random_state': 99, 'min_samples_split': 2,
             'max_depth': 5}
```

In the evaluation phase, the models have an average r-squared similar to each other, but in general the Ada boost model has better performance than the second model. In particular, the evaluations curves tend to be similar and it is quite difficult to say which one is the best model.

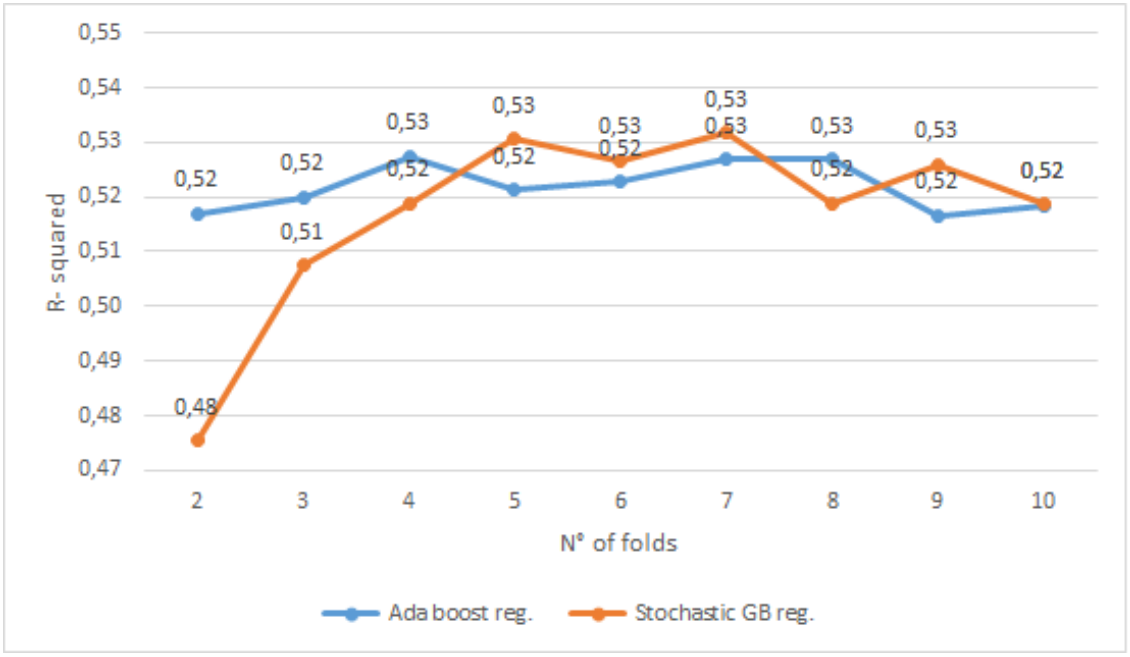

Figure 3.9: CV on Ada boost and Stochastic GB regressors

4 Classification problem

The linear regression model assumes that the response variable Y is quantitative. But in many situations, the response variable is instead qualitative. For example, eye color is qualitative, taking qualitative on values blue, brown, or green. Often qualitative variables are referred to as categorical. The *classification* is the process for predicting qualitative responses. In machine learning and statistics, classification is the problem of identifying to which of a set of categories a new observation belongs, on the basis of a training set of data containing observations whose category membership is known. An algorithm that implements classification, especially in a concrete implementation, is known as a *classifier*. The term classifier sometimes also refers to the mathematical function, implemented by a classification algorithm, that maps input data to a category.

4.1 Logistic regression and Decision tree classifier

The classification analysis begins with the *Logistic regression* model and the *Decision tree classifier*. These models and their outcomes are stored at the following link logistic_and_decision_tree_classifier on Github. Logistic regression is named for the function used at the core of the method, the logistic function. The logistic function, also called the sigmoid function was developed by statisticians to describe properties of population growth in ecology, rising quickly and maxing out at the carrying capacity of the environment. It is an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits. Logistic regression uses an equation as the representation, very much like linear regression. Input values are combined linearly using weights or coefficient values to predict an output value. A key difference from linear regression is that the output value being modeled is a binary values rather than a numeric value. The coefficients of the logistic regression algorithm are estimated by means of the maximum-likelihood estimation.

A classification tree is very similar to a regression tree, except that it is classification used to predict a qualitative response rather than a quantitative one. The classification tree predicts each observation belongs to the most commonly occurring class of training observations in the region to which it belongs. The task of growing a classification tree is quite similar to the task of growing a regression tree. Just as in the regression setting, the recursive binary splitting is used to grow a classification tree. However,

in the classification setting, RSS cannot be used as a criterion for making the binary splits. A natural alternative to RSS is the classification error rate. The classification error rate is simply the fraction of the training observations in a region that do not belong to the most common class.

As seen before for the regression problem, the RFECV approach was used in order to select the features which will be used in the models creation phase. The attributes selected are more than those used in the regression problem and they are the following:

RFECV: Selected attributes for classification problem

```
{

'seq', 'mode', 'artist_pop', 'acousticness',
'danceability', 'energy', 'instrumentalness',
'liveness', 'loudness', 'speechiness', 'valence'
}
```

Obviously, the importance of the *artist_pop* is well known and in fact it was selected by the RFECV. In addition, the RFECV has selected some attributes related to the quality of a song such as *liveness*, *loudness* and so on.

The training stage is the same already seen in the regression problem, in particular, each model was created by means of the GridSearchCV technique in two different versions and these versions differ in the number of folds considered in the cross validation. In general, the training stage shows that the Logistic regression estimator was performed in few seconds, this may depend on the number of parameters. Both versions of the logistic regression estimator have an accuracy of 53%

GridSearchCV: Logistic regression outcomes

```
# set of parameters to test
param_logistic = {"solver" : ["newton-cg", "lbfgs",
                              "sag", "liblinear"],
                }

-- Grid Parameter Search via 5-fold CV

GridSearchCV took 0.25 seconds
for 4 candidate parameter settings.
```

```
Model with rank: 1
Mean validation score: 0.535 (std: 0.026)
Parameters: {'solver': 'newton-cg'}

-- Grid Parameter Search via 10-fold CV

GridSearchCV took 0.44 seconds
for 4 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.535 (std: 0.070)
Parameters: {'solver': 'newton-cg'}
```

On the other hand, the Decision tree estimator took more time than the one above. Similar to the Decision tree regressor, Decision tree classifier has several parameters to be set, such as the depth of the tree and so on. But this estimator differs from the Decision regressor for the functions used to measure the quality of a split.

GridSearchCV: Decision tree classifier outcomes

```
# set of parameters to test
param_DTC = {"criterion": ["gini", "entropy"],
             "min_samples_split": [2, 5, 10, 15],
             "max_depth": [2, 5, 10, 15],
             "min_samples_leaf": [1, 5, 10, 15],
             "max_leaf_nodes": [None, 5, 10, 15],
             "max_features": [None, 3, 4, 5],
             }

-- Grid Parameter Search via 5-fold CV

GridSearchCV took 43.71 seconds
for 2048 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.600 (std: 0.065)
Parameters: {'max_leaf_nodes': None, 'min_samples_leaf': 5,
             'min_samples_split': 15, 'criterion':'gini',
             'max_features': 5, 'max_depth': 5}
```

```
19
20 -- Grid Parameter Search via 10-fold CV
21
22 GridSearchCV took 89.05 seconds
23 for 2048 candidate parameter settings.
24 Model with rank: 1
25 Mean validation score: 0.605 (std: 0.109)
26 Parameters: {'max_leaf_nodes': None, 'min_samples_leaf': 5,
27                'min_samples_split': 5, 'criterion': 'entropy',
28                'max_features': None, 'max_depth': 15}
```

The above results show that both versions of the Decision tree classifier have an accuracy of 60%, it seems that the classifier estimators generate better models than the regression estimators. As mentioned before, the evaluation phase was performed by means of the cross validation approach. The evaluation curves show the variation of the accuracy on the basis of the number of folds considered and the following image shows this relationship for both models.

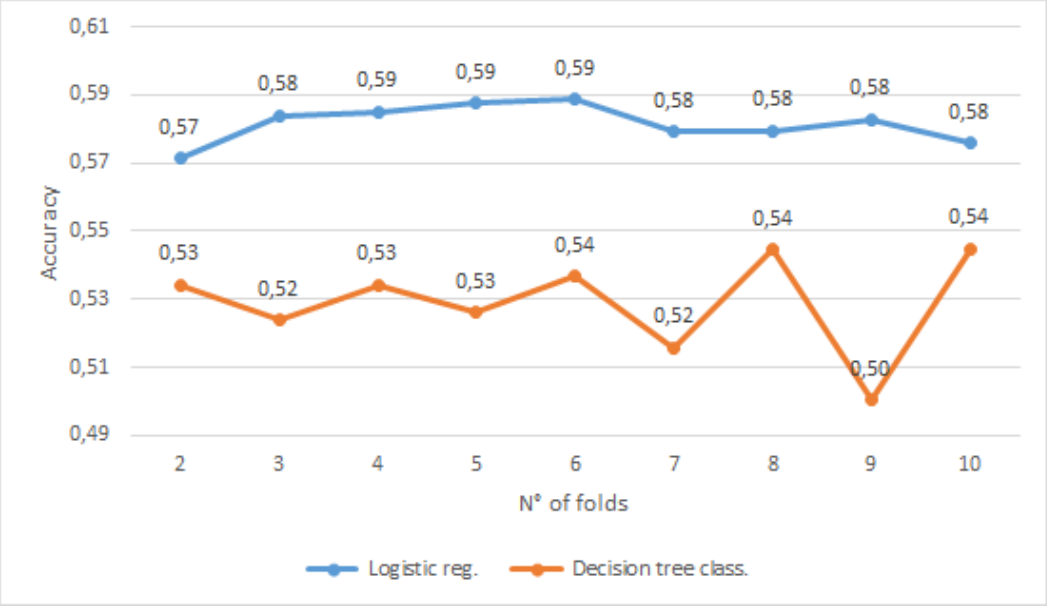

Figure 3.10: CV on Logistic regression and Decision tree classifier

The outcomes obtained in the evaluation stage are better than those obtained in the regression analysis, in fact, the logistic regression model has an accuracy of 58% in all folds considered. The second model tends to be less accurate, this may depend on the fact that the Decision trees approach have a high variance.

The above results are based on the accuracy metric but this measure is not good enough to explain which is the best model. Therefore, the confusion matrix was calculated on both models in order to show the prediction errors. Additionally, the *precision* and the *recall* metrics were calculated because they are crucial to choose the correct model. The following table shows these matrices: the left matrix concerns the Logistic model and the second matrix refers to the Decision tree model.

		Predicted			
		HP	LP	MP	Total
Actual	HP	200	19	64	283
	LP	29	228	34	291
	MP	128	82	79	289
	Total	357	329	177	

		Predicted			
		HP	LP	MP	Total
Actual	HP	157	27	99	283
	LP	36	194	61	291
	MP	108	78	103	289
	Total	301	299	263	

Table 3.1: Confusion matrices: Logistic reg. on the left and Decision tree classifier on the right;

The precision and the recall of the *HP* class are respectively 0.56 and 0.71 for the Logistic model while 0.52 and 0.55 for the Decision tree model. The precision and the recall of the *LP* class are 0.69 and 0.78 for the Logistic model while 0.65 and 0.67 for the Decision model. Finally, the precision and the recall of the *MP* class are 0.45 and 0.27 for the Logistic model while 0.39 and 0.36 for the Decision model.

In general, the first model seems to be better than the second one, in fact, the Logistic model tends to predict correctly the *LP* and *HP* classes whereas the *MP* class has several errors. Both models predict songs with a medium popularity as less popular songs or high popular songs. In addition, the second model predicts correctly only the *LP* class whereas the other classes have more errors than those in the Logistic model. Similar to the Linear regression estimator, the Logistic estimator has high bias but low variance whereas the Decision tree methods have low bias and high variance. This is the reason why the first model tends to be more accurate than the second one.

Once the confusions matrices were obtained, the last task performed concerns the visualization of the models just created. The Logistic model has three lists of results, in particular, these lists contain the beta coefficients of a specific song popularity class which will be used as the response value. These outcomes can be found at the link

mentioned before. Finally, the following link tree_classifier_best_v2.png shows the Decision tree classifier obtained.

4.2 Bagging classifier and Random forest classifier

The classification analysis proceed to improve the outcomes already obtained. In particular, the Bagging algorithms were used and these analysis can be found at the following link Analysis/Classification/bagging/ on Github. These analysis concern the Bagging classifier which is stored in the file called *bagging_classifier.ipynb* and the Random forest classifier which is store in the file *random_forest_classifier.ipynb*.

As mentioned before, the Bagging or *Bootstrap Aggregation* is a general procedure that can be used to reduce the variance for those algorithm that have high variance. An algorithm that has high variance are decision trees, like classification and regression trees. Decision trees are sensitive to the specific data on which they are trained. If the training data is changed the resulting decision tree can be quite different and in turn the predictions can be quite different. In particular,a Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions, either by voting or by averaging, to form a final prediction. On the other hand, the Random forest estimator is an improvement over bagged decision trees.

Similar to the Bagging regressor estimator, the Bagging classifier estimator allows to set which will be the base estimator to fit on random subsets of the dataset. This parameter was set with the list of the estimators and their best parameters already seen before. Since the models creation phase took too much time, the Bagging classifier was performed only on the first version of the GridSearchCV. In fact, this phase was completed in 6 minutes.

GridSearchCV: Bagging classifier outcomes

```
1 # set of parameters to test
2 DTC = DecisionTreeClassifier(**best_param_DTC)
3 LR = LogisticRegression(**best_param_LR)
4
5 param_BC = {"base_estimator": [DTC, LR],
6               "n_estimators": [5, 10, 20, 50],
7               "max_samples": [2, 5, 8, 10, 15],
```

```
8                "random_state": [99],
9                "bootstrap": [True, False],
10               "bootstrap_features": [True, False],
11              }
12
13 -- Grid Parameter Search via 5-fold CV
14
15 GridSearchCV took 369.21 seconds
16 for 160 candidate parameter settings.
17 Model with rank: 1
18 Mean validation score: 0.581 (std: 0.035)
19 Parameters: {'max_samples': 5, 'base_estimator':
20               LogisticRegression(C=1.0, class_weight=None,
21               dual=False, fit_intercept=True,
22               intercept_scaling=1, max_iter=100,
23               multi_class='ovr', n_jobs=1,
24               penalty='l2', random_state=None,
25               solver='newton-cg', tol=0.0001,
26               verbose=0, warm_start=False),
27               'bootstrap': True, 'n_estimators': 50,
28               'random_state': 99, 'bootstrap_features': True}
```

The Logistic Regressor was selected as base estimator and the model just created has 58% of accuracy. On the other hand, the Random classifier took less time than the Bagging estimator. As the Decision tree classifier, the Random forest classifier has several parameters to set such as the depth of the tree and so on. This training stage was completed in one minute and the model selected has a mean validation score of 59%.

GridSearchCV: Random forest classifier outcomes

```
1 # set of parameters to test
2 param_RFC = {"n_estimators": [10],
3               "criterion": ["gini"],
4               "max_depth": [5, 10, 15, 20],
5               "min_samples_split": [5, 10, 15, 20],
6               "min_samples_leaf": [1, 5, 10, 15, 20],
```

```
                "max_leaf_nodes": [None, 5, 10, 15],
               }

-- Grid Parameter Search via 5-fold CV

GridSearchCV took 65.79 seconds
for 320 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.591 (std: 0.063)
Parameters: {'max_leaf_nodes': 10, 'min_samples_leaf': 5,
             'n_estimators': 10, 'criterion': 'gini',
             'min_samples_split': 20, 'max_depth': 20}
```

Once the models were created, the evaluation stage was performed on both models. The following image shows the accuracy variation of both models on the basis of the number of folds considered.

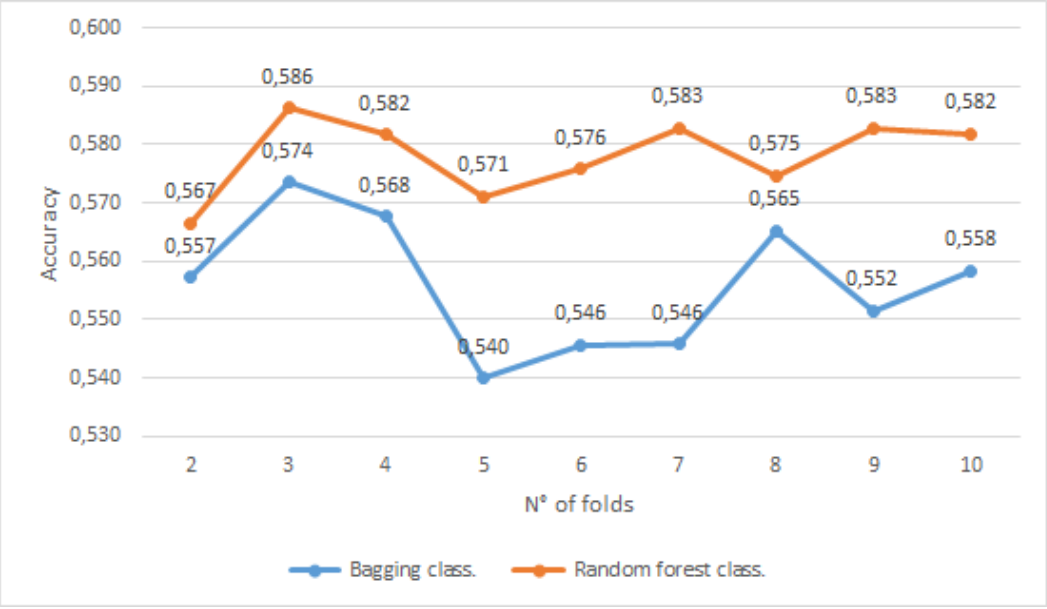

Figure 3.11: CV on Bagging classifier and Random forest classifier

The evaluation phase shows that the Random classifier has a better average accuracy than the second model. In fact, the accuracy of the Random classifier is always higher than that of the Bagging classifier. Consequently, the Random model should be more accurate but it is necessary to compare the confusion matrices in order to choose which model is the best.

Actual \ Predicted	HP	LP	MP	Total
HP	200	22	61	283
LP	27	229	35	291
MP	130	83	76	289
Total	357	334	172	

Actual \ Predicted	HP	LP	MP	Total
HP	210	22	51	283
LP	34	219	38	291
MP	127	83	79	289
Total	371	324	168	

Table 3.2: Confusion matrices: Bagging classifier on the left and Random forest classifier on the right;

As seen before, the models predict very well the classes *LP* and *HP* whereas the *MP* class tends to have some errors. This situation could be depend on certain instances, for example some songs with medium popularity could be similar to those of the low and high popularity classes. The *artist_pop* attribute could influence the popularity of these songs, particularly, an ugly song could be predicted in the high class because it was created by a famous singer.

Finally, the following image shows the tree which is obtained by the Random classifier. For simplicity, only one tree over 10 was considered. The predictors space was divided into 10 regions. The attribute *acousticness* is used as root node and the internal nodes use the *artist_pop* attribute or the songs quality attributes in order to do a split in the tree.

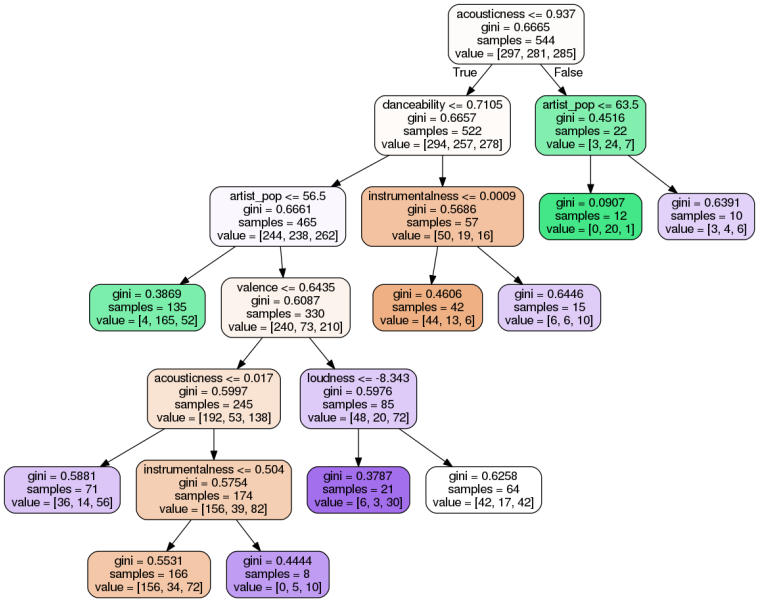

Figure 3.12: Decision tree classifier model

4.3 Ada boost classifier and Stochastic gradient boost classifier

The classification analysis ends with the boosting algorithms. These analysis can be found at the following link Analysis/Classification/boosting/. As mentioned before, boosting is a general ensemble method that creates a strong classifier from a number of weak classifiers. This is done by building a model from the training data, then creating a second model that attempts to correct the errors from the first model. Models are added until the training set is predicted perfectly or a maximum number of models are added. Modern boosting methods build on AdaBoost, most notably stochastic gradient boosting machines.

Similar to Ada regressor, the Ada classifier has a parameter for the so-called base estimator. The best versions of the Decision tree classifier and the Logistic regression are used to set the base estimator parameter. The training stage was completed in 2 minutes and the models obtained have respectively an accuracy of 56% for the first version of the GridSearchCV whereas the second version has a mean validation score of 57%.

GridSearchCV: Ada boost classifier outcomes

```
# set of parameters to test
DTC = DecisionTreeClassifier(**best_param_DTC)
LR = LogisticRegression(**best_param_LR)

param_ADA = {"base_estimator": [DTC, LR],
             "n_estimators": [3],
             "algorithm": ['SAMME', 'SAMME.R'],
             "random_state": [99],
            }

-- Grid Parameter Search via 5-fold CV

GridSearchCV took 1.06 seconds
for 4 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.563 (std: 0.043)
Parameters: {'n_estimators': 3, 'base_estimator':
             LogisticRegression(C=1.0, class_weight=None,
```

```
                dual=False, fit_intercept=True,
                intercept_scaling=1, max_iter=100,
                multi_class='ovr', n_jobs=1,
                penalty='l2', random_state=None,
                solver='newton-cg', tol=0.0001,
                verbose=0, warm_start=False),
                'random_state': 99, 'algorithm': 'SAMME'}

-- Grid Parameter Search via 10-fold CV

GridSearchCV took 1.86 seconds
for 4 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.572 (std: 0.063)
Parameters: {'n_estimators': 3, 'base_estimator':
                LogisticRegression(C=1.0, class_weight=None,
                dual=False, fit_intercept=True,
                intercept_scaling=1, max_iter=100,
                multi_class='ovr', n_jobs=1,
                penalty='l2', random_state=None,
                solver='newton-cg', tol=0.0001,
                verbose=0, warm_start=False),
                'random_state': 99, 'algorithm': 'SAMME'}
```

The above results are stored in the file called *ada_boost_classifier.ipynb*. The models just created use the same estimator as base estimator parameter, in fact, both models use the Logistic regression. The second model tends to have better accuracy than the first one, therefore only the second model was considered in the evaluation stage.

On the other hand, the outcomes of the Stochastic estimator are stored in the file called *stochastic_gradient_boosting_classifier.ipynb*. The creation phase took over 3 minutes, in fact, this is the reason why the training stage was performed only in the first version of the GridSearchCV. The model obtained has a mean validation score of 60%.

GridSearchCV: Stochastic GB classifier outcomes

```
# set of parameters to test
param_SGBC = {"loss": ['deviance'],
              "n_estimators": [100],
              "max_depth": [2, 5, 10, 15],
              "min_samples_split": [2, 5, 10, 15],
              "min_samples_leaf": [ 5, 10, 15],
              "max_leaf_nodes": [None, 5, 10, 15],
              "random_state": [99],
             }

-- Grid Parameter Search via 5-fold CV

GridSearchCV took 228.83 seconds
for 192 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.600 (std: 0.057)
Parameters: {'loss': 'deviance', 'max_leaf_nodes': 15,
             'min_samples_leaf': 5, 'n_estimators': 100,
             'random_state': 99, 'min_samples_split': 2,
             'max_depth': 10}
```

Once the models were created, the analysis proceed with the evaluation phase. In general, the Stochastic model has better performance than the Ada boost model, in fact, the accuracy of the Stochastic model is higher than that of the other model.

As seen before, the confusion matrices were calculated on both models in order to show the prediction errors. The models predict very well the classes *HP* and *LP* whereas the *MP* class tends to have several errors. In particular, the stochastic model has a better accuracy and it produces less errors in the *MP* class than the Ada model. On the other hand, the Ada model predicts better the *LP* and *HP* classes than the Stochastic model, but the Ada confusion matrix is the same obtained by the Bagging classifier.

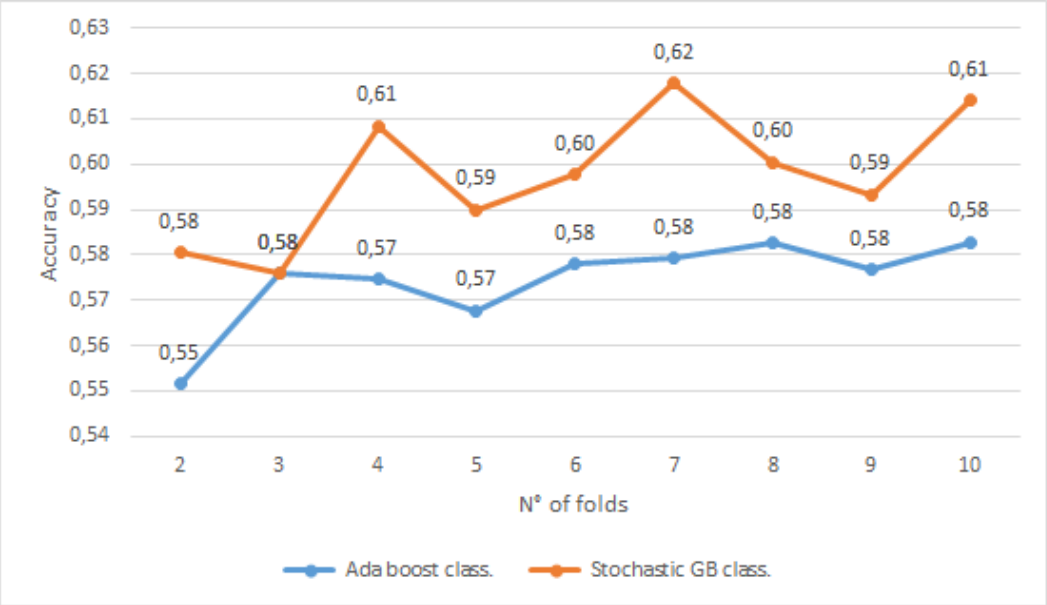

Figure 3.13: CV on Ada boost classifier and Stochastic GB classifier

		Predicted			
		HP	LP	MP	Total
Actual	HP	200	22	61	283
	LP	27	229	35	291
	MP	130	83	76	289
	Total	357	334	172	

		Predicted			
		HP	LP	MP	Total
Actual	HP	179	19	85	283
	LP	26	202	63	291
	MP	99	62	128	289
	Total	304	283	276	

Table 3.3: Confusion matrices: Ada boost classifier on the left and Stochastic GB classifier on the right;

Chapter 4

INTERPRETATION OF MODEL AND SIMULATION

1 Summary

This last chapter gives an answer to the myth of four chords by analyzing the Decision tree model already obtained before. In particular, the rules of the model were extracted and interpreted. Finally, some particular instances were created in order to prove that there is not a relationship between chord progressions and songs popularity.

2 Model interpretation

The last phase of this project focus the attention on the analysis of the model already obtained before. In particular, two different tasks were performed: the first one concerns the interpretation of the model and the extrapolation of the rules used to predict the popularity of songs; the other task is called *Simulation* in which the model was tested with particular instances in order to check if a relationship exist or not between the chord progressions and the songs popularity. The sources codes can be found at the following link Simulation on Github.

For simplicity, the model considered is the *Decision tree regressor*, in fact the decision tree obtained by means of this estimator is generally easier to understand than others regression models. The model was recreate by means of the hyper-parameters already obtained before, i.e.:

Decision tree regressor best parameters

```
"criterion": "mse", "max_depth": 15,
"max_features": 6, "max_leaf_nodes": 10,
"min_samples_leaf": 10, "min_samples_split": 10
```

Once the model was recreated, the next step concerns the extrapolation of the model rules. These rules are used to predict the popularity of songs. The following image shows the decision tree regressor model which was used to extract the rules.

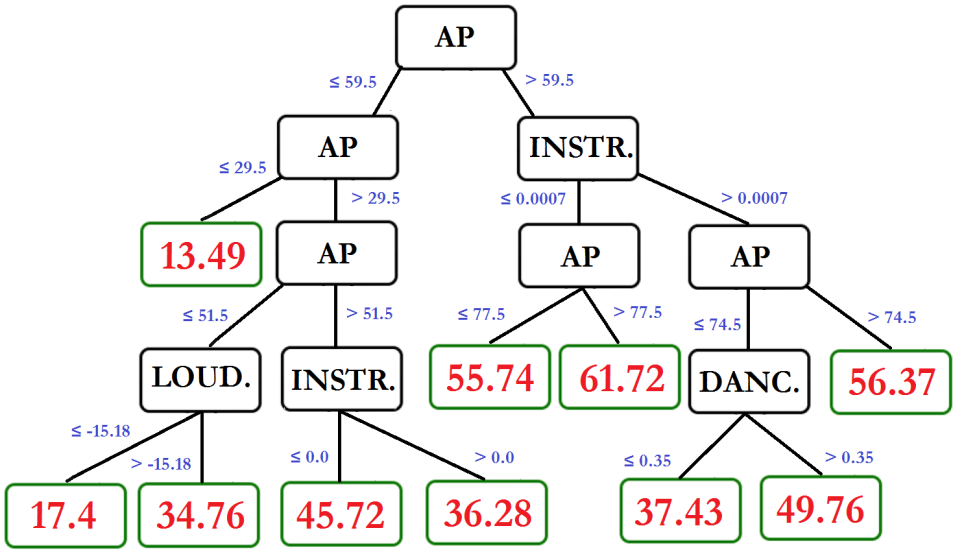

Figure 4.1: Decision tree regressor model

As seen before, the model splits the predictors space into ten regions. Each region specifies the value of songs popularity which will be assigned to the respectively instances. In particular, the model uses a subset of the available attributes and the majority of splits are based on the *artist_pop* attribute. Furthermore, the chord progressions attribute was not used as split node whereas it looks like the model prefers to use the qualitative attributes such as for example the *instrumentalness* and the *danceability*. The following image shows the rules extracted from the model.

	Artist pop.	Instrum.	Dance.	Loud.	freq %
RULE 1	]59.5 , 77.5]	≤ 0.0007			30.01
RULE 2	> 77.5	≤ 0.0007			16.80
RULE 3	]29.5 , 51.5]			> -15.18	14.37
RULE 4	]59.5 , 74.5]	> 0.0007	> 0.35		9.39
RULE 5	]51.5 , 59.5]	> 0.0			7.53
RULE 6	> 74.5	> 0.0007			6.26
RULE 7	]51.5 , 59.5]	≤ 0.0			6.14
RULE 8	< 29.5				5.91
RULE 9	]59.5 , 74.5]	> 0.0007	≤ 0.35		2.43
RULE 10	]29.5 , 51.5]			≤ -15.18	1.16

Figure 4.2: Decision tree regressor rules

The majority of rules use the *artist_pop* attribute to determine which will be songs popularity values. The most frequently rules are the top 3. In particular, the rules 1 and 2 use the *instrumentalness* level in order to determine songs popularity. On the other hand, the rule 3 considers the *loudness* attribute. The first rule and the second one are related to songs which have medium/high popularity whereas the third rule is used to predict the less popular songs.

3 Conclusions

Based on the rules extracted, we perform a simulation task where we select instances describing real songs and investigate the model's behaviour. We consider two groups of instances: the first one contains highly popular songs, the second one contains little popular songs. Each group contains each of the chord progressions analyzed in this thesis. The songs from popular artists are those having a value of *artist_pop* 75, whereas the songs from the less popular artists are those having *artist_pop* 35. These prototypes are stored in two different files and used as input to the Decision tree model.

We find that the most popular artists use more chord progressions than the less popular artists. Moreover, the songs' harmonic structures do not influence significantly songs popularity. Consequently, there is not a chord progression which increases the chances of a song to become popular. These results suggest that songs popularity mainly depend on the popularity of its creator.

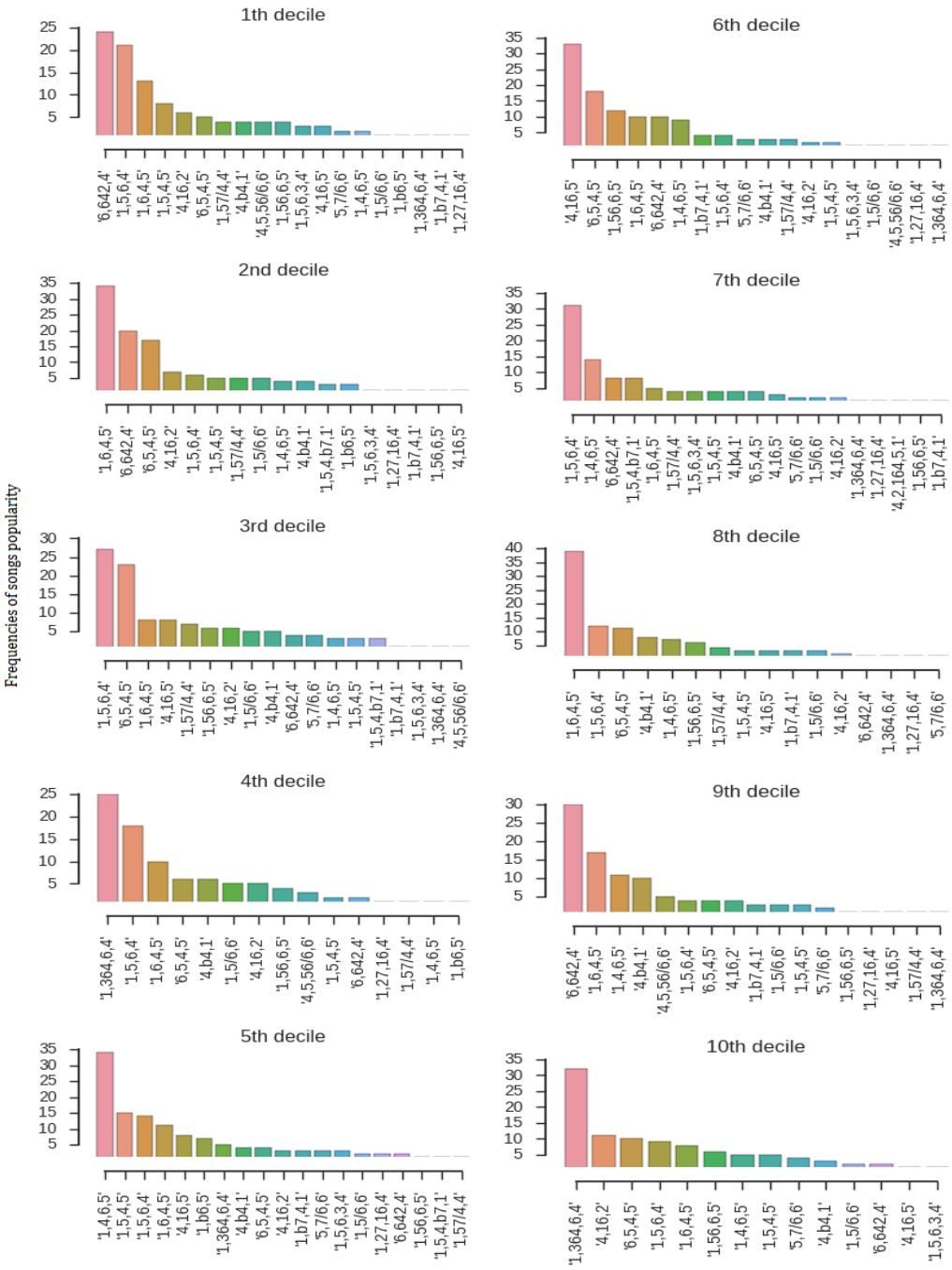

Figure 4.3: Frequencies of songs popularity for each decile of songs popularity

As mentioned before, the myth of four chords asserts that all popular songs consists of the same 4 chords. The most common chord progression is *1,5,6,4* and in fact this progression appears in each decile of songs popularity. In general, this pattern could be used to play several songs but this choice does not ensure if they will be popular or not. Similar to *1,5,6,4*, others chord progressions are used regardless of songs popularity. The choice of chords is just one aspect of writing a song. The chord progression provides the harmony of a song, but music contains many other elements. As well as harmony, there is melody, rhythm, tempo, meter, dynamics, articulation, and timbre. And beyond the music, many artists feel that the story told by the lyrics is just as important to the success of a song.

Bibliography

[1] G. Casella, S. Fienberg, I. Olkin *An Introduction to Statistical Learning*, Springer texts in statistics

[2] K. Markham http://www.dataschool.io/15-hours-of-expert-machine-learning-videos/ *Data school*

[3] H. Hamilton http://www2.cs.uregina.ca/ dbd/cs831/index.html *Knowledge Discovery in Databases*

[4] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth *From Data Mining to Knowledge Discovery: An overview* in Advances in Knowledge discovery and Data Mining. Fayyad U, Piatetsky-Shapiro G, Smyth P, Uthurusamy R. MIT Press. Cambridge, Mass.. 1996 pp. 1-36

[5] U. Fayyad *Data Mining and Knowledge Discovery: Making Sense Out of Data* in IEEE Expert October 1996 pp. 20-25

[6] E. Simoudis *Reality Check for Data Mining* in IEEE Expert October 1996 pp. 26-33

[7] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth *The KKD Process for Extracting Useful Knowledge from Volumes of Data* in Communications of the ACM, November 1996/Vol 39, No.11 pp.27-34

[8] Nick Long https://www.musical-u.com/learn/four-chords-and-the-truth/ *Four Chords and the Truth* January 2011

[9] jack Hook https://www.theodysseyonline.com/quick-guide-chords *A Quick Guide To Chords* May 2016

[10] http://www.rollingstone.com/music/lists/the-500-greatest-songs-of-all-time-20110407/the-cure-just-like-heaven-20110526 500 Greatest Songs of All Time

[11] F. Pellacini http://pellacini.di.uniroma1.it/teaching/fondamenti14/lectures.html *python*

[12] https://docs.python.org/3.5/tutorial/index.html *the python tutorial*

[13] https://doc.scrapy.org/en/latest/intro/tutorial.html *Scrapy Tutorial*

[14] Jason Brownlee http://machinelearningmastery.com

[15] https://en.wikipedia.org/wiki/Data_mining

[16] https://en.wikipedia.org/wiki/Music_theory#Chord

[17] http://jupyter.org/

[18] http://docs.python-requests.org/en/master/

[19] http://ipython.readthedocs.io/en/stable/interactive/magics.html

[20] http://scikit-learn.org/stable/modules/classes.html#module-sklearn.ensemble

[21] http://seaborn.pydata.org/api.html

[22] http://pandas.pydata.org/pandas-docs/stable/10min.html#csv