UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI INGEGNERIA INFORMATICA, MODELLISTICA, ELETTRONICA E SISTEMISTICA

DIMES

University degree in

Computer Engineering for the Internet of Things

Course of

Big Data Analytics

Project

A tracks-grouping service for Spotify users relying on clustering and classification

GitHub Link

click here

Professors

Andrea Tagarelli

Antonio Caliò

Candidate

Francesco Raco
214443

INDEX

INTRODUCTION

Spotify become very popular thanks not only to the high number and variety of tracks and podcasts it offers, but also to the possibility it gives to users to organize their favorite songs and podcasts as they prefer. In such app, each user can add existing playlists to its favorite ones and create its own playlist by putting into it its favorite songs/podcasts. Also, each user has a “favorite songs” playlist: it contains all the songs the user gives a “like”, which are automatically added into such playlist.
Since this particular playlist will likely grow during time in terms of items, it would be nice to have a service for the Spotify final user to rearrange the tracks such playlist contains by grouping them into multiple smaller playlists, each containing tracks of the same genre or tracks of few similar genders: the first service we implemented does exactly this. First, we implement a classifier that can predict the genre of a track given its audio features. Then, through the **spotipy** API (a python library for Spotify), we get the “favorite songs” playlist of a given user, use the classifier to predict the genre of each track, group the tracks by genre and then suggest the user to split the playlist into multiple playlists, each belonging to a “macro-genre”: since there’re tons of possible music genres, we necessarily have to group them into a smaller subset where each genre is a representative of a lot of real genres very similar to each other.
In the following, the implementation of this service is explained in detail.

THE CLASSIFIER

The initial idea was to take a dataset found in Kaggle which contains audio features for almost 600.000 Spotify tracks, and through the spotipy API to add the genre of each track, and then preprocess such obtained dataset and use it to train the classifier. Unfortunately, in the Spotify API there's no way to retrieve the genre of the single track; as an alternative, we thought to retrieve the genres of the album the track belongs to (which is possible), but for the majority of retrieved album objects the genre was missing (null value).
So, for assigning the genre to tracks, we decided to take another Kaggle dataset containing for each genre the average audio features values for the songs belonging to such genre. The idea is to implement a clustering algorithm in order to find the proper number of "macro-genres" to group the tracks, since in this dataset we've almost 3000 genres and it would be meaningless to suggest the user to create playlists with very few tracks. Then, once the clustering is done, we'll use the centroids to assign the genre to each track in the previous dataset, in the following way: for each track, we'll compute the difference in terms of audio features with respect to all the centroids, and then assign to the track the genre of the centroid which minimizes such difference.
So, let's see the clustering. The genres dataset contains the following attributes:

- **Mode:** the most frequent mode (major/minor) of songs in the genre.
- **Genres.**
- **Acousticness:** how acoustic the songs in genre are.
- **Danceability:** how suitable the songs in genre are for dancing.
- **Duration_ms:** average time duration of songs in milliseconds.
- **Energy:** how energetic the songs in genre are.
- **Instrumentalness:** ratio of instrumental sounds.
- **Liveness:** presence of audience.
- **Loudness:** how loud the songs in genre are in dB.
- **Speechiness:** ratio of spoken words.
- **Tempo:** tempo in BPM.
- **Valence:** the positivity of songs in genre.
- **Popularity:** average popularity of songs in genre.
- **Key:** the most frequent key in songs in genre.

Other than "genres", which is a categorical attribute, the other attributes are all numerical: mode is a binary one, key is a discrete one, and the others are all continuous attributes.
Genres attribute of course has been removed, since it has a different value for each data object. Then, we initially thought to perform the following operations:

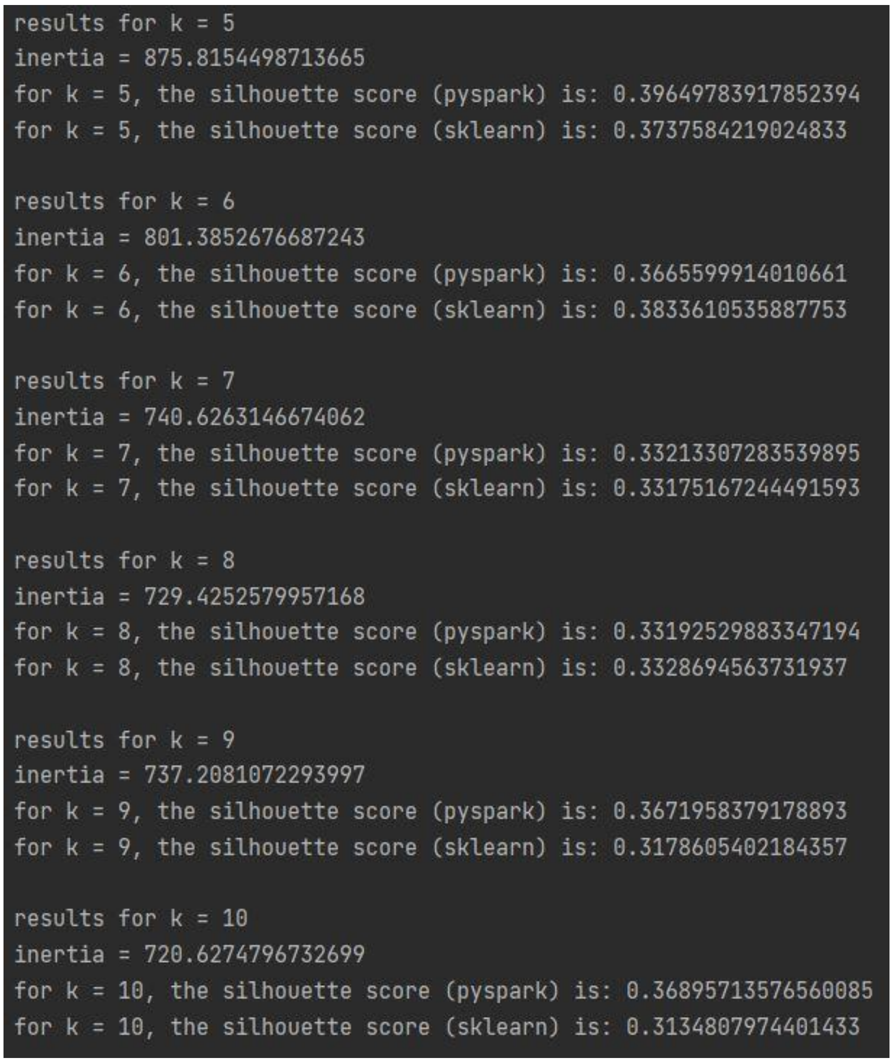
- Remove the **mode** attribute, because it is almost homogeneous: 2477 genres out of 2973 have the same value.
- Remove the **Speechiness** attribute, because its values range from 0 to 1, and 2489 genres out of 2973 have value between 0.02 and 0.12.

But this didn't improve the performance of clustering algorithms, they became a little worse. Then, we thought to substitute Speechiness with **log(Speechiness)** in order to have an attribute with a less skewed distribution, but also this didn't improve the performance. So, in the end I decided to keep all the attributes, because they're few (13) and they all have an important meaning.
As final step regarding the preprocessing, we grouped the features into an attribute containing vectors of doubles, which is the format needed as input for machine learning algorithms, and then we scaled such values to have all values between 0 and 1. This last step has been implemented through a MinMaxScaler in Spark.

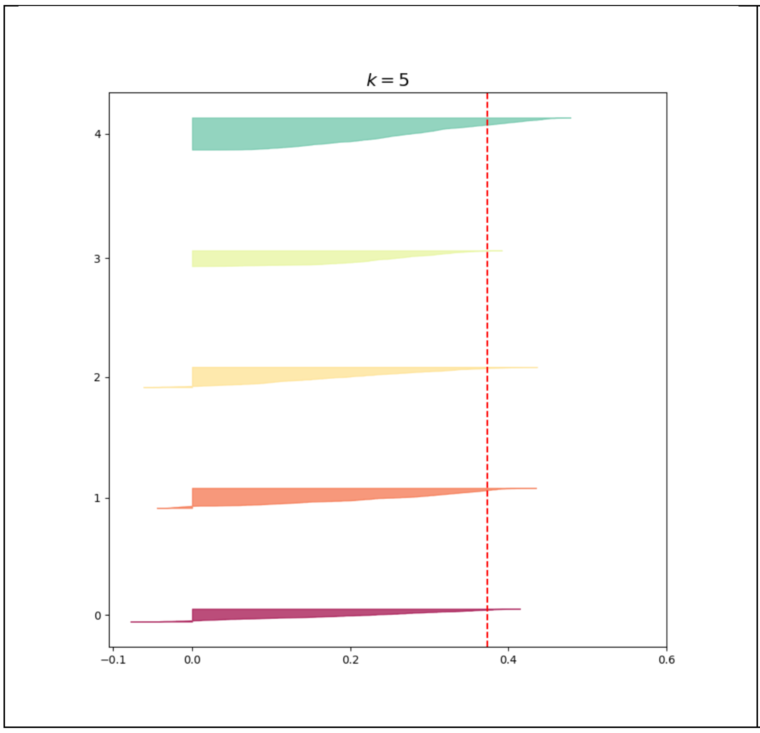
Clustering

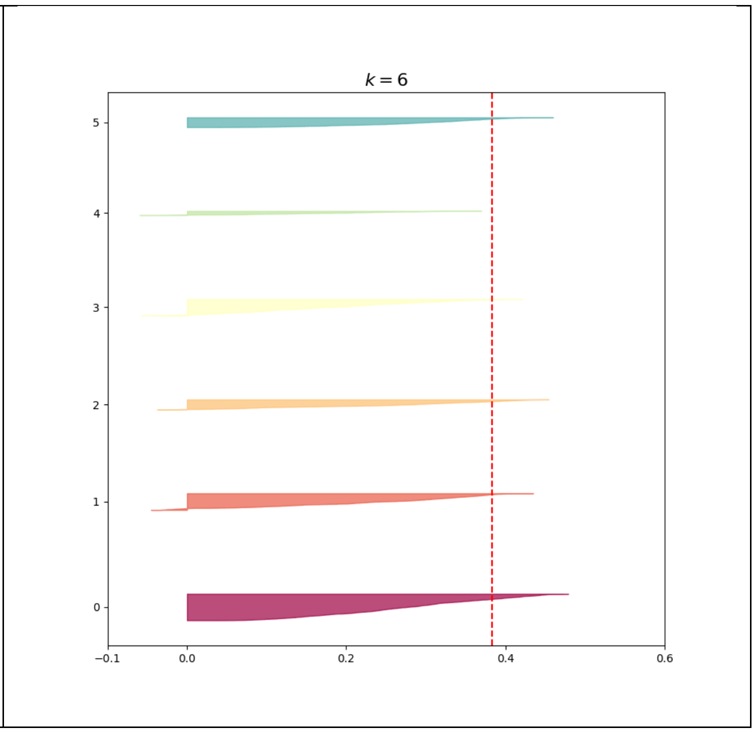
At this point, we started with the clustering. We utilized both KMeans and DBSCAN, to see which one performs better, trying both algorithms with different sets of hyperparameters and utilizing the silhouette score for performances measure.
Let's first show results regarding DBSCAN. We tried this algorithm by varying the eps and the minimum number of samples needed to label a zone as a dense one. The results obtained are not so good (you can see them in the "DBSCAN_results.txt" file): for each configuration, we either obtained an acceptable number of clusters but a silhouette score near to 0, or a very small number of clusters (very often 3 clusters, one of which consisting in the outliers) with a good silhouette score (0.3 or above). So, DBSCAN ended up being not a good choice as clustering algorithm, making KMeans the proper choice. At this point we tried KMeans with different values of K (number of clusters). We used both implementations provided by Spark and Scikit-learn, which provided very similar results, and relied on the last one being it crucial for choosing the proper K value: this because Scikit-learn allows to get the silhouette values associated to each sample of the dataset, while this is not possible through Spark.

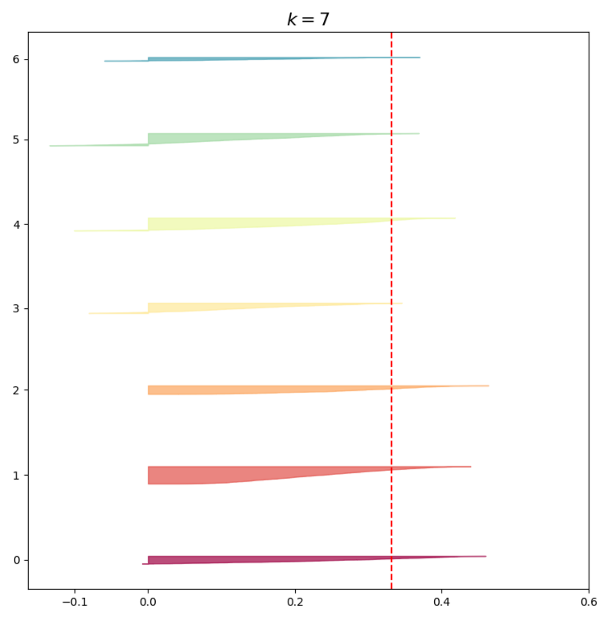
Such values were utilized to create the silhouette diagrams for growing K values in order to choose the best one: this was necessary because for values between 5 and 10, which we think are the most suited ones for the service we want to implement, the silhouette scores of the dataset were very similar (around 0.35). In the following, we first show the silhouette scores we just mentioned, and then the silhouette diagrams of our interest.

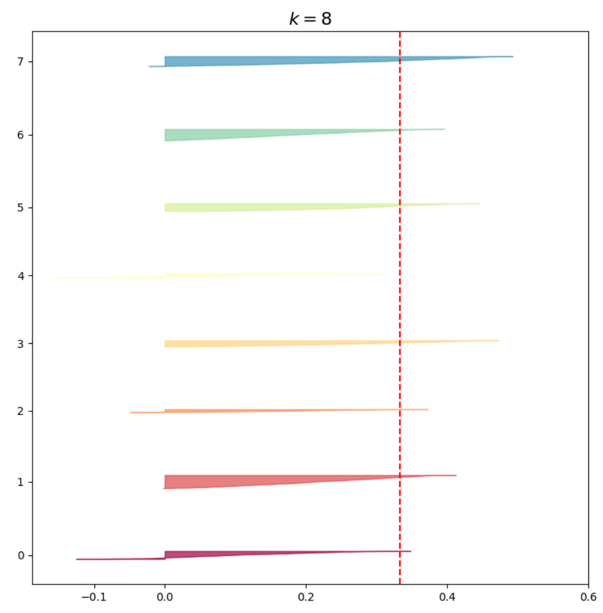


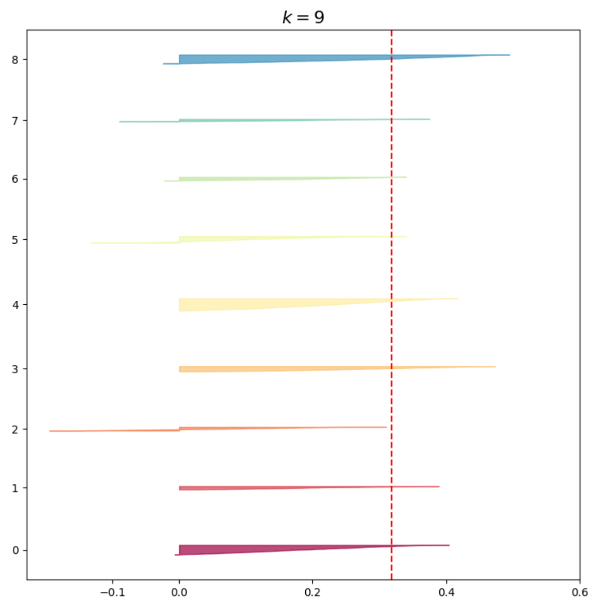
```
results for k = 5
inertia = 875.8154498713665
for k = 5, the silhouette score (pyspark) is: 0.39649783917852394
for k = 5, the silhouette score (sklearn) is: 0.3737584219024833

results for k = 6
inertia = 801.3852676687243
for k = 6, the silhouette score (pyspark) is: 0.3665599914010661
for k = 6, the silhouette score (sklearn) is: 0.3833610535887753

results for k = 7
inertia = 740.6263146674062
for k = 7, the silhouette score (pyspark) is: 0.33213307283539895
for k = 7, the silhouette score (sklearn) is: 0.33175167244491593

results for k = 8
inertia = 729.4252579957168
for k = 8, the silhouette score (pyspark) is: 0.33192529883347194
for k = 8, the silhouette score (sklearn) is: 0.3328694563731937

results for k = 9
inertia = 737.2081072293997
for k = 9, the silhouette score (pyspark) is: 0.3671958379178893
for k = 9, the silhouette score (sklearn) is: 0.3178605402184357

results for k = 10
inertia = 720.6274796732699
for k = 10, the silhouette score (pyspark) is: 0.36895713576560085
for k = 10, the silhouette score (sklearn) is: 0.3134807974401433
```

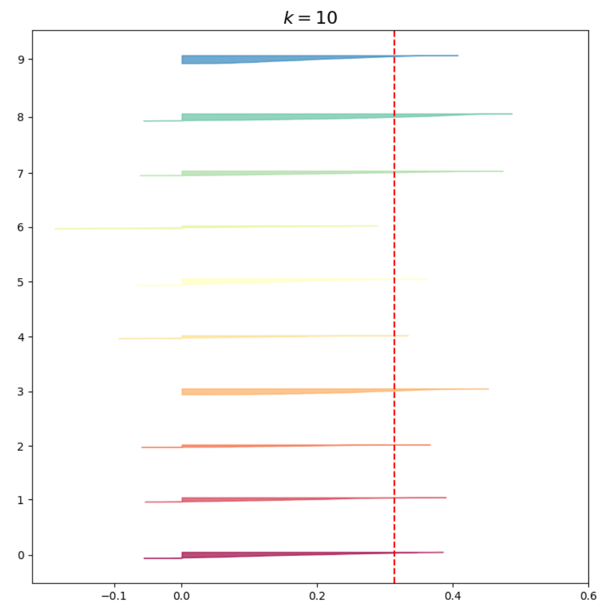






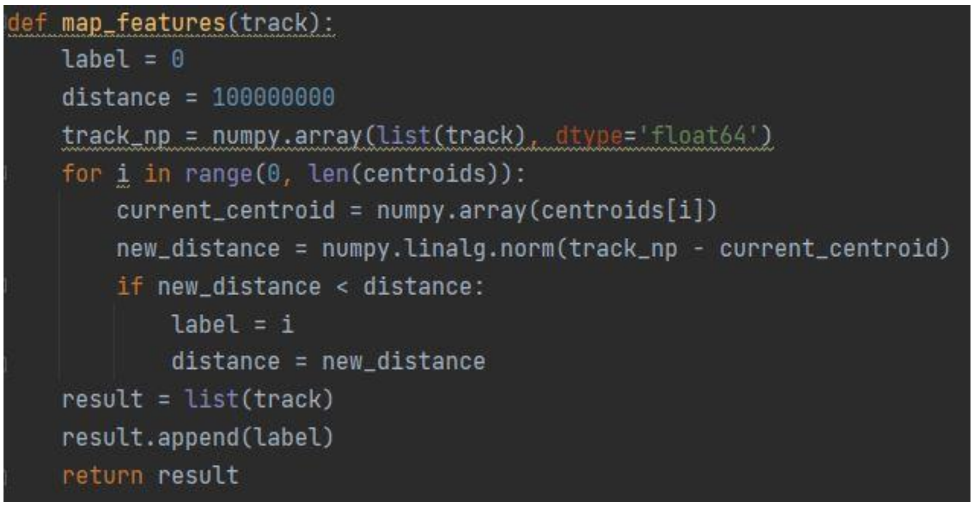






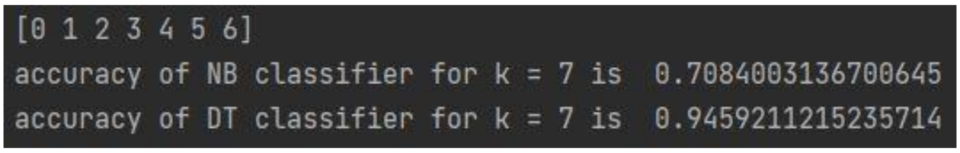
By looking at these diagrams, it's clear that K=5 and K=7 are the best choices. We chose K=7 since we think 5 would be too small, and 7 is a good tradeoff for achieving a proper grouping of tracks.

At this point, it was necessary to assign to each track its macro-genre. We did it by defining a function called "map_features" and using it for the assignment: the DataFrame containing the tracks was first preprocessed in the same way as the genre classifier (putting values into a vector of doubles and using a MinMaxScaler to scale them to have values between 0 and 1). Then, we transformed the dataframe into a Spark RDD, then the map_features function was passed to the RDD’s map function. The resulting RDD was then used to create a new DataFrame, which we utilized for training and testing the classifier. In the following we see the code regarding the map_features function, where centroids is an array containing the centroids obtained through the clustering



```python
def map_features(track):
    label = 0
    distance = 100000000
    track_np = numpy.array(list(track), dtype='float64')
    for i in range(0, len(centroids)):
        current_centroid = numpy.array(centroids[i])
        new_distance = numpy.linalg.norm(track_np - current_centroid)
        if new_distance < distance:
            label = i
            distance = new_distance
    result = list(track)
    result.append(label)
    return result
```

Once the assignment was done, we went through the implementation of the classifier. We tried both Decision Tree (by varying the maximum depth and the impurity measure) and the Naïve Bayes (by varying the smoothing). We ended up with the following results

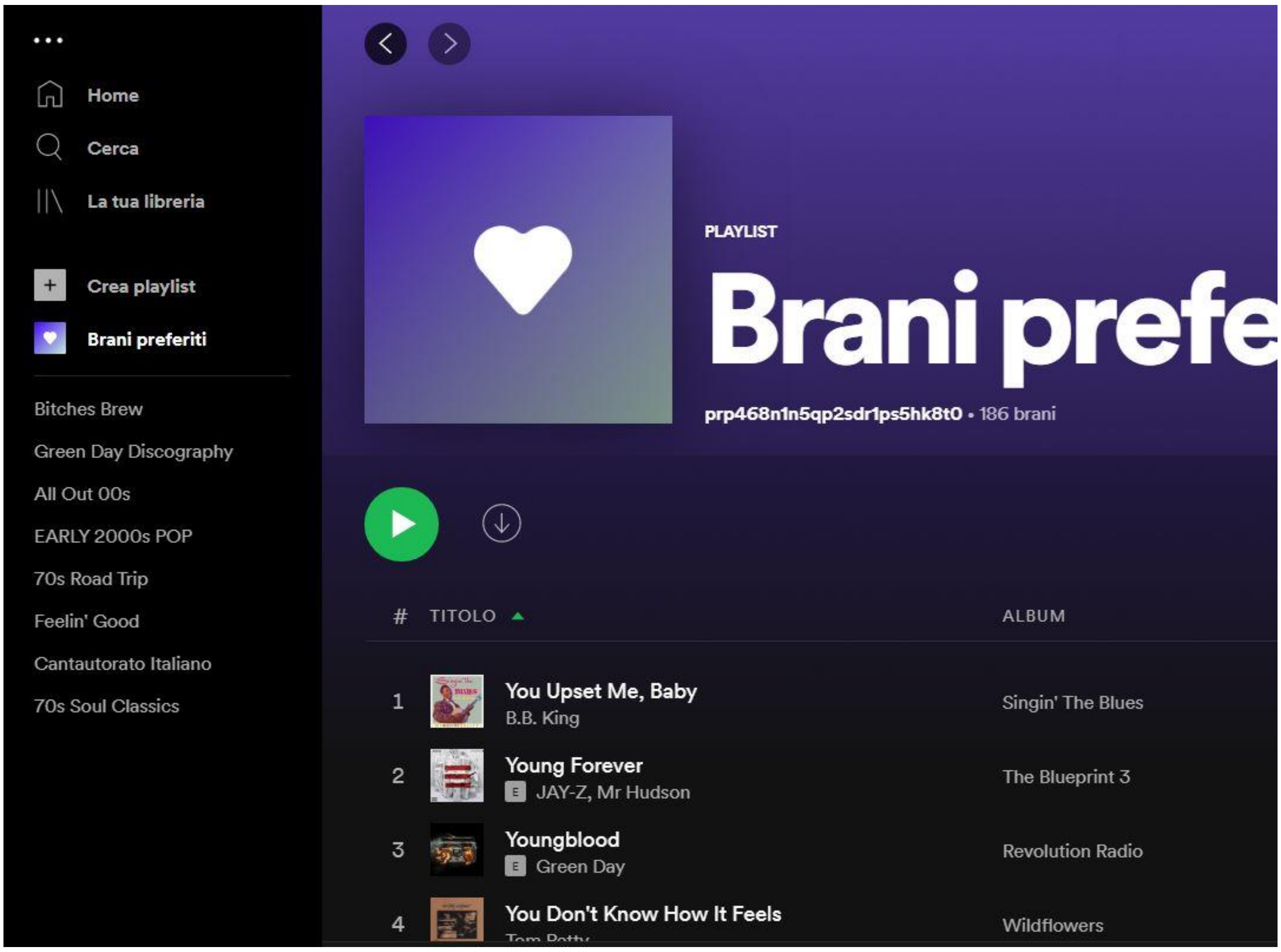


```
[0 1 2 3 4 5 6]
accuracy of NB classifier for k = 7 is  0.7084003136700645
accuracy of DT classifier for k = 7 is  0.9459211215235714
```

In the first row we see the different values of genres we ended up with: this proves the correctness of the genre assignment since we got 7 different values. Then, we see the accuracy of the two classifiers we tested (the tracks dataset was split by assigning the 80% of them to the training set and the rest to the test set, a cross-validation was used). It turns out that Decision Tree is the best choice: we get 94.6% accuracy, which is a very good result.
We tried also to implement it by first discretizing the dataset values by using a Spark Bucketizer, but the accuracy was slightly lower (around 91%). Since the number of attributes is low, we think that using the non-discretized version is a better choice due to the better accuracy.

THE SERVICE

At this point, we implemented the final part of this first service: by using the **spotipy** library, we first collected all the tracks of the "favorite songs" of one of us and classified them through the classifier previously implemented. Finally, for each cluster we created a playlist and added each track to the proper playlist. In the following, we can see the status of the Spotify profile prior to and after the execution of the script



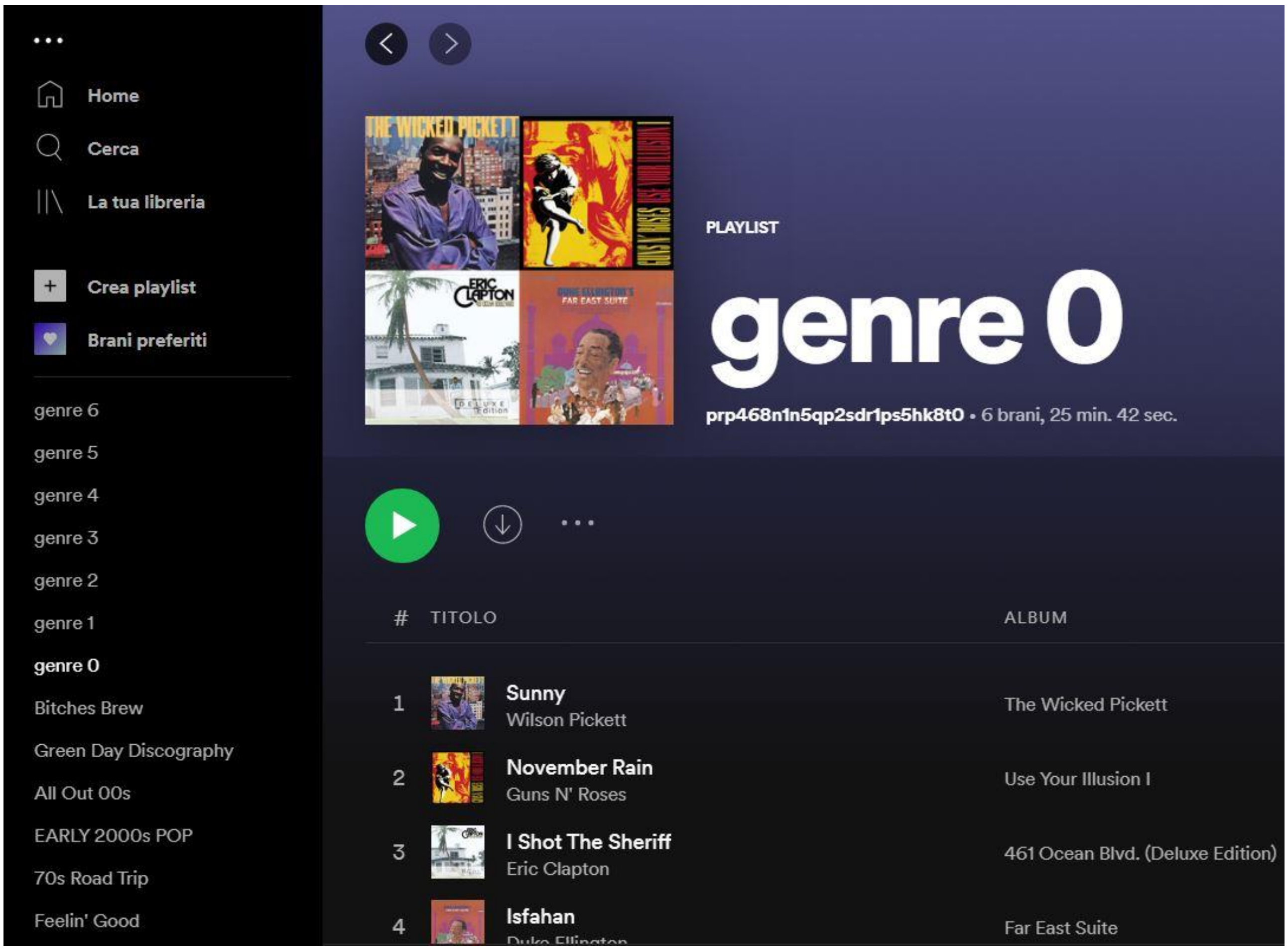
Bitches Brew
Green Day Discography
All Out 00s
EARLY 2000s POP
70s Road Trip
Feelin' Good
Cantautorato Italiano
70s Soul Classics

PLAYLIST

Brani prefe

prp468n1n5qp2sdr1ps5hk8t0 • 186 brani

#	TITOLO	ALBUM
1	You Upset Me, Baby B.B. King	Singin' The Blues
2	Young Forever E JAY-Z, Mr Hudson	The Blueprint 3
3	Youngblood E Green Day	Revolution Radio
4	You Don't Know How It Feels	Wildflowers



genre 0

prp468n1n5qp2sdr1ps5hk8t0 • 6 brani, 25 min. 42 sec.

#	TITOLO	ALBUM
1	Sunny Wilson Pickett	The Wicked Pickett
2	November Rain Guns N' Roses	Use Your Illusion I
3	I Shot The Sheriff Eric Clapton	461 Ocean Blvd. (Deluxe Edition)
4	Isfahan	Far East Suite

As we can see, all the playlist have been created correctly; some of them are empty, meaning that for such genres there weren't no tracks. In the "filled" ones, we can find all the songs belonging to the "favorite songs" playlist. If we take a look, we see that in each playlist there're some tracks that are not so similar apparently, but due to the way we proceeded, we can assume that the audio features are at least a little similar. We tried also to increase the number of clusters to see if the grouping would become better, but we ended up with a lot of empty playlist and some others with one or two tracks each other; you can see this in an attached text file.
In conclusion, the implemented service cannot group perfectly the songs per genre, but it can still be useful to group the tracks belonging to the "favorite songs" playlist into smaller playlists which will be a little more heterogeneous than expected, but still useful for the end user since both the clustering and the classification activities have been mainly focused on audio features of a huge amount of tracks.