

Building a Classifier Model for The Billboard Hot 100

Gianmarco Forcella (1725967)

Taranciuc Gabriel (1693558)

Sommario

1. Motivation.....	3
2. Project's Goal.....	3
2.1 Setup	3
3. Dataset overview.....	4
3.1 Dataset Description	4
3.2 Scraping from the web.....	5
4. Choosing The Algorithm.....	8
4.1 Decision Tree	8

1. MOTIVATION

Music has always been an interesting market for any programming application: there are lots of researches that try to create a predictive model that is able to guess some contest's winner, or to understand which song somebody is listening to (e. g. Shazam).

The reason we chose, among all subjects, to work on a Machine Learning project that involved music is that we are really passionate about it and we wanted to experiment our new knowledge on something we are familiar with.

2. PROJECT'S GOAL

Music, as already noted, has always been an interesting market for any programming application. But it's not just that: though the retail compartment might be suffering a crisis, music continues **to expand** and reach new platforms, and **so do charts**: most of the modern charts, in fact, also enlist songs that are only available via Streaming Services such as Spotify.

For this project, we decided to build a classifier whose goal is to determine whether a certain song can enter the American Chart or not.

Why considering only the American charts and not any other country? There are two main reasons:

1. The US Market **offers more data** than any other country: for instance, we asked the [FIMI](#) if it was possible to have the Italian chart as a .csv file and we were told that it wasn't possible, due to the fact that the chart contains sensible data;
2. It was one of the first charts that started including, back in 2007, Streaming Services in their calculations.

The chart used as a reference will be the **Billboard Top 100**, which gets an update every week since 1936 and is considered one of the most famous all over the world.

2.1 SETUP

As for the project's setup, this is what was used for developing our classifier:

- **Python 3.x**;
- **Visual Studio Code** as our Script Editor;
- **SciKit-Learn** for the ML algorithms;
- **Spotipy** as the wrapper for Spotify's APIs;
- [This](#) dataset for the information regarding the Billboard. This is the most complete dataset available about it, since it collects data from the '70s up to 2018.

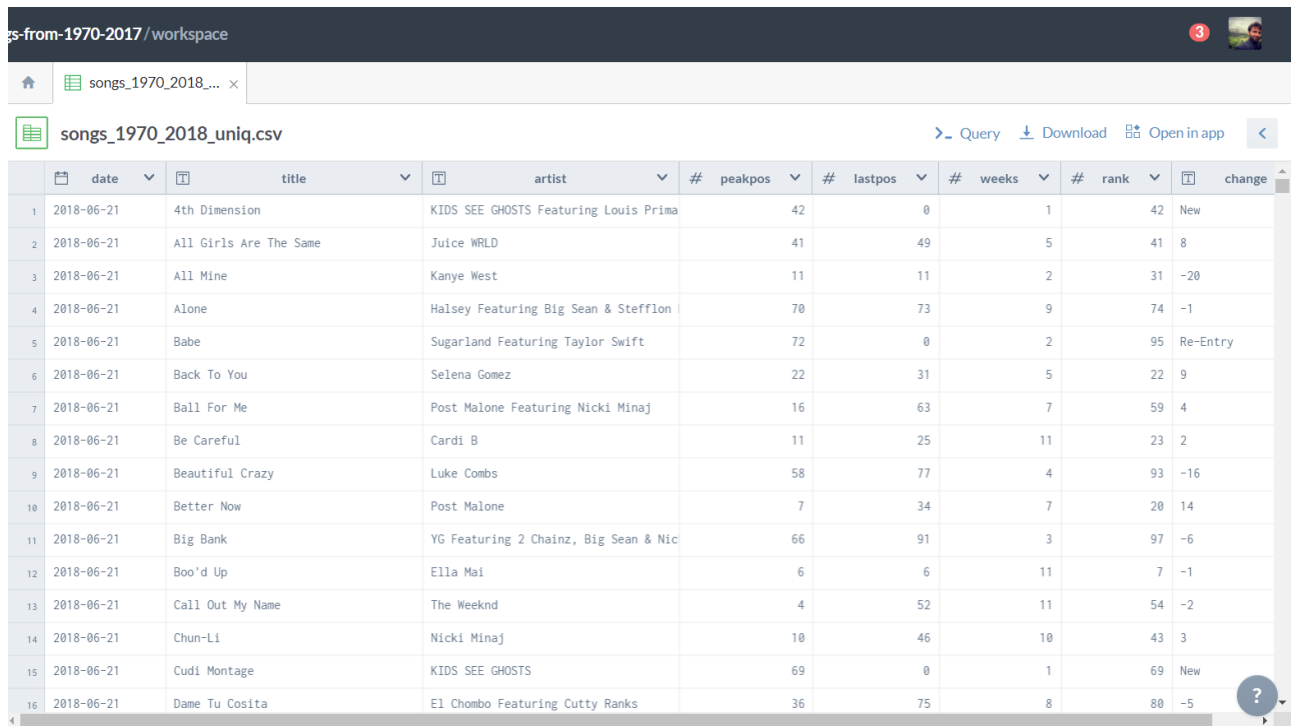
The final code can be checked out [here](#), in our GitHub Repository.

3. DATASET OVERVIEW

We will now have a brief description of the dataset and why we felt that the data proposed wasn't enough.

3.1 DATASET DESCRIPTION

The dataset, that from now on will also be called **Billboard's Chart**, contains about 20104 rows. In the following screenshot, we present an insight of the available columns on it.



	date	title	artist	# peakpos	# lastpos	# weeks	# rank	change
1	2018-06-21	4th Dimension	KIDS SEE GHOSTS Featuring Louis Prima	42	0	1	42	New
2	2018-06-21	All Girls Are The Same	Juice WRLD	41	49	5	41	8
3	2018-06-21	All Mine	Kanye West	11	11	2	31	-20
4	2018-06-21	Alone	Halsey Featuring Big Sean & Stefflon	70	73	9	74	-1
5	2018-06-21	Babe	Sugarland Featuring Taylor Swift	72	0	2	95	Re-Entry
6	2018-06-21	Back To You	Selena Gomez	22	31	5	22	9
7	2018-06-21	Ball For Me	Post Malone Featuring Nicki Minaj	16	63	7	59	4
8	2018-06-21	Be Careful	Cardi B	11	25	11	23	2
9	2018-06-21	Beautiful Crazy	Luke Combs	58	77	4	93	-16
10	2018-06-21	Better Now	Post Malone	7	34	7	20	14
11	2018-06-21	Big Bank	YG Featuring 2 Chainz, Big Sean & Nic	66	91	3	97	-6
12	2018-06-21	Boo'd Up	Ella Mai	6	6	11	7	-1
13	2018-06-21	Call Out My Name	The Weeknd	4	52	11	54	-2
14	2018-06-21	Chun-Li	Nicki Minaj	10	46	10	43	3
15	2018-06-21	Cudi Montage	KIDS SEE GHOSTS	69	0	1	69	New
16	2018-06-21	Dame Tu Cosita	El Chombo Featuring Cutty Ranks	36	75	8	80	-5

- **Date** column represents the date which the information is relative to (i. e. the date of that chart info)
- **Title** is the name of the track
- **Artist** is the name of the artist
- **Peakpos** represents the maximum position which that song ever reached in the chart
- **Lastpos** is the position in the chart the entry had in the preceding week
- **Weeks** describes how many weeks the song remained on the billboard
- **Rank**
- **Change** will tell the reader the change in rank in the chart of the song from the preceding week. For example, if the song is a new hit or a returning one there will respectively be "New" or "Re-Entry". Numerical numbers will symbolize the position that the track gained/loss in the last week.

While the columns of this dataset give a lot of information, they are **not representative** of a song at its best, since this data only refers to the performance of a track in the chart. We felt that, in order to describe a song in a Machine Learning system, more accurate data was needed: something that could, somehow, describe the audio features of the hit.

For this reason, starting from the **Billboard's Chart**, we decided to **create our own** dataset.

3.2 SCRAPING FROM THE WEB

In order to build our new dataset, though, first we need to define what will be our features for the new dataset.

As said before, we need something that can describe the audio features of the given track and **Spotify** might be of good help for this.

The Music Streaming giant, in fact, offers [an API](#) that, given the song's ID (referring to the corresponding ID on their database), can return the description of the track with its musical features. The values that we are going to extract from this API are listed below:

Feature	Description
Duration_ms	The duration of the track in milliseconds
Key	The estimated overall key of the track. Integers map to pitches using standard Pitch Class notation. I.e. 0 = C, 1 = C#/Db, 2 = D, and so on. If no key was detected, the value is -1.
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
Acousticness	A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic.
Danceability	Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable.
Energy	Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy.
Instrumentalness	Predicts whether a track contains no vocals. "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. Values above 0.5 are intended to represent instrumental tracks, but confidence is higher as the values approaches to 1.0
Liveness	Detects the presence of an audience in the recording. Higher liveness values represent

	an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.
Loudness	The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typically range between -60 and 0db.
Speechiness	Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), 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, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks.

With that being said, we now present the code that was used for scraping this information from Spotify.

```

1. from spotipy.oauth2 import SpotifyClientCredentials
2. import json
3. import spotipy
4. import requests
5. import pandas
6. import time
7. import csv
8.
9.
10. def splitterFunction(artista):
11.     if "featuring" in artista.lower():
12.         toReturn = artista.lower().split("featuring")
13.         return toReturn[0].strip()
14.     if "feature" in artista.lower():
15.         toReturn = artista.lower().split("feature")
16.         return toReturn[0].strip()
17.     if "&" in artista.lower():
18.         toReturn = artista.lower().split("&")
19.         return toReturn[0].strip()
20.     return artista
21.
22.
23. artistNames = pandas.read_csv("billboard.csv", encoding = 'UTF-8')['artist']
24. trackNames = pandas.read_csv("billboard.csv", encoding = 'UTF-8')['title']
25.
26. client_credentials_manager = SpotifyClientCredentials("75d944d4a2f64af3ada75b8d846451a8", "399a7ee3bdc947b799148755314b4753")
27. spotify = spotipy.Spotify(client_credentials_manager=client_credentials_manager)
28. toSave = []
29. added = set()
30. contatore = 0

```

```

31. firstTime = True
32.
33. for i in range(0, len(artistNames)):
34.     print("Iterazione su " + str(artistNames[i]))
35.     if contatore < 5000 and artistNames[i] and not trackNames[i] in added:
36.         toFind = splitterFunction(artistNames[i])
37.         results = spotify.search(q='artist:' + toFind + " track:" + trackNames[i], type='track')
38.         time.sleep(2)
39.         if results['tracks']['total'] == 0:
40.             print("Non ho trovato nulla per " + str(artistNames[i]))
41.             continue
42.         idTraccia = results['tracks']['items'][0]['id']
43.         analisiTraccia = spotify.audio_features(idTraccia)
44.         contatore += 2
45.         if None in analisiTraccia:
46.             print("ANALISI NON DISPONIBILI PER " + str(trackNames[i]) + " DI " + str(artistNames[i]))
47.             continue
48.         toAppend = []
49.         toAppend.append(trackNames[i])
50.         toAppend.append(artistNames[i])
51.         toAppend.append(1)
52.         toAppend.append(analisiTraccia[0]['danceability'])
53.         toAppend.append(analisiTraccia[0]['energy'])
54.         toAppend.append(analisiTraccia[0]['loudness'])
55.         toAppend.append(analisiTraccia[0]['speechiness'])
56.         toAppend.append(analisiTraccia[0]['acousticness'])
57.         toAppend.append(analisiTraccia[0]['instrumentalness'])
58.         toAppend.append(analisiTraccia[0]['liveness'])
59.         toAppend.append(analisiTraccia[0]['key'])
60.         toAppend.append(analisiTraccia[0]['mode'])
61.         toAppend.append(analisiTraccia[0]['duration_ms']/1000)
62.         with open("datasetFINAL.csv", "a", encoding="UTF-8", newline='') as myfile:
63.             wr = csv.writer(myfile)
64.             if firstTime:
65.                 wr.writerow(("track", "artist", "billboarder", "danceability", "energy", "loudness", "speechiness", "acousticness", "instrumentalness", "liveness", "key", "mode", "duration"))
66.                 firstTime = False
67.             try:
68.                 wr.writerows([toAppend])
69.             except Exception:
70.                 print("Si è verificato un errore mentre cercavo di effettuare qualcosa sull'artista " + str(artistNames[i]) + " che ha fatto la traccia " + str(trackNames[i]))
71.                 myfile.close()
72.                 print("Aggiunto con successo " + str(artistNames[i]))
73.                 added.add(trackNames[i])
74.         if contatore >= 5000:
75.             time.sleep(180)
76.             contatore = 0

```

For the dataset's creation, we decided not to use pandas and use, instead, the built-in statement open()-with-resources.

After loading the two necessary columns from the **Billboard's Chart**, we begin several iterations (as many as the number of tracks loaded) to question Spotify's APIs in order to get all the information we need. To avoid a possible ban from Spotify, we also inserted a counter that would stop the program for 3 minutes if it reached a value of 5000.

Lastly, we convert the **duration_ms** feature to seconds, and set the value presented in line 51 as 1. This is done because it will represent the test feature of the dataset (from there, the name "billboard").

There is still something missing, though: our fresh dataset is just full of **positive** values. If we train whatever Machine Learning algorithm with this dataset, it will be composed of just one class and it will make bad predictions.

For this reason, we searched for a new dataset that had to be "merged" with **datasetFinal**: during our research on Google, we found the link to [this](#) resource which seemed what we needed, since it has 5000 rows.

4. CHOOSING THE ALGORITHM

Having worked in building the dataset and made some feature engineering, it is now time to decide which Machine Learning algorithm to use in order to build our classifier.

4.1 DECISION TREE

The first thing we consider is a Decision Tree. As one can imagine, this algorithm just builds a tree that will make decisions.

```
1. data = pd.read_csv('datasetFINAL.csv', encoding='UTF-8')
2. features = data[["danceability", "energy", "loudness", "speechiness", "acousticness", "instrumentalness", "liveness",
3.                 "key", "mode", "duration"]]
4. target = data["billboard"]
5.
6. train_features, test_features, train_target, test_target = train_test_split(features, target)
7.
8. clf = tree.DecisionTreeClassifier()
9. clf = clf.fit(train_features, train_target)
10. predictions = clf.predict(test_features)
11.
12. print(confusion_matrix(test_target, predictions))
13.
14. print(classification_report(test_target, predictions))
```

With the following code, we will make an evaluation of the data gathered so far and will try to see if it can meet our standards or not.

An output that the above code snippet can produce is something like this:

Confusion Matrix

```
[[ 437 579]
```

```
 [ 670 3225]]
```

Classification Report

	precision	recall	f1-score	support
0	0.39	0.43	0.41	1016

1 0.85 0.83 0.84 3895

Is it good? Can a simple Decision Tree be enough? Well, not exactly: from the classification report, it is easy to see how the dataset still is inclined towards the positive values. Putting it in percent, about 75% of **datasetFinal** is made of positive values, which still seems to be too much for the algorithm.

What would happen if we try to select random values from the **datasetFinal**, in order to make it smaller?

```
1. positives = data[data["billboarder"] == 1]
2. negatives = data[data["billboarder"] == 0]
3.
4. less_positives = positives.sample(len(negatives))
5.
6. dataset = pd.concat([less_positives, negatives])
7. dataset.to_csv('datasetRidotto.csv')
```

We let **pandas** do the job on choosing the elements. First, we create two variables that will hold the positive and negative part of the dataset; then, we tell pandas to choose a fixed length of the new dataset (depending on how many negative samples we have) and, finally, we save it as **datasetRidotto.csv**.

Let's see now what happens.

Confusion Matrix

[[740 270]

[292 759]]

Classification Report

precision recall f1-score support

0 0.72 0.73 0.72 1010

1 0.74 0.72 0.73 1051

This looks way better: with a dataset of approximately 8700 elements, we have a perfect and balanced situation, with a precision that is higher than 65% in both cases.

4.2 MULTILAYER NEURAL NETWORKS

What would happen, though, if we try to apply the same thoughts to a Multilayer neural network?

For predicting something in the future (because it may happen that songs don't go straight to the Billboard when they get released to the market!), Neural Networks are said to be the best option but let's see what happens in terms of precisions.

As before, we will first consider the case in which the algorithm takes in input the complete **datasetFINAL**.

```
1. data = pd.read_csv("datasetFINAL.csv", encoding="latin1", header=0)
2.
```

```

3. features = data[["danceability", "energy", "loudness", "speechiness", "acousticness", "instrumentalness", "liveness",
4.                 "key", "mode", "duration"]]
5. target = data["billboarder"]
6.
7. scaler = RobustScaler()
8.
9. train_features, test_features, train_target, test_target = train_test_split(features, target)
10.
11. scaler.fit(train_features)
12.
13. train_features = scaler.transform(train_features)
14. test_features = scaler.transform(test_features)
15.
16. clf = MLPClassifier(hidden_layer_sizes=(10, 100, 1000), verbose=True, max_iter=1000)
17. clf = clf.fit(train_features, train_target)
18.
19. predictions = clf.predict(test_features)
20.
21. print(confusion_matrix(test_target, predictions))
22.
23. print(classification_report(test_target, predictions))

```

As it can be seen, there's a new operation before the training of the network: that is, we scale our data to make our model converge in less time.

Let's see what happened this time: since it is a Neural Network, there will also be a section dedicated to the plot of the loss curve.

Confusion Matrix

```

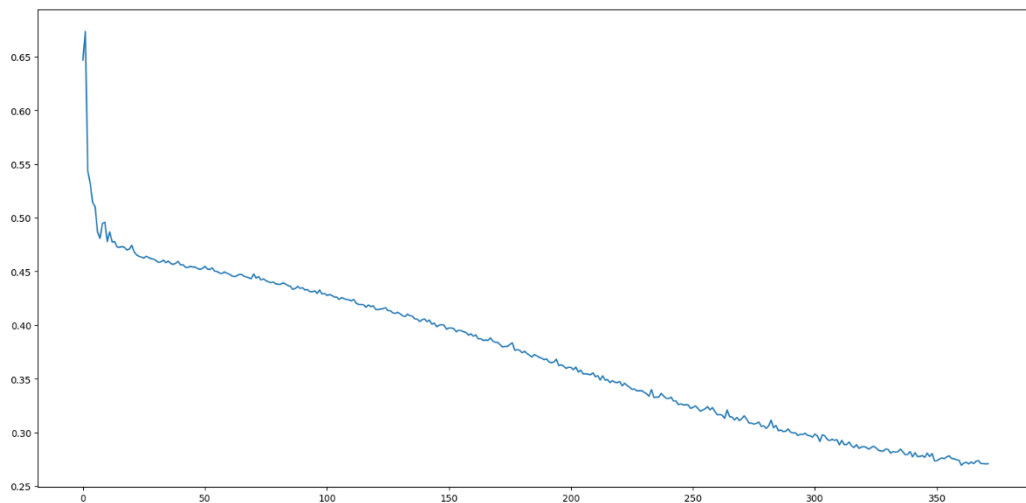
[[ 154 890]
 [ 454 3413]]

```

Classification Report

	precision	recall	f1-score	support
0	0.25	0.15	0.19	1044
1	0.79	0.88	0.84	3867

Loss Curve



What can we learn from this?

Our Neural Network surely did a good job, *by decreasing its loss at every epoch*, but it never reached the maximum iterations that we specified in its Constructor (1000), because at some point it stopped changing its loss value.

And what about the classification report? The report clearly shows that the Network hardly recognizes the false class, being *almost unbalanced* towards the *true class*.

Will the “shrunk” dataset change the result?

Confusion Matrix

[[568 460]

[474 559]]

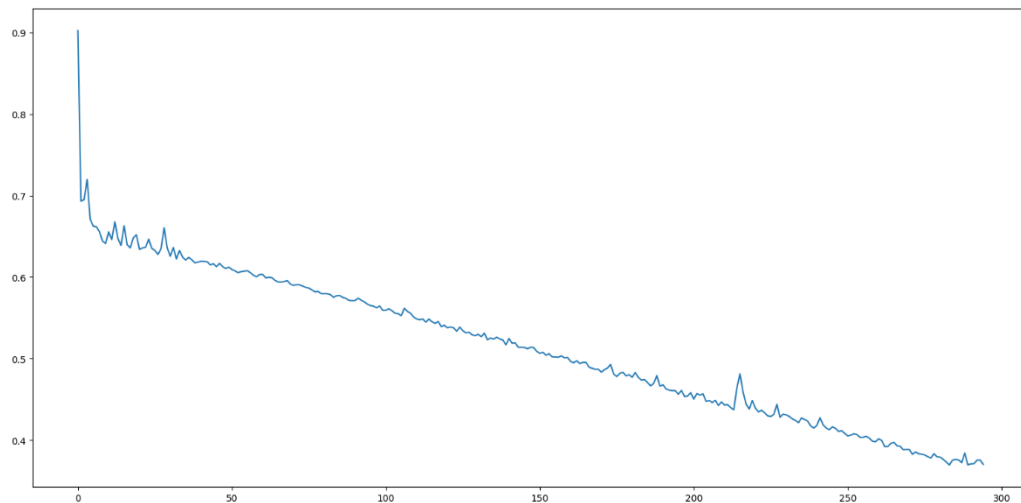
Classification Report

precision recall f1-score support

0 0.55 0.55 0.55 1028

1 0.55 0.54 0.54 1033

Loss Curve



Well, something *did* change: the neural network is now perfectly balanced to recognize both classes but their precision is still **lower** than what we obtained from the Decision Tree. Thus, we will not consider the Neural Network for this task.

4.3 RANDOM FOREST CLASSIFIER

Let's now see how an Ensemble method would perform with this task. We opted for using a **Random Forest Classifier**, since a single Decision Tree performed well.

```
1. data = pd.read_csv('datasetFINAL.csv', encoding='UTF-8')
2. features = data[["danceability", "energy", "loudness", "speechiness", "acousticness", "instrumentalness", "liveness",
3.                 "key", "mode", "duration"]]
4. target = data["billboarder"]
5.
6. train_features, test_features, train_target, test_target = train_test_split(features, target)
7.
8. clf = RandomForestClassifier()
9. clf = clf.fit(train_features, train_target)
10. predictions = clf.predict(test_features)
11.
12. print(confusion_matrix(test_target, predictions))
13.
14. print(classification_report(test_target, predictions))
```

Confusion Matrix

[[255 774]

[498 3384]]

Classification Report

	precision	recall	f1-score	support
0	0.34	0.25	0.29	1029
1	0.81	0.87	0.84	3882

As it is possible to read, the Random Forest trained with the complete dataset gets a huge unbalance towards the true class. Let's see how the situation changes with the "shrunk" version.

Confusion Matrix

```
[[625 416]
```

```
[464 556]]
```

Classification Report

	precision	recall	f1-score	support
0	0.57	0.60	0.59	1041
1	0.57	0.55	0.56	1020

This is *slightly* better, and it grants a more precise Random Forest. Though, the **Decision Tree remains the best option**.

4.4 SVM

And how would a Support Vector Machine perform? We are still making an operation that it can easily fit in the regression field, so it seems appropriate to use it.

```
1. data = pd.read_csv("datasetRidotto.csv", encoding="latin1", header=0)
2.
3. features = data[["danceability", "energy", "loudness", "speechiness", "acousticness", "ins
   trumentalness", "liveness",
4.                "key", "mode", "duration"]]
5. target = data["billboarder"]
6. clf = svm.SVC(gamma='scale')
7.
8. scaler = RobustScaler()
9.
10. train_features, test_features, train_target, test_target = train_test_split(features, targ
    et)
11.
12. scaler.fit(train_features)
13.
14. train_features = scaler.transform(train_features)
15. test_features = scaler.transform(test_features)
16.
```

```

17. clf = clf.fit(train_features, train_target)
18.
19. predictions = clf.predict(test_features)
20.
21. print(confusion_matrix(test_target, predictions))
22.
23. print(classification_report(test_target, predictions))

```

Confusion Matrix

[[460 577]

[238 786]]

Classification Report

precision recall f1-score support

0 0.66 0.44 0.53 1037

1 0.58 0.77 0.66 1024

This is just one run we're showing, but the SVM, if run several times, has higher precision values to the false class (up to 0.7), while the true class only reaches 0.6.

4.5 EVALUATION

For the reasons we reported above, it seems obvious that our choice falls on the **Decision Tree**, which was able to grant us a higher precision value on both classes.

But how did the Decision Tree perform in a 100-Fold evaluation?

```

1. fold = KFold(100, True)
2.
3. scores = []
4. for train_index, test_index in fold.split(data):
5.     train_set = data.iloc[train_index]
6.     test_set = data.iloc[test_index]
7.     train_features = train_set[["danceability", "energy", "loudness", "speechiness", "acousticness", "instrumentalness", "liveness", "key", "mode", "duration"]]
8.     test_features = test_set[["danceability", "energy", "loudness", "speechiness", "acousticness", "instrumentalness", "liveness", "key", "mode", "duration"]]
9.     train_target = train_set["billboarder"]
10.    test_target = test_set["billboarder"]
11.
12.    clf = tree.DecisionTreeClassifier()
13.    clf.fit(train_features, train_target)
14.    predictions = clf.predict(test_features)
15.    scores.append(accuracy_score(test_target, predictions))
16.
17. print("best: " + str(np.max(scores)), " average: " + str(np.mean(scores)) + " worst: " + str(np.min(scores)))

```

Here are the results, considering that we worked, as showed above, on the **datasetRidotto.csv** for better performances.

best: 0.8414634146341463 average: 0.6780517190714075 worst: 0.5421686746987951

What can these results tell us?

That our Decision Tree will, in the worst case, have a 50% possibility of making a good guess.
Thus, for us, the model seems to have been trained in a good way and satisfies our expectations.