Prediction of music genre

Data Mining (GEI)

Spring semester, course 2021/22

Pere Arnau Alegre
Andrés Jiménez González
Victor Teixidó López
Max Vives Ribera
You Wu

Pere Arnau Alegre
Andrés Jiménez González
Victor Teixidó López
Max Vives Ribera
You Wu

Index

1. Motivation

When we started to think of a topic we all liked and were interested in, we couldn't find a specific topic that appealed to all of us. That's why we decided to start scrolling through different data sets until we found one that seemed interesting enough to make a project about. After almost half a laboratory session, we ended up choosing the data set we are going to work with, prediction of music genre. The topic is interesting and everyone has a minimal knowledge of music so the subject seemed perfect to the group.

Moreover, choosing this topic has allowed us to learn more about music and how different qualities can define different genres quite well, which in the end is more knowledge about general music culture that we know. Unlike the first project we did, the learning curve in this one was small and not complicated at all. Despite everything we mentioned, although everyone knew at least a bit about music, there were concepts we didn't know a lot about so we had to do some research.So in the end, and after finishing the project, we can assure that we made the right choice with this topic and we would choose it again.

2. Data Source presentation

We got our data by searching in the kaggle website classification datasets with a theme that was interesting to all the members of our group and was big enough so that we could work properly with it. Once we found a dataset that fulfilled our needs we downloaded it and loaded it using and mounted it in our *Google Colab* so that we could work with it and make the necessary changes. Using *Google Colab* we reduced the size of the dataset to obtain a similar structure of data matrix as the one the assignment requires. The dataset chosen gathers information about different songs and all their characteristics and, of course, its music genre. In the next section, we will explain all different features of our data set. In addition, if we look up where the data has been collected, we can see that the author mentions that all the information it's from the Spotify database.

3. Metadata

First thing to mention are the different labels of music genres we have to classify all the songs in our data set. In total, there are 10 music genres which are the following: 'Electronic', 'Anime', 'Jazz', 'Alternative', 'Country', 'Rap', 'Blues', 'Rock', 'Classical' and 'Hip-Hop'. Originally the dataset had approximately 42,000 values and we had to reduce it to meet the conditions imposed by the guidelines. At the end, we had around 5,000 values and all the labels had, proportionally, the same amount of observations as the others.

Now, one by one, we are going to explain all the features we have in our data set, what they mean and the importance and relevance we think they can have when predicting a song genre. To mention that although in the guidelines it is mentioned that there must be at least 20 columns, the data set we chose has 16 but after talking to our laboratory teacher, she agreed that there was no problem at all.

During the metadata we are going to differentiate between 2 types of features: the common ones in the musical environment and the ones generated by the *Spotify API*.

3.1 Common used features

artist_name

This column shows who the author, singer or group of the song is, as the name says, the artist of the track. At first sight, having a data set of only 5,000 observations doesn't seem that this feature is going to be of great importance when predicting the genre of a track.

track_name

This column shows what the name of the song is and it happens the same as we saw in the last variable. At first sight, having a data set of only 5,000 observations, it doesn't seem that this feature is going to be of great importance when predicting the genre of a track.

popularity

This variable represents the popularity among *Spotify* users from each one of the observations we have. Important to mention again that all the information has been extracted from the *Spotify* database. This feature seems to have some more important relevance when deciding the genre of a song.

duration_s

This column is self-explanatory as it represents the duration of each one of the songs represented in our dataset. About the relevance of this feature it's more difficult as the majority of tracks nowadays have a duration of 3-4 minutes. Despite that, we did realize that for example, classical music has a long duration. So it could be that this feature has importance in some cases but not in general.

key

For this feature we had to do a bit of research because our musical knowledge was not that great. The key of a song is the note or chord the music is centered around, music genres have specific keys sometimes that they are played. Knowing that we can assume that the key of a song could be deterministic to decide which genre belongs to.

mode

This variable also required some seeking of information about it. In music, the mode describes any of several ways of ordering the notes of a scale according to the intervals they

form with the tonic, providing this way a theoretical framework for the melody. As far as we know, we didn't find any relevant information which indicates that different genres have different enough modes between them in order to distinguish them.

tempo

The tempo of a track defines the rate of speed indicated by one of a series of directions and often by an exact metronome marking. In other words, can be explained as the velocity of the rhythm that the song has. This seems like a variable that could distinguish different music genres apart, so it may be an important feature when classifying songs.

music_genre

This feature is in fact our classification feature, also known as the response variable. We have in total 10 different possible values for this column which are the following: 'Electronic', 'Anime', 'Jazz', 'Alternative', 'Country', 'Rap', 'Blues', 'Rock', 'Classical' and 'Hip-Hop'. As we said before, the number of observations of each label is approximately the same so the data is not unbalanced.

3.2 Spotify features

All these columns could in some way help us to distinguish between the different labels of the dataset to predict. Since the values are given by the *Spotify API* they all proceed from the same source and we can ensure that the values will be useful and well taken. Although, probably there will be variables more deterministic than other ones.

valence

The valence of a song describes the musical positiveness conveyed by a track. We could say that tracks with high valence sound more happy or cheerful when, on the other hand, tracks with low valence sound more sad or depressed. Although the positiveness of a song depende more on the artist rather than the genre, this feature could help, in some cases, to differentiate different genres.

acousticness

This value describes how acoustic a song is, where a score of 1.0 means the song is most likely to be an acoustic one and a value of 0 means that probably it's not.

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

This feature represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy

instrumentalness

This value represents the amount of vocals in the song. The closer it is to 1.0, the more instrumental the song is.

liveness

This value describes the probability that the song was recorded with a live audience. According to the official documentation, a value above 0.8 provides strong likelihood that the track is live. From our point of view, this variable doesn't seem as important as other ones when deciding which genre belongs to one song. The main reason is that any song could be recorded with a live audience.

loudness

This variable is also one introduced by the *Spotify API* that, as the name suggests, describes how loud a track is, independently obviously of the volume of the device. Values around 1 means that the song is quite loud and low values means that the track is not loud at all.

speechiness

This column represents the presence of spoken words in a track. Higher values around 1 means that the song is probably made of spoken words. On the other hand, a score between 0.33 and 0.66 is a song that may contain both music and words, and a score below 0.33 means the song does not have any speech. Again, from our point of view, this feature doesn't seem a deterministic one in general circumstances.

4. Preprocessing

Before starting with the machine learning algorithms, we started with the preprocessing of our data for the sake of eliminating errors, outliers, and missing values. Error values can

generate bad results when using a machine learning algorithm as they can create false biases. And missing values have to be treated as some models do not work with missings.

As a first data analysis, we developed a correlation plot with all our numeric variables so that we could understand the relationship between each other. We can see a strong positive correlation between variables loudness and energy and a strong negative correlation between loudness and acousticness, for example.

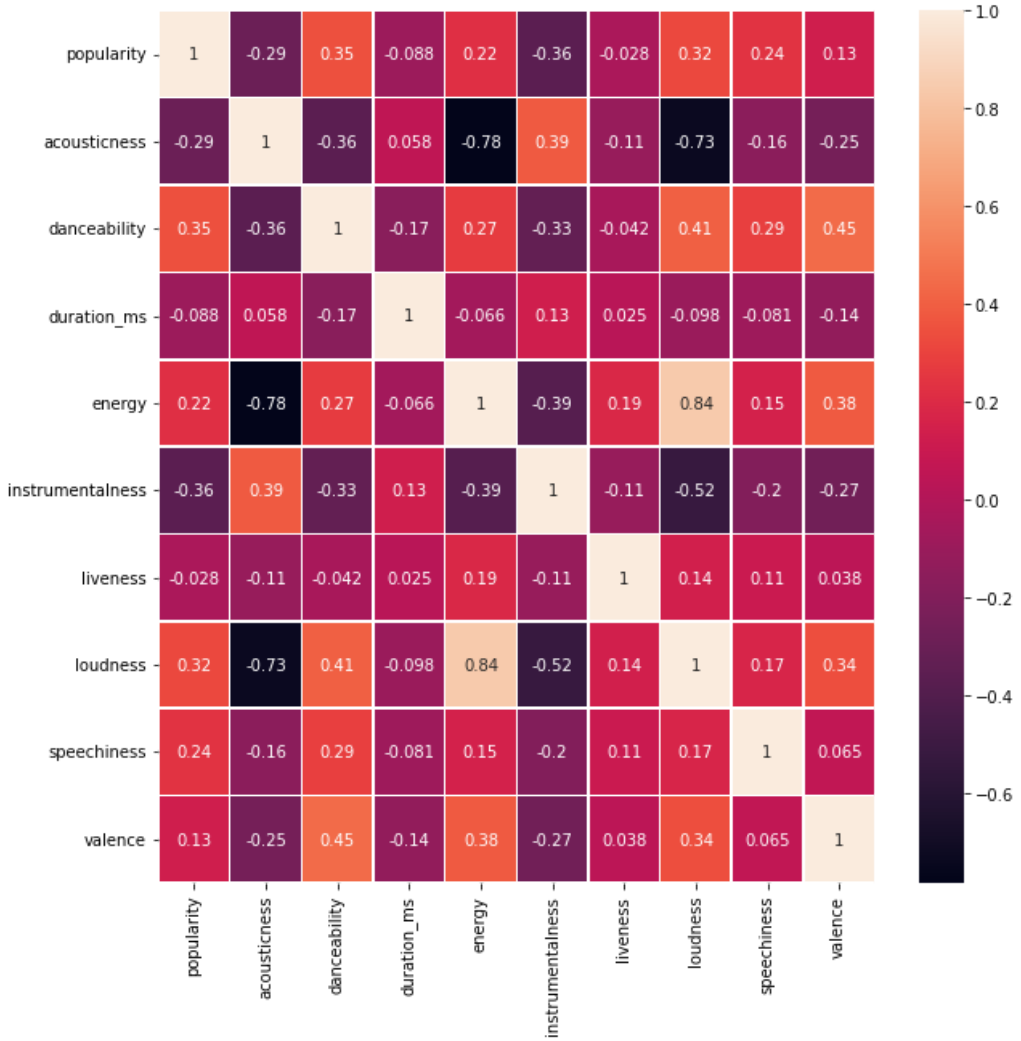


Figure 4.1: Correlation plot between all numeric variables

4.1. Data cleaning and variable analysis

For every variable in our dataset, we checked for severe outliers, errors and missings. To check for severe outliers, we drew a boxplot and used Tukey's threshold to gather the severe outliers and we replaced those values with a missing value. To check for errors, we just checked if the values for each variable were within the value range of the variable. We also performed a bit of profiling with our target variable.

Pere Arnau Alegre
Andrés Jiménez González
Victor Teixidó López
Max Vives Ribera
You Wu

Variable instance_id

We will delete this variable because it does not provide any significant data to help us with our predictions.

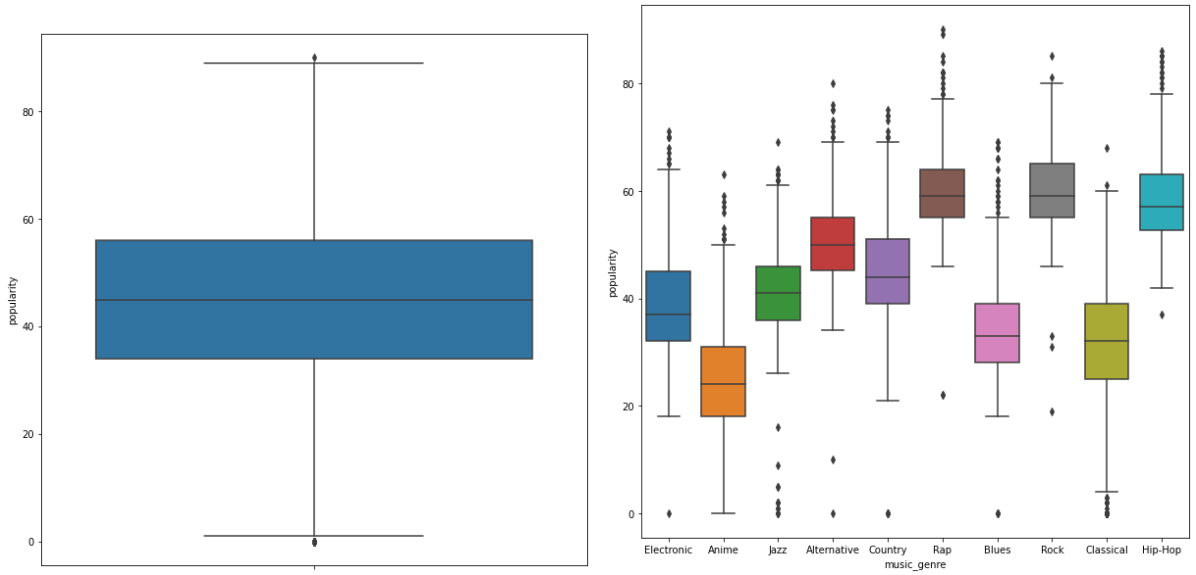
Variable artist_name

We do nothing for this variable.

Variable track_name

We do nothing for this variable

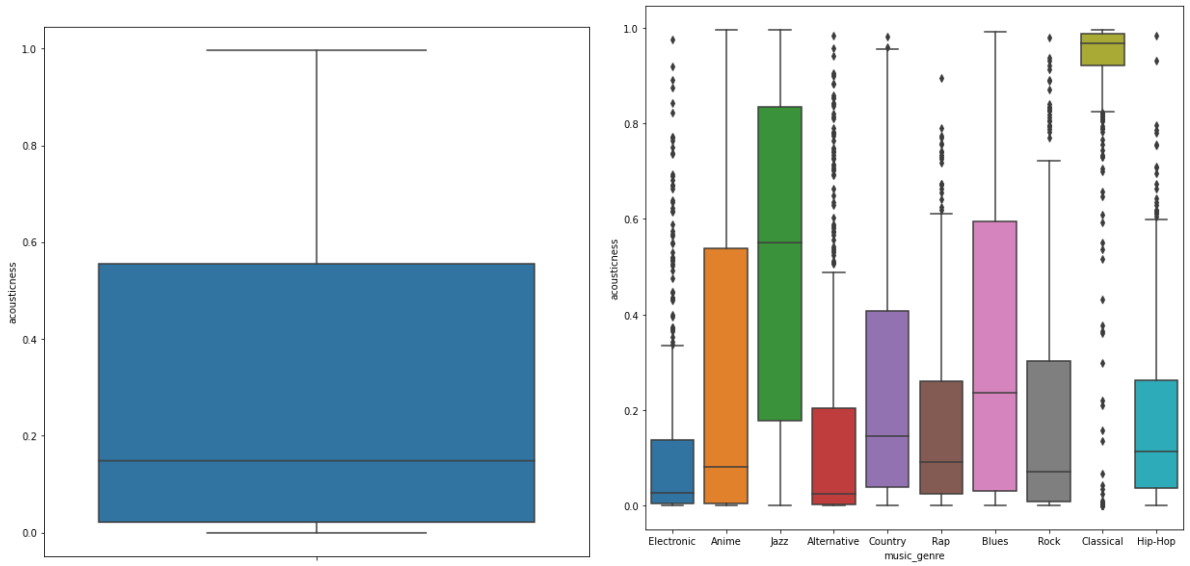
Variable popularity



Figures 4.2 & 4.3: Boxplot of popularity & boxplots of popularity for every music_genre.

From the boxplots we can see that the variable does not present any severe outliers. We can also see that popularity is very correlated with the genre of the music.

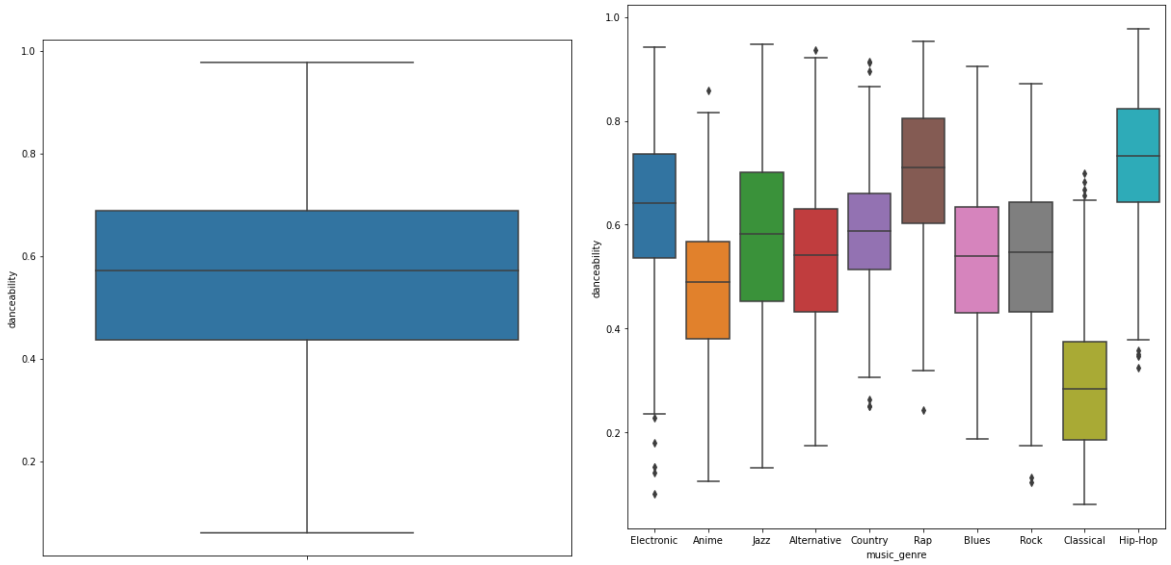
Variable acousticness



Figures 4.4 & 4.5: Boxplot of acousticness & boxplots of acousticness for every music_genre.

From the boxplots we can see that the variable does not present any severe outliers. We can also see that acousticness is lightly correlated with the genre of the music.

Variable danceability



Figures 4.6 & 4.7: Boxplot of danceability & boxplots of danceability for every music_genre.

From the boxplots we can see that the variable does not present any severe outliers. We can also see that danceability is lightly correlated with the genre of the music. As expected, classical music presents very low danceability.

Variable duration_ms

Pere Arnau Alegre
Andrés Jiménez González
Victor Teixidó López
Max Vives Ribera
You Wu

In this variable we encountered severe outliers and also negative values for the duration. We identified the values with severe outliers and also with errors and we replaced them with NA's so that we can impute them later.

```
def find_outliers_tukey(x):
    q1 = np.nanpercentile(x, 25)
    q3 = np.nanpercentile(x, 75)
    iqr = q3-q1
    floor = q1 - 3*iqr
    ceiling = q3 + 3*iqr
    outlier_indices = list(x.index[(x < floor)|(x > ceiling)])
    outlier_values = list(x[outlier_indices])

    return outlier_indices, outlier_values
```

Figure 4.8: Tukey function to identify severe outliers

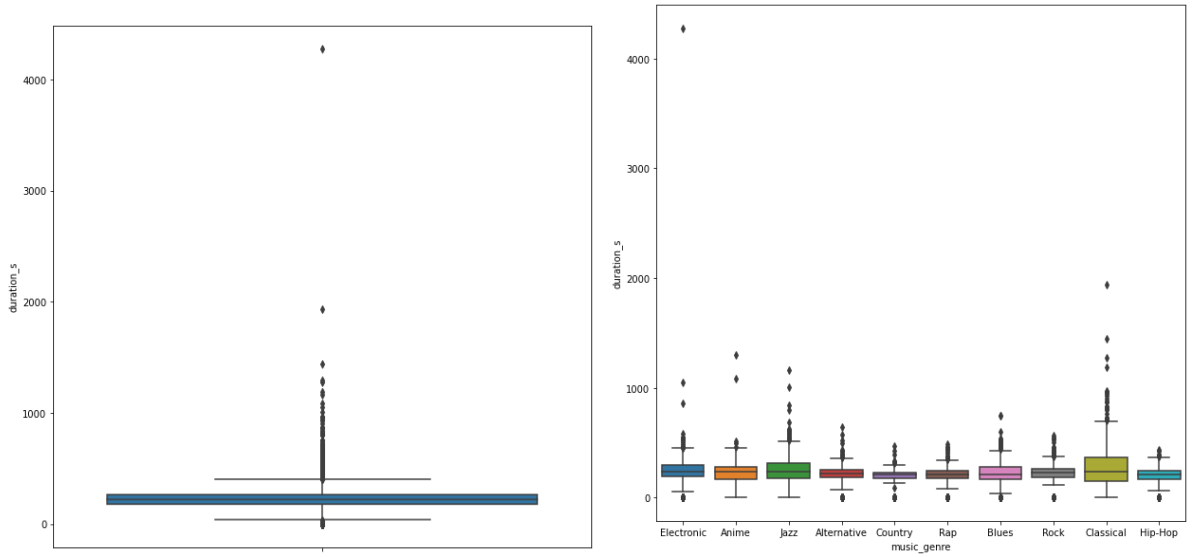
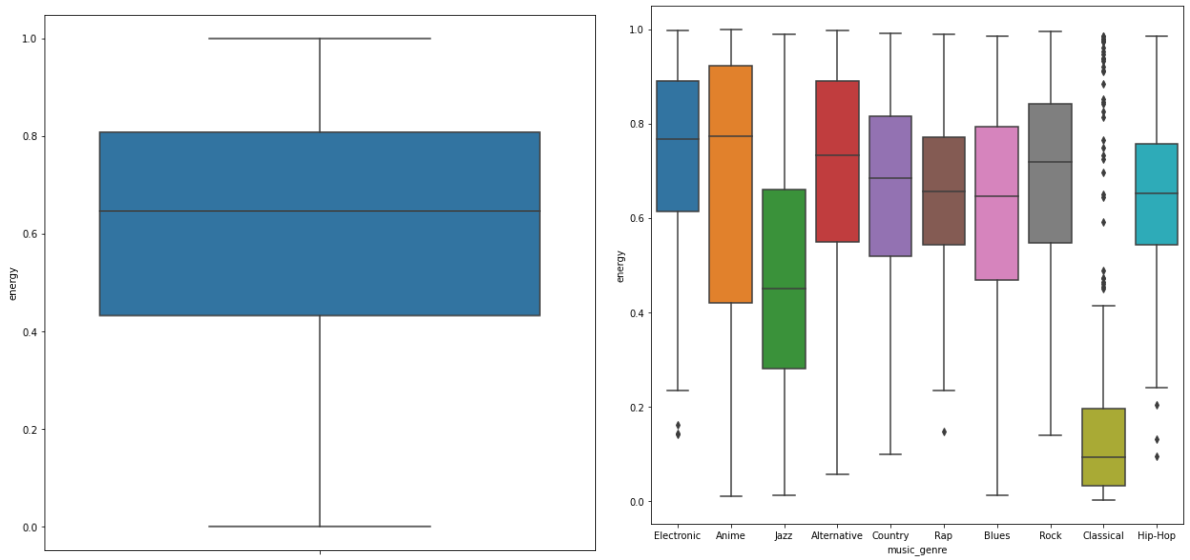


Figure 4.9 & 4.10: Boxplot of duration_s & boxplots of duration_s for every music_genre previous from deletion of severe outliers and errors

We can see that the variable of duration is not very correlated with genres of music as the mean for every genre is more or less the same but the quantiles are quite different.

Variable energy

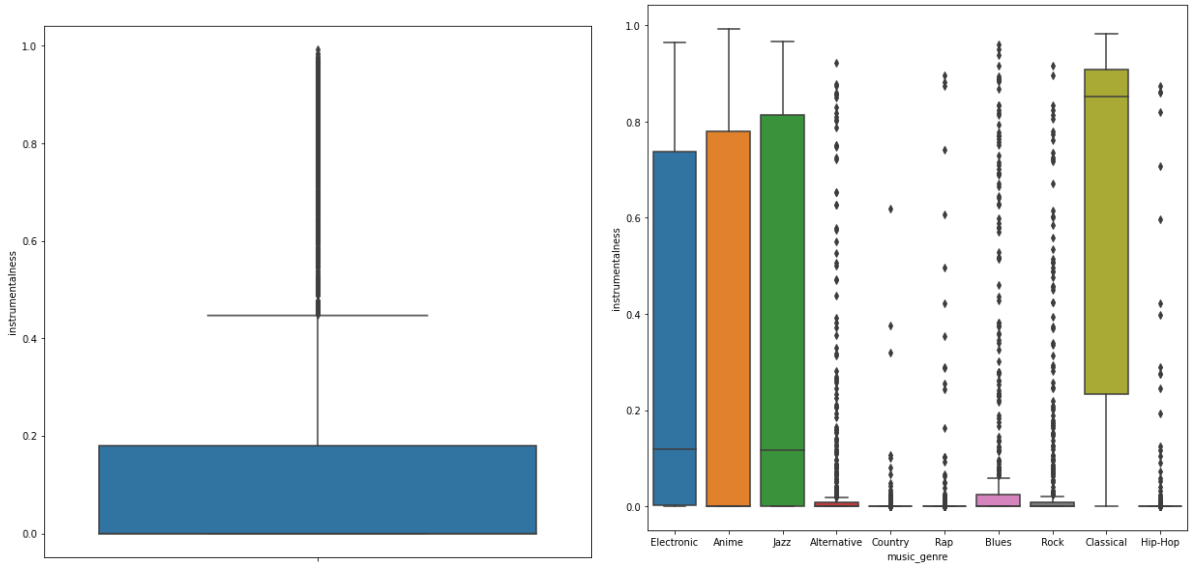


Figures 4.11 & 4.12: Boxplot of energy & boxplots of energy for every music_genre.

From the boxplots we can see that the variable does not present any severe outliers. We can also see that energy is lightly correlated with the genre of the music. As expected, classical music presents very low energy alongside Jazz.

Variable instrumentalness

In this variable we encountered severe outliers and we treated them like before.



Figures 4.13 & 4.14: Boxplot of instrumentalness & boxplots of instrumentalness for every music_genre.

Variable key

Pere Arnau Alegre
Andrés Jiménez González
Victor Teixidó López
Max Vives Ribera
You Wu

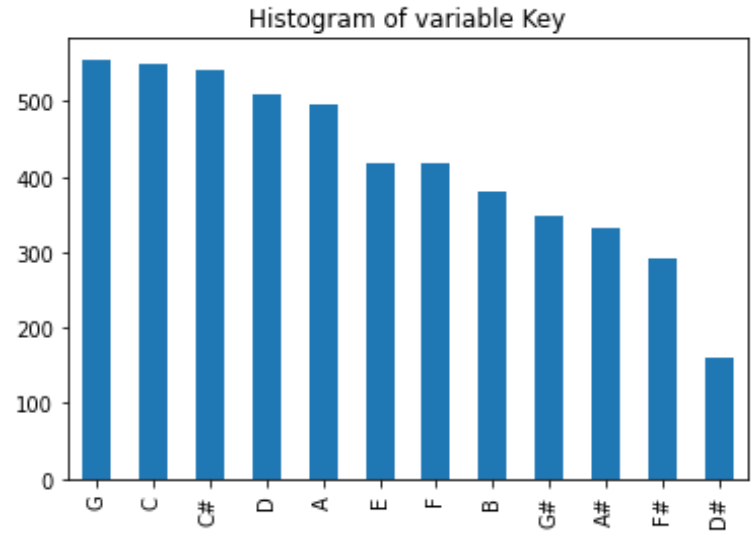


Figure 4.15: Histogram of variable key.

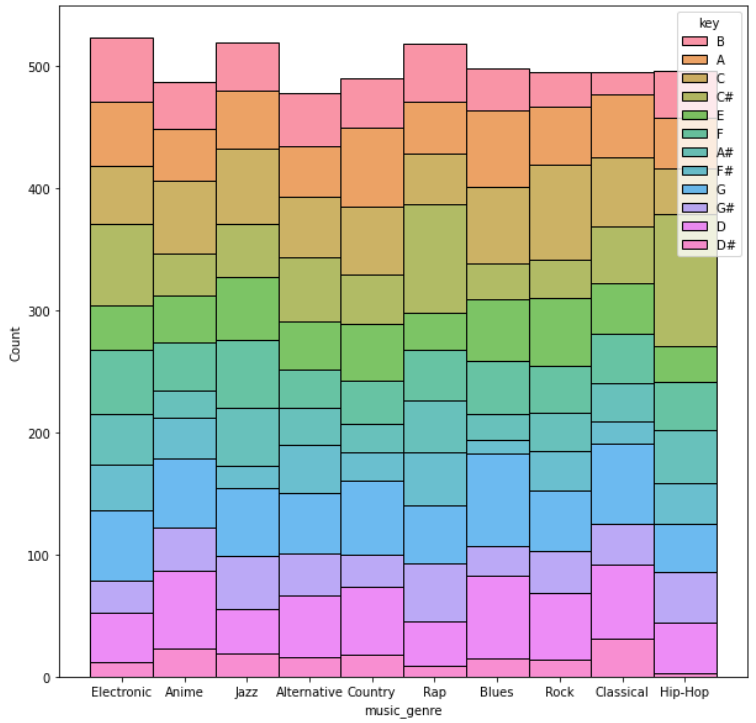
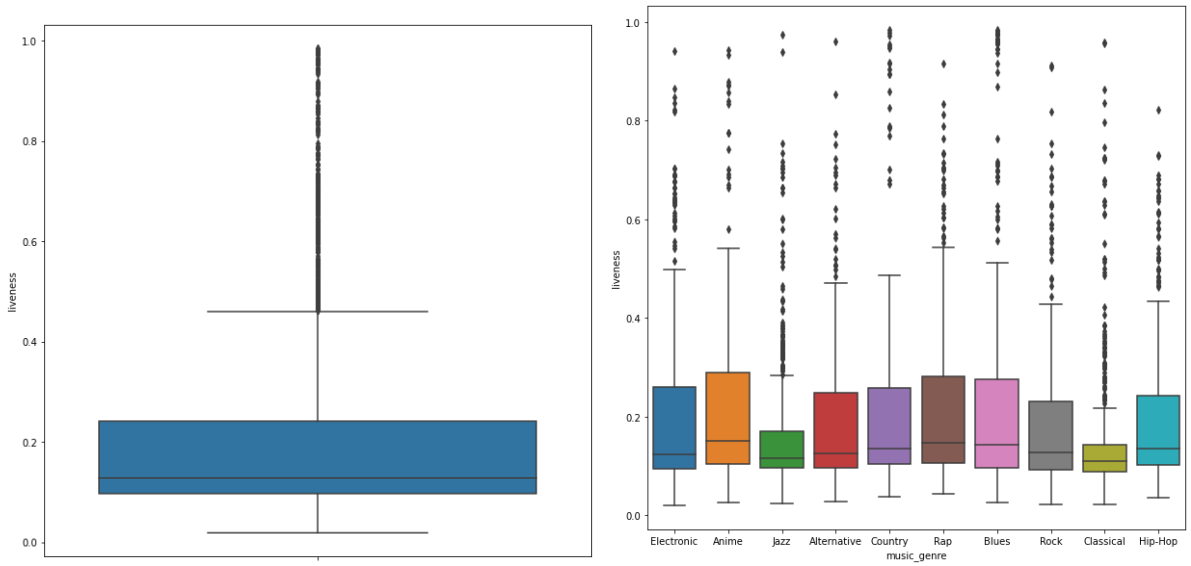


Figure 4.16: Added variable plot of key vs music_genre.

If we developed a contingency table and tested for correlation between key and music_genre with the Chi squared test, we get a p-value of 7.72e-30, so there exists correlation between music_genre and key.

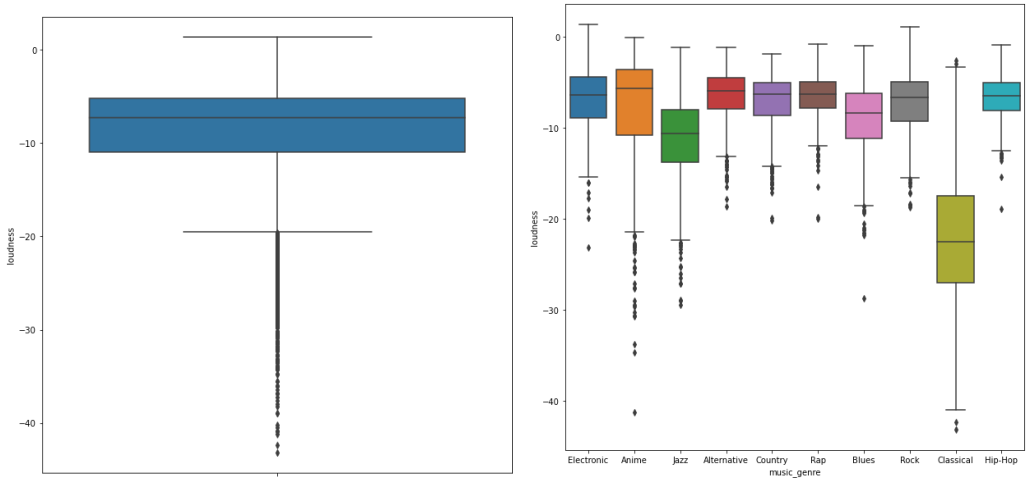
Variable liveness



Figures 4.17 & 4.18: Boxplot of liveness & boxplots of liveness for every music_genre.

From the boxplots we can see that the variable presents severe outliers so we will treat them. The variable does not present any errors as all the values are between 0 and 1. We can also see that liveness is lightly correlated with the genre of the music. As expected, classical music and jazz present lower liveness than other genres.

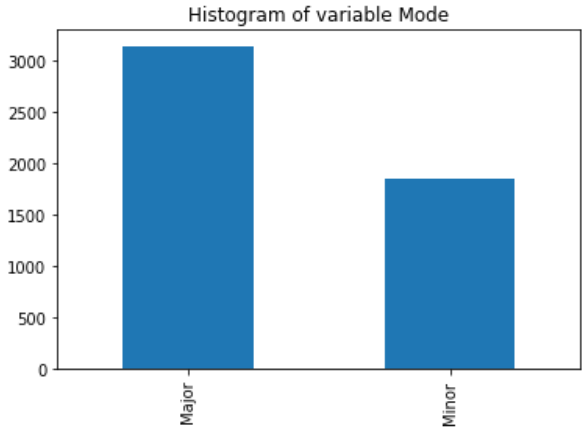
Variable loudness

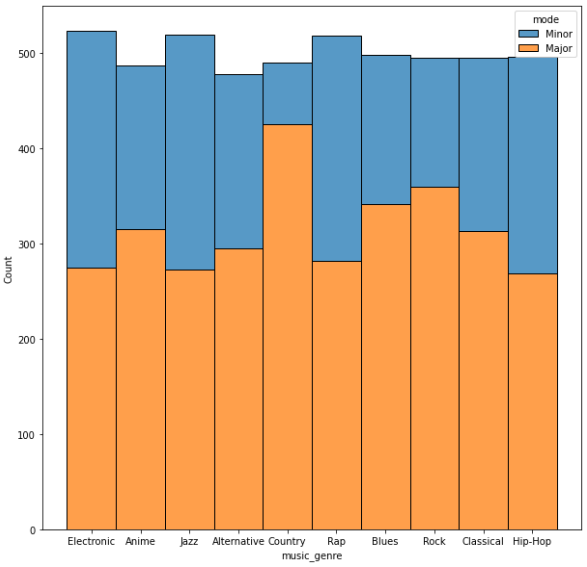


Figures 4.19 & 4.20: Boxplot of loudness & boxplots of loudness for every music_genre.

From the boxplots we can see that the variable presents severe outliers so we will treat them. We can also see that liveness is very correlated with the genre of the music. As expected, classical music presents significant lower loudness than other genres.

Variable mode

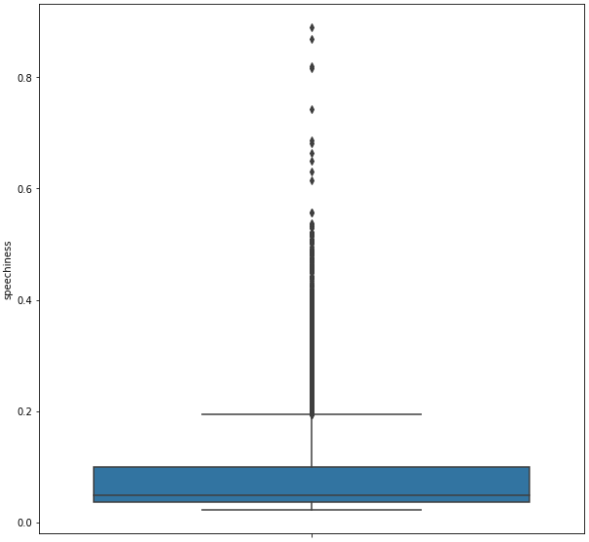


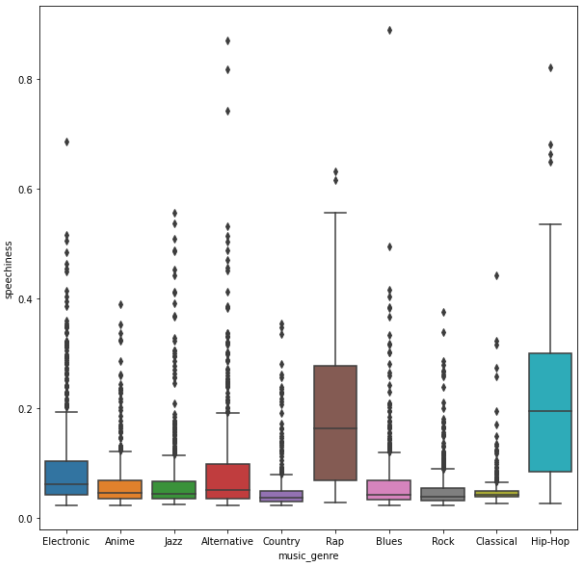


Figures 4.21 & 4.22: Histogram of variable mode and an added variable plot of variable mode vs variable music_genre

With a Chi squared test from mode vs music_genre we get a p-value of 7.58e-44 so there exists a correlation between the 2 variables.

Variable speechiness

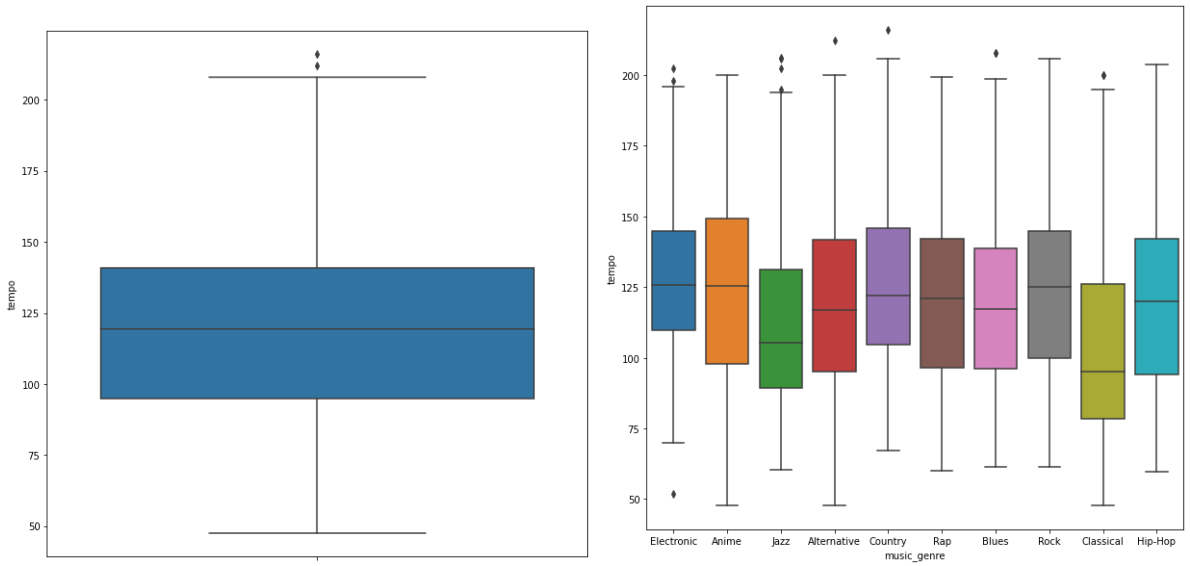




Figures 4.23 & 4.24: Boxplot of speechiness & boxplots of speechiness for every music_genre.

From the boxplots we can see that the variable presents severe outliers so we will treat them. The variable does not present any errors as all the values are between 0 and 1. We can also see that speechiness is very correlated with the genre of the music. As expected, rap and hip-hop present remarkably higher speechiness than other genres.

Variable tempo



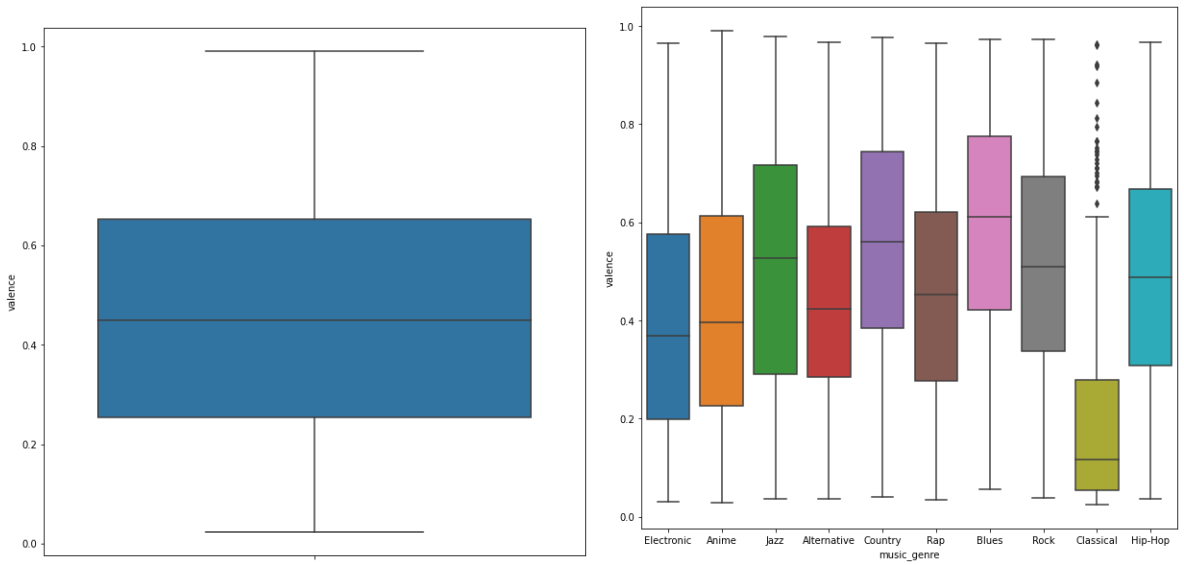
Figures 4.25 & 4.26: Boxplot of tempo & boxplots of tempo for every music_genre.

From the boxplots we can see that the variable present does not present severe outliers. We can also see that tempo is lightly correlated with the genre of the music.

Variable obtained date

We will delete this variable because it does not provide any significant data to help us with our predictions.

Variable valence



Figures 4.27 & 4.28: Boxplot of valence & boxplots of valence for every music_genre.

From the boxplots we can see that the variable does not present severe outliers. The variable does not present any errors as all the values are between 0 and 1. We can also see

that valence is correlated with the genre of the music. As expected, classical music presents remarkably lower valence as it is a more melancholic genre of music and blues presents more valence as it is a more euphoric genre.

Variable music genre

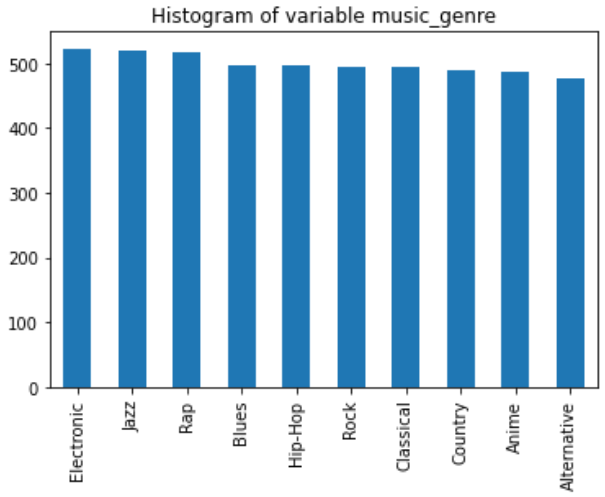


Figure 4.29: Histogram of variable music_genre.

This is our target variable. We can see that we have a very balanced dataset, so that will aid our machine learning models.

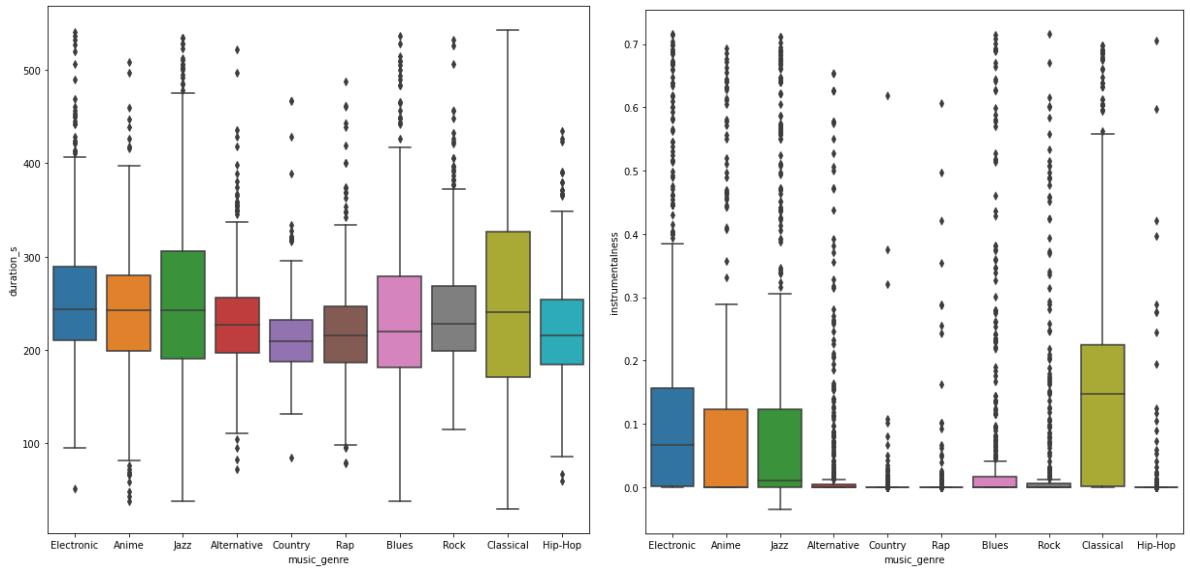
4.2. Imputation of missing data

For some variables we deleted severe outliers and errors in an effort of trying to achieve a more balanced dataset. In order to maintain all the observations and not lose information, we will use an iterative imputer to refill all the missing values.

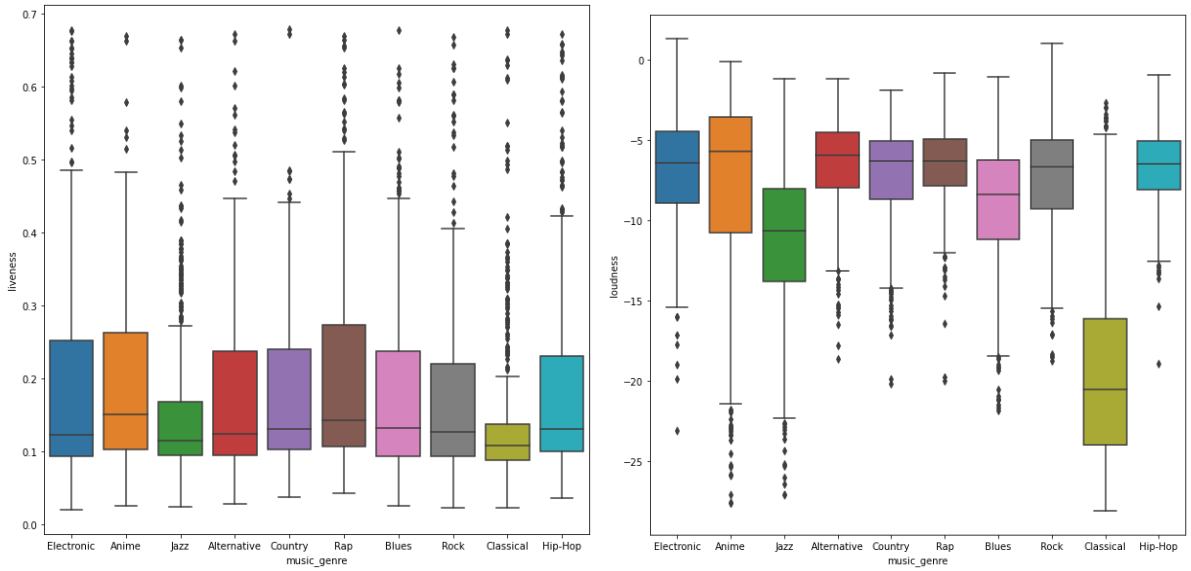
```
data_num = data.select_dtypes(include=np.number)
imp = IterativeImputer(max_iter=10, random_state=0)
imputed_vals = pd.DataFrame(imp.fit_transform(data_num), columns=data_num.columns)
data = data.combine_first(imputed_vals)
```

Figure 4.30:

We will now check the quality of imputation of the algorithm by redrawing the boxplots for the variables that we imputed.



Figures 4.31 & 4.31: Boxplots of duration_s and instrumentalness for every music_genre after imputation.



Figures 4.33 & 4.34: Boxplots of liveness and loudness for every music_genre after imputation.

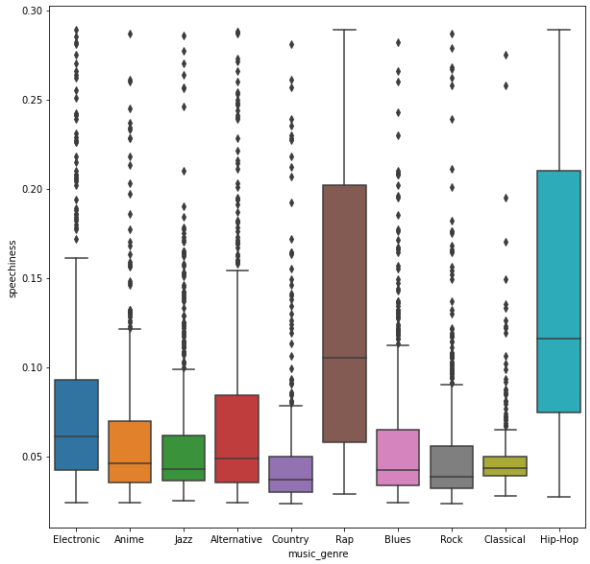


Figure 4.35: Boxplots of speechiness for every music_genre after imputation.

From the boxplots we can see that all variables follow the same pattern as before the deletion of outliers and imputation, so we can conclude that the imputation was successful.

4.3. Multivariate outliers

As a final preprocessing cleaning, we searched the dataset for multivariate outliers using the Mahalanobis distance and we created a variable in order to classify these outliers. In some later algorithms we will try to exclude from the computed dataset these outliers because they produce bad results as they do not tally with the rest of observations.

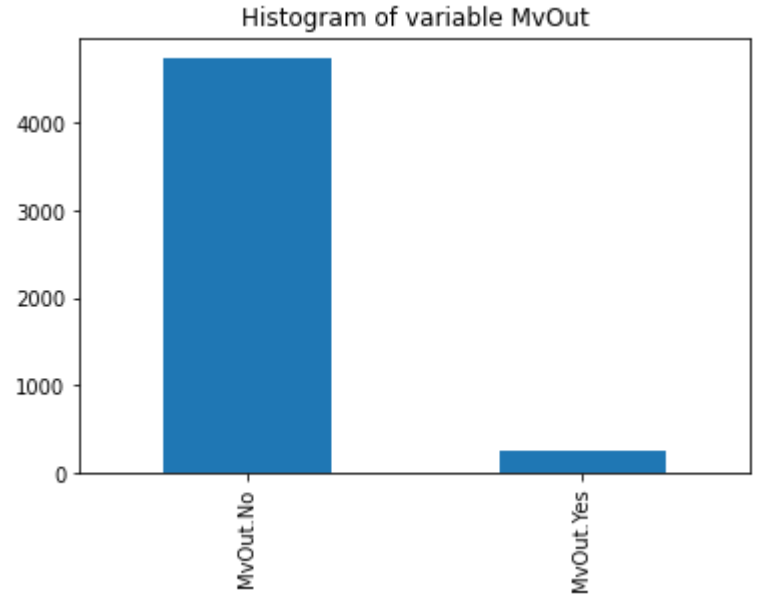


Figure 4.36: Histogram of variable MvOut.

With the purpose of visualizing these outliers, we computed a PCA to reduce the dimensionality of the dataset and be able to visualize the dataset in 2 dimensions. Then we

plotted the observations in these 2 dimensions and colored as red the multivariate outliers. As expected, we can see that most of the multivariate outliers lie around the periphery of the central cluster of observations.

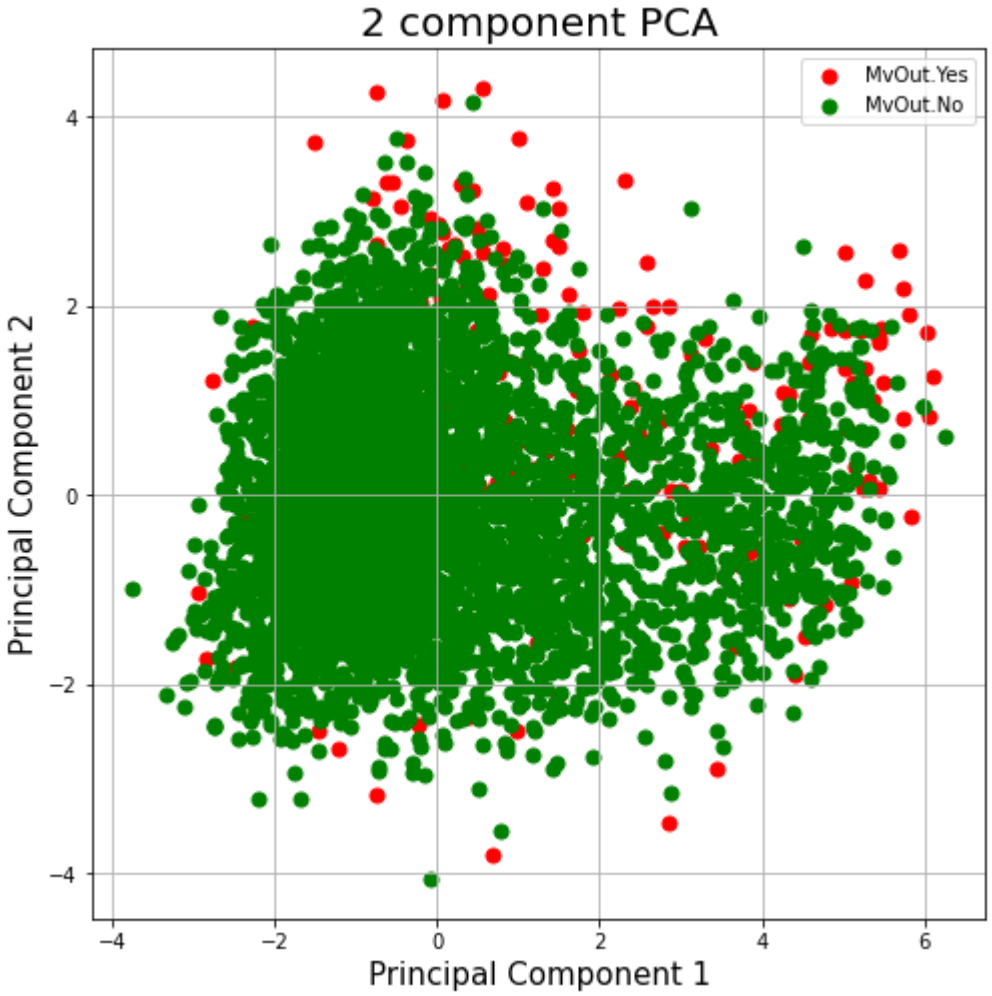


Figure 4.37: PCA with the representation of multivariate outliers vs normal observations.

5. Machine learning methods

5.1 Naïve-Bayes

5.1.1. Conclusions

Naive Bayes is a probabilistic classifier based on the theorem of Bayes and some simplifying assumptions. These assumptions can be simplified in assuming that the explanatory variables are independent, because that is called “Naive”.

Bayes’ theorem:

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}$$

Figure 5.1.1: Formula of Bayes' theorem.

The Bayes theorem tells us that the probability of A being B is equal to the probability B being A by the probability of and divided by the probability of B.

The theorem can be used to obtain the probability of a label y given the values of some explanatory variables X.

$$P(y|x1, x2, x3..xN) = \frac{P(x1|y).P(x2|y).P(x3|y)...P(xN|y).P(y)}{P(x1).P(x2).P(x3)...P(xN)}$$

Figure 5.1.2: Formula of Bayes' theorem applied to predict a label given explanatory variables.

There are different versions of the algorithm Naive Bayes and the first thing that we will do is to consider which one fits our data better.

Bernoulli Naive Bayes assumes that all our features are binary, this is not our case so we will not use this method.

Multinomial Naive Bayes is used when we have discrete data, but it isn't our case.

Bernoulli Naive Bayes are also suitable for classification with discrete data. The difference is that while MultinomialNB works with occurrence counts, BernoulliNB is designed for binary/boolean features.

The categorical naive bayes as the name indicates is also for categorical features.

As our data is continuous we will be using the Gaussian Naive Bayes, because of the assumption of the normal distribution.

5.1.2. Data with outliers

First of all we will divide our dataset into the explanatory variables (X) and the response variable (y). As in our dataset we have some variables that are not continuous, we will use the get_dummies method that converts categorical variables into dummy/indicator variables.

```
# Separation into data and label
X = data.drop('music_genre', axis=1)
X = pd.get_dummies(X)
y = data['music_genre']
```

Figure 5.1.3: Commands to separate data into label and explanatory variables.

After that we will normalize the explanatory data to be able to apply the Gaussian Naive Bayes correctly.

```
# Normalization:
X = preprocessing.normalize(X)
```

Figure 5.1.4: Command to normalize explanatory variables.

It's important to divide the data in a training dataset and a testing dataset. The first one will be used to train our model and the second one to test how well it performs. This is important because the model should be tested with data that has not been used to build it.

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42)
```

Figure 5.1.5: Command to divide data into training data and testing data.

Cross validation is a way that we can use our training data in order to estimate how well our model will perform on data we haven't seen before.

The cross_val_score method divides the dataset into cv parts and uses one part to test the model and the other ones to train it. We use StratifiedKFold to indicate how many divisions the cross_val_score will use, because the folds are made by preserving the percentage of samples for each class.

```
cv = StratifiedKFold(n_splits=10)
gnb = GaussianNB()
cv_scores = cross_val_score(gnb,X=x_train,y=y_train,cv=cv)
np.mean(cv_scores)
```

Figure 5.1.6: Commands get the mean of cross validation scores.

The mean of the accuracy of the 10 different models generated by the cross_val_score is 0,49586.

Finally, we use cross_val_predict to get a prediction of the response variable and then diagnose the goodness of our model.

```
predicted = cross_val_predict(gnb, X=x_test, y=y_test,  cv=cv)
```

Figure 5.1.7: Command to get the predicted values of music_genre using a cross validation predict.

We obtain the next confusion matrix:

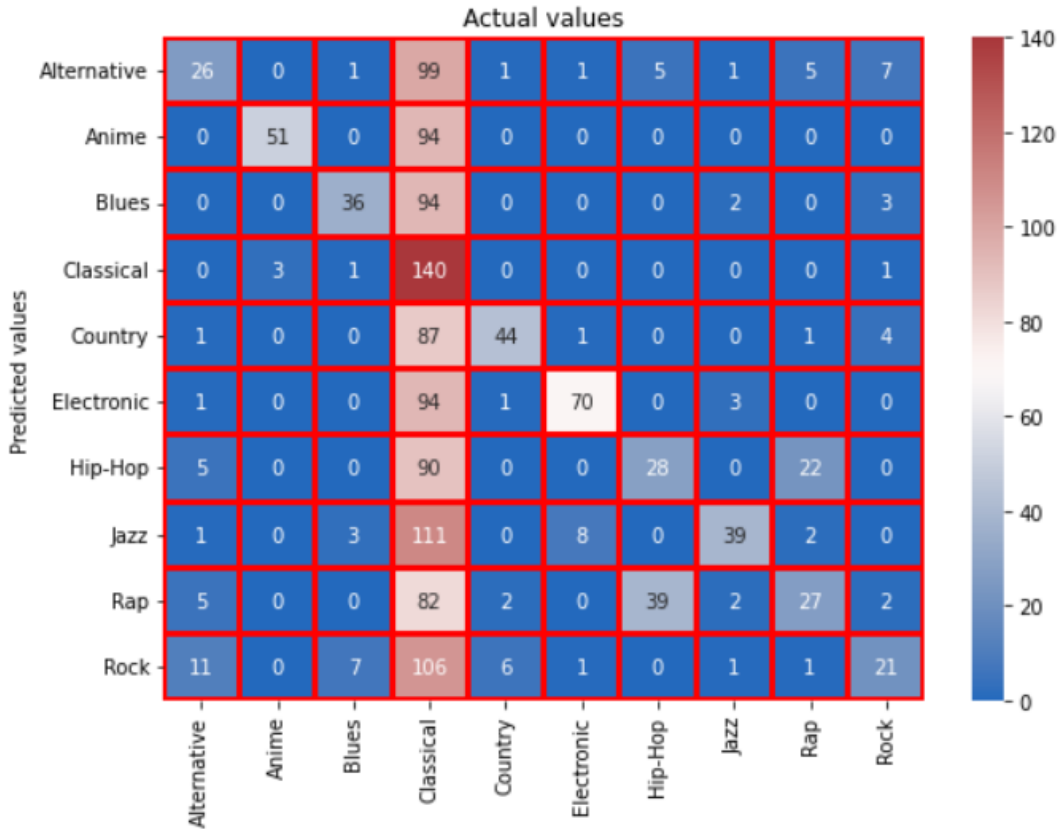


Figure 5.1.8: Confusion matrix obtained from the predicted values of the cross_val_predict.

As we can see a lot of songs are predicted to be classical music when they are not, but in the other hand the songs that are actually classical are very well predicted.

This can be confirmed since the accuracy of the model is only 0,321.

	precision	recall	f1-score	support
Alternative	0.52	0.18	0.27	146
Anime	0.94	0.35	0.51	145
Blues	0.75	0.27	0.39	135
Classical	0.14	0.97	0.25	145
Country	0.81	0.32	0.46	138
Electronic	0.86	0.41	0.56	169
Hip-Hop	0.39	0.19	0.26	145
Jazz	0.81	0.24	0.37	164
Rap	0.47	0.17	0.25	159
Rock	0.55	0.14	0.22	154
accuracy			0.32	1500
macro avg	0.63	0.32	0.35	1500
weighted avg	0.63	0.32	0.35	1500

Figure 5.1.9: Classification report obtained from the predicted values of the cross_val_predict.

Moreover, we can get this classification report. The precision values are high but the recall ones are very low.
Precision is defined as the ratio of true positives to the sum of true and false positives, so if it has high values indicate that the values predicted to be x are usually well predicted.
Recall is defined as the ratio of true positives to the sum of true positives and false negatives, so if it has low values indicate that we have a lot of false negatives predictions.

We have also tested the model using all our data, without using the cross_val_predict, and we have getted better results.

```
clf = GaussianNB()
clf.fit(x_train, y_train)
pred=clf.predict(x_test)
```

Figure 5.1.10: Commands to get the predicted values of music_genre using all the training data.

We get an accuracy of 0.5 and the next classification report:

	precision	recall	f1-score	support
Alternative	0.59	0.34	0.43	146
Anime	0.71	0.74	0.73	145
Blues	0.76	0.52	0.62	135
Classical	0.22	0.97	0.36	145
Country	0.82	0.56	0.66	138
Electronic	0.90	0.64	0.75	169
Hip-Hop	0.54	0.23	0.33	145
Jazz	0.95	0.37	0.53	164
Rap	0.52	0.44	0.48	159
Rock	0.66	0.21	0.32	154
accuracy			0.50	1500
macro avg	0.67	0.50	0.52	1500
weighted avg	0.67	0.50	0.52	1500

Figure 5.1.11: Classification report obtained from the predicted values using all the training data with outliers.

The F1 is the weighted harmonic mean of precision and recall. The closer the value of the F1 score is to 1.0, the better the expected performance of the model is, and in this case we get higher values.

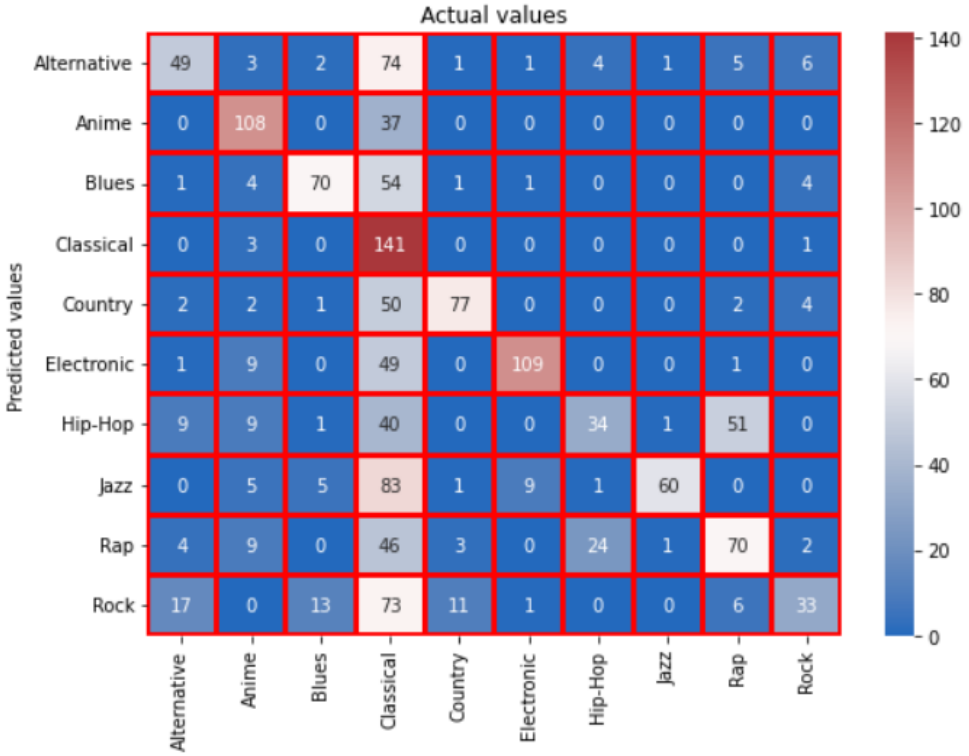


Figure 5.1.12: Confusion matrix obtained from the predicted values using all the training data with outliers.

That improvement is clearly seen in the confusion matrix, but the predictions of classical music are still harming our model considerably.

5.1.3. Data without outliers

Now we will use the data without outliers to diagnose the last model (using all the data to train) which is the one giving us the better results and see if it improves.

	precision	recall	f1-score	support
Alternative	0.45	0.37	0.41	146
Anime	0.68	0.82	0.75	145
Blues	0.61	0.50	0.55	135
Classical	0.29	0.99	0.45	145
Country	0.84	0.55	0.67	138
Electronic	0.97	0.66	0.79	169
Hip-Hop	0.70	0.76	0.73	145
Jazz	0.85	0.44	0.58	164
Rap	0.61	0.35	0.45	159
Rock	0.71	0.30	0.42	154
accuracy			0.57	1500
macro avg	0.67	0.57	0.58	1500
weighted avg	0.68	0.57	0.58	1500

Figure 5.1.13: Classification report obtained from the predicted values using all the training data without outliers.

The accuracy has and the recall have improved from 0.5 to 0.57. The precision has remained the same and the f1-score has improved from 0.52 to 0.58.

We can assure that the presence of outliers makes the model worse.

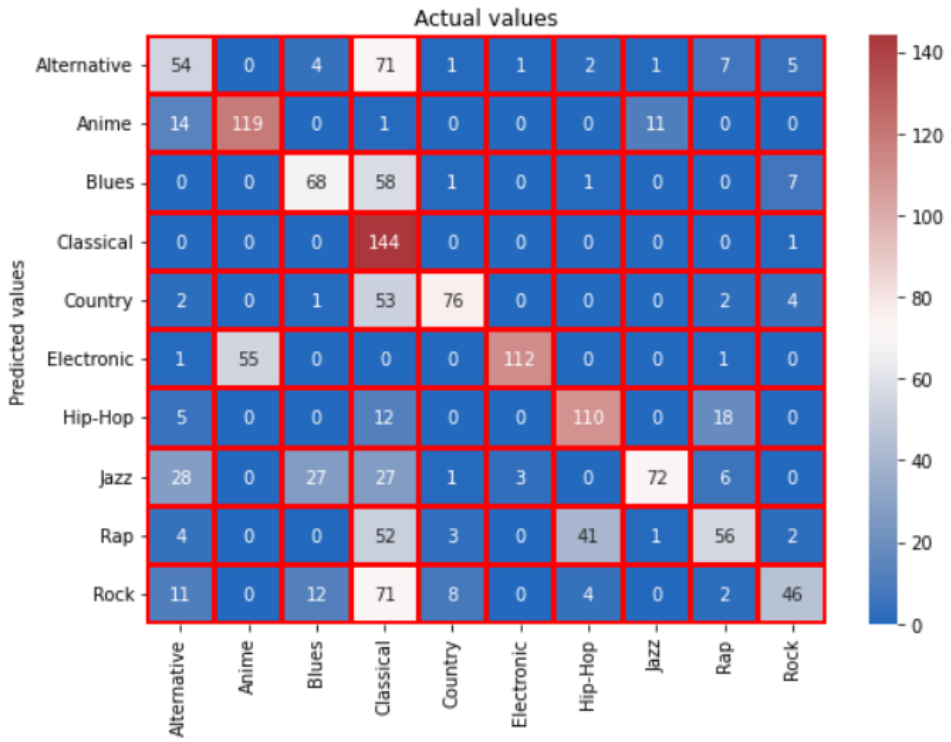


Figure 5.1.14: Confusion matrix obtained from the predicted values using all the training data without outliers.

Without outliers the predictions are less influenced by Classical Music, so maybe the outliers were classical songs.

5.1.4. Conclusions

In conclusion, Naive Bayes is a very simple model that has a very low time of execution. Despite assuming a false hypothesis, that the explanatory variables are independent, it gets very decent results in the prediction. It has to be said that this model is very influenced by the outliers.

5.2 K-NN

5.2.1. Introduction

KNN stands for K Nearest Neighbor and is one of the fundamental machine learning algorithms. Machine learning models anticipate output values based on a set of input values. KNN is a type of machine learning algorithm that is primarily used for classification. It classifies the data point based on the classification of its neighbors.

The advantages of KNN are:

- No Training Period: Because the data itself is a model that will be the reference for future prediction, KNN modeling does not require a training period, and as a result it is very time efficient.This makes the KNN algorithm much faster than other training-based algorithms (SVM, Linear Regression etc.).
- Because the KNN algorithm does not require any training before making predictions, new data can be added without affecting the algorithm's accuracy.
- Easy Implementation. There are only two parameters required to implement KNN: the value of n_neighbors(K) and the distance function (Euclidean,Manhattan,Chebyshev,Minkowski).

The disadvantages of KNN are:

- Does not work well with large datasets: The cost of calculating the distance between each data instance in large datasets is enormous, which lowers the algorithm's efficiency.
- Does not work well with high dimensions: The KNN algorithm does not perform well with high-dimensional data because calculating the distance in each dimension becomes challenging for the algorithm with a large number of dimensions.
- Feature scaling: Before applying the KNN algorithm to any dataset, feature scaling (standardization and normalization) is required. If we don't, KNN may make incorrect predictions.
- Sensitive to noisy data, missing values and outliers: The KNN algorithm is sensitive to dataset noise. Missing values must be manually imputed, and outliers must be removed.

5.2.2. KNN with multivariate outliers

For our first attempt at using the KNN we'll use a version of our dataset that hasn't had any severe or multivariate outliers removed. Because the knn method can only use numerical variables, we'll remove all categorical variables. After that, we'll divide the dataset into training and testing data, do feature scaling and use the GridSearchCV function to identify the algorithm's optimized parameters. The prediction will be computed after we obtain the best parameters.

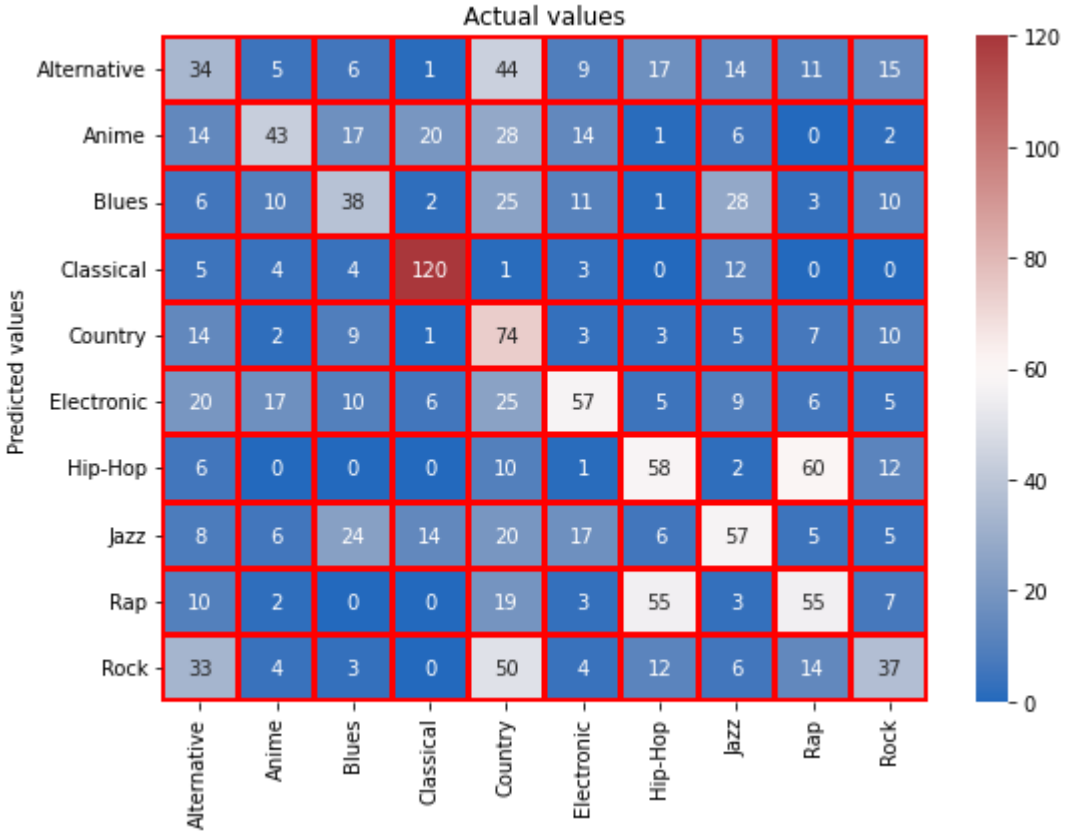


Figure 5.2.1: Confusion matrix of the model with outliers.

	precision	recall	f1-score	support
Alternative	0.23	0.22	0.22	156
Anime	0.46	0.30	0.36	145
Blues	0.34	0.28	0.31	134
Classical	0.73	0.81	0.77	149
Country	0.25	0.58	0.35	128
Electronic	0.47	0.36	0.40	160
Hip-Hop	0.37	0.39	0.38	149
Jazz	0.40	0.35	0.38	162
Rap	0.34	0.36	0.35	154
Rock	0.36	0.23	0.28	163
accuracy			0.38	1500
macro avg	0.39	0.39	0.38	1500
weighted avg	0.40	0.38	0.38	1500

Interval of confidence: (0.357639282378951, 0.40698172923371395)

Accuracy: 0.382

Figure 5.2.2: Metrics obtained from the confusion matrix with outliers.

The resulting confusion matrix can be seen on figure 5.2.1 and it shows an average accuracy of 0.382. With a quick look at the confusion matrix we can see that the model predicts some classes better than others.

5.2.3. KNN without outliers

With this second approach at using the KNN, we will use a dataset where we did some more exhaustive preprocessing and we deleted severe and multivariate outliers. Then we followed the same steps as before.

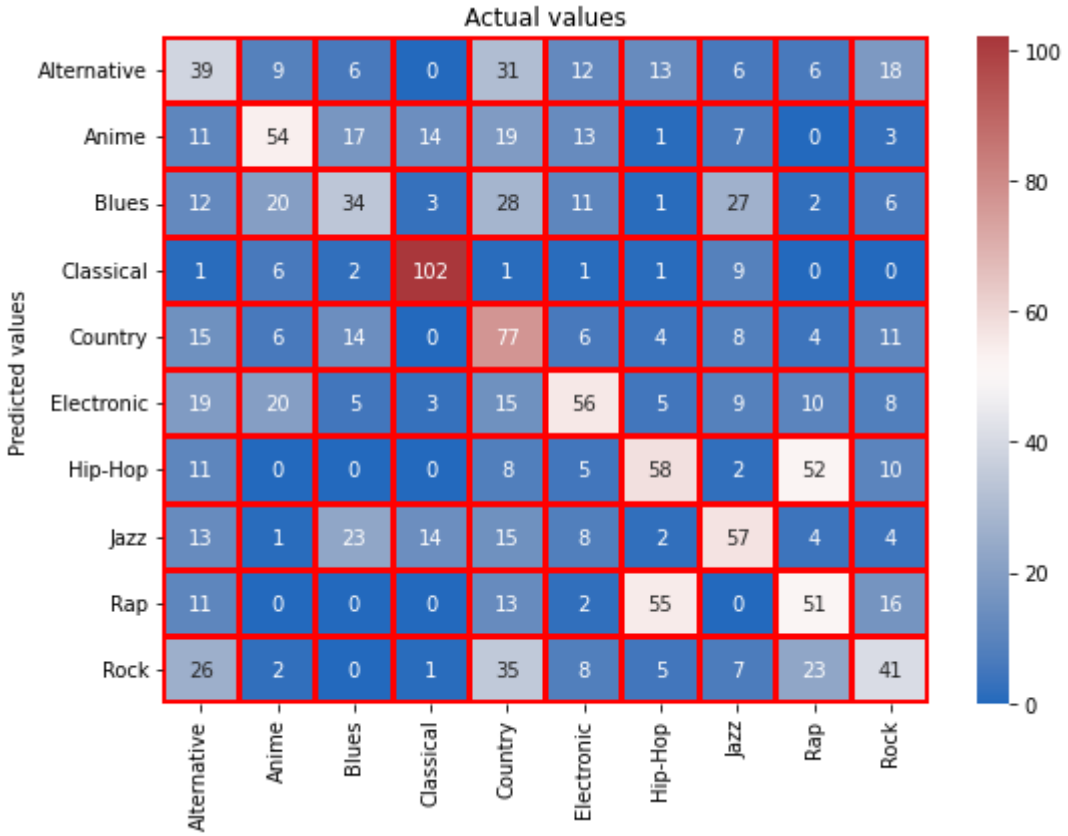


Figure 5.2.3: Confusion matrix of the model without outliers.

	precision	recall	f1-score	support
Alternative	0.25	0.28	0.26	140
Anime	0.46	0.39	0.42	139
Blues	0.34	0.24	0.28	144
Classical	0.74	0.83	0.78	123
Country	0.32	0.53	0.40	145
Electronic	0.46	0.37	0.41	150
Hip-Hop	0.40	0.40	0.40	146
Jazz	0.43	0.40	0.42	141
Rap	0.34	0.34	0.34	148
Rock	0.35	0.28	0.31	148
accuracy			0.40	1424
macro avg	0.41	0.41	0.40	1424
weighted avg	0.40	0.40	0.40	1424

Interval of confidence: (0.374271686687884, 0.4255455517365586)

Accuracy: 0.39957865168539325

Figure 5.2.4: Metrics obtained from the confusion matrix without outliers.

The resulting confusion matrix can be seen on figure 5.2.3 and it shows an average accuracy of 0.399, therefore the treatment of outliers improves the performance of our model.

5.2.4. Finding best parameters to use

The K-NN algorithm has three parameters: k (number of neighbors),distance weighting and distance metric. To begin, K is the number of closest neighbors, which is employed in majority voting. Second, distance weighting is a parameter that, rather than collecting the votes of the k-nearest neighbors directly, allows each vote to be weighted by the distance of that instance from the new data point. Finally,the distance metric is the approach for determining a query point's closest neighbors

In order to find the best parameters we used the GridSearchCV, the parameters we used on the GridSearchCV were:

1. N_neighbors: list(range(1,30,2))
2. Metric values: (‘euclidean’, ‘manhattan’, ‘chebyshev’, ‘minkowski’)
3. Weight values: (distance, uniform)

The results were the following:

- Best Params= {'metric': 'manhattan', 'n_neighbors': 7, 'weights': 'distance'}

5.2.5. Conclusions

The KNN algorithm offers us an easily implemented and without need of training model to predict categories. We have seen through its development that without proper preprocessing the accuracy can be affected resulting in a decrease of accuracy. However, with proper preprocessing we got a decent accuracy of 0.399 which isn't as high as we expected but it's an average of the accuracy of the predictions of each genre. In some genres such as "Classical" and "Anime" we get a precision of 0.74 and 0.46 respectively which are notoriously higher than other genres such as "Alternative" and "Country" with precision of 0.25 and 0.32 respectively. We concluded that this low average accuracy is caused due to the lack of highly correlated variables with each music genre and the fact that a big fraction of the genres had similar features.

5.3 Decision Trees

5.3.1 Introduction

Decision Trees are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

Some advantages of decision trees are:

- Simple to understand and to interpret. Trees can be visualized.
- Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed. Note however that this algorithm does not support missing values.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.

The disadvantages of decision trees include:

- Decision-tree learners can create over-complex trees that do not generalize the data well. This is called overfitting. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
- Predictions of decision trees are neither smooth nor continuous, but piecewise constant approximations as seen in the above figure. Therefore, they are not good at extrapolation.

- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

5.3.2 First approach

With our first approach at creating a decision tree, we will use a version of our dataset where we did not delete severe or multivariate outliers. We will drop all categorical variables, because we can only use numerical variables in our decision tree (later on we'll use them). After that, we will split the dataset into training and testing data and we will invoke the function to create a decision tree. We will use the standard way to predict the tree, therefore we will not use any control mechanism to control the depth of the tree or the minimum sample split or the minimum impurity decrease.

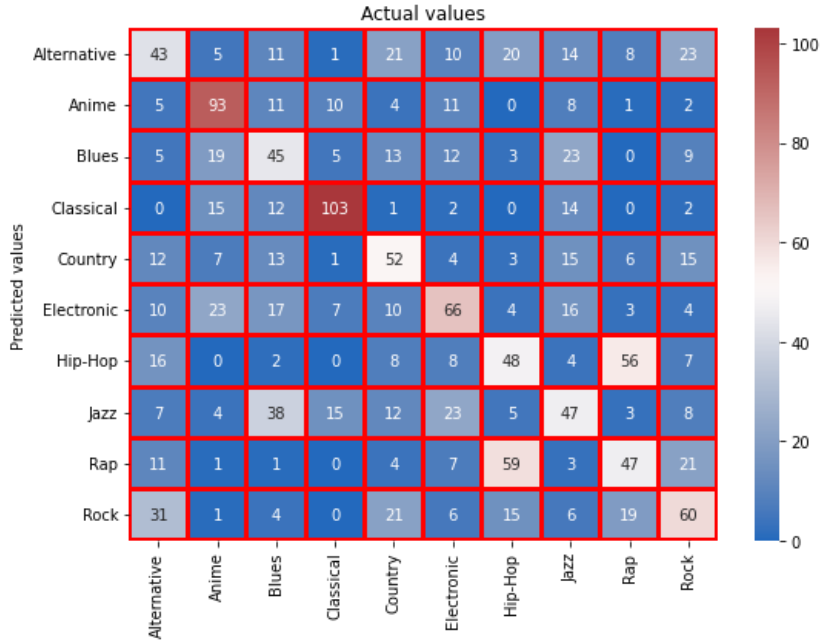


Figure 5.3.1: Confusion matrix of the model with outliers.

The resulting confusion matrix can be seen on figure 5.3.1 and it shows an accuracy of 0.4067. With a quick look at the confusion matrix we can see that the model predicts some classes better than others.

5.3.3 Decision tree without outliers

With this second version of our decision tree, we will use a dataset where we did some more exhaustive preprocessing and we deleted severe and multivariate outliers. Then we followed the same steps as before: We dropped all categorical variables, we splitted the dataset into training and testing and we used the standard way to predict the tree.

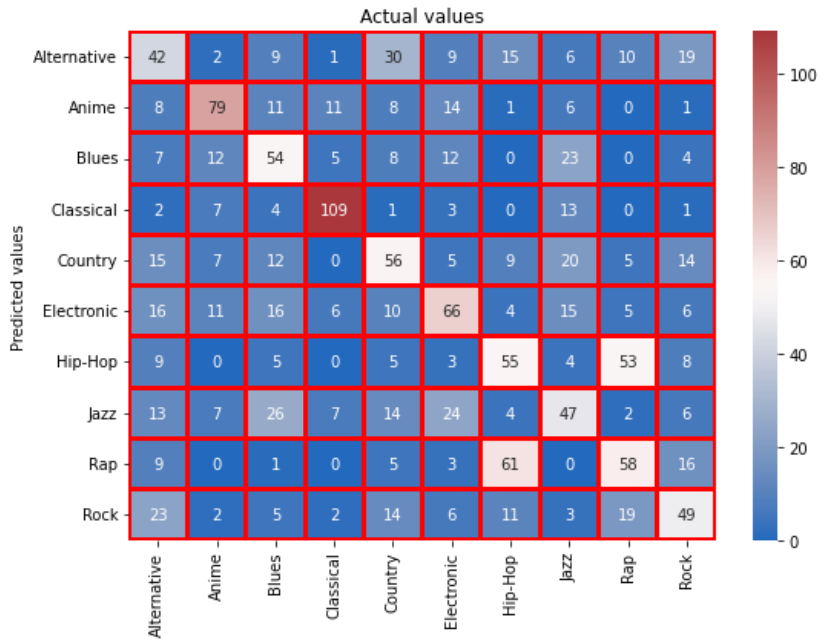


Figure 5.3.2: Confusion matrix of the model without outliers.

The resulting confusion matrix can be seen on figure 5.3.2 and it shows an accuracy of 0.432, therefore the treatment of outliers improves the performance of our model.

5.3.4 Decision tree with some optimizations

In the previous sections we developed a decision tree without any control mechanism, so we most likely ended up with a decision tree that had a big depth and was not readable. We would like to create a decision tree that was interpretable in order to better understand the algorithm and the steps that it makes in order to predict the results. We will also include some optimizations like the inclusion of our categorical variables as numeric variables so that the algorithm can make use of them.

```
X = pd.get_dummies(X)
#convert categorical variables to numeric so we can use them in decision tree
(X_train, X_test,  y_train, y_test) = cv.train_test_split(X, y, test_size=.3, random_state=42)
clf=tree.DecisionTreeClassifier(criterion='entropy', min_samples_split=2,min_impurity_decrease=0.02)
```

Figure 5.3.3: Used commands to create the decision tree.

As figure 5.3.3 shows, we used the function get_dummies to convert our categorical variables into numeric variables and we used 2 control mechanisms, the minimum sample split and the minimum impurity decrease.
But as a result of adding control mechanisms to the tree, we worsen our performance to 0.406.

5.3.5 Decision tree with further optimizations

To improve the performance a bit, we reduced the minimum impurity decrease and added a max depth of 4 so that we can end up with a readable tree. We also reduced the test size to 10%, so that our tree does not overfit too much.

```
(X_train, X_test,  y_train, y_test) = cv.train_test_split(X, y, test_size=.1, random_state=42)
clf=tree.DecisionTreeClassifier(criterion='entropy', max_depth=4, min_impurity_decrease=0.01)
```

Figure 5.3.4: Used commands to create the decision tree.

With those small changes, we achieved an accuracy of 0.423 and the following confusion matrix. As seen before, there are classes where the model has lower accuracy at predicting than others. The worst performing classes are *alternative* with a score of 0.22 and *blues* with a score of 0.3. The best performing class is classical music with a 75% accuracy. This high score could be due to the high difference between values of other genres and classical music, as we saw during the variable analysis section.

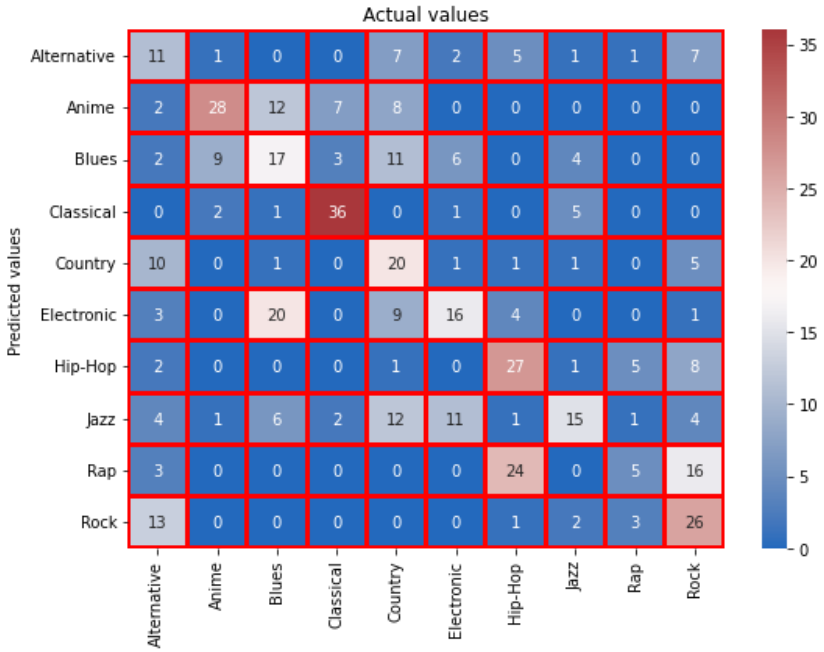


Figure 5.3.5: Confusion matrix of the model without outliers and with optimizations.

In figure 5.3.6 we can see in more detail the precision and other metrics for all the music categories. As said before, there are categories that perform better than others.

```
Accuracy: 0.4231578947368421

              precision    recall  f1-score   support

 Alternative       0.22      0.31      0.26        35
       Anime       0.68      0.49      0.57        57
       Blues       0.30      0.33      0.31        52
   Classical       0.75      0.80      0.77        45
     Country       0.29      0.51      0.37        39
  Electronic       0.43      0.30      0.36        53
     Hip-Hop       0.43      0.61      0.50        44
        Jazz       0.52      0.26      0.35        57
         Rap       0.33      0.10      0.16        48
        Rock       0.39      0.58      0.46        45

    accuracy                           0.42       475
   macro avg       0.43      0.43      0.41       475
weighted avg       0.45      0.42      0.41       475

Interval of confidence: (0.3788671374867574, 0.4684009521881719)
```

Figure 5.3.6: Goodness of fit metrics from the model.

From our model, we can also extract the importance of the features that the model is using to predict the music genre. In our model, we can see that out of the 14 different variables, our model uses just only 5 and the most important variable, by far, is *popularity*. As we saw in the variable analysis section, the variables with most correlation with the music genre were the ones that appear in this feature importance table.

	Feature	Importance
0	popularity	0.618
1	acousticness	0.192
2	speechiness	0.087
3	instrumentalness	0.062
4	danceability	0.041

Figure 5.3.7: Feature importance table.

The resulting decision tree can be seen in figure 5.3.7 For each intermediate node, we have a condition so that we can flow through the nodes according to the values of our song we are trying to predict.

Pere Arnau Alegre
Andrés Jiménez González
Victor Teixidó López
Max Vives Ribera
You Wu

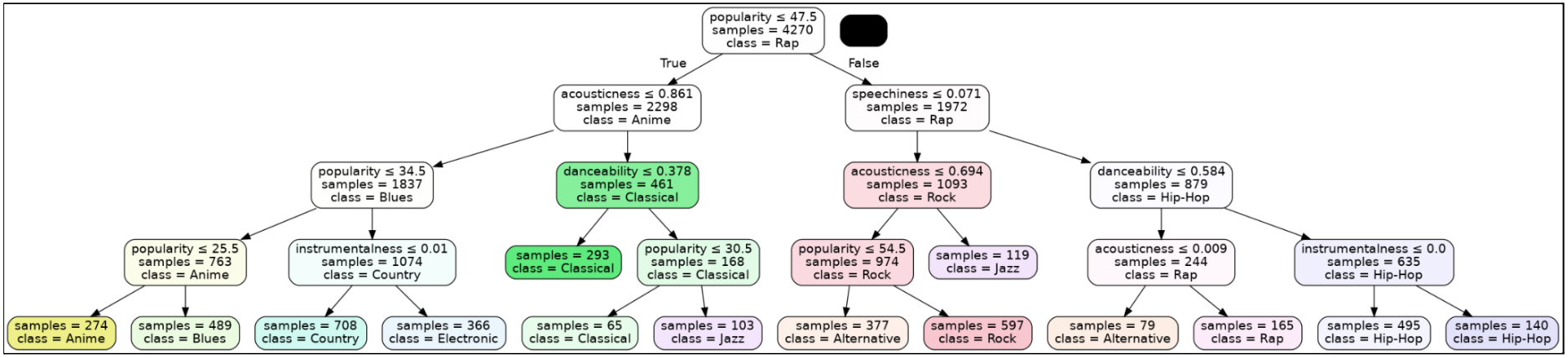


Figure 5.3.8: Decision tree model.

5.3.5.1 Practical Example of usage:

As an example on how to interpret a decision tree, we will try to predict the genre of a song of our dataset. We chose a classical song so that we are likely to get a right prediction. In order to predict our song, in this particular case, we just need 3 out of the 14 variables that the song contains. Those are *popularity=0, acousticness=0.951, danceability=0.394.*

We will start the prediction with the root node that has the condition *popularity<=47.5.* Our value for *popularity* is 0 so the condition is true, therefore we will go to the left node. Following the other conditions, we end up in a node of *class=classical*, so our model made a right prediction.

Pere Arnau Alegre
Andrés Jiménez González
Victor Teixidó López
Max Vives Ribera
You Wu

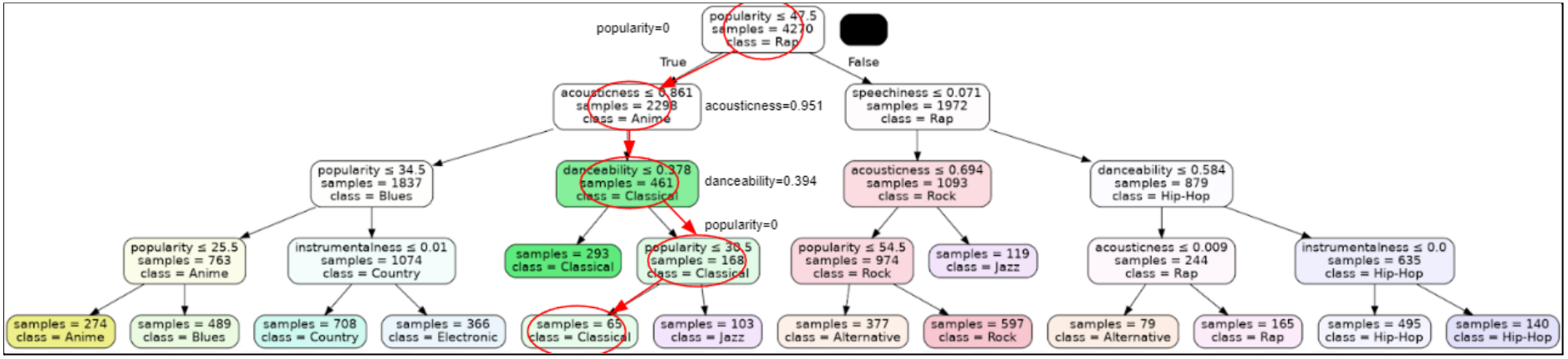


Figure 5.3.9: Decision tree model with the path followed to predict a classical song.

5.3.6 Performance driven model

From our previous models, we saw that some of the depth control mechanisms that were added hurt the accuracy of the model, despite being necessary in order to achieve a readable model. We also saw that some of the music genres of our dataset had really poor accuracy because they shared common traits with other genres or just because our model did not fit well with their values.
As a final experiment, we will try to fix the previous problems as an attempt at achieving a better model.

```
depth = []
for i in range(3,20):
    clf = tree.DecisionTreeClassifier(criterion='entropy', max_depth=i)
    # Perform 7-fold cross validation
    scores = cross_val_score(estimator=clf, X=X, y=y, n_jobs=-1)
    depth.append((i,scores.mean()))
print(depth)
```

Figure 5.3.10: Code snippet of the loop used to calculate the best depth.

Using a small loop and a cross validation function, we can compute the best depth to use for our model. We get that with a depth of 6 the model accuracy is the best, as it is high enough so that the tree can grow and use different variables as conditions for the nodes, and low enough so that the tree does not overfit the data too much.
To improve performance, we dropped from our dataset all the observations with music genres of *Alternative, Blues, Country* and *Hip-Hop* as they were the worst performing categories.
As a result of the model we get the following confusion matrix with an accuracy of 0.672

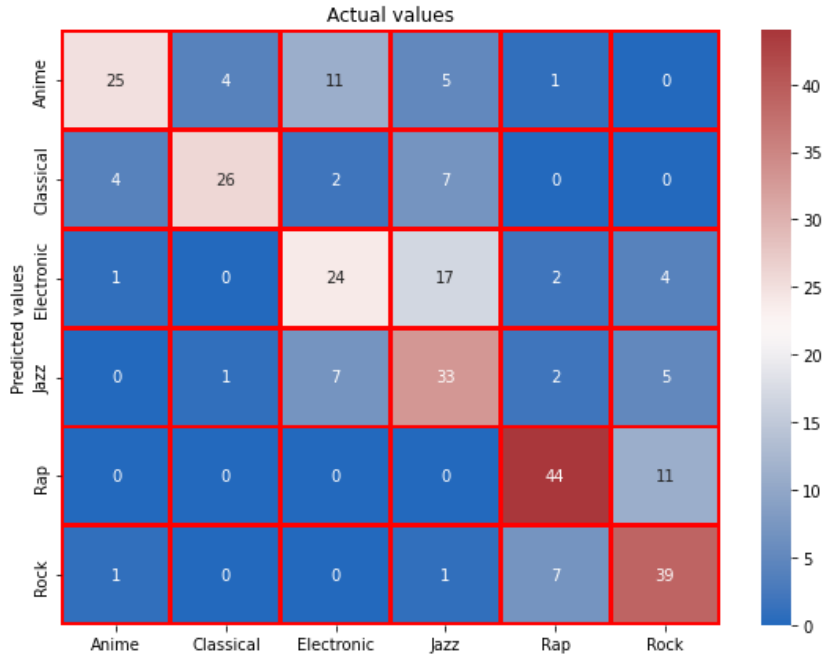


Figure 5.3.11: Confusion matrix of our final model with the deletion of some music genres.

5.3.7 Conclusions

Decision trees offer us a very simple and understandable model with very simple rules to predict categories. We have seen that due to its ease of development, it can produce big problems like overfitting and low accuracy. We also saw that we were able to fix these problems by tweaking some parameters like the maximum depth or the minimum impurity decrease. With our final and simplest model, we got an accuracy of 0.423. We determined that this low score was due to the lack of highly correlated variables with the music genre and to the fact that some genres had very similar features, and also due to the maximum depth of the tree.

5.4 Meta-Learning algorithms

Meta-learning refers to learning algorithms that learn from other learning algorithms, making predictions and providing information about the performance of these learning algorithms, this method only uses variables numericals: *popularity, durations_s, tempo, valence, acounticness, danceability, energy, instrumentalness, liveness, loundness, speechiness*.

During the analyzation of the methods we use a cross validation 50. All the results are calculated in the notebook called "*MetaMethods_con_outliers.ipynb*"

In this section we will do 5 different Meta-Learning methods:

5.4.1. Voting Scheme

This methods trains various base estimators and predicts on the basis of aggregating the findings of each base estimator.

In our case we use Naive Bayes, K-NN and Deciscion Tree algorithms and apply it to the music genere dataset.

First, we make a train for each algorithm classifier, we use GridSearchCV to find the best parameters for K-NN. And we obtain the following results:

```
Best Params fo Knn= {'n_neighbors': 27, 'weights': 'distance'} Accuracy= 0.32086666666666663
Accuracy: 0.417 [Naive Bayes]
Accuracy: 0.321 [Knn (3)]
Accuracy: 0.431 [Dec. Tree]
```

The Decision Tree is the classifier which we obtain best results and Knn the worst classifier.

The voting scheme can be either voting='hard' or voting='soft':
Hard/Majority Voting: calculated on the predicted output class.
Soft/Weighted Voting: calculated on the predicted probability of the output class.

In practical the output accuracy will be more for weighted voting than majority voting, as Knn effect negatively in the result, so in weighted voting we put more weight for Naïve Bayes and Decision Tree methods.

```
Accuracy: 0.428 [Majority Voting]
```

```
Accuracy: 0.437 [Weighted Voting]
```

43.7% is a great accuracy score, we can compare the ensemble's performance to the theoretical performance of each individual classifier, using this method we can improve the performance of the model and climb up the rank ladder.

5.4.2. Bagging

In this algorithm, we will do Decision Tree bagging for each classifier with different estimators(1,2,5,10,20,50,50,100,200). We are going to fix the parameter max_features: 1.0 (default value) and 0.35.

```
Accuracy: 0.408 [1]
Accuracy: 0.399 [2]
Accuracy: 0.469 [5]
Accuracy: 0.503 [10]
Accuracy: 0.520 [20]
Accuracy: 0.535 [50]
Accuracy: 0.535 [100]
Accuracy: 0.537 [200]
```

Bagging method with max_features = 1.0

```
Accuracy: 0.235 [1]
Accuracy: 0.252 [2]
Accuracy: 0.305 [5]
Accuracy: 0.346 [10]
Accuracy: 0.392 [20]
Accuracy: 0.442 [50]
Accuracy: 0.466 [100]
Accuracy: 0.504 [200]
```

Bagging method with max_features = 0.35

We can observe all accuracy use the default max_feature is above 53.7%, and the accuracy with max_feature = 0.35 (that's mean only take 35% of features) increases as we increase number of estimators. The best accuracy we obtained in this classification is 53.7%, with default max_features and n_estimador = 200.

5.4.3. Random Forest

The next is the Random Forest, this method fits a number of classifier(n_estimators) and we decided to train modifying the n_estimators, the following we have results.

```
Accuracy: 0.392 [1]
Accuracy: 0.366 [2]
Accuracy: 0.468 [5]
Accuracy: 0.506 [10]
Accuracy: 0.521 [20]
Accuracy: 0.537 [50]
Accuracy: 0.551 [100]
Accuracy: 0.552 [200]
```

Random Forest method

We have also used Extra Trees method, it is similar to Random Forest but this method we only get a single reason prediction(yes/no) for each classifier. In this dataset, we have not obtained better accuracy.

```
Accuracy: 0.355 [1]
Accuracy: 0.352 [2]
Accuracy: 0.442 [5]
Accuracy: 0.487 [10]
Accuracy: 0.514 [20]
Accuracy: 0.541 [50]
Accuracy: 0.545 [100]
Accuracy: 0.550 [200]
```

Extra Trees method

5.4.4. Boosting

Boosting tries to reduce variance of base classifiers by building different bootstrapping datasets.

* Selecting a decision stump
* Increasing the weighting of cases that the decision stump labeled incorrectly, while reducing the weighting of correctly labeled cases

We use Ada Boost Classifier and Gradient Boosting Classifier with different value of n_estimators to improve the performance.

Ada Boost method, we apply base estimator Decision Tree Classifier with max_depth value None and 5. In the following, we can see the results obtained.

```
Accuracy: 0.197 [1]
Accuracy: 0.265 [2]
Accuracy: 0.346 [5]
Accuracy: 0.398 [10]
Accuracy: 0.432 [20]
Accuracy: 0.426 [50]
Accuracy: 0.392 [100]
Accuracy: 0.364 [200]
```

Ada Boost method with max_depth = None

```
Accuracy: 0.441 [1]
Accuracy: 0.420 [2]
Accuracy: 0.400 [5]
Accuracy: 0.380 [10]
Accuracy: 0.412 [20]
Accuracy: 0.446 [50]
Accuracy: 0.474 [100]
Accuracy: 0.489 [200]
```

Ada Boost method with max_depth = 5

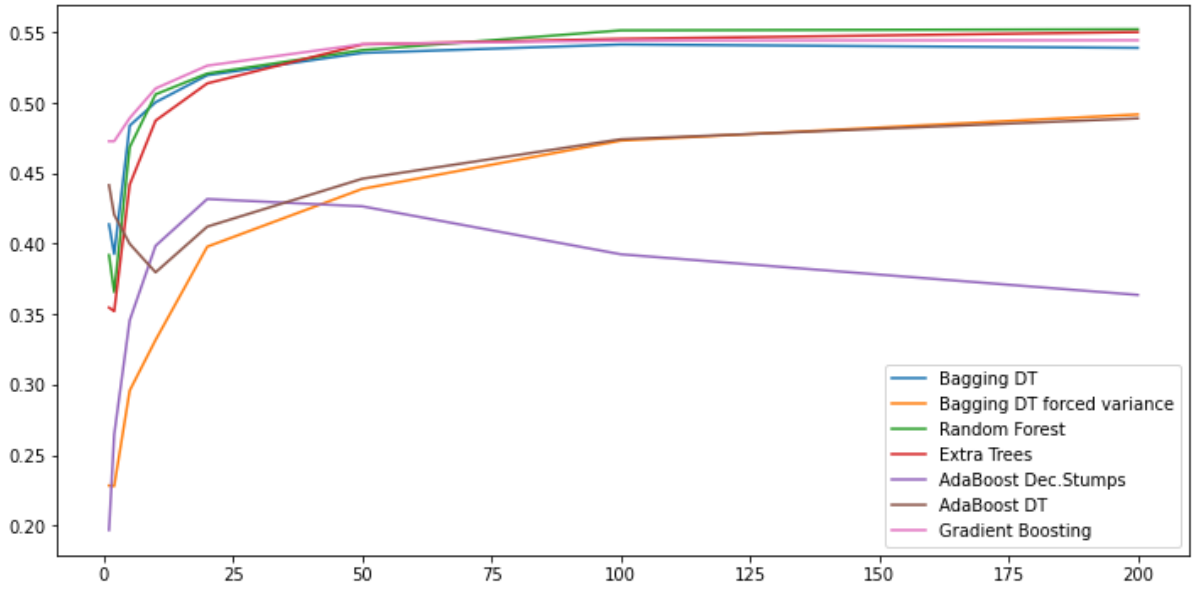
```
Accuracy: 0.473 [1]
Accuracy: 0.473 [2]
Accuracy: 0.489 [5]
Accuracy: 0.510 [10]
Accuracy: 0.526 [20]
Accuracy: 0.541 [50]
Accuracy: 0.544 [100]
Accuracy: 0.544 [200]
```

Gradient Boosting method

With Gradient Boosting Classifier we achieve the better results we had seen.

5.4.5. Conclusion Meta Learning Algorithms

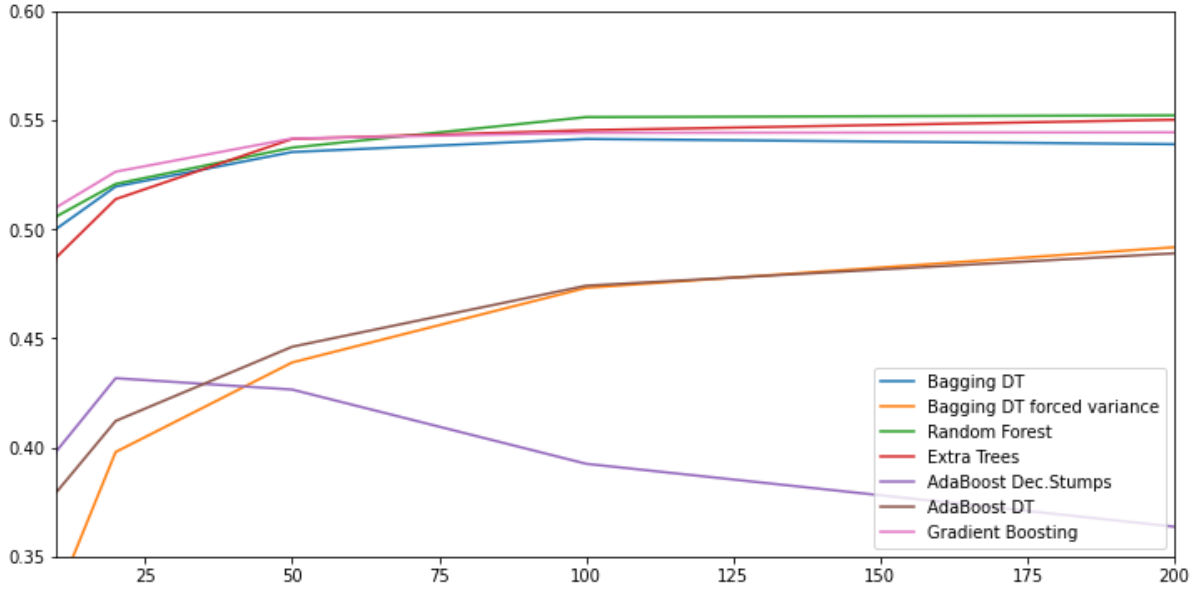
We decide to use a line plot to compare easlier all accuracy obtained using diferent classifers of Meta Learning, below you can see the plot, where x-axis indicate the accuracy obtained and y-aixis the number of estimadors:



Observing the plot, we achieve good results. The results of Bagging DT, Random Forest, Extra Trees and Gradient Bosting are quite similar.

All of them improves as the number of estimators increases, excepting the Adaboost Dec. Stump, when the n_estimators arrives 25 decrease caused by overfitting, thats mean our model doesn't generalize well from training datas to unseen data.

The below we can see the plot centralized in accuracy between 0.35 and 0.6.



We can observe the Bagging DT, Random Forest, Extra Trees and Gradient Bosting remains almost constant independent of number of estimators. Once the estimator number exceeds around 75 estimators the Random Forest becomes the method in which has the highest accuracy, which the better result was the 55,2% of accuracy.

In the notebook we do all previous calculations with dataset with outliers, we also have made a test for dataset without outliers in "*MetaMethods_without_outliers.ipynb"* but we obtain a quite worse results than with outliers, It's because we have music genre that have special features in it that we've eliminated as if they were outliers.

We also applied this methods on the dataset with and without outliers in the notebook "*MetaMethods_with_outliers_fimportances_features.ipynb"* and *"MetaMethods_wihtout_outliers_importances_features.ipynb"* respectively, we only choose the most correlated variables and most important features of the dataset, but both two don't have any feature which are very correlated, that was the reason why we only choose the importants features. In importants features with outliers we take features: *popularity, valence, acounticness, danceability, energy, instrumentalness, loundness* and *speechiness* to predict label *music genre,* and the best result of this notebook are 53.8% using the Random Forest algorithm. And without outliers we only have three features: *acounticness, danceability* and *speechiness,* using Gradient Boosting method we got the best result in this notebook 31.2% accuracy. Both of them we don't have get best result than the notebook "*MetaMethods_con_outliers.ipynb".*

to conclude this, using meta-learning with outliers, we obtain highlight results were majority of them have a accuracy between 45% and 55%, having a dataset with 10 different music genres.

5.5 Support Vector Machines

When talking about supervised learning models in machine learning, Support Vector Machines, or SVMs, are one of the most robust prediction models. Those are characterized for analyzing data for classification and regression analysis. Given an example training set we can label the different classes and use it to train an SVM in order to build a model that can perform a proper classification of a new sample.

The main idea of the performance of this algorithm is the representation of the sample points in the space and to separate every 2 classes at the maximum distance possible through a separation hyperplane, called support-vector. This vector is defined by the 2 closest points of both classes. When a new point is introduced in the model, dependending in the spaces which it belongs to, it can be classified in one class or another. In Figure 5.5.1, we can see an example of a separation between 3 classes and their corresponding support-vectors.

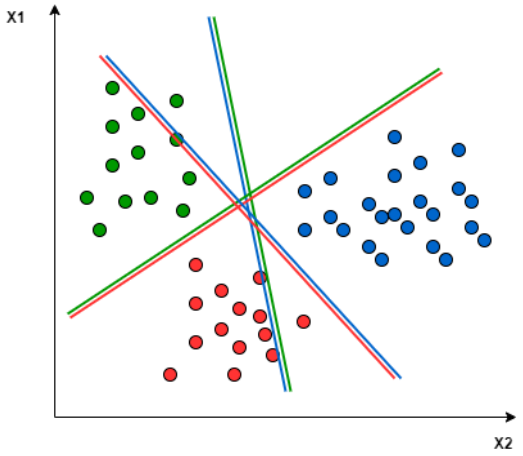


Figure 5.5.1: Example of a SVM between three classes

Focusing on our database, we will be using the sample of the data without outliers and after that the one with outliers. The main problem that resides in Support Vector Machines are the never ending execution times that functions like *GridSearch* needs to perform. After executing the script with both datasets we will compare the results and reach the conclusions made.

The first step will be dividing our data in training and test samples, as their names suggest, we will use the training sample to train our Support Vector Machine and the test samples will be used to check the accuracy of the model we built. The library *model_selection* from *sklearn* will help us to separate the data, after doing a bit of a research we found that the best relation would be using a 80/20 ratio between training and testing, so that's the ratio we chose also. We used the variable *music_genre*, the one to predict, to stratify the data using also the option to shuffle the data before stratifying. It's important to clarify that, before dividing the data and starting to train the model, we took only the numerical variables of our dataset and normalized them.

First step when working with Support Vector Machines is choosing the kernel to work with. Typically when starting with a new data set, it is recommended starting with kernels with a simple hypothesis space, such as *Linear*. So during this section of the project, we will work with *Linear* and then we will jump into more complex hypothesis spaces, specifically, with *RBF* (Radial Basis Function). As we were told in laboratory class, those 2 methods are enough to obtain good results and have an idea of which kernel is the best for our dataset.

But before starting training without a clear objective in mind, we have to know what we are looking for when testing the different kernels in our dataset. Having in mind that our data has more than 2 dimensions, 10 to be precise, we will be looking to compare the different kernels through hyperparameter search such as *grid search*. More precisely, we will choose which kernel is the best one for our data set based on the accuracy they obtain.

5.5.1 Without outliers

As we mentioned above, first of all we will run the script through the sample of the dataset cleaned from missings, NAs and extreme outliers.

Linear Kernel

When using Linear SVM, what we are really doing is executing a Support Vector Machine with default parameters, in other words, we are not using any kernel to move the data to a higher dimensional space. In the first touchdown with the script, we simply execute the *Linear* Kernel without any specific variables or metrics with the code we see in Figure 5.5.1.1.

```
knc = SVC(kernel='linear')
knc.fit(X_train, y_train)
pred=knc.predict(X_test)
```

Figure 5.5.1.1: Default Linear Kernel script

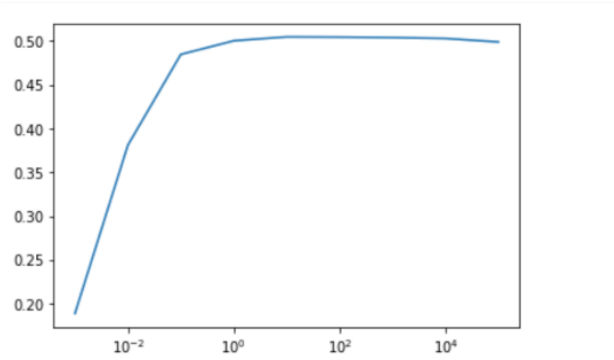
Through this we obtain an accuracy around 0.49, approximately, which in the context of having 10 label categories and almost 5000 rows, are decent results. Still, we can probably achieve more accuracy with some improvements. Next step will be finding the optimal C value, which default is 1, through the GridSearch method. In Figure 5.1.1.2, we can see the initial fragment of the script used to find that value. Before commenting on the results, let's talk about the parameters chosen for the execution.

```
Cs = np.logspace(-3, 5, num=9, base=10.0)
param_grid = {'C': Cs}

grid_search = GridSearchCV(SVC(kernel='linear'), param_grid, cv=5)
grid_search.fit(X_train,y_train)
```

Figure 5.1.1.2: Script fragment to find optimal C value in Linear Kernel

First of all, we chose a wide range of C values to search through in order to maximize the probabilities of finding the best one. Parameter *cv*, defines how many sections are we gonna split our data each of these folds are going to be used as testing at some point. As it seems reasonable, the higher the number of folds the bigger the computational time. At the beginning we tried to run the script with 10 as the value of cross-validation parameter, but *Google Collab* was killing our process constantly due to long time execution. To solve that, we decided to use instead 5 as a value, which, moreover, tends to be the default value.

In Figure 5.1.1.3 we can see the results obtained from the script mentioned above. The best value of parameter C found is 10, and we can see this, although it may not be big enough, in the left plot where, when C has value 10, reaches its maximum peak of accuracy. Moreover, the accuracy on the test set it's around 0.49 percent, basically the same we obtained with the default C value.




```
Acc. 5-fold cross on train data=  0.5044813119755912

Confusion matrix on test set:
 [[37  2  1  0 12  9 10  5  5 12]
 [ 2 51  9 15  4  7  0  1  0  3]
 [ 1 11 43  2 13  7  1 11  0  6]
 [ 1  3  3 71  1  2  0  4  0  1]
 [ 7  4 10  0 44  3  4  7  2 17]
 [11 10  6  2  6 50  3  5  0  3]
 [ 8  0  0  0  4  0 35  3 36 10]
 [ 4  4 12  6  9 14  2 40  1  4]
 [ 9  0  0  0  2  0 40  0 33 17]
 [11  1  0  0  6  1  2  2 11 62]]

Accuracy on test set:  0.4910432033719705

Best value of parameter C found:  {'C': 10.0}

Number of supports:  3271 ( 9418 of them have slacks)
Prop. of supports:  0.8616965226554267
```

Figure 5.1.1.3: Output of the script of the Linear Kernel

RBF Kernel

The other kernel we are going to work with and analyze the results is the RBF Kernel or Radial Basis Function, it is used to find a non-linear classifier or regression line. First of all, we are going to execute the default version, without changing any parameters, of this Kernel. Important to mention that RBF Kernel is the default kernel implementation of SVMs in sklearn, so we don't need to specify the kernel to use. In figure 5.1.1.4, we can see the default RBF Kernel script.

```
knc = SVC()
knc.fit(X_train, y_train)
pred=knc.predict(X_test)
```

Figure 5.1.1.4: Default RBF Kernel script

Using the script above, we obtain an accuracy around 0.508, approximately, which is better of both accuracies we found with the Linear Kernel. Still, we can probably achieve more accuracy with some improvements. Next step will be finding the optimal C and gamma values through the GridSearch method. In Figure 5.1.1.5, we can see the initial fragment of the script used to find that value. Before commenting on the results, let's talk about the parameters chosen for the execution.

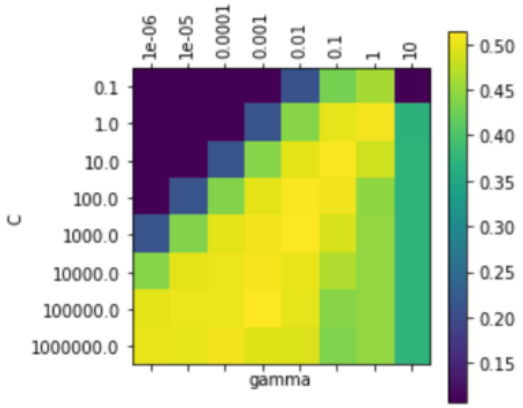
```
gammas = [0.000001,0.00001, 0.0001,0.001,0.01,0.1,1,10]
Cs = np.logspace(-1, 6, num=8, base=10.0)
param_grid = {'C': Cs, 'gamma' : gammas}
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train,y_train)
parval=grid_search.best_params_
```

Figure 5.1.1.5: Script fragment to find the optimal values of C and gamma in RBF Kernel

Respecting the value of cross-validation parameters and C values, the justifications are quite similar as it were with the Linear Kernel. We are still using cv as 5, because it has a good balance between computation and good performance. The gamma variable will let us explore between different combinations of possibilities and *GridSearch* will pick the best one.

The range of this last variable is not so wide due to computational complexity, despite that, the range is wide enough to get good results.

In Figure 5.1.1.6, we can see the best combination between C value and gamma that *GridSearch* has found. Having 100000 as a C value and 0.001 as gamma value our kernel manages to obtain a little bit more than 0.51 percent of accuracy using the train data. So, as it seems logical, the next step is to test these values with test data to see how much accuracy we really obtain.




```
Best combination of parameters found:  {'C': 100000.0, 'gamma': 0.001}

Acc. 5-fold cross on train data=  0.5102714763543117
```

Figure 5.1.1.6: Displaying grid of best combination of parameters found

Checking the Figure 5.1.1.7, we can see the accuracy we obtained with the optimal parameters of the RBF Kernel. We obtain a little bit more than 0.511 percent. So we can confirm that, with this kernel we increased in, more less, a 2% our accuracy respect the Linear Kernel we used in the beginning. It's quite difficult for a dataset to be perfectly linear separable so, in general, this kernel should provide better results. At the end of this section, we will discuss in more depth which is the better option to pick.

```
Confusion matrix on test set:
 [[37  1  3  0 14  3 12  5  5 13]
 [ 2 50 11 14  5  6  0  1  0  3]
 [ 1  7 45  4 12  6  0 12  0  8]
 [ 2  3  2 73  1  0  0  5  0  0]
 [ 4  2 11  0 43  4  2  9  5 18]
 [ 8 14  4  1  7 51  1  5  2  3]
 [ 5  0  0  0  3  1 40  1 40  6]
 [ 4  3  9  4 12 18  2 41  1  2]
 [ 6  0  0  0  3  0 36  0 42 14]
 [13  1  0  0  4  1  4  2  8 63]]

Accuracy on test set:  0.5110642781875658

Number of supports:  3179 ( 7892 of them have slacks)
Prop. of supports:  0.8374604847207587
```

Figure 5.1.1.7: Output of the script with the C and gamma optimal values found

If we had to choose a kernel between Linear and RBF ones, we would pick the second one. The main reason is the better accuracy this kernel gives us and, for instance, the better performance we will give when trying to predict new songs into different genres. Is important to mention that, in our case, both kernels had a computation time quite similar, around 2 or 3 hours, knowing that, we cannot discard any of the kernels just for being or slower than the other one. As we imagined, our dataset is not linear and that's why RBF Kernel can achieve better performance on accuracy.

5.1.2 With outliers

Now we are gonna execute the same script we did before but this time without having the extreme outliers removed from the dataset. Missing values and NAs will be treated normally.

Linear Kernel

As we mentioned before, linear kernel does not use complex hypothesis spaces to train the model, in fact, it is designed to be used in linear separable datasets. All variables and values we are gonna use in this execution are the same we used with the dataset without outliers. That's why we are not going to explain again what each variable does and what we choose for each value.

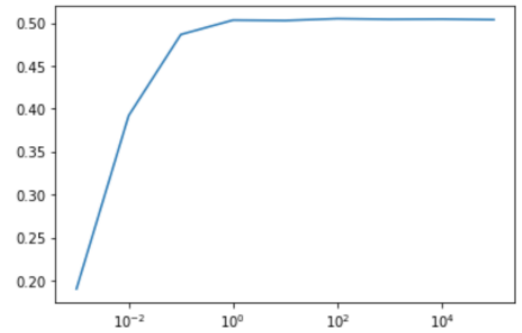
With the first execution having all the parameters by default, for instance without choosing any specific *C* value, we obtained an accuracy of 0.51, we can see the output in Figure 5.1.2.1. It's impressive how this result it's almost as high as the best one we had in the last section, with the data without outliers. In said figure we can also see the confusion matrix generated during the prediction of our dataset.

```
Confusion matrix on test set:
 [[30  1  3  0 17  9  6  7  7 15]
 [ 3 57 12  9  4  7  0  3  1  1]
 [ 4 16 37  4 11  8  1 16  0  3]
 [ 0  3  1 86  0  3  0  5  0  1]
 [ 4  2  6  0 46  9  5 12  2 12]
 [ 9  9  3  0  2 63  3 11  4  1]
 [ 4  0  0  0  7  2 40  4 34  8]
 [ 0  5 12  8 12 18  2 44  0  3]
 [ 8  0  0  0  2  1 33  0 45 15]
 [12  0  1  0  7  1  7  6  3 62]]

Accuracy on test set:  0.51
```

Figure 5.1.2.1: Confusion matrix and accuracy obtained by the Linear Kernel with default parameters

Next step, as we did before, will be finding the best *C* value through the *GridSearch* tool. Again, we will use a cross-validation value of 5 instead of 10, because with our software capacity using *Google Collab* is not possible in terms of computation cost. After executing the script, we obtained the results we see in Figure 5.1.2.2. Comparing these results with the ones we got without the outliers, we can see that, although the accuracy on the train data is

quite similar, in the end on the test set, we got an accuracy more than 2 points higher. To be mentioned also that the best parameter we got for *C* was 100. Despite that, as we can see in the left plot, all *C* values above 1 have almost the same accuracy, and that's why this result and the default one are almost the same.




```
Acc. 5-fold cross on train data=  0.5051311013767209
```

```
Confusion matrix on test set:
 [[31  1  3  0 18  8  7  7  6 14]
 [ 3 60 10  7  6  6  0  3  1  1]
 [ 4 15 38  4 10  8  1 17  0  3]
 [ 0  5  1 84  1  3  0  5  0  0]
 [ 4  2  7  0 45  9  4 12  1 14]
 [ 8  9  3  0  3 63  3 11  4  1]
 [ 6  0  0  0  6  3 40  2 35  7]
 [ 0  4 14  8 12 16  3 44  0  3]
 [ 7  0  0  0  3  2 34  0 44 14]
 [13  0  1  0  6  1  7  5  3 63]]

Accuracy on test set:  0.512

Best value of parameter C found:  {'C': 100.0}

Number of supports:  3407 ( 9737 of them have slacks)
Prop. of supports:  0.8519629907476869
```

Figure 5.1.2.2: Output generated by the script to find optimal parameters for Linear Kernel

RBF Kernel

Now let's take a look at the second kernel we choose to work with, RBF Kernel. We are going to follow the same steps we did with the data without outliers. First of all we are going to execute the default RBF Kernel without changing any of the parameters we have and then we will find which is the best combination of *C* value and gamma and use them to find the best accuracy with RBF Kernel. We will not explain again what every parameter means and does because we already explained all these concepts in the section before.

In Figure 5.1.2.3, we can see the confusion matrix obtained and the accuracy we got. Unlike what happened with Linear Kernel, where we obtained higher accuracy than the dataset without outliers, this time we got the same accuracy with both datasets using the default RBF Kernel, and accuracy of 0.503 percent.

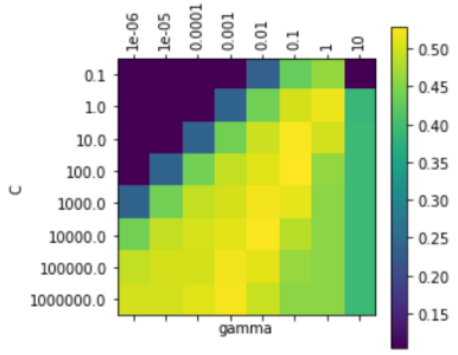
```
Confusion matrix on test set:
 [[29  2  2  0 18  4 10  8  6 16]
 [ 4 55  9 10  6  7  0  4  1  1]
 [ 6 11 32  4 11  7  1 26  0  2]
 [ 1  4  0 88  0  1  0  4  0  1]
 [10  1  8  0 41  9  4 12  2 11]
 [ 4  6  3  0  5 65  1 13  6  2]
 [ 3  0  0  0  8  1 38  3 41  5]
 [ 2  3 12  8 11 11  2 52  0  3]
 [ 4  0  0  0  4  1 41  0 42 12]
 [12  0  0  0  7  1  5  9  4 61]]

Accuracy on test set:  0.503
```

Figure 5.1.2.3: Confusion matrix and accuracy obtained by the RBF Kernel with default parameters

Now our next step is to find which combination of parameters adjust better to our dataset and use them to obtain as much accuracy as we can. The parameters we are looking for in this kernel are the *C* value and the gamma value. Important to mention that the ranges to

explore and cross-validation value remains the same that were in the previous section. Remember that *C* value tells the SVM how much we want to avoid misclassifying each training example. On the other hand, gamme value represents the decision region the kernel will use.

After executing the script, that is the same one we used with the dataset without outliers, we obtained the results we see in Figure 5.1.2.4. This time, the best parameters found by *GridSearch* were 100 for *C* and 0.1 for gamma. We can see that, on train data we achieved an accuracy of almost 0.53 which is a value quite good, having in mind that we have 10 possible labels to fit in. Next step, will be testing this parameter with the test data set to prove how well they are trying to predict the music genre of different songs.




```
Best combination of parameters found:  {'C': 100.0, 'gamma': 0.1}

Acc. 5-fold cross on train data=  0.5296357947434294
```

Figure 5.1.2.4: Displaying grid of best combination of parameters found

Checking the Figure 5.1.2.5, we can see the accuracy we obtained with the optimal parameters of the RBF Kernel. We got a little bit more than 0.515 percent. So in this case, both Linear and RBF kernels get almost the same accuracy, the Linear Kernel we saw before got 0.512 of accuracy. It's quite difficult for a dataset to be perfectly linear separable so, in general, this kernel should provide better results.

```
Confusion matrix on test set:
 [[33  2  3  1 16  3 10  7  5 15]
 [ 4 58 11  9  5  7  0  2  1  0]
 [ 7 12 41  4 12  5  0 17  0  2]
 [ 1  4  0 84  1  1  0  7  0  1]
 [ 9  2  8  0 49  5  5  9  2  9]
 [11  8  3  0  5 59  1  9  7  2]
 [ 5  0  0  0  5  2 41  3 39  4]
 [ 4  3 15  9 12 13  1 44  0  3]
 [ 7  0  0  0  2  1 36  0 48 10]
 [11  0  2  0 11  2  5  5  5 58]]

Accuracy on test set:  0.515

Number of supports:  3271 ( 6537 of them have slacks)
Prop. of supports:  0.8179544886221556
```

Figure 5.1.2.5: Output of the script with the C and gamma optimal values found

Now let's see which of the both kernels we saw should we pick when working with a real model. If we had to choose a kernel between Linear and RBF ones, we would pick the second one. The main reason is the little better accuracy this kernel gives us and, for instance, the better performance we will give when trying to predict new songs into different genres. Is important to mention that, in our case, both kernels had a computation time quite similar, around 2 or 3 hours, knowing that, we cannot discard any of the kernels just for being or slower than the other one. As we imagined, our dataset is not linear and that's why RBF Kernel can achieve better performance on accuracy.

SVM conclusions

In both executions, without outliers and with outliers, the results we obtained show us that the best kernel we can use in our dataset is the RBF Kernel. Now the question is, should we use the dataset with outliers or without outliers? Well, the accuracy values we obtained show that with outliers we get better results when identifying the genre of a song. As a first thought, this idea may seem a little contradictory because theoretically a dataset with outliers tends to have worse training and, for instance, tends to predict worse.

Having a dataset with 10 different music genres leads us to a huge variety of possible values in all the different features and each genre may have higher values for features than all the others one. A clear example of that is the variable *duration_ms*, the average duration of a song is 3-4 minutes, but having a music genre as classical musical, leads to song duration of more than 10 minutes. Of course, all these values will be treated as outliers and it may be treated in the dataset, but in the end, if we treat them we are losing valuable information that we could use to detect if a song is a classical music piece.

As an important mention, highlight that all the results we obtained shows that there is a big presence of overfitting, the proportion of supports is more than 0.5 percent which normally tends to be a clear indicator. To solve that problem, normally the best way is to get more data to work with, other options would be to make use of *PCA* to select which variables are the best and work only with them. At the beginning we tried something like that but we got worse accuracies so we discarded that idea to use more in depth.

6. Comparison between models

With the aim of comparing the performance of all the models we used on this dataset and drawing conclusions we have created a table showcasing the values of the accuracy,precision,recall and F1-Score of each of the models.

Model	Accuracy	Precision	Recall	F1-Score
Naive Bayes	0.57	0.67	0.58	0.58
KNN	0.399	0.40	0.40	0.40
Decision Trees	0.423	0.43	0.43	0.41

Support Vector Machines	0.515	-	-	-
Voting scheme	0.437	-	-	-
Bagging	0.537	-	-	-
Random Forest	0.552	-	-	-
Boosting	0.544	-	-	-

7. Conclusion

The preprocessing step has been very important to reduce the size of the data set, originally we had 50.000 rows with 18 columns, we decide to work with around 5.000 rows, with two datasets, one dataset with outliers and one without outliers, both of them are balanced datasets.

As we can see in the previous section table, for all methods, we obtained results between 0.399 and 0.57. The best performing model is Naïve Bayes, in which we obtained an accuracy of 0.57. Although the accuracies that we were obtaining at the beginning didn't convince us at all, at the end, having a total of 10 possible classifications, having an accuracy of almosy 60% is quite good considering the large number of music classes that we had.

Moreover, apart from the results obtained, we have been able to not only learn about music and song attributes, but also about machine learning methods. We now know how to apply different methods, why we should apply them and when to, we also have learned what to do when we run into inconveniences like, not removing the outliers for example.

8. Bibliography

Naive-Bayes:
Cross Validation in Scikit Learn
Scikit-learn
Classification Report in Machine Learning
Naive Bayes, Clearly Explained!!!
Implementing Gaussian Naive Bayes in Python - Analytics Vidhya

K-NN:
Advantages and Disadvantages of KNN Algorithm in Machine Learning
KNN Algorithm: When? Why? How?. KNN: K Nearest Neighbour is one of the… | by Aditya Kumar | Towards Data Science
Advantages And Disadvantages of KNN | by Anuuz Soni | Medium

Preporocessing:
Spotify LUFS: How Loud Should Your Songs Be?
PCA using Python (scikit-learn) | by Michael Galarnyk | Towards Data Science
Detecting And Treating Outliers In Python — Part 2 | Towards Data Science
Python Scatter Plot - Machine Learning Plus
Check for NaN in Pandas DataFrame (examples included) - Data to Fish
Grouped Boxplots in Python with Seaborn - GeeksforGeeks

Decision trees:
1.10. Decision Trees — scikit-learn 1.1.1 documentation
Decision Tree Classifier with Sklearn in Python • datagy
Python Decision Tree Classification Tutorial: Scikit-Learn DecisionTreeClassifier | DataCamp

Meta-learning methods:
opencv-machine-learning/10.05-Combining-Different-Models-Into-a-Voting-Classifier.ipynb
Ensemble/Voting Classification in Python with Scikit-Learn
sklearn.ensemble.GradientBoostingClassifier — scikit-learn 1.1.1 documentation

Support Vector Machines:
How to Select Support Vector Machine Kernels - KDnuggets
K-Fold Cross Validation. Evaluating a Machine Learning model | by Krishni
The Complete Guide on Overfitting and Underfitting in Machine Learning
Radial Basis Function Kernel - Machine Learning - GeeksforGeeks
SVM: Number of support vectors - Cross Validated