UNIVERSITÀ DI PISA

Project DM2:
Data Mining applications on the Free Music Archive dataset

Giovanni Scognamiglio
Agnese Simonelli
Martina Trigilia

Module 1

1.1 Introduction

The objective of this paper is to go beyond the introductory descriptive Data Mining tasks and explore the more advanced predictive capabilities offered by Data Mining algorithms.

The dataset used for the study is the Free Music Archive dataset, hereafter referred to as FMA dataset. The FMA is an open and accessible source of track records derived from the Free Music Archive website, a free and open online music repository. The dataset is designed for music information retrieval (MIR) tasks and contains more than 100.000 tracks along with a wide variety of related metadata ranging from audio features to genre and artist information.

The dataset was arranged by its creators in numerous tables according to their origin and type. The data pulled from the FMA API for each track was stored in the table tracks.csv and was composed of approximately 106'500 tracks and its related metadata along with album and artist metadata for each record. Other data such as pre-computed features for each track, a hierarchical taxonomy of track's genres and audio features extracted from Spotify was placed in other tables, respectively in features.csv, genres.csv and echonest.csv. Other information is also included in the FMA data such as raw mp3 audio; however such data is out of the scope of this paper and thus will not be considered. A selection of attributes from the former tables presented in a tabular can be found in Table **α** in the appendix. As the FMA encompassed a wide range and variety of data, our first step was to proceed with a thorough exploration of the tables with their records and related attributes.

The tracks.csv dataset is a merged multi index relational table where each row represents a single track. Metadata variables relating to the artist and album of each song are respectively represented by the columns with the "album" and "artist" top level index. As expected, there were many repetitions in the albums' and artists' attributes for every song belonging to the same album/artist.

The tracks.csv contained many features which were compiled by humans and thus required a consistent data cleaning. Features with a significant number of missing values were dropped or converted to binary, those included album.engineer, artist.active_year_end and track.date_recorded.

Value inconsistencies were also addressed on numerica features such as comments and listens count. Around 3% of records contained "-1" value for their album or track comments and listens; those values were replaced with their respective mode. An inconsistency was also found in attribute date_released, where around 6% of albums and tracks have a creation date prior to the release date. (ie: the album was uploaded on the platform before being released). By cross-checking with data on the FMA website, we noticed that the upload date (date_created) appears to be generated automatically whilst the release date looks like an attribute which was manually input by the artist when uploading and was therefore prone to data entry errors. We asserted that, for those rows with an inconsistent date_released, a good approximator was date_created, and proceeded to impute as such. Numerous outliers were also noticed among many other numeric features such as tracks' bit rate, tracks' duration and albums' number of tracks. After verifying each outlier against the data available on the FMA website, we proceeded to replace erroneous outliers with mode values. Outliers which were not erroneous (eg. songs with very poor bit-rate or abnormous duration) were left for the anomaly detection task discussed later.

Further exploration was done on the variable artist.location as we figured it could be an interesting attribute for any future data mining implementation. As the variable appeared revealing but had over 60% of missing values, we envisaged keeping it and imputing the missing values from its latitude and longitude attributes. We tried to use Google's location API to impute the artist's location based on his

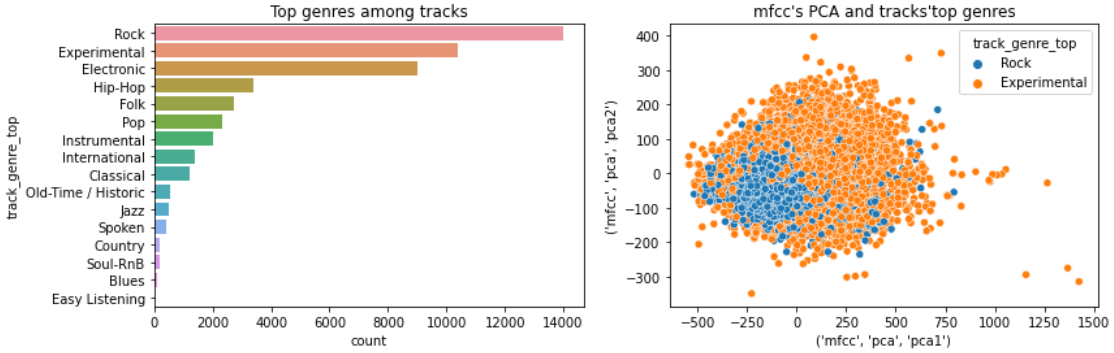
latitude and longitude; however, many records also did not have location coordinates, thus making imputation from coordinates unfeasible.

Our main target variable for the classification part, track.genre_top, was also adjusted. The genre top feature displayed the single hierarchical top genre of the track from the genres present in the genres.csv table. We first noticed that around half of the tracks did not have a genre_top. Every track that had an empty genre_top attribute, had two or more genres or sub genres belonging to different top level genres and therefore was impossible to allocate to a single top level genre. We then noticed that around 2'000 rows had no genres at all in any of the genres columns; rendering them useless for any genres classification purposes. We analysed the various top level genres combination present in the genres_all with the intent to create new top level genres which would be the resulting possible combination (eg. "Pop&Rock" and "Electronic&Funk"). This approach was also inappropriate as we found over 590 combinations of top level genres present in genres_all. To ease and render the various classification tasks more understandable, we eventually decided to drop all the records with mixed top level genres; which left us with around 40'000 records with unique top level genres, which we deemed sufficient for our study.

The other tables of the dataset (features.csv, genres.csv and echonest.csv) were by contrast much cleaner. Indeed, features.csv and genres.csv contained only numerical features obtained from the raw audio features, thus leaving no room for human input and errors. echonest.csv although contained many missing data, rendering many of its features unusable, also had many numerical temporal features of high quality. However, the former were features with low value that were repeated in the tracks.csv table and we thus dropped the features with many missing values and kept the valuable temporal features. With the intent of using the temporal features as the independent variable of various classification tasks, we also transformed the features.csv table. For each main temporal feature of the table (mfcc, chrwoma_stft, …) we added a copy of the values of their PCA transformation. The table feature.csv now has a copy of a two component PCA for each of each main index. We decided to keep both the original values and the PCA's two components variables in order to more easily visualize the effect of temporal features on our target variables and to later compare classification results obtained from PCA features against the original ones.

We finally analysed many of the features' distribution in order to understand how the data was distributed across the features space. For instance, we saw that most of the numerical features in tracks.csv such as tracks.duration, albums.listens., track.favorites had extremely skewed distributions. Whilst numerical features from features.csv were mostly normally distributed, making them practical for our classification tasks. We reported some selected features distribution graphs in the appendix and reported here in Figure 1 the most informative ones. We see that Rock and Experimental are the most popular top genres among the records and when plotting the genres against in a scatter of mfcc's coefficients, we understand that temporal features like mfcc might be discriminant in a classification problem.

Figure 1: Top genres distribution and separation by mfcc coefficients' pca



1.2 Balanced classification

For our first classification task was a simple genre recognition classifier. We decided to build a classifier that would identify whether a song is "Rock" or "Experimental".

After the preprocessing part mentioned previously, we filtered our dataset to obtain only records with "Rock" and "Experimental" labels. Our new dataset was composed of approximately 24'000 records (14'000 records labelled as "Rock" and 10'000 labelled as "Experimental"). As mentioned above, we used the temporal features of features.csv as independent variables and the top level genre variable of the tracks.csv as target variable.

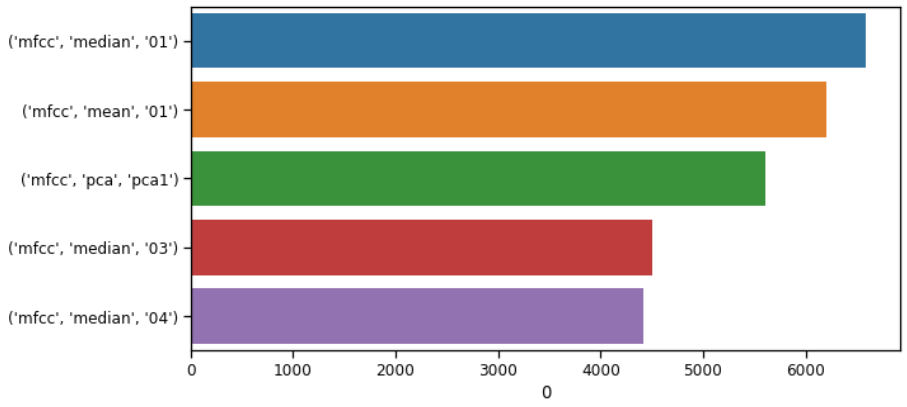
With a 57 to 43 percent ratio we can say the target feature is fairly balanced. We therefore used our common classification techniques.

The first classifier we built was a Decision Tree Classifier. Out of curiosity, our first trial was to use the PCA features of the temporal features (created in data transformation) as the independent variables. After tuning the parameter with GridSearch, we obtained the following cross-validated (5-fold cross validation) results:

- Accuracy: 76,3 % (+/- 0.05)
- Macro avg. F1 - score: 75,8 % (+/- 0.05)

In our second approach, we first used ANOVA to search for the best attributes to use for our classification task (ie. the ones that are most dependent on the target attribute). From Figure 2: we can see the F-scores obtained from ANOVA are incredibly high, telling us that many independent variables are highly dependent on the tranck's genre.

Figure 2: top 5 F-score from one way ANOVA with target variable



We used the top 30 features with the highest F-score and then used GridSearch for tuning the classifiers parameters and obtained the following parameters: max_depth=10, min_samples_split=700, min_samples_leaf=100.
Interestingly, the first intuitive thing we saw is that the results were not particularly distant from the ones obtained with the PCA transformed features. Showing the potential valuable characteristics of adopting PCA for dimensionality reduction.
Our classifier obtained the following cross validated scores:

- Accuracy: 78,1 % (+/- 0.04)
- Macro avg. F1 - score: 77,7 % (+/- 0.04)
- Roc AUC: 0.85

By fitting our model on a random stratified sample, we obtained results in line with the cross-validation. From the obtained confusion matrix we saw that overall the Decision Tree classifier did perform well. Test accuracy was 78,4%, which is significantly better than the 58% one would get by always predicting the majority class Rock.
By comparing the F1-scores of the various classes, we can see that our model performs better with the Rock class.

- F1-score class Experimental: 0.74%
- F1-score class Rock: 0.81%

Interpreting the recall metrics, we saw that our model was slightly better at correctly identifying records with Rock genre than records with Experimental genre. Indeed 83% of records with Rock label were correctly identified against the 73% recall of class Experimental. Furthermore, the model provided a higher precision percentage for the Rock class (80% against 76%), telling us that out of all the labels classified as Rock, 80% were correct. Finally, for understanding how the model behaves under different acceptance thresholds, we plotted the ROC curve in Figure 3. From the ROC curve, we can see that the model performs slightly better with class Rock at higher threshold level and performs slightly better with class Experimental at lower threshold level.

Our second classifier is the K-Nearest Neighbour. In order to make results comparable, we used the same independent features previously obtained from ANOVA. We first standardardized the input variables as KNN is distance based and could be offset by variables with different spread. After analysing the validation curve with various numbers of neighbours, we settled on 12 neighbours (n = 12). The results obtained were relatively better than the Decision Tree ones.
Cross validated metrics were as follows:

- Accuracy: 82.0 % (+/- 0.04)
- Macro avg. F1 - score: 80.1 % (+/- 0.04)

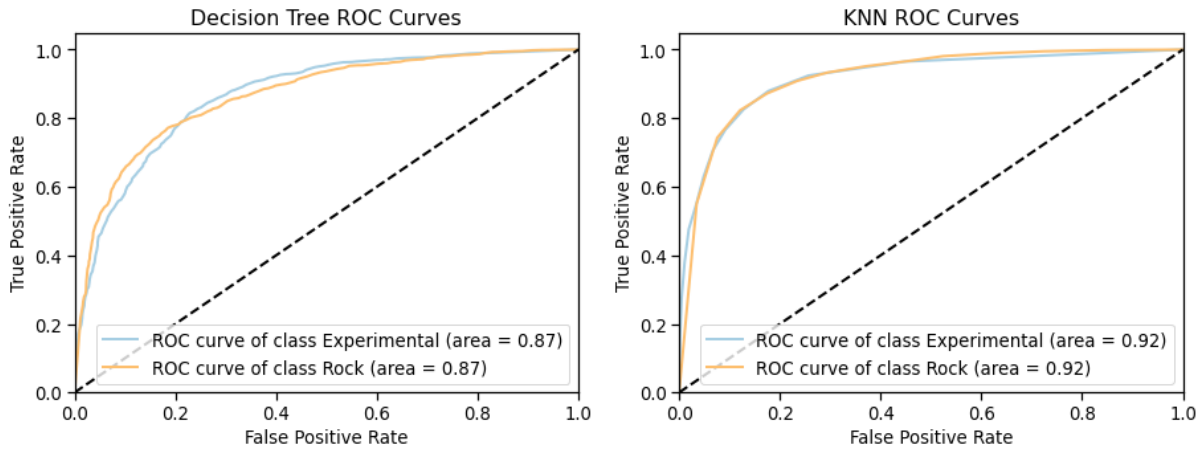
Fitting the model on a stratified sample, we obtained both a test accuracy and macro F1 of 83%. As with the Decision Tree classifier, the confusion matrix showed us that the model performs significantly better when dealing with Rock labelled records.

- F1-score class Experimental: 0.79%
- F1-score class Rock: 0.87%

With KNN, the differences between precision and recall for both classes were less accentuated. The classifier correctly identified 91% of Rock labeled records against the 71% recall of “Experimental” class. However, the model had a higher precision when classifying class Experimental, indeed the precision of Experimental class was 89% against 81% of Rock class.

Overall, comparing the results obtained from the Decision Tree classifier and the KNN one, and leaving aside the different in performance between the two classes, we saw that the KNN was a relatively more performant option, a result also confirmed by the AUC ROC (area under roc curve) calculated for both methods, 0.87 for the Decision Tree against 0.92 for KNN.

Figure 3: ROC curves for Decision Tree and KNN classifiers on balanced dataset.

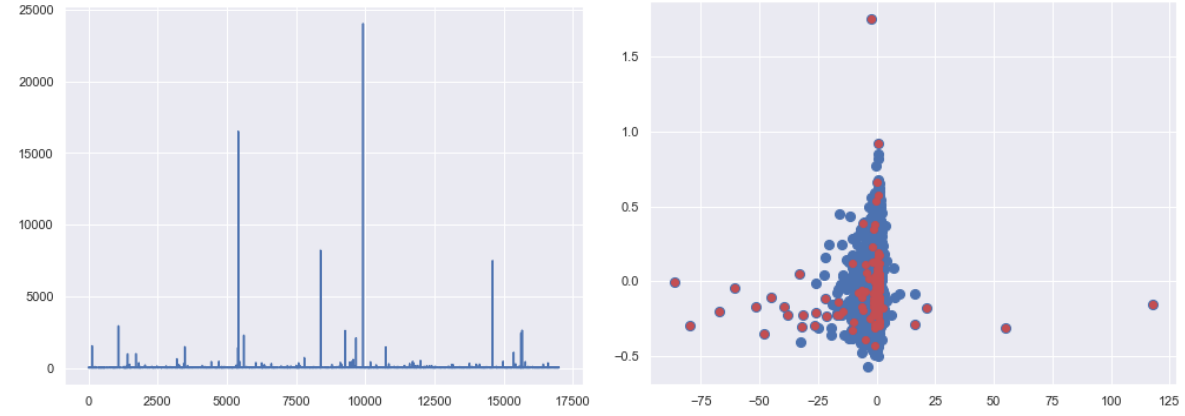


1.3 Outlier detection

For the Distance-based approach we choose to implement KNN and detect outliers based on their distance from other points.

The further a point is from its k-th neighbor, the more likely it is an outlier. We use two methods, the classic KNN and KNN from PyOD library that let us choose the percentage of outliers we want to detect.Since this method is based on measuring the distances between points, we should normalize the data frame. However, in presence of outliers the StandardScaler does not guarantee balanced feature scales, due to the influence of the outliers while computing the empirical mean and standard deviation that leads to the shrinkage in the range of the feature values. Also MinMaxScaler scales all the data features in the range [0, 1] or else in the range [-1, 1] if there are negative values in the dataset. This scaling compresses all the inliers in the narrow range [0, 0.005]. For this reason we choose to normalize the dataset with the RobustScaler method. This approach is based on percentiles and therefore is not influenced by a small number of very large marginal outliers.

Figure 4: KKN distance based approach outlier detection



The classic KNN method returns us the distances and the indexes of each point and by plotting the means of the k-distances, as shown in Figure 4, we visually choose 77 as the threshold in order to

select about the 1% outliers. In fact, this threshold lets us locate 170 rows, representing around 1% of the data. The outliers have been highlighted in red color on the scatter plot above.
In order to confirm the results obtained with KNN, we proceeded with the same operation using the PyOD KNN, that given a certain number of contamination equal to 0.01 provided very similar results. We also created a Venn diagram to visually illustrate the relation between the sets of outliers that these two different algorithms find out. As the reader can see in Figure 5, the sets outliers are very close together with over 150 common outliers out of the 189 combined outliers.

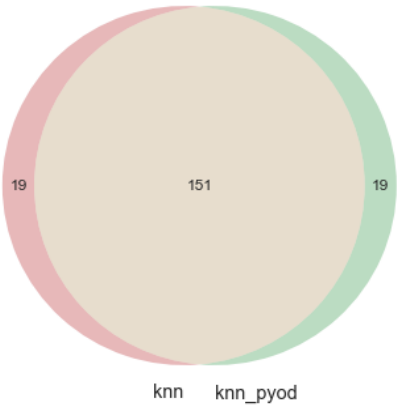
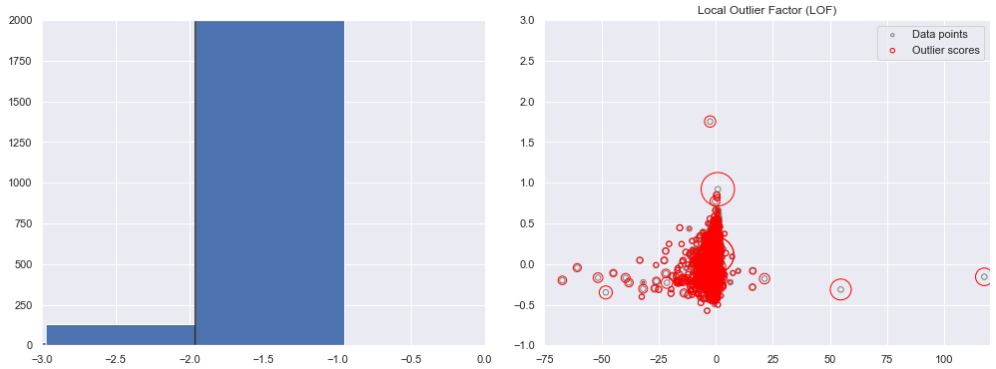


Figure 5: Venn diagram for KNN and KKN_pyod outliers

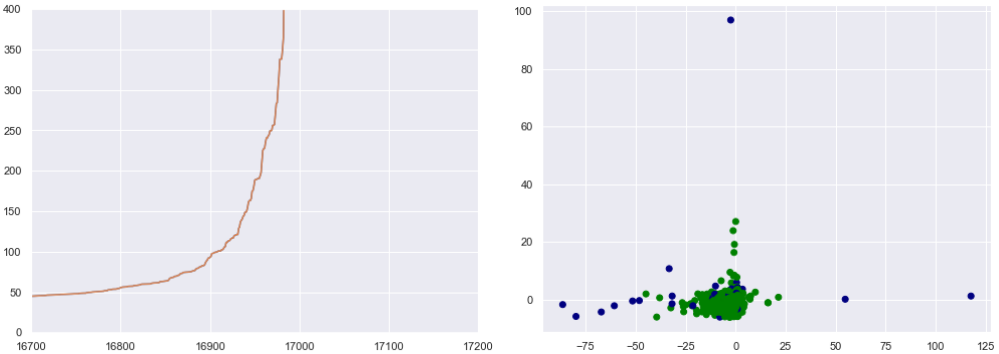
For the Density-based approach we choose to implement LOF and detect outliers based on their density of a point compared to the density of their neighbors. Also in this case we tried two algorithms, first we used the Sklearn Local Outlier Factor with contamination values set to 0.01 and n_neighbors = 10. The identified outliers were outlined in red Figure 6. We once again repeated the procedure with Pyod and obtained the same results.

Figure 6: Density approach outlier detection with LOF



For the Cluster-based approach we choose to implement DBSCAN and detect outliers by dividing the points in clusters and detect the points that do not belong to any of the clusters.
In order to choose eps properly we calculate the distance from each point to its n closest neighbour using the NearestNeighbors, and then sorting and plotting the results. The optimal value for epsilon will be found at the point of maximum curvature, which as we can see in Figure 7, is where mean eps is equal to 130.

Figure 7: Cluster based approach outlier detection with DBSCAN



We further also checked outliers with the High Dimensional approach so as to make a general graphical cross comparison (Figure **δ** in appendix). We choose to implement ABOD and detect outliers by analyzing the location of a point's neighbours. When looking at the cross comparisons, we can see that overall our outlier detection is quite successful. For the most part, the different methods identified the same as outliers.

We eventually treated the outliers found with the various methods and proceeded to remove them from the dataset. Indeed, as we have over 100'000 records and over 500 features, we deemed acceptable to drop the few hundred records considered outliers.

1.4 Imbalanced classification

The class distribution of the target variable used in the classification task for point 1.2 was fairly balanced, with 14'000 records labelled as Rock and 10'300 labelled as Experimental (respectively 57% and 43%). As requested, we transformed the dataset into an imbalanced version.

Our objective was to create a 96% class Rock to 4% class Experimental imbalance in the target variable. Starting from the dataset used in point 1.2, we randomly selected around 9750 records labelled as Experimental and created a new copy of the data dropping those records. Our new dataset had the following target variable distribution:

- class Rock: 14'000
- class Experimental: 614

The new distribution is imbalanced with class Experimental being represented in only 4% of the records. To solve the classification task we therefore adopted some common imbalanced learning techniques.

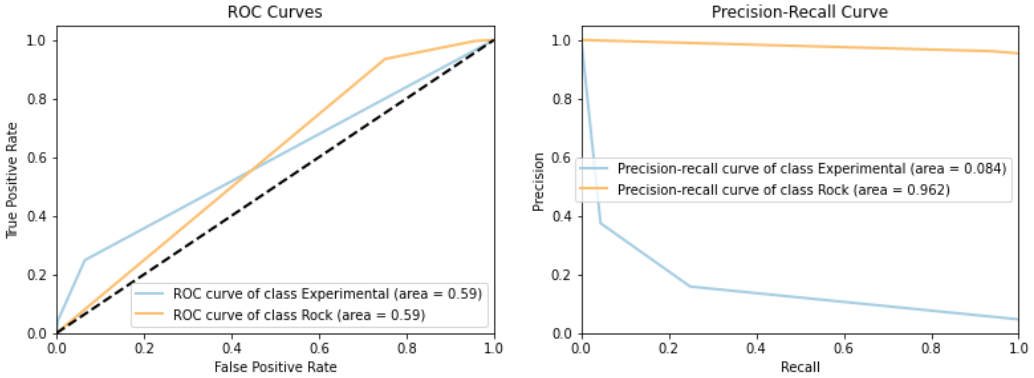
We started by building a KNN classifier on the raw imbalanced data to use as a benchmark for future comparisons. For ease of comparison, we used the same explanatory variables used in the classification tasks in point 1.2. As for the hyper parameter tuning, we noticed that by relaxing the number of neighbours of the classifier, we obtained better results; indeed, by letting our model expand more we increased its chance of finding the imbalanced class.

As expected, the cross validated results obtained appeared good. Compared to the results on a balanced dataset previously obtained, we got a much higher accuracy (92% against 76%) and a relatively smaller macro F1-score (75% against 60%).

- Accuracy: 92.8% (+/- 0.03)
- Macro F1-score: 60.0% (+/- 0.02)

Looking at these overall global metrics, one could be tricked into thinking that the model actually did perform well. However; when dealing with imbalanced classes, the overall metrics are not particularly informative. Indeed, the first thing we noticed is that a classifier always classifying as the majority class would be more performant than ours (96% against 92.8%). Secondly, when looking at the metrics of each individual class obtained from the confusion matrix, we can see that our model is indeed quite good at classifying instances with class Rock: it correctly recognizes 94% of Rock records and out of the ones it classifies as Rock, it has correctly classified 96% of them (F1-score class Rock: 0.95). However, the model performs relatively poorly when dealing with class Experimental, as it correctly recognizes only 25% of Experimental records and has a 16% precision (F1-score class Experimental: 0.19). Such figures appear poor when compared to the majority class; however, are in truth acceptable when considering the degree of class imbalance. As shown in Figure 8, the poor metrics of class Experimental are reflected in the low area under Precision-Recall value (0.084).

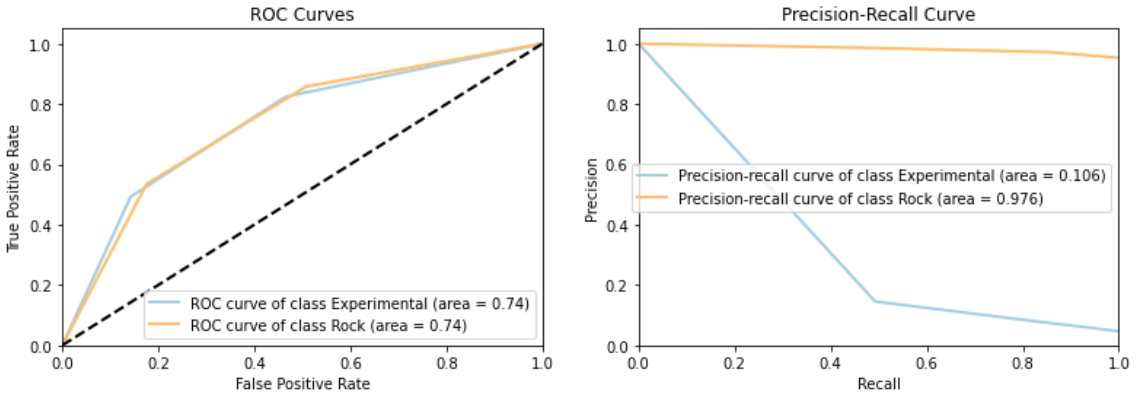
Figure 8: Unbalanced classification results with unbalanced training



Our first approach to imbalanced learning was to act on the classifier. We tried to improve the performance of the classifier by using class weights. After multiple trials, we noticed that by pushing the classes' weight to 10 to 1 trying to balance the classes, we obtained a small but not significant improvement for the precision and recall of the underrepresented class.

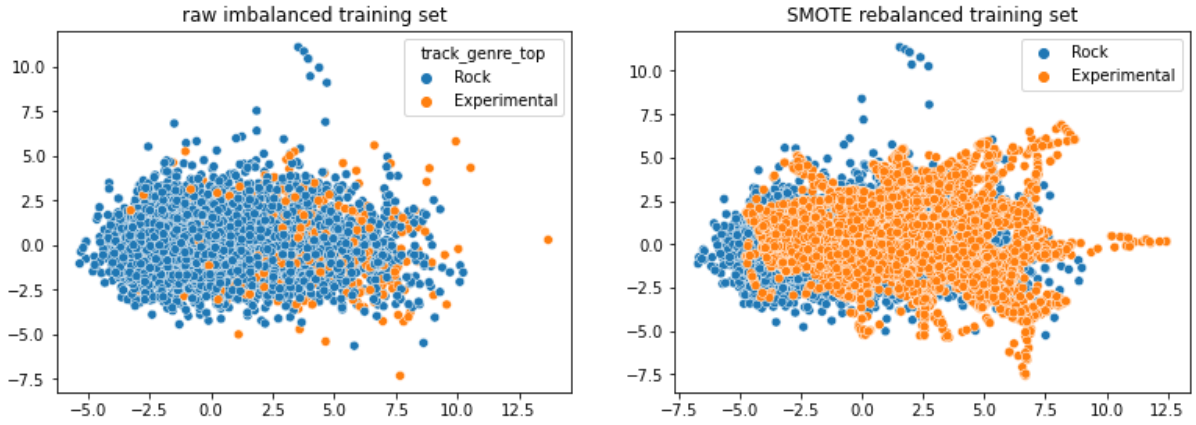
Our second approach was to act on the data. We first tried undersampling the majority class of the training set both randomly and with Condensed Nearest Neighbour (CNN). In both cases we of course only resampled the training set and left the test set untouched. The classifier was therefore trained on a sample with balanced classes. We did obtain an overall improvement with both methods, with CNN providing slightly better results but also taking far longer to run on all the data.
Test accuracy (54%) and both F1 scores (class Rock: 0.69; class Experimental: 0.15) were much lower when trained on balanced sub samples than when trained on the original imbalanced data set.
However, when looking at the AUC ROC and AUC PR in Figure 9, we can see that in reality our model actually performed better than depicted by the accuracy and f1 scores. AUC ROC improved from 0.59 to 0.74 and AUC PR of class Experimental improved from0.084 to 0.106.

Figure 9: Results of classifier trained of CNN rebalanced test set



As our dataset was not particularly large (~15'000 records), our third approach was to oversample the data rather than undersampling it.
We therefore tried both random oversampling (taking into consideration the random selection bias) and SMOTE oversampling. As the reader can see in Figure 10, we can see the effect of SMOTE generating synthetic points in between Experimental features.

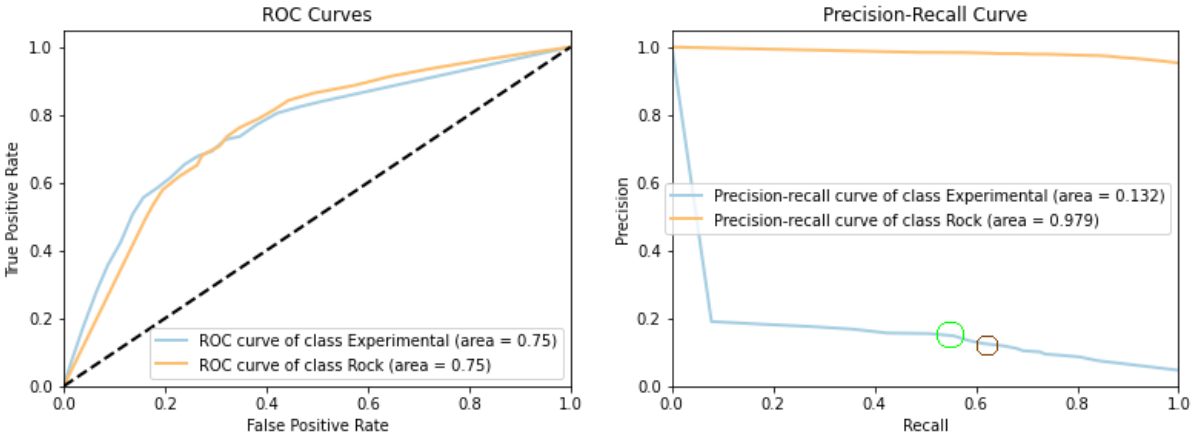
Figure 10: rebalancing training set with SMOTE (PCA visualization)



We trained the classifier on the rebalanced dataset and then tested it on the imbalanced test set. The results were better both than the benchmark and than the classifier trained on the CNN undersampled set.

Similarly to the results obtained with the CNN resampled set, accuracy and f1 scores lowered compared to the benchmark. Test accuracy was 75%, classes Rock and Experimental F1 score were respectively 0.85 and 0.19. As mentioned in the CNN analysis, the confusion matrix metrics are not informative. Indeed, as shown in Figure 11, the AUC ROC and AUC PR depict another version of the story. From these metrics we can see that our model performed better than the benchmark trained on imbalanced data. Furthermore, since the data is imbalanced and we care more about the underrepresented class Experimental, we can use the AUC PR of class Experimental to deduct that the classifier trained on SMOTE data performed better than the one trained on CNN (AUC PR SMOTE: 0.131 vs AUC PR CNN: 0.106).

Figure 11: Roc and PR curve for SMOTE rebalanced classification.



Finally, our final touch was to adjust the threshold. Using the results obtained with the SMOTE resampled training set. As mentioned above, the confusion matrix results gave us f1 scores for classes Rock and Experimental of respectively 0.85 and 0.19.

More precisely, metrics of class Experimental were as follows: Precision: 12%; Recall: 61% (highlighted in a brown circle on Figure 11).

We noticed a small peak on the PR curve at a slightly higher threshold (highlighted in a green circle) and we proceeded to raise the threshold to 0.7 in order to obtain such a level of performance in our

test confusion matrix. The result was that we improved the f1 score of class Experimental from 0.19 to 0.24. whilst also improving the class Rock f1 score from 0.85 to 0.92.

1.5 Conclusions

Applying imbalanced learning techniques to improve a classification's results turned out to be a very instructive task. We learned how the confusion matrix results are to be interpreted in situations where we have high class imbalances and how the traditional overall metrics can be misleading if used as the sole discriminant of a model's performance.

As the reader might have also noticed in our model comparison in point 1.3, traditional overall classification metrics such as accuracy and macro f1 scores do not convey meaningful information in imbalanced classification. On its own, a very high accuracy on a very imbalanced classification does not guarantee a good model performance as a very high accuracy could also be obtained by a dummy classifier always guessing the majority class. The overall metrics which are quite informative in a normally balanced classification, such as accuracy and AUC ROC, are considering the performance of both classes together. In an imbalance problem, the classifier is usually extremely talented in classifying the majority class and proportionally bad in classifying the minority. Such results get averaged out in the overall metrics and might missguide the model creator. For example, this can be seen in the case of the AUC ROC, where recall is plotted against the false positive rate (FPR). FPR stays low if the model is very good at classifying the opposite class thus showing the reader an acceptable ROC curve for the underrepresented class when in reality the curve is primarily supported by the capacity of the classifier to classify the majority class. This is not to say that these overall metrics are to be ignored all together, however the model creator should have well in mind that, especially in situations where we are more interested in predicting the minority class than the majority, the overall metrics alone can be misleading and other more informative metrics are to be considered to get a proper idea of our model's capabilities. As the reader can see in the previous point 1.3, when comparing classification models, we used the AUC PR (area under precision-recall curve) as the main discriminant to decide which model was better. We did this because we were more interested in a classifier that would perform well on the minority class. By plotting the precision and recall values for various threshold levels, AUC PR of class Experimental gave us an unbiased picture of the classifiers capacity to recognize Experimental records.

Another interesting aspect of the PR curve is that, similarly to the ROC curve in balanced learning, it provides an idea of the global performance of the classifier for every threshold level. In imbalanced learning where we are mostly interested in the minority class, having an overall idea of the classifier for various thresholds is particularly valuable as we can freely modify the threshold for advantaging the minority class without having to worry too much about the majority class as it tends to performance strongly for almost any threshold (contrary to a balanced classification task where we care about all classes and modifying the threshold in favor of one might badly affect the performance of another class). As the reader can see in Figure 11, due to the class imbalance, the PR curve of the majority is so high that we can adjust the threshold in order to maximize the f1 score of our minority class without having to worry about the majority class. The reasoning was applied in the final part of point 1.4, where we managed to improve the minority class results by analyzing the PR curve and shifting along the threshold to obtain our desired performance.

Module 2

2.1 & 2.2 Advanced classification methods

The first classification task we performed was with the Naive Bayes classifier. The Naive Bayes is a statistical framework for classifying records based on the Bayes theorem. The classifier selects the class that provides the highest posterior probability for a record belonging to a certain class given its set of attributes. We first selected only features that were perfectly independent to one-another so as to satisfy the Naive Bayes' naivety assumptions. Secondly, we also made sure that the continuous input features were normally distributed, as Bayes uses probability density estimation to calculate probabilities of continuous attributes.

The classifier itself was quite easy to train as it did not require any hyper parameter input. We increasingly added input variables to the model to see how it was reacting to larger features set, especially considering its assumption about independent features. With a training set of more than 200 features, we obtained strong cross-validated results.

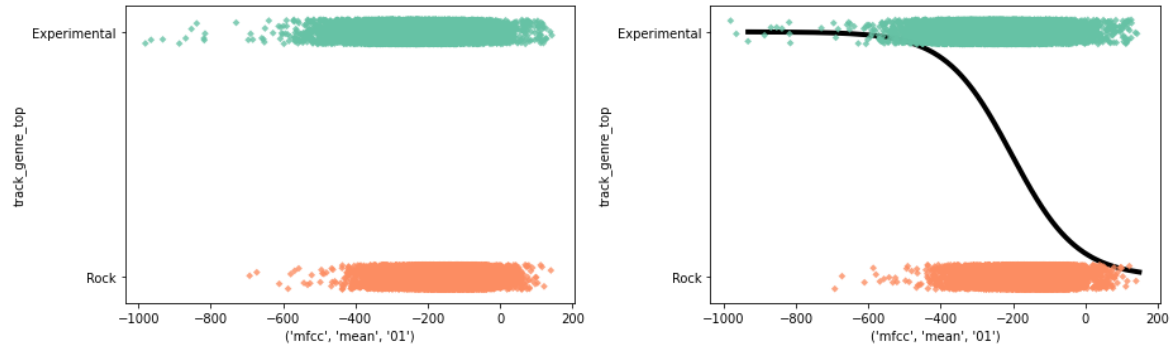
- Test Accuracy: 79,41 % (+/- 0.05)
- Macro avg. F1 - score: 78,72 % (+/- 0.05)
- AUC ROC: 0.87

Those results were as good as the ones obtained with the tuned decision tree classifier after hyper-parameter tuning (reported in Module 1.2). We were positively surprised both by the speed at which Bayes is capable of learning and by the performance it achieved. Indeed, despite its core assumptions, Bayes was able to handle multicollinearity in the data it was fed.

The second family of classifiers we analysed was the regression classifier. The linear regression could not be used for our categorical classification task and is reported later in this section for another task. We thus proceeded with the logistic regression, which after logit transforming the Y estimator function, results in a sigmoid function modelling the data from which is interpretable as a probability outcome. We can thus use these probability as a threshold and classify our records.

For better understanding and visualization, we first proceeded with a simple logistic regression. We selected the most discriminant dependent variable (the median of mfcc's coefficient 1) and fitted the model. Similarly to the Bayes one, this classifier has no main hyper parameter to tune. The model obtained a 70% accuracy and similar F1-score. Looking at the respective class accuracy, we saw that only class Rock had a strong performance (F1: 0.80) whilst class Experimental had a quite poor performance (F1: 0.60). This can be understood in Figure 12 where we can neatly see the sigmoid function resulting from the logit transformation. We can also see that a single attribute is not enough to make the problem easily separable by the sigmoid curve, and in this case it is unfairly advantaging class Rock, explaining the discrepancy between class performance.

Figure 12: Visual representation of simple logistic regression on training set with single attribute.



We increased the number of independent variables to 200 and proceeded with a multiple logistic regression. The accuracy drastically improved. Both the L1 and L2 regularized regressors achieved identical impressive cross-validated accuracy:

- Test Accuracy: 84 % (+/- 0.10)
- Macro avg. F1 - score: 85 % (+/- 0.10)
- AUC ROC: 0.93

We further compared the coefficients of the regularized versus the non regularized regressors. As expected, with more independent features, the regularizations had a significant impact on the coefficients. Almost all coefficients were significantly decreased in the regularized settings with the L1 regularization showing the most effect and even pushing around ten coefficients to zero. Indeed, as the number of input features was increased, both the complexity and the chance of multicollinearity increased and thus significant effects of regularization were visible.

The third type of classification method part of our analysis is the Rule-based classification. Rule-based are similar to decision trees but they are not hierarchical, instead they are based on a set of rules that map every record to the class upon which those were generated. We used the Ripper algorithm from the Wittgenstein[1] library to generate the rules set.

As we had a binary classification, we only needed to grow the rules for one class. Therefore, the first choice was on which class to build the classifier. We tried both and chose the one which resulted in the best performance (class Experimental). We also noticed that the Rule-based classifier performs particularly worse than the other classifiers with few input features and we thus gradually increased the number of independent features in order to have results in line with the other classifiers. We noticed that, apart from being quite slow to learn the model, RIPPER also crashes with more than 200 features. The first application with the out-of-box setting on 100 features gave us relatively good results:

- Accuracy: 76,41 %; Macro avg.
- F1 - score: 76,72 %; AUC ROC: 0.85

The RIPPER classifier has few main hyper parameters that include the number of final optimization runs once all rules have been made, the proportion of training data left for pruning a rule once its fully grown and the maximum complexity threshold for the Description Length. Although testing parameters on this algorithm was particularly time and computationally consuming, we tried different setups to analyse the dynamics inside the algorithm. By playing with the maximum Description Length we saw that it impacted the training accuracy and that we achieved slightly better results by dropping the Description Length allowance from the default 64 to 34. Indeed by lowering the maximum threshold accepted, the algorithm stops growing rules earlier and we can improve generalization to a certain extent.

We finally tried to proceed to tune the hyper parameter with randomsearch but eventually settled with the default options and the updated Description Length allowance as the chosen parameters as we were not obtaining significantly better results with other parameters tested. The final results of the RIPPER classifier were: Accuracy: 73,97 %; Macro avg. F1 - score: 74,86 %; AUC ROC: 0,82.

Table 1: Classification result extracted from Rule-based classifier RIPPER

Predicted class	Rule applied for prediction

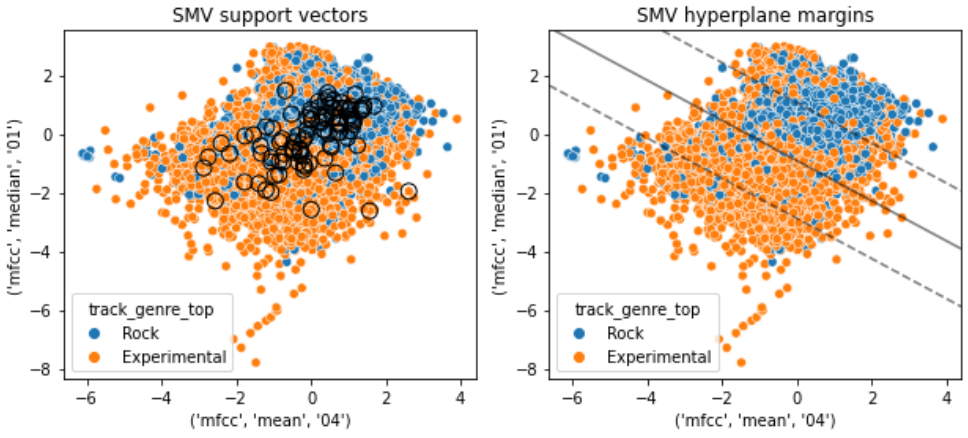
[1] https://github.com/imoscovitz/wittgenstein

True (Experimental)	<Rule ["(mfcc,mean,01)=-324.76--271.08"&"(mfcc,median,06)=<-1.0"]> OR <Rule ["(mfcc,median,06)=<-1.0"&"(mfcc,mean,04)=15.14-21.65"]> OR <Rule ["(mfcc,mean,01)=-324.76--271.08"]>
False (Rock)	[ ]

Looking at Table 1, the reader can understand both the logic of the rule sets grown by RIPPER and the noticeable advantage of the Rule-based classification method. We can see an example of two records classified by the RIPPER classifier. We can see that the first record was classified as Experimental as it was covered by one or more rules. In this case three rules were covering the instance, the first rule states the conjunctions of two conditions: that the record had the mean of mfcc's coefficient 1 between -324 and 271 AND that it had the median of mfcc's coefficient 6 smaller or equal to -1. The reading is the same for the two other rules related to the first record. The second record was instead labelled as Rock as, no rules were covering the record and following the RIPPER's methodology, the record was classified as negative.
Concluding on Ripper, we saw that, similarly to the decision tree, Rule-based classifiers are quite expressive and particularly easy to interpret. Indeed the classifier can tell us the reason behind every classified record in a way understandable even to non technical readers; which in some situations can be a strong advantage.

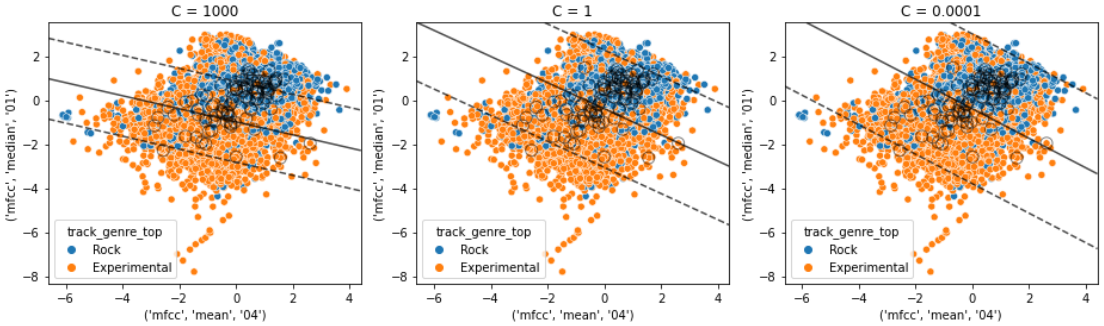
Continuing with our analysis, we proceeded with a Support Vector Machine (SVM) classification task. The model is a maximal margin classifier and will try to find a hyperplane that best separates the class based on the margin between their support vectors. We first selected two independent features (('mfcc','mean','04') and ('mfcc',median,'01')) as they were graphically relatively well separated and we standardized the features as we know that SVM uses Euclidean distances and unstandardized data can unfairly set more weight to certain features. We fitted the out-of-the-box linear SVM to the data and obtained quite good results: 75.2% accuracy and 74% F1-score (Auc ROC was not available as the classifier has no method to predict probability). Indeed the results are slightly higher than the other classifiers analysed so far with out-of-the-box settings, especially considering there are only two input features and that we are trying to solve a nonlinear problem with a linear classifier. As we know from the course theory, for solving this classification, the classifier used slack variables to relax the optimization problem. In Figure 13, we can neatly see both the support vectors and the results separation margins.

Figure 13: Graphical representation of fitting linear SVM model to



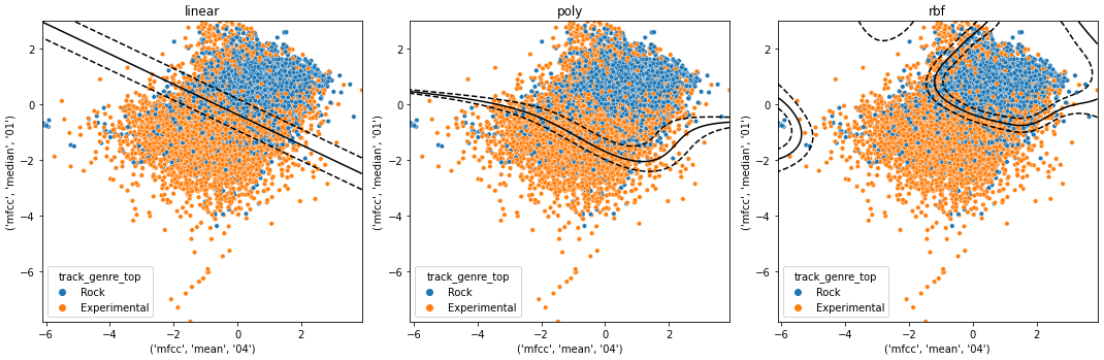
Regarding the hyper-parameter tuning of the linear SVM classifier, we performed various tests with the regularization parameters. We tried various setups, especially with variations in the C parameters and tried to find a good balance between training accuracy and generalization. By setting C to 0.001, we improved the classifier's accuracy to 76.5%. Indeed, the higher the C value, the lower the regularization. Whilst the linear model is not particularly adapted to solve this nonlinear task, we could still see the effect of the regularization with extreme setups. We saw that by drastically increasing the model's regularization, we could slightly reduce the training accuracy and improve the validation accuracy. Vice versa, by drastically reducing the regularization, we allowed the model to overfit and obtained slightly worse validation accuracy compared to the training one. In Figure 14, the reader can see the behaviour of the linear SVM under different regularization settings. Starting from the left on Figure 14, we can see SVM trained with a very laxed regularization, thus margins are relatively close together. Shifting to the right, we see that by increasing the model's regularization, we obtain higher generalization and thus the margins are pushed outwards.

Figure 14: Effect of regularization parameter C on Linear SVM model fitted on training data



We then moved on to solve the same classification problem with SVM with a kernel function so as to better model the non-linear relationship of the two input features. We saw that overall the 'rbf' kernel was the most performant one and the polynomial kernel was the least performant model regardless of the regularization setup. The difference in kernels was particularly noticeable when plotting the separation hyperplanes between the two input features (Figure 15). The linear function has the same shape as with the linear SVM classifier, whilst the polynomial and 'rbf' have their shape that model the data. Graphically, we can see that the 'rbf' kernel provides a hyperplane that best separates the data.

Figure 15: Effect of kernel function on SVM classifier



Although we are still in a simple two features input setting, the SVM with 'rbf' kernel function with C set to 0.1 achieved a 77% accuracy. As we gradually increased the number of input features in the model and we noticed the 'rbf' kernel kept delivering higher performance the more features we put

into the model in contrast to the other kernel which stopped performing after 100 features. With 200 input features, SMV with 'rbf' kernel obtained very strong results:

- Test accuracy: 87,58 % (+/- 0.05)
- Macro avg. F1 - score: 86,94 % (+/- 0.05)

These results show us that the capacity of SVM to handle large data input and thus multicollinearity, like regression and rule-based, derive from the regularization penalties. Optimizing the regularization level allows to find the balance between overfitting and strong regularization.

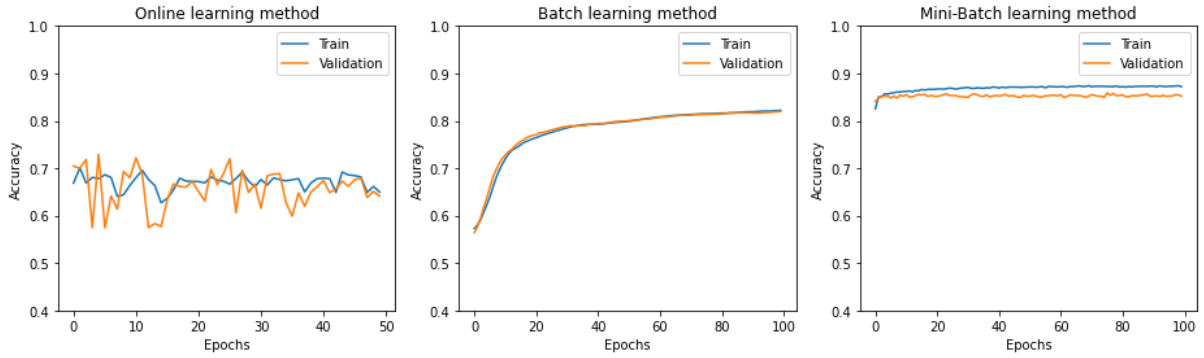
We eventually performed the classification with the Neural Network. Our first step was to standardize our input features so as to speed up the learning process and facilitate convergence. We started by testing various parameters and performance with sklearn's multilayer perceptron classifier; however, as we wanted to be able to select a different activation function for the output node than for the hidden layers, we felt the need to upgrade to a more dedicated library such as Keras.
Using Keras we created a sequential model and gradually added dense hidden layers in order to understand the behaviour of the model and monitorate its performance. Our goal was to create a model large enough to perform well and small enough to avoid overfitting.
We first tested how much performance a linear perceptron could deliver and thus built our first model as a single layer network. We chose a sigmoid activation function to obtain a probability outcome for our binary classification and compiled the model with binary cross entropy loss and ADAM optimizer. Finally, our model was fitted with the default batch size of 36 and 50 epochs. From the loss results, we could see our model converged pretty quickly (ie. it needed few epochs to learn the model) which was expected as our classification problem was quite simple. With only one input feature (the same one selected in the simple logistic regression), the accuracy obtained was around 75%; whilst with 200 features, we obtained a 85% accuracy and 84% F1-score. As we expected, the results are very similar to the ones previously obtained with the simple logistic regression classifier, as indeed the task is almost identical.
We transformed the perceptron into a multi-layer network and started testing the model behaviour to new layers and neurons. As we increased the number of neurons, we immediately realized the high tendency of the neural network to overfit even with a single hidden layer. When comparing the validation accuracy to the training accuracy for every epoch, we realize that, for a neural network with one hidden layer, few neurons will not fit enough whilst adding many neurons will simply overfit without improving validation accuracy. Furthermore, even when adding new hidden layers and nodes, our model did not generalize better as it kept overfitting the training data.
We thus proceeded to use regularization methods to address overfitting. Our goal was to be able to build a complex neural network whilst avoiding overfitting with weights and drop-out regularization hoping to increase the model's generalization. The difficulty we encountered was to set the right equilibrium between the model's capacity (number of layers and nodes) and the strength of the regularization parameters. We proceeded with a systematic experimentation with multiple different layers and regularization setups to understand the generalization behaviour (we used a GPU backend from Google Compute Engine to speed up the process). We further searched for the best level of regularization for the kernel and the percentage of dropout nodes at every layer with gridsearch. We eventually used early stopping to get the best possible model across all the epochs. Below are reported two of the setups providing the best results (both used ADAM optimization):

- Setup 1: NN (40/ 40/ 40/ 1); Relu for hidden layers; sigmoid for final layer; 40% dropout rate:
  - Test accuracy: 89,68 % - Macro F1-score: 0.89

- ❖ Setup 2: NN (100/ 100/ 100/ 1); Relu for hidden layers; sigmoid for final layer; 0.01 kernel regularizer:
    - Test accuracy: 88,63 % - Macro F1-score: 0.88

The setups were the default 36 batch size, thus using the so-called mini-batch learning mode, where in every epoch, the model updates its weights after estimating the error with every 36 records. We chose this setting as it was offering the best performance; however, while proceeding with our experiments, we also gained some valuable insights on the relationship between batch size and the way the model learns. To better illustrate our thought, we provided some graphical representation of the effect of the learning mode on the network's dynamics. We considered a simple neural network optimized with a gradient descent algorithm with one hidden layer and three nodes. We trained the model with the different learning methods: Batch, Stochastic/Online and Mini-batch.

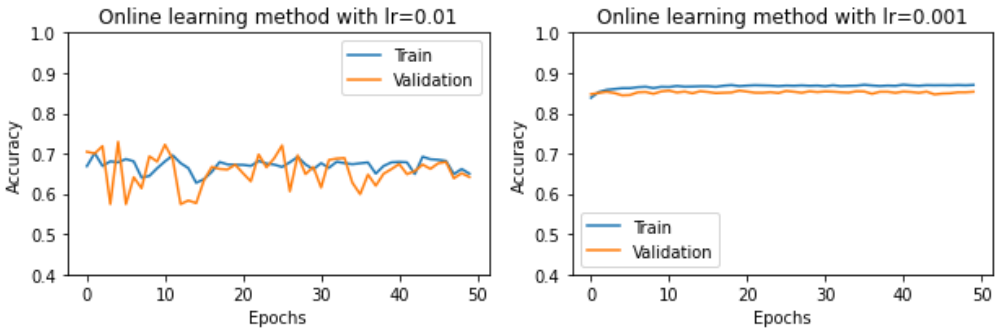
Figure 16: NN trained with three different learning method



From the accuracy curves in Figure 16, we can see how well and how quickly they learned the problem and how noisy the updates were. As expected, the Mini-batch learning algorithm provided the best results. We can see that the model both learns fast and stabilizes fast. When comparing it to the other learning methods we can see that the Mini-batch offers a sort of compromise between the stability of the Batch method and relatively fast learning of the Online method. Analysing the Batch learning method, we can see a very stable but slow training. Indeed, it updated its weight on all the trained instances only once for every epoch and thus took many more epochs (around 40) to converge. Moving on the Stochastic gradient descent, we can see that changes to the model are extremely noisy, resulting in a much worse performance than the Batch mode. In fact, the algorithm updates the weights after every single training instance in every epoch and as every weight update is done on the error obtained by a single record classified, the updates are extremely volatile.

Interestingly, we tried to strongly reduce the learning rate of the online learning network (without changing any other parameter) and obtained impressively strong results. Indeed, by reducing the learning rate, we reduced the amount of weight update and thus stabilized the volatility previously mentioned. As shown in Figure 17, the validation accuracy increases to around 85% which is inline with the mini-batch results.

Figure 17: Comparison of NN with online learning with different learning rate of SGB optimization algorithm



Concluding on NN parameter tuning, we saw that Batch size is directly related to the speed and stability of the learning process. We discovered that although it is extremely time and computationally consuming, the Online learning method with a controlled learning rate can provide good results. We proved that the Mini-batch method was probably the best option as a good compromise between the advantages of the Batch and online algorithms, especially considering its computational ease compared to the Online method.

The last classifier we analysed is the Ensemble method. Ensemble method classifiers method consist in training multiple classifiers (called estimators) on resamplings of the training data and leveraging on the principle of the wisdom of the crowd to obtain the "wisest" answer. In Ensemble, every estimator is trained on a new sample (with replacement) of the training data. Two main ways of sampling the data are proposed in Ensemble learning: "bagging" and "boosting". We analysed both methods with their respective most used algorithms. The first one tested was the Random Forest (RF) classifier which is based on the bagging technique. The algorithm randomly resamples the records and trains each decision tree on a new subselection of the features. We proceeded to systematically experiment multiple different setups of the classifier in order to understand its behaviour and reported the results of the performance with different hyper parameters in Table 2.
We first tested the RF with 1 decision tree (estimator) and as with the decision tree analysed earlier, we obtained a model that overfits the training data and generalizes poorly. We compared the results of the RF with various levels of regularization on different estimator setups. We noticed that with no regularization (out-of-the-box settings), the RF easely tended to overfit. However, interestingly as the number of estimators grew, the model was increasingly fitting the training data (reaching 100%), yet it was also impressively capable of generalizing quite well on the test set (reaching 85% accuracy and 0.93 Auc ROC). From this first example, we understand that the aggregation of many fully developed decision trees, that would each individually overfit the training data and generalize poorly, has the positive effect of transforming the overfitting into a strong learning with strong generalization. Indeed, we can appreciate the power of this aggregation by noticing that out-of-the-box RF with 50 estimators achieved a nine percentage points increase compared to the out-of-the-box decision tree classifier. We also tested the model on a more stringent regularization (max depth = 10) and noticed less overfit for few estimators but also less capacity to generalize with higher numbers of estimators. We eventually searched for the best hyper parameters with grid search and obtained very similar parameters to the minimum settings, with no significant differences in performance compared to the non-regularized version.

Table2: Accuracy results of systematic experimentation with Random Forest (R.F.) classifier

	R.F. (no regularization)	R.F. (max depth = 10)	R.F. (grid_search best)*
1 estimator	train accuracy: 90% test accuracy: 74%	train accuracy: 84% test accuracy: 77%	train accuracy: 82% test accuracy: 76%
10 estimators	train accuracy: 98% test accuracy: 83%	train accuracy: 91% test accuracy: 82%	train accuracy: 90% test: 84%
50 estimators	train accuracy: 100% test accuracy: 85% Auc ROC: 0.931	train accuracy: 93% test accuracy: 84% Auc ROC: 0.922	train accuracy: 92% test accuracy: 85% Auc ROC: 0.926

*best parameters found with Grid Search: min_sample_split = 10 and min_samples_leaf = 11)

It is also worth mentioning that, as it is built on decision tree classifiers, we can leverage many of the advantages of the single decision tree classification. Namely, simplicity, feature importance and explainability. We extracted and looked at the decision trees created by RF to see which were the main features it used for the splits. We also extracted the list of the most used features across all the estimators and plotted in Figure **β** (see appendix) the distribution of their usage in box plots for the main most used features.

Moving on to the other main Ensemble classifier method, we analysed the 'AdaBoost' (AB) classifier which is based on the "boosting" method. Boosting and bagging are quite alike, with the difference that in boosting, for each new estimator, records are resampled based on their weights and the final decision of the classification is not democratic but rather based on the importance of each estimator derived from the estimator's accuracy.

Understanding the key characteristic of boosting also helps understanding why the default settings of out-of-the-box AdaBoost uses stumps (decision tree with max_depth = 1) instead of fully grown trees that proved to be the most accurate learners with bagging. As we can see from Table 3, AdaBoost performs significantly better than Random Forest on the same setup. This can be explained by the effect of boosting on the learning process. In every iteration, boosting updates the weights of the instances so as to increase the probability of picking misclassified instances (and vice versa for well classified ones) in the next sampling for the next estimator. This enables AdaBoost to use a simple but efficient classifier (ie. the stump) and feed new misclassified instances to each new estimator thus enabling the stumps to split on potentially new features and as a consequence improving the learning of the whole classifier. On the opposite, bagging does not enable learning with a too simplistic base classifier as the new estimators keeps receiving instances it has already correctly classified (among the randomly selected instances of the bootstrap)

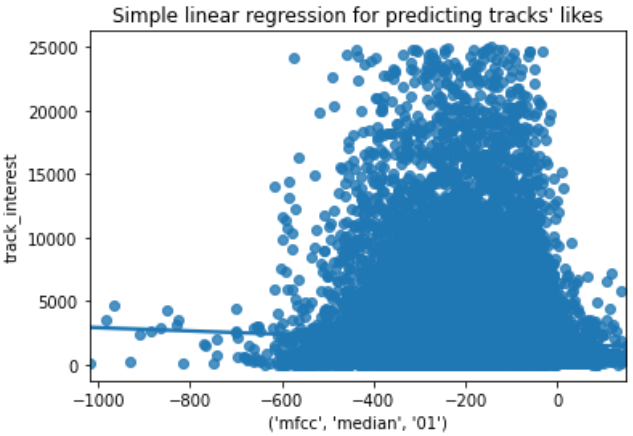
Table 3: AdaBoost and Random Forest comparison on stumps setup

	AdaBoost (max_depth = 1)	Random Forest (max_depth = 1)
50 estimators	train accuracy: 85% test accuracy: 84% Auc ROC: 0.911	train accuracy: 74% test accuracy: 74% Auc ROC: 0.850

We also tried alternatives combinations of bagging and boosting with other classifiers such KNN, SVM and Neural Networks. However we did not obtain any results better than the ones previously reported. Furthermore, when trying the Ensemble methods with other classifiers, we also lost part of

the advantage of using Random Forest and AdaBoost, which is explainability and ease of implementation. Indeed, compared to other classifiers, both Ensembles work particularly well out-of-the-box and do not require standardization as they are based on decision trees.

Our experience in training Random Forest and AdaBoost was very positive. Indeed, we think these Ensemble methods provide the best compromise between the distinctive characteristics of the classifiers we discussed above.
Bayes was effortless to fit as it has no hyperparameter and despite its assumptions of independent data which we thought would penalize the model, actually proved to be a valid light-weight classifier. Similarly, logistic regression was also easy to fit and provided slightly stronger performance than Bayes although being slower to build the model. Rule-based RIPPER was impractical as it was particularly slow both to learn and classify and provided the lowest results of the advanced classifiers; however, it provided very helpful human-understandable reasons behind every classification made. Moving on, SVM did require standardization of the data and some fine tuning between regularization and kernel choice; however, it gave us very strong results and proved efficient on large datasets. Neural Network, it clearly was a step ahead from the other classifiers in terms of performance. Furthermore, considering how much more one could have explored in NN, it felt like it did not belong with the other methods in terms of potential; however, NN did require a much higher level of parameter tuning.
As classification is all about compromises (accuracy vs learning time; regularization vs overfitting; ease of implementation vs performance; …) we believe Ensemble's AdaBoost and Random Forest to be the right compromise as it does not need particular preprocessing (no need to standardize, no need to drop redundant features), it learns and classifies relatively fast, it provide clear explainability of the classification and provide strong classification results.



2.3 Regression

Similarly to the logistic regression explained previously, the linear regression tries to find the best linear hyperplane that best "fits" the data so as to minimize the residual error.
The two features we deemed interesting for a linear regression are:

- ☐ track_interest (number of time track was liked) as dependent variable X
- ☐ 'mfcc', 'mean', '01' (mean of mfcc's coef. 1) as independent variable Y

After fitting the regression model to the data, we obtained the following parameters:

- Coefficient (slope): -1.247
- Intercept (bias): 1639.574

We can also get a glimpse of the regression's performance should the figure not be enough:

- R2: 0.003

Meaning that the independent variable explains 0.3% of the variance of the dependent variable.
Indeed, both from the regression's results and from the figure, we can tell that the features are almost linearly non dependent, meaning that we can't predict a variable with the other one.

We expressly chose two highly uncorrelated variables so as to check whether by adding many more features and using regression techniques able to handle them, we could improve the generalization and find a relationship not visible from a simple regression. In fact, the simple regression remains a linear model (with linear parameters), thus would be unable to find a nonlinear relationship in the data.

We then increased the number of independent variables and proceeded with multiple linear regression and neural networks. We also tried to adapt other methods, such as Naive Bayes, to solve a regression but we had unsuccessful results. For both methods we used 200 features from the features.csv (the same 200 features used for the previous classification tasks).

The Neural Network was tuned to be adapted to its new purpose. We changed the activation function of the output layer to a linear one in order to provide a continuous output. We also changed the loss function used for backpropagation to the mean of square error, once again for matching the numerical task at hand. Finally we fine tuned the model and settled on a (10/10/1) architecture with relu activation function for the hidden layers.

With the multiple regression standardized the data and fitted the model. The results are reported in Table 4. We obtained a slight improvement from before. With the new added features, the model increased its R2 by a relatively large amount (from 0.003 to 0.059) and decreased overall MSE by a couple hundred thousand. Same pattern was shown by the Neural Network, which obtained slightly better results than the multiple linear regression managing to explain around 8% of the variability in the data.

Table 4: Multiple regression results.

model	simple linear regression	multiple linear regression	Neural Network
R2	0.003	0.059	0.078
Cross validated Mean of Squared Error (MSE)	7’977’376	7’525’884	7’163’544

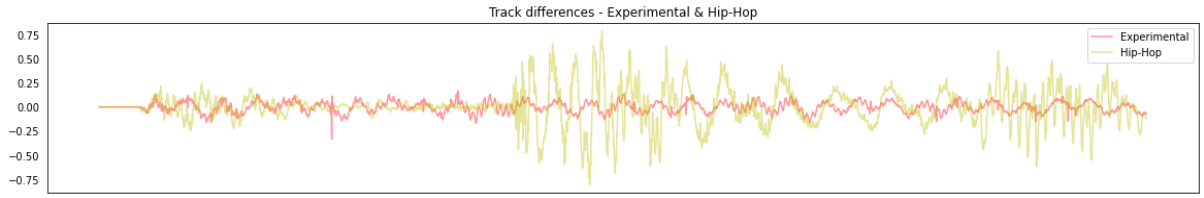
In conclusion, we did not obtain the results we hoped. Although the multiple linear regression and NN showed small improvements from simple linear regression, they did not manage to clearly predict the number of likes on a track.

Module 3

3.1 Time Series

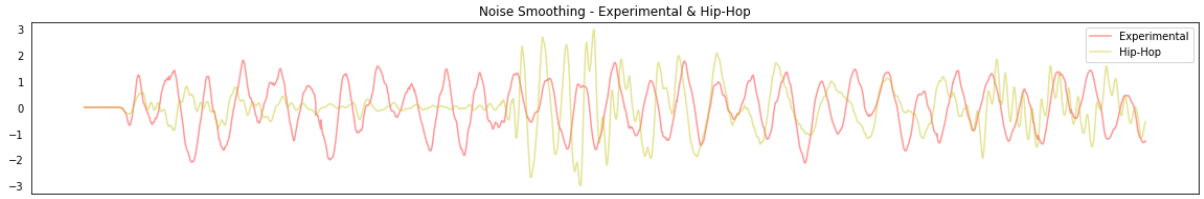
For this module, we decided to create a new dataset of time series. The time series have been extracted from the fma_small with librosa, considered a sampling rate of 8000 and segments of 10 seconds. On the dataset created, we first decided to look at time series with different genres, because we expected that different genres will generate different spectrogram waves; for example, in Figure 18 we can observe differences between an Experimental song wave and a Hip Hop song wave.

Figure 18: the time series of a hip-hop and a experimental song before normalization



In order to normalize the time series, we decided to apply offset translation, amplitude scaling and noise removal (smoothing) with a window of size 20. In Figure 19, we can observe an Experimental song time series and a Hip Hop one, after the transformations: they still have a distinct shape but now they look cleaner than before.

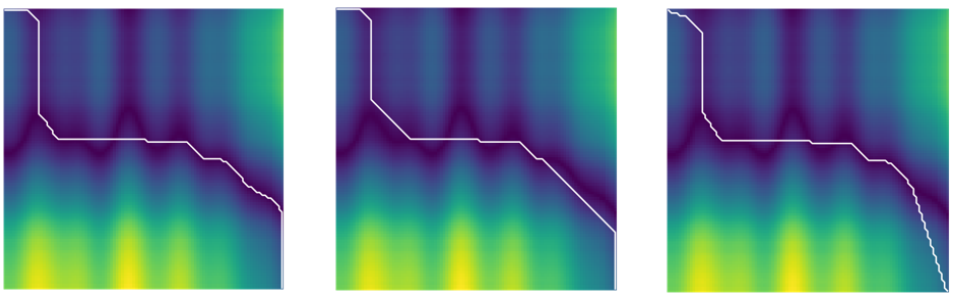
Figure 19: the time series of a hip-hop and a experimental song after normalization



3.2 Clustering

Before applying TimeSeriesKMeans algorithm by tslearn, we make some comparisons between time series. We evaluate the distances between two Hip-Hop songs, between two Experimental songs and between an Experimental song and a Hip-Hop song (different genres). The results obtained, for the different distances, are shown in Table 5. Observing the results, we can see how comparing two songs belonging to different genres (Experimental vs Hip-Hop) we obtain greater distances than the distance obtained by comparing songs belonging to the same genre, as we could have imagined.
In particular, Figure 20 shows the optimal path found by DTW, with and without constraints, as regards the comparison between two Hip-Hop songs.

Figure 20: the path of Dynamic Time Warping between 2 Hip-Hop songs



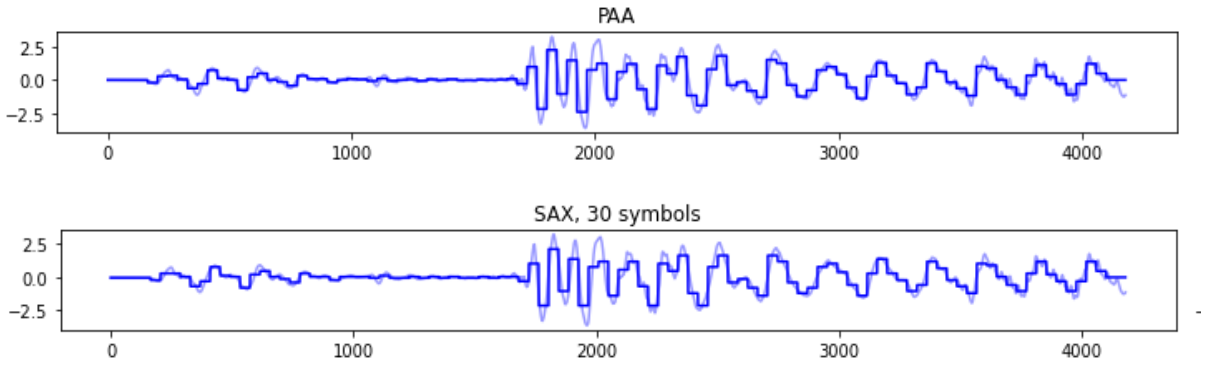
Furthermore, we can observe that using Dynamic Time Warping the difference between the first two distances (calculated over two songs of the same genre) and the third one distance (calculated over two songs of different genre) is proportionally larger. So it can be said that DTW is a better discriminant when it comes to genre.

Table 5: the different distances between 2 instances

	Euclidean	Manhattan	DTW	Sakoe Chiba (radius:20)	Itakura (max_slope:2)
Hip-Hop\Hip-Hop	50.92	2633.79	16.84	29.85	17.86
Experimental\Experimental	50.14	2630.79	22.34	23.20	22.38
Experimental\Hip-Hop	89.25	4487.84	31.74	68.51	33.82

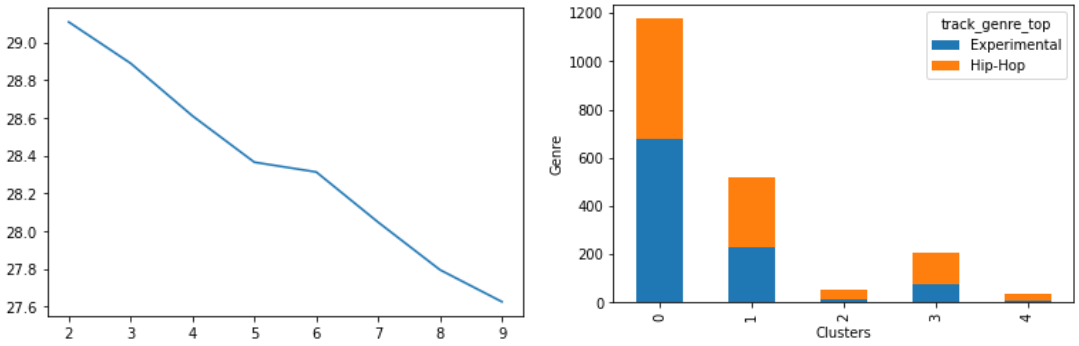
To perform the clustering task, we created a dataset containing only time series belonging to the two genres Hip-Hop and Experimental, with the aim of finding two clusters that differentiate the genres, obtaining a dataset consisting of 1998 timeseries.
Despite this reduction, we soon realized that running any clustering algorithm without having first made an approximation was computationally too expensive. In fact, we tried to perform the TimeSeriesKMeans algorithm with the Dynamic Time Warping metrics but it took us a very long time and it finished only when we reduced the time series to 50. For this reason we approximate the time series of 4180 attributes in 80 segments with Piecewise Aggregate Approximation (PAA). And also with Symbolic Aggregate Approximation we splitted the space in 30 equiprobable parts, which means transforming the time series in a combination of length 80 of 30 different symbols. In Figure 21 we can observe the various approximations.

Figure 21: the approximation of the times series over the original time series



Thanks to the approximation Kmeans is faster and can be applied to the entire dataset of 1998 instances.
Initially, K-Means was performed with metric = Euclidean and n_clusters = 2, hoping to split the dataset into tw cluster representing Experimental and Hip-Hop genres; however, the two clusters that the algorithm actually produced had an equal division of the two genres. We therefore hypothesized the existence of subgenres related to the top genre and we investigated the mean SSE of the clusters as k increases, as shown in Figure 22, but also in this case the results were poor since there is not a value of k capable of decreasing the SSE and choosing k=5 we obtain the 5 clusters.

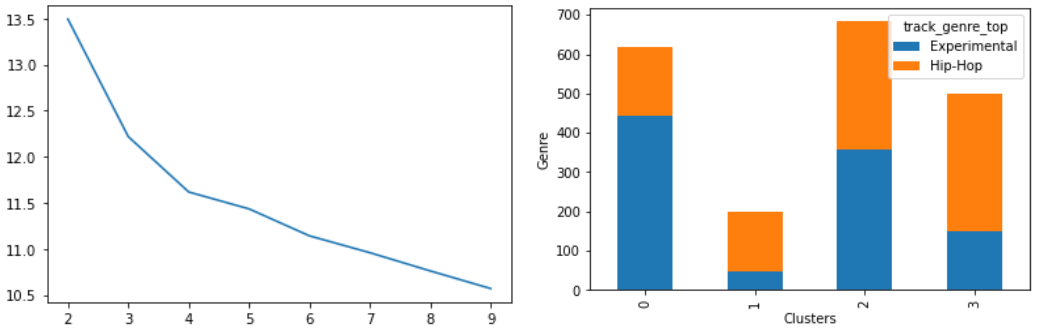
Figure 22: the sse of the clusters as the number of k changes and the distributions of the classes over the clusters find using euclidean distance



The same procedure was performed with metric = DTW and we choose k=4 as the plot in Figure 23 suggested to minimize the SSE. The distribution of clusters that Kmeans produce is shown in Figure 23, where we can see that Cluster0 has a clear majority of Experimental songs, while Cluster3 has a majority of Hip-Hop songs. We can say that the results in terms of SSE are better with DTW than with Euclidean, although the desired results have not been obtained. Furthermore, the results obtained

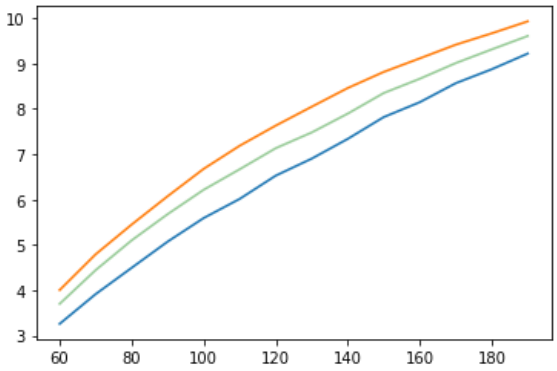
through the SAX approximation are worse than those obtained with PAA, therefore they are not reported.

Figure 23: the sse of the clusters as the number of k changes and the distributions of the classes over the clusters using DTW



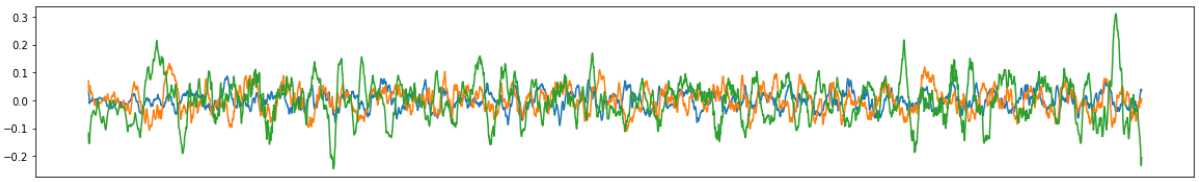
We thought that the uninspiring results were due to the approximation and in an attempt to find a value of n_segments that did not compromise the distances between the time series we found that the average distance calculated with dtw in a dataset composed only of HipHop songs (orange line) is greater than the average distance of a mixed dataset (green line), how we can see in Figure 24. Therefore K-means and also other distance based algorithms are not suitable for the task of discriminating between genres.

Figure 24: the internal mean distance between all the instances of three datasets composed: the orange by only hip-hop song, the blue one by only experimental song and the green one by a mix of the two genres; as the Number of segments of PAA change



Finally, we tried running K-means with a feature-based approach. To perform the feature based clustering, we extract the following statistics: mean, standard deviation, median, various percentiles and kurtosis. Clustering was then performed using the Euclidean distance in the statistics space. As for the previous application of KMeans we investigated and found K = 3 as the optimal value. The centroids obtained are shown in Figure 25.

Figure 25: the centroid founded by the cluster algorithms

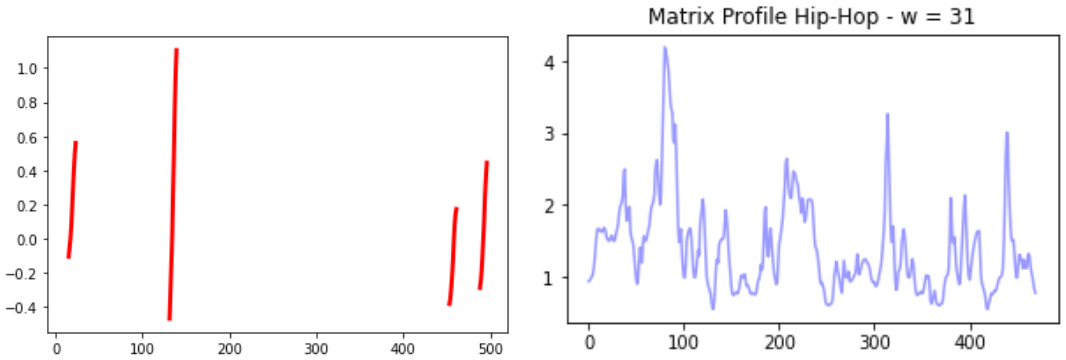


3.3 Motifs

The goal of this task was to discover motifs, which are repeated subsequences in the analysed time series. We decided to analyze an Hip-Hop song, taking just 500 features from the central part of the timeseries, in order to make them more visible. At that point several Matrix Profiles of the time series,

with different time windows, were calculated. In particular, we analyzed the Matrix Profiles with windows $\in$ [5,40]. The motifs obtained from the matrix profiles with low windows values were not that interesting. In Figure 26 we can, in fact, see one of the motifs obtained with window = 9, which shows almost straight lines that correspond to the amplitude of the wave song, which can be found in all the songs. For this reason we finally decided to analyze the motifs obtained from the Matrix Profile built with a window = 31, and we obtained different values close to zero, as shown in Figure 27, which indicates similarity with other subsequences in time series.

Figure 26: Motifs w=9 Figure 27: Motifs w=31



In Figure 28 the motifs obtained are shown: the black (Figure 29, right) and red motifs simply resume repeated patterns of a curved wave, while the green(Figure29, left) and yellow ones show more particular and specific patterns.

Figure 28: Motifs w=31

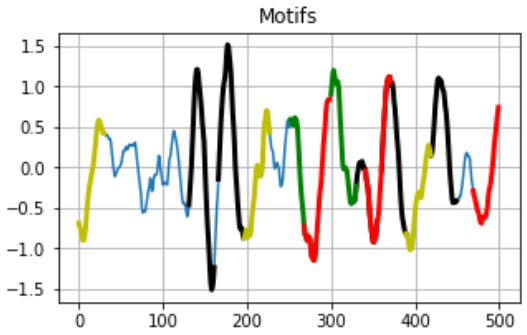
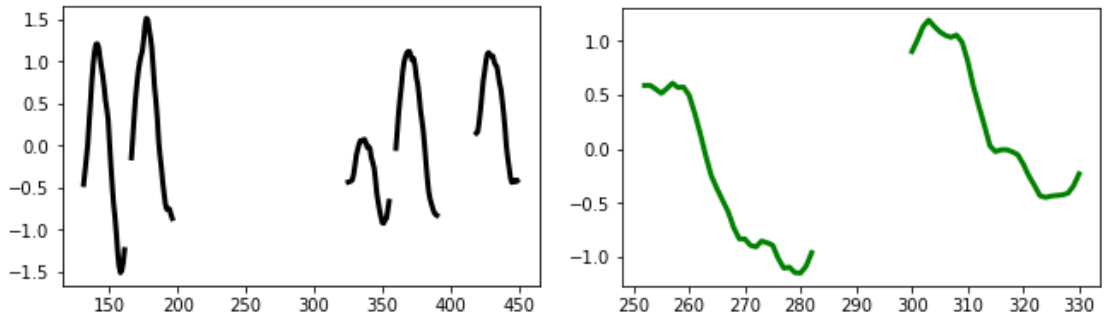


Figure 29: Motifs w=31



On the other hand, Figure 30 shows the anomalies found in the same time series, which could be due to possible incorrect registrations.

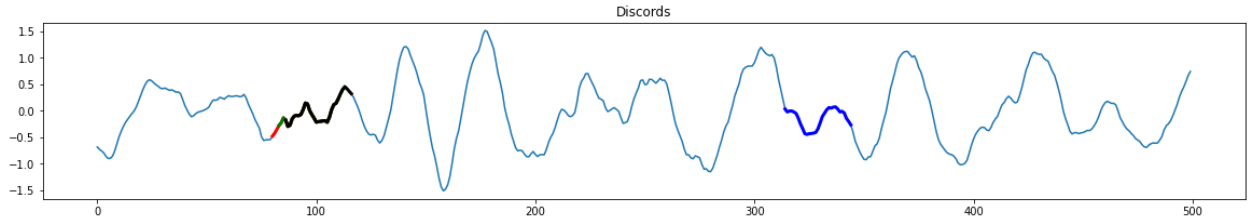


Figure 30: Discords in the time series

3.4 Classification

For the classification task we used two different datasets and compared the results:

1. The data frame of the time series of Experimental and Hip-Hop tracks approximated with PAA in 100 segments in order to be computationally less expensive.
2. A data set obtained by extracting the shapelets from the time series and measuring the distance between each time series and each shapelet.

To extract the shapelets we use the method ShapeletModel by tslearn which uses brute force with some optimization techniques. This method required to know the number of shapelets and their length, to chose this parameters we use Grabocka's euristics "grabocka-params-to-shapelet-size-dict" which taking as input the size of the dataset and the number of classes to be classified, suggests that we search for 5 shapelets of length 10.

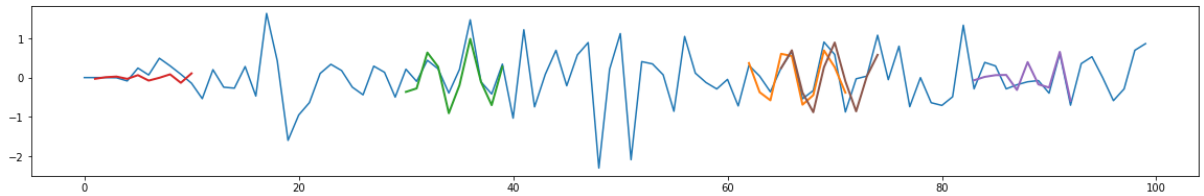


Figure 31: the 5 shaplets founded

Figure 31 shows a time series and the 5 shaplets found by the algorithm in the positions where the distance with the ts is minimum. As you can see in some places the shapelets overlap, this appen because they are also very similar to each other.

ShapeletModel can act also as classifier and can predict unseen instances, in this case, with an accuracy of 0.55, which is very low since the set is perfectly balanced (50% Hip-Hop and 50%Experimental) and a dum classifier who always assigns the Hip-Hop genre would have an accuracy of 0.50.

Next, we build the second dataset of shape (2000,5) that we mentioned above. Finally, any vector-based classification algorithm can now be applied to the shapelet-transformed dataset.

K-Nearest Neighbour

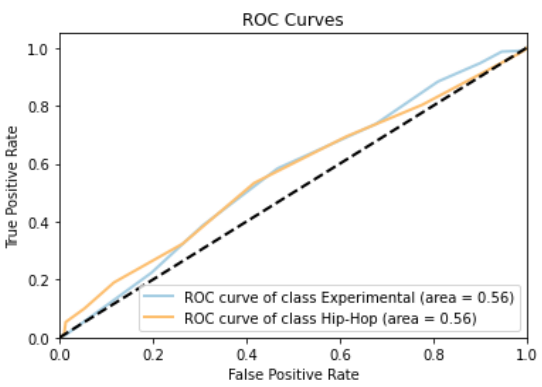
The first classifier used is the K-Nearest Neighbour, which has been applied first to the shapelet dataset. Thanks to GridSearch we estimate various combinations of parameters and find that the best results are given by setting neighbors = 13 and using Manhattan as metric.

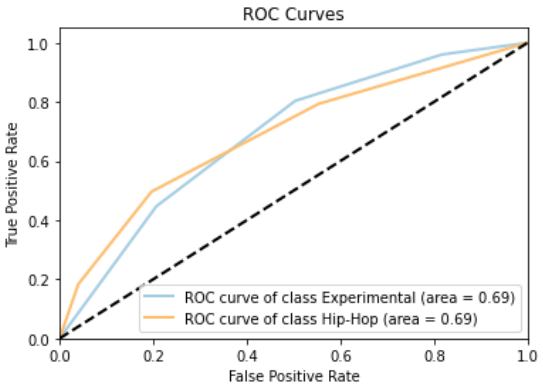
The results obtained are the following. Cross validated metrics:

- Accuracy: 54.0 % (+/- 0.03)
- Macro avg. F1 - score: 45.6 % (+/- 0.06)

While fitting the model on a test set, we obtained a slightly better F1-score for the class Experimental, but still very low.

Figure 32: the ROC curve of KNN with shaplet dataset to the left and time series dataset to the right





Then we applied KNN to the time series dataset. For this task we could choose between two libraries, the first one by sklearn did not let us choose the Dynamic Time Warping as the metric for calculating the distances and a second one by pyts who instead had the option. We used both and compared the results reported in table 6.

Table 6: the accuracy and F1 score of the different KNN

	Shapelet ds metric = manhattan	Timeseries ds metric = euclidian	Timeseries metric = dtw
Accuracy	54.0 %	53.4 %	65.2 %
F1 Scores	45.6 %	62.5 %	69.6 %

The classifier able to use DTW performs much better than the one using Euclidian, this is due to the fact that Euclidean distance is not able to discover time series who present similarity at different times or with different "speed"

.

Decision Tree Classifier

The second classifier applied is Decision Tree Classifier, as before we started applying DCT at the shaplet dataset. We used GridSearch for tuning the classifiers parameters and obtained the following parameters: max_depht = 6, min sample leaf = 1, min sample split = 50.

Our classifier obtained the following cross validated scores:

- Accuracy: 52,65 % (+/- 0.04)
- F1 - score: 47,37 % (+/- 0.05)

By fitting our model on a test set we obtained results in line with the cross-validation regarding the accuracy while the F1-scores perform slightly better: 60,8%. By comparing the F1-scores of the various classes, we can see that our model performs better with the Experimental class.

- F1-score class Experimental: 0.61%
- F1-score class Hip-Hop: 0.50%

Interpreting the recall metrics, we saw that our model was better at correctly identifying records with Experimental genre (recal: 68%) than records with Hip-Hop genre (recal: 44%).

After that we build a Decision Tree for the time series dataset, this time the parameters obtained by GridSearch are max_depht = 9, min sample leaf = 30, min sample split = 20.

Our classifier obtained the following cross validated scores:

- Accuracy: 52,57 % (+/- 0.07)
- F1 - score: 51,41 % (+/- 0.05)

By fitting our model on a test set we obtained better results: Accuracy of 60% and F1-score of 65%. As before our model performed better with the Experimental class.

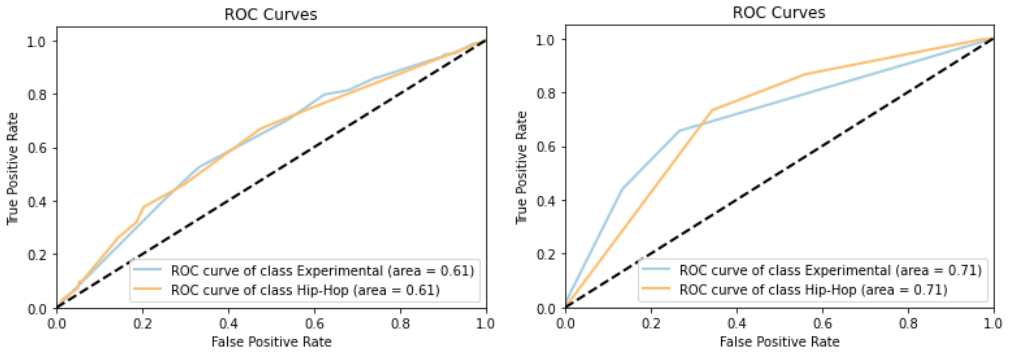
- F1-score class Experimental: 0.65%, precision: 77%
- F1-score class Hip-Hop: 0.50%, precision 41%

The Decision Tree Classifier did not perform well for both the datasets, which was predictable since the attributes over which it took decision were not features of the instances but measures of the frequency at a specific time. For this reason we also applied the DTC to the dataset of features extracted by the time series that we already used for clustering. And this time the classifier obtained the following cross validated scores:

- ❖ Accuracy: 67,38 % (+/- 0.07)
- ❖ F1 - score: 67,15 % (+/- 0.05)

And by the model on a test set we obtained very similar results and a better ROC curve, to the right in the figure 33.

Figure 33: the ROC curve of DTC with shaplet dataset to the left and feature dataset to the right



Module 4

4.1 Sequential Pattern Mining

For the analysis of Sequential Pattern Mining, we used the Time Series dataset with 7997 songs and 4180 features. We decided to divide this dataset into two datasets: one containing the Rock songs and another containing the Electronic ones. Furthermore, this analysis was carried out on the central part of the time series, in order to exclude the initial and final parts of the songs and to reduce the number of features, which were reduced to 500. SAX was then applied to reduce the size of the series, dividing it into 100 segments of equal length, whose values were then replaced with the average of the segment to which they belong; finally they were divided into 8 equiprobable regions and renamed according to a specified alphabet. We first tried to apply SAX with the parameters n_sax_symbols = 30 and n_paa_segments = 80 like before but the resulting patterns were too repetitive. The idea was to consider each song as a sequence of transactions (each of length 500) and apply the PrefixSpan algorithm on both Rock and Electronic datasets to find frequent patterns. Figure 34 shows the sequential patterns and related supports for the two datasets (on the left the one relating to Rock songs, on the right the dataset consisting only of Electronic songs).

Figure 34: Patterns with their support

Support	Pattern
1000	[5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 4]
1000	[5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 4, 4]
1000	[5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 4, 4, 4]
1000	[5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 4, 4, 4, 4]
1000	[5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 4, 4, 4, 4, 4]
1000	[5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 4, 4, 4, 4, 4, 3]
1000	[5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 4, 4, 4, 4, 4, 3, 3]
1000	[5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 4, 4, 4, 4, 4, 3, 3, 3]

Support	Pattern
1000	[4, 4, 4, 4, 4, 2]
1000	[4, 4, 4, 4, 4, 2, 2]
1000	[4, 4, 4, 4, 4, 2, 2, 2]
1000	[4, 4, 4, 4, 4, 2, 2, 2, 2]
1000	[4, 4, 4, 4, 4, 2, 2, 2, 2, 2]
1000	[4, 4, 4, 4, 4, 5]
1000	[4, 4, 4, 4, 4, 5, 5]
1000	[4, 4, 4, 4, 4, 5, 5, 5, 5, 5]

The first thing we can notice is that all sequential patterns have support equal to 1000, which is equal to the number of Time Series in each dataset, and this could be caused by the approximation made by SAX, which made the Time Series look very similar to each other. All Rock songs' patterns have decreasing values of symbols, instead Electronic ones have both increasing and decreasing symbols in patterns. Also of note is that all symbols in Rock's patterns are equal or above 3, which is close to the median symbol, instead in some Electronic's patterns there is the symbol 2. By analysing these patterns we can then assume that some of them could appear contiguously in the same time series, such as [5,5,5,5,5,6,6,6,6,6,6,6,6,6,6,4,4,4,4,4] and [5,5,5,5,5,6,6,6,6,6,6,6,6,6,6,4,4,4,4,4,3] because it's the same pattern plus 3. Most of the patterns found are of this kind, but clearly these observations are not the real goal of sequential pattern mining, because it was not feasible to interpret and analyze patterns of symbols extracted from audio waveforms.

4.2 Advanced Clustering

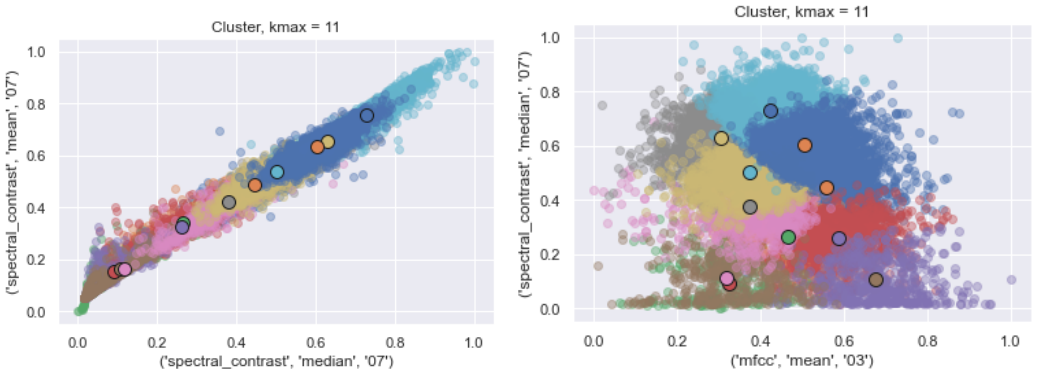
The dataset used for this module is the same we used for the classification task, that is features.csv after the appropriate removal of the outliers. However, given the high number of features present, it was appropriate to select just some of them before proceeding with the execution of the clustering algorithms. In particular, it was decided to use the dataset with the 9 features most correlated to the target variable. The new dataset is therefore composed of: ('mfcc', 'median', '01'), ('mfcc', 'mean', '01'), ('mfcc', 'median', '03'), ('mfcc', 'median', '04'), ('mfcc', 'skew', '03'), ('mfcc', 'mean', '04'), ('mfcc', 'mean' , '03'), ('spectral_contrast', 'median', '07'), ('spectral_contrast', 'mean', '07'). In order to proceed with the execution of the cluster algorithms, it was necessary to normalize these numerical features using MinMax Scaler.

The first algorithm used to perform Advanced Clustering is X-Means, which is an extended version of K-Means. The python library used is PyClustering. This algorithm doesn't need much hyperparameter optimization like that required by K-Means. The splitting criterion used is BIC; pyclustering implementation of the algorithm also provides features to define the amount of centers that should be initialized (amount_centers) and the amount of candidates that can be considered as a center in the second step (amount_candidates). Different values were taken into consideration for these parameters, but the results in terms of SSE and cluster composition were not very different from each other and it was therefore decided to initialize them both to 2. As for the number of clusters, unlike K-Means, it is possible to choose a maximum number of clusters to allocate (kmax). For kmax between 3 and 20, the results of SSE and Silhouette Score were plotted, in order to find a good trade off between the two, as shown in Figure 35. We therefore considered it appropriate to run the algorithm with kmax = 11, obtaining SSE = 653.15 and Silhouette Score = 0.29.

Figure 35: SSE vs Silhouette Score

In order to better visualize the results and analyse them, combinations of features were chosen to obtain plots in two dimensions, as shown in Figure 36. As for the first plot, we can observe a good separation with different values for each one of the cluster, however we cannot see clearly all 11 clusters: eight clusters are visible, while the others overlap, such as the green cluster with the brown one; the centroids are also well separated, with the exception of those of the red and pink clusters. We can see that in the second combination plotted, centroids are closer to each other, some of them almost arise, and are distributed diagonally, so the respective clusters are distinguish themselves to have values of ('spectral_contrast', 'median' , '07') and ('spectral_contrast', 'mean', '07') in the same range.

Figure 36: Clusters in 2D



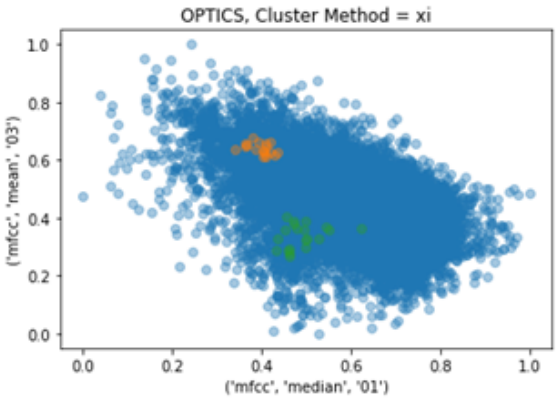
OPTICS is a density based clustering algorithm derived directly from the DBSCAN, which is best suited for large datasets. SKLearn implementation of OPTICS performs k-nearest-neighborhood on all points to identify core sizes with high density and then expands clusters from them. The OPTICS clustering technique does not need to maintain the epsilon parameter but it can be given to reduce the time taken. This leads to the reduction of the analytical process of parameter tuning. We chose to execute the algorithm with the method “xi” (the algorithm was also executed with the "dbscan" method, but the results were not better). The min_samples parameter (the number of samples in a neighborhood for a point to be considered as a core point) was chosen by observing the measurement of the Silhouette Score: in a range between 3 and 20, min_samples = 16 is the only configuration to return a positive score. We then proceeded with the execution of the algorithm, which unfortunately classified most of the points (99%) as noise points, as can also be seen from Figure 37.

4.3 Transactional Clustering

For this task, we decided to use the two dataset echonest.csv and track.csv. This decision was made in order to be able to take advantage of some categorical attributes present in track.csv, like “track_genre_top”. We merged the two dataset by “track_id”, appropriately deleting outliers. We also decided to delete rows with more than one value in the “track_genre_top” column. Most of the features were obtained by discretizing them according to their distribution. In particular, we created discrete attributes by observing the distribution of the attributes “danceability”, “energy”, “listeners”,

"liveness", "speechiness", "track_duration", "track_ favorites", "track_listens". The final dataset is therefore composed of 10 attributes and 9192 records.

Figure 37: OPTICS



We decided to approach this task with the K-Modes algorithm, comparing the results with a different number of clusters. The results are compared by referring to the cost of clustering, defined as the sum distance of all points to their respective cluster centroids. For k in range (2,20), the results of SSE were plotted in order to find the best hyperparameter. The graphic suggested us to use k = 8, but the clusters found were not so different and the number of records wasn't quite balanced into the 8 clusters. It was therefore decided to analyze the clustering obtained with k = 3, using the "Huang" method with 50 repetitions: the cost is 47185, obtaining three clusters with fairly separate centroids, furthermore the number of records is distributed in an equal manner between them. In Figure 38 are shown the details about the centers of each centroid. Cluster 2 and Cluster 3 capture the 'Rock' genre, while Cluster 1 the 'Electronic' genre. Cluster 1, in fact, has higher values of Energy, Danceability, Speechiness than the other two clusters, while Cluster 2 and 3 have higher values of Acousticness and Instrumentalness. Also, Cluster 2 has a low value in Liveness, instead Cluster 3 has a high one. Also of note is that all three clusters have not too distant low values regarding the track_listens, track_duration and track_favorites (None of them is "Liked").

Figure 38: Kmodes centroids

Clusters with k=3	Centroid
Cluster 1	'Medium_acousticness', 'Very_High_danceability', 'High_energy', 'Low_instrumentalness', 'Very_Low_liveness', 'Very_High_speechiness', 'Low_track_duration', 'Not_Liked', 'Electronic', 'Very_Low_listens_track'
Cluster 2	'High_acousticness', 'Low_danceability', 'Medium_energy', 'High_instrumentalness', 'Low_liveness', 'Low_speechiness', 'Low_track_duration', 'Not_Liked', 'Rock', 'Very_Low_listens_track'
Cluster 3	'Very_High_acousticness', 'Very_Low_danceability', 'Very_Low_energy', 'Very_High_instrumentalness', 'High_liveness', 'Very_Low_speechiness', 'Very_Low_track_duration', 'Not_Liked', 'Rock', 'Very_Low_listens_track'

APPENDIX

Table **α**: selected features explanation

Table	Variable Name	Category	Description	Values
track.csv	tracks.genres	Categorical Nominal	List of the track's genre IDs (related to genre.csv)	[n]
track.csv	tracks.date_created	Numeric interval	Date of upload on the FMA platform	dates
features.csv	mfcc.mean.01	Numeric ratio	Mean of MFCC spectral features of each frame for coefficient 1	q
genre.csv	title	Categorical Nominal	Name of the genre (where each record is a unique genre)	-
echonest.csv	acousticness	Numeric ratio	Song acousticness derived from Spotify	q

Figure **δ**: cross comparisons between all outlier methods detection

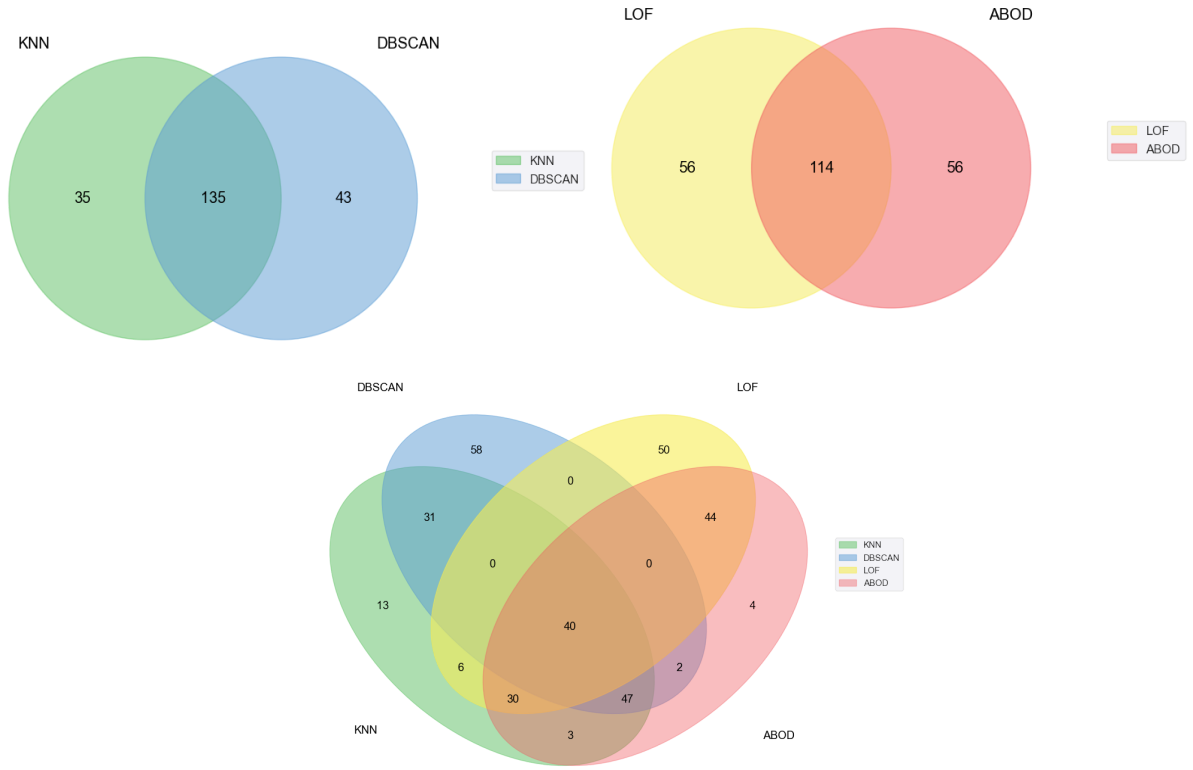


Figure **β**: Distribution of features usage for splitting across every estimators