Data Mining Project for Spotify Dataset

Greta Montera

Chapters

1 Understanding

In this report I will present observations and a general analysis of the Spotify dataset. In the first part I will try to discover correlations and trends. in the second part i will try to create model for prediction. This dataset is composed of 24 attributes. Some of them, such as *name*, *artists*, *album_name*, and *genre*, are categorical variables. The attributes *danceability*, *energy*, *loudness*, *mode*, *speechiness*, *acousticness*, *instrumentalness*, *liveness*, *valence*, *time_signature*, *n_beats*, *n_bars*, *popularity_confidence*, and *processing* and *duration_ms* are of type float64. The remaining attributes, *popularity*, *key*, and *features_duration_ms*, are of type integer, and *explicit* is a boolean.

I began my data understanding phase by checking the shape of the dataset and gathering information about the data. To accomplish this, I used Python libraries for data analysis, such as Pandas, NumPy, Seaborn and Matplotlib for visualization. There are 24 attributes or columns and 15000 rows. Next, statistical indices were determined for each numeric attribute of the dataset to facilitate a comprehensive analysis. For now, we can't discern much about the data, but statistical indices reveal some of the characteristics and shape of the data. In fact, there is also information about the quantiles and the median, measures which will be very useful in the following chapters.

Statistic	Danceability
Count	15,000
Mean	0.551
Std Dev	0.194
Min	0
25%	0.441
50% (Median)	0.580
75%	0.695
Max	0.980

Table 1: Statistical summary of danceability.

Description	Value
Columns	24
Rows	15,000

Table 2: Number of rows and columns.

1.1 Correlations and other observations on data

The next step was to determine the possible correlations between the variables. I decided to use the simple Pearson 's correlation at this stage instead of Kendall's and Spearman's. In this part of the heatmap, it is possible to observe that some attributes are positively and negatively correlated. For example, there is

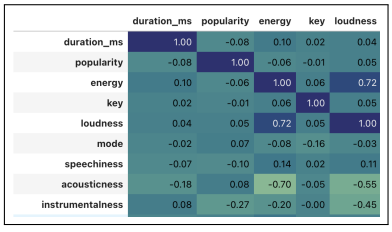


Figure 1: Correlation between variables

a strong positive correlation between the pair *energy* and *loudness*, while *energy* and *acousticness* are negatively correlated.

There is another interesting group of attributes we can focus on: *time_signature*, *n_beats*, *n_bars*, *features_duration_ms*, *duration_ms*. It emerges a very high correlation, most of them show a score of 1.00, which is a perfect correlation. in other words *features_duration_ms* and *duration_ms* are redundant attributes. Other positive correlations exist, like *duration_ms* and *n_beats* are weaker but still high.

features_duration_ms	time_signature	n_beats	popularity_confidence
1.00	0.01	0.84	-0.01
-0.08	-0.00	-0.08	-0.00
0.10	0.20	0.24	0.04
0.02	0.03	0.03	0.02
0.04	0.25	0.17	0.05
-0.02	-0.02	-0.03	0.00
-0.07	0.08	-0.04	0.03
-0.18	-0.14	-0.28	-0.02
0.08	-0.12	0.03	-0.03
-0.02	-0.06	-0.02	-0.03
-0.14	0.19	-0.07	0.00
0.05	0.22	0.46	0.04
1.00	0.01	0.84	-0.01
0.01	1.00	0.10	0.02
0.84	0.10	1.00	0.01
-0.01	0.02	0.01	1.00

Figure 2: Correlation between variables

I also checked the correlations between *n_beats* and *n_bars* because a lot of values were similar. In fact, their Pearson's correlation score is: 0.98. Even in this case the strong correlation majke these attributes redundant
In the dataset, I also found attributes that are not correlated at all. For example, one pair is *key* and *processing*, but the majority of the attributes are not correlated.
While searching for correlations, I noticed that some characteristics describing how a song sounds change based on the genre. Therefore, I attempted to highlight the differences between genres while analyzing the correlation between *energy* and *loudness*.
Then I did the same with other attributes that describe how the song sounds, such as *loudness*, *instrumentalness*, *key*, *acousticness*, *danceability*, *energy*, *liveness*, and the genre of the song and visualize it in a pairplot.

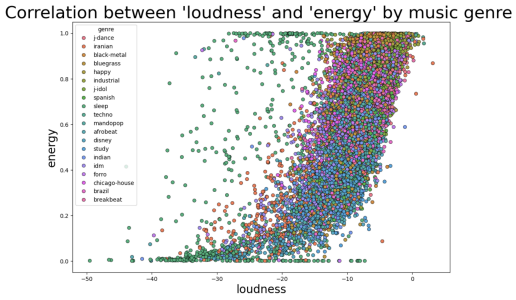


Figure 3: Scatter plot by genre

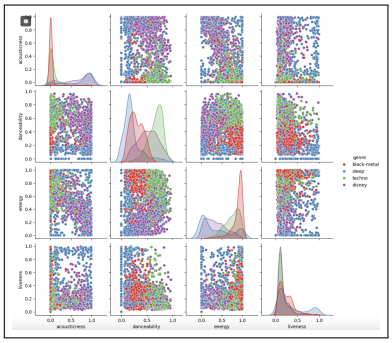


Figure 4: Difference between some characteristics of the sound based on genre

Based on this observation, I decided to visualize a histogram of the frequency of musical notation keys. The histogram reveals that the seventh key is the most popular, while the third key is the least popular.

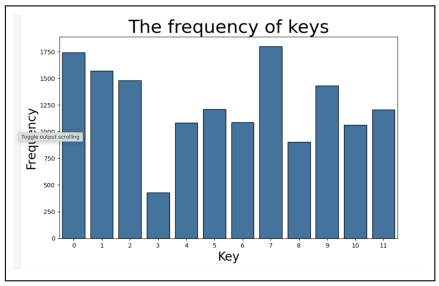


Figure 5: Barplot of Pitch Notation

1.	B/C
2.	C/D
3.	D
4.	D/E
5.	E
6.	F
7.	F/G
8.	G
9.	G/A
10.	A
11.	A/B
12.	B

Figure 6: Pitch Notation with Numerical Labels

After learning more about keys and their frequency, I wanted to represent this situation using a normal distribution to illustrate the shape of the data. To conduct this analysis, I created a new temporary table, referred to as *df_key*, which consists of two columns: *key* and *frequency*. The first column contains the musical keys, while the second column represents the frequency associated with each key. I utilized this table to visualize how the frequency of the keys follow a normal distribution. This phenomenon is highlighted in *Figure 7* which suggests that the keys frequencies are symmetrically distributed around a central value between 1000 and 1500.

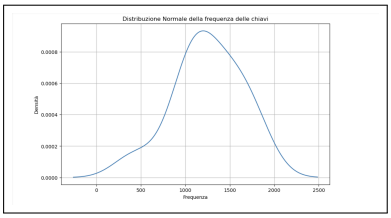
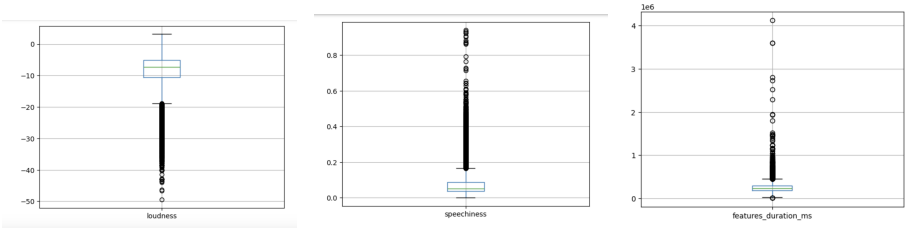


Figure 7: Normal distribution of the frequencies of the keys

1.2 Detecting outliers

In general there are a lot of outlier values, i found them in attributes like: *loudness*, *speechiness*, *duration_ms*, *n_bars*, *n_beats*, *features_duration_ms* and *liveness*. I used a boxplots to visualize them in *Figure 8*. I detected more than 400 rows with outlier values using the interquartile range between the 25th and 75th percentiles of each attribute, leaving 10,033 rows remaining. Expecially In the *loudness* attribute, I observed many negative outliers. I then calculated the Z-scores to assess how much this attribute deviates from the normal distribution and the result is -1.30, indicating that it is significantly below the mean. During the outlier detection phase, I began examining the *duration_ms* attribute (in this case i temporary converted milliseconds in minutes). I discovered that the genre with the highest mean duration is *chicagohouse*, with a score of 5.45, while the genre with the lowest mean duration is *study*, with a score of 2.4. I also ordered all the genres in ascending order and calculated the median, which I found to be 4.26. then, I used a boxplot to visualize the distribution of song durations (in milliseconds) for each genre. The boxplot graph highlights that there are extreme outliers in the *duration_minutes* attribute for genres such as *sleep*, *forro*, and *breakbeat* Anyway I decided to use the full dataframe in the following chapters due to the amount of information and the influence that outliers bring to the analysis.



(a) Outliers in *loudness*

(b) Outliers in *speechiness*

(c) Outliers in *features_duration_ms*

Figure 8: Outliers detected.

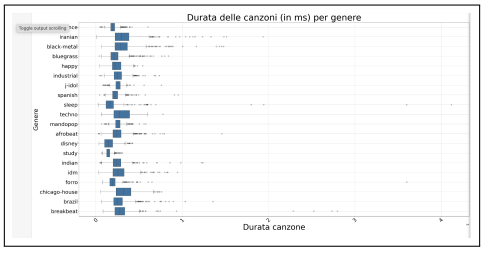


Figure 9: Boxplot that shows the distribution of songs duration for each genre

1.3 Detecting missing values

For the cleaning phase of the analysis I utilized the *isna().sum()* function from Pandas to calculate the number of missing values for each attribute. I noticed that attributes like *mode*, *time_signature*, and *popularity_confidence* have a considerable number of missing values and just 2.217 not-null values . Therefore, during the data preparation phase, I removed the entire column to clean the dataset.

Column	Missing values
name	0
duration_ms	0
explicit	0
popularity	0
artists	0
album_name	0
danceability	0
energy	0
key	0
loudness	0
mode	4450
speechiness	0
acousticness	0
instrumentalness	0
liveness	0
valence	0
tempo	0
features_duration_ms	0
time_signature	2062
n_beats	0
n_bars	0
popularity_confidence	12783
processing	0
genre	0

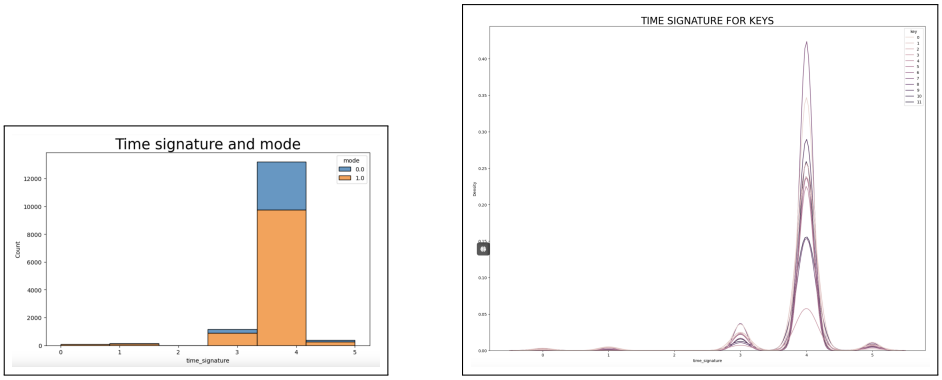
2 Preparation

2.1 Data cleaning and replacing missing values

After the information about missing values found during the Understanding phase, now i will clean the dataset from them if necessary. The first element detected was the column:

- *popularity_confidence* due to its large number of *null* values

There are also two other attributes,*time_signature* and *mode*, which are known to have some missing values like observed in the table above. I handled these attributes by replacing the missing values with the median of each attribute with the Pandas function *fillna()*. I also observed the trend of the *mode* attribute in relation to the *time_signature* attribute and *key* just to have an idea of which attribute take as principal element for the next chapters:



(a) Boundaries for outliers

(b) Key and time signature

Figure 10: Comparison of figures

I also converted the *explicit* column from boolean values (True and False) to 1 and 0, and I dropped the *artists*, *name*, and *album_name* columns, as they are not needed for my analysis. This was done to create a more homogeneous dataset.

3 Clustering

For clustering I utilized the entire dataset because, in the understanding phase, I observed that certain outliers—such as the extreme negative outliers detected in *loudness*—were highly significant for the analysis. After scaling the entire dataset using *MinMaxScaler()*, I proceeded with clustering using K-means, Hierarchical Clustering, DBSCAN and HDBSCAN. To accomplish this, I imported *KMeans*, *DBSCAN*, *HDBSCAN*, *AgglomerativeClustering*, and *KNeighbors_graph*.

3.1 K-means

Initially, I employed the Elbow Method to determine the optimal number of clusters for the K-means algorithm. The Elbow Method utilizes a measure known as the Sum of Squared Errors (SSE), which quantifies the compactness of the clusters and aids in selecting the number of clusters that strikes a balance between variance and model complexity. From the graph, it is evident that the optimal number of clusters is 6 or 7.

Additionally, I calculated the optimal number of clusters based on the Silhouette score, which suggested a value of 3. Consequently, I chose 3 as number of clusters.

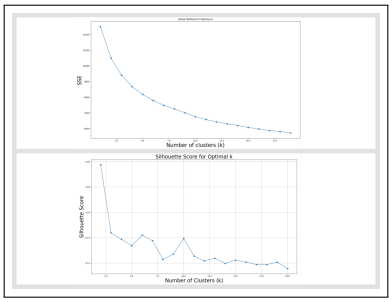
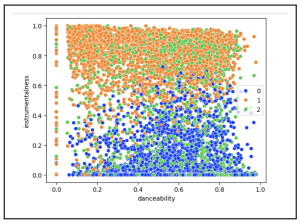
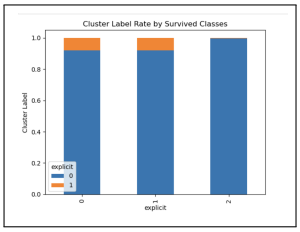


Figure 11: Silhouette score method

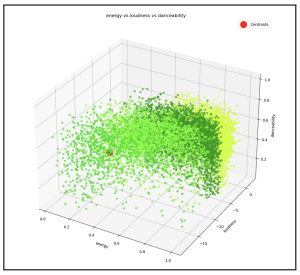
In the picture below (a), we can see the result of K-means algorithm: three clusters, with each color representing a different cluster. The scatter plot (in which the y variable is *instrumentalness* and the x is the *danceability*) shows that some areas are clearly separated from others but at the same time, they are not well-defined and overlap in certain areas, with the green cluster primarily overlapping with both the blue and orange clusters. This reflects the Silhouette score, which is not high enough to guarantee well-separated clusters. I also examined other characteristics within the clusters, such as the number of explicit songs in each cluster. In picture *b*, we can see that some songs in Clusters 0 and 1 are explicit,



(a) Four clusters after K-means



(b) Bar graph of explicit



(c) 3D visualization

but there are even fewer in Cluster 2. I also wanted to visualize the clusters in a 3-dimensional space, so I selected three variables: *danceability*, *loudness*, and *energy* as the z axis. The result is shown in *c*, with three different shades of green representing the three clusters. Even in a 3-dimensional space, it is clear that some clusters overlap in certain areas, and much of the data is quite sparse.

3.2 DBSCAN

In this subsection, I decided to use the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm as an alternative to the K-means algorithm for clustering. Unlike K-means, DBSCAN does not require specifying the number of clusters in advance and can identify clusters of arbitrary shapes. DBSCAN relies on two parameters: *epsilon*, that defines how close points must be to each other to be considered part of the same cluster and the minimum number of points required to form a cluster. To determine appropriate values for these parameters, I used the K-Distance Graph, which helps identify a suitable *epsilon* value in a manner similar to the Elbow method used before. The K-Distance Graph plots the distance from each point to its k-th nearest neighbor (in this case, the 3rd nearest neighbor), sorted in ascending order. The elbow of the graph indicates the optimal *epsilon* value. I observed the elbow at a value of 0.7 on the y-axis, so I chose $epsilon = 0.7$. For the minimum number of points, I selected 10. Now, the Silhouette score with DBSCAN was 0.19, which is very low and also DBSCAN has a speed of 18 s against 22.1 s of K-means. I noticed that points are assigned to their respective cluster but there is one cluster in DBSCAN that is a lot denser than others. In fact DBSCAN generated 2 clusters with*Cluster 0* a lot denser than the other and few noise points.

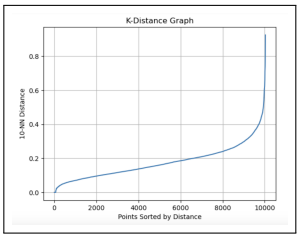


Figure 13: K-Distance Graph illustrating the optimal *epsilon* value.

I attempted to address the high dimensionality of the data and the density problem using HDBSCAN, a variation of DBSCAN that is not influenced by the dimensionality of the data, noise, or outliers. Instead of the *epsilon* and *eps* parameters, HDBSCAN uses *min_cluster_size* and *min_samples*. With HDBSCAN, the silhouette score is 0.2, and the execution time is 14 seconds, making it faster than DBSCAN. Consequently, I decided to incorporate PCA to improve both the performance of the HDBSCAN algorithm and its visualization

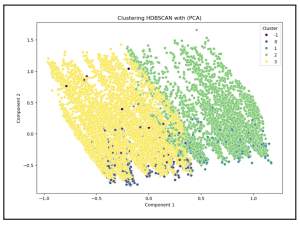


Figure 14: HDBSCAN with PCA

3.3 Hierarchical Clustering

The last clustering method I used was Hierarchical Clustering, a technique that builds a hierarchy of clusters. I employed Agglomerative Clustering, which starts with each object as an individual cluster and then successively merges the closest clusters until all objects are contained in a single cluster. I set the parameters:

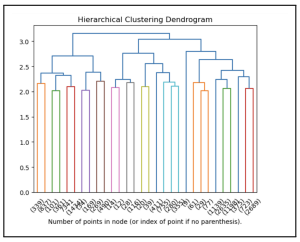


Figure 15: Hierarchical Clustering

- distance threshold = ”0”.
- n_clusters = ”None”.
- Linkage = ”complete”

Setting distance_threshold=0 essentailly means that
the algorithm will not stop merging clusters until all objects are combined into a single cluster. *N_clusters* specifies the number of clusters to find. Linkage set to ”complete” calculates the distance between two clusters as the maximum distance between any two points in the clusters. It tends to create more compact clusters and is less sensitive to outliers. I chose ”complete” value for this parameter because i have a lot of outliers in the dataframe.

3.4 Conclusion

In conclusion, I used four different techniques for clustering my dataset and labeling the points: K-means with 3 clusters, DBSCAN with PCA and 3 clusters, HDBSCAN with 4 and Hierarchical Clustering to obtain a hierarchy. The best algorithm is definitely HDBSCAN because it has the highest Silhouette score and is also faster in terms of execution time.

	Silhouette Score	**Execution Time**
K-means	0.199	22.1 s
DBSCAN	0.194	18 s
HDBSCAN	0.200	14 s
Hierarchical	0.078	24.1 s

Table 3: Comparison of Clustering Algorithms

4 Classification

Classification is a supervised learning technique where the goal is to predict categorical labels. A crucial step in classification is training the model on a portion of the dataset and testing it on another portion of unseen data to evaluate its performance. At this point, I decided on the goal I wanted to achieve: the

correct assignment of a song to a specific key or music genre. I created three different models: the first one using KNN to assign a song to a specific key, followed by Naive Bayes and Decision Tree models to assign a song to a precise music genre

4.1 Partitioning the Dataset

The first step in building the model was to split the dataset into two parts using the standard holdout method. This method splits the entire dataset into two subsets: one for training the model and one for testing it. The training set should be larger because we need a substantial amount of data for generalization and to avoid overfitting. Once the model is fully trained, the next step is to test the model on the remaining portion of the original dataset, which should be new to the model. This testing phase is crucial for assessing the model's performance and evaluating the classification task on unseen data.

For this phase I imported *train_test_split* from *sklearn* and used it for splitting: at the end of the splitting phase the result is:

- (10500, 20: The feature matrix used for training the model.
- (4500, 20): The feature matrix used for testing the model.
- (10500,) The target vector corresponding to the training feature matrix.
- (4500,) The target vector corrisponding to the test feature matrix.

After that the analysis proceeds with the normalization phase in which will be used the *StandardScaler()* imported from the library *sklearn.preprocessing* to normalize the training set.

4.2 KNN for Classification

K-Nearest Neighbors (KNN) is used to assign a class label to a new data point based on the majority class of its nearest neighbors, after calculating the distance. For this algorithm, I used standard values based on my dataset: the Euclidean metric for distance and 13 as the number of nearest neighbors

Accuracy, Recall, Precision for Evaluating KNN: I checked the model's performance using other metrics like recall, precision, F1-score (which indicates the harmonic mean of precision and recall), and support. Then, i used the confusion matrix in *Figure 16* to visualize the performance of the model.

These results indicate that the model sufficiently assign the data to the correct a key. The performance is also represented in the ROC curve, which shows the threshold for considering the performance good. The proximity of the curve to the top left corner of the graph (where TPR is maximum and FPR

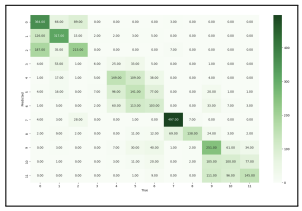
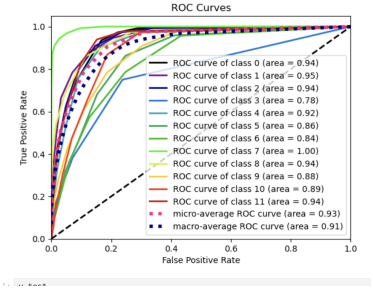


Figure 16: Confusion matrix of predicted data

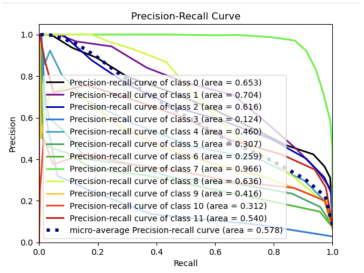
Class	Precision	Recall	F1-score	Support
0	0.53	0.70	0.60	522
1	0.60	0.67	0.64	470
2	0.61	0.48	0.54	444
Accuracy		0.54		
Macro avg	0.52	0.48	0.49	4500
Weighted avg	0.55	0.54	0.53	4500

Table 4: Part of the measures

is minimum) indicates strong performance. I used the roc_auc_score function to calculate the mean AUC score for a multi-class classification problem by treating each class separately (one-vs-rest) and then averaging the results. The average ROC curve score is 0.91, while the Precision-Recall curve score is 0.58. These results suggest that while the model is effective at distinguishing between different keys, it sometimes misclassifies songs into incorrect keys, leading to lower precision in its predictions.



(a) Roc curve



(b) Precision-recall curve

Figure 17: Results of the performance

4.3 Alternative method for K-NN

Repeated Holdout: I used this method as an alternative to the standard hold-out validation in the model evaluation and validation processes described in the previous pages. In this case, I set the number of iterations to 50, meaning the entire process of partitioning the data, training, and testing is repeated 50 times, each time using a different portion of the data. Additionally, an overall error is estimated by averaging the errors from each iteration. The result is an error rate of 0.458, which indicates that the model is not very accurate. However, with an accuracy of 0.54, we can say that the performance is sufficient, considering the large amount of data and the presence of many outliers.

Cross Validation: This method uses a parameter called *K*, which determines the number of portions into which the dataset is splitted for the training and validation phases. The model is trained on *K-1* folds and tested on the remaining fold, with this process being repeated *K* times so that each fold is used as the test set once. In terms of time, it is slower than Repeated Holdout. I tried $K=13$ and the overall error is 0.453.Accuracy is also computed and it has score 0.54, so basically the results are the same for both methods.

4.4 Finding the best value of *KNN*

Hyperparameters Tuning: Previously i observed that the model's performance is good but it could be improved by searching for better values for parameters. I used Hyperparameters Tuning to find the best value of neighbors indicated in the *x* axis. The higher the line the best the accuracy. The graph in *Figure 18* shows that we could try to set the number of neighbors to 16 instead of 13 to increase the accuracy score to 0.56. So I set the parameter *K* to 16 and the overall error calculated by using Cross Validation is minimized a little from 0.453 to 0.452.

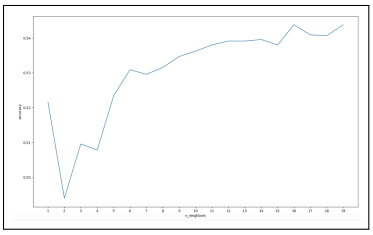


Figure 18: Best accuracy

Grid Search: Then, I used the second method: Grid Search, a technique to help the model find other new best parameters such as weights, metric, and number of neighbors. It works by evaluating 10 different combinations of these parameters, calculating the cross-validation score for each. In the end, the combination with the highest score is chosen. The resulting combination is: The

Parameter	**Value**
Weights	distance
Number of Neighbors	29
Metric	cityblock

Table 5: Best hyperparameters found by Randomized Grid Search

Best cross-validation score:
0.78

Table 6: Best cross-validation score achieved

accuracy of the model with these parameters is 0.53

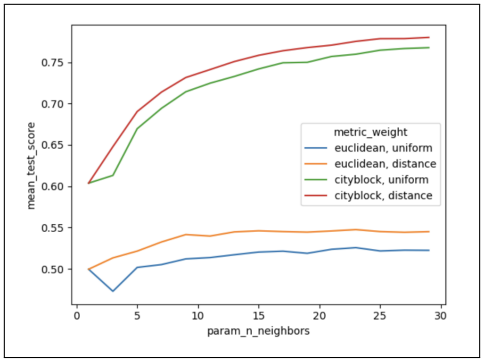


Figure 19: Best parameters

4.5 Naive Bayes

So far, the task I implemented was to assign a song to a specific key. Now, I aimed to use Gaussian Naive Bayes to categorize a song into a genre based on its characteristics. The model created with Gaussian Naive Bayes has proven to be less effective than the previous one due to the challenging nature of the task, in fact the accuracy is higher for the previous model (0.54), now it is very slow 0.21.

Genre	Precision	Recall	F1-score	Support
Afrobeat	0.05	0.00	0.01	290
Black-metal	0.14	0.06	0.08	282
Forro	0.50	0.07	0.12	294
Happy	0.22	0.19	0.20	291
IDM	0.29	0.10	0.15	321
Indian	0.26	0.02	0.03	318
Accuracy	0.21	-	-	6000
Macro Avg	0.26	0.21	0.17	6000
Weighted Avg	0.26	0.21	0.17	6000

Table 7: Classification report for the Gaussian Naive Bayes model

I used the same evaluation methods used for K-NN, specifically the ROC curve, to assess the performance of the model in this task. It is evident that the model does not perform well, as some lines in the graph do not exceed the threshold line. Additionally, I utilized cross-validation to estimate the overall error, which

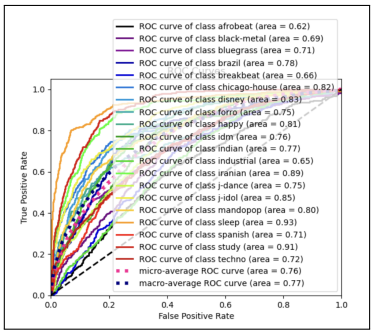


Figure 20: ROC Curve

turned out to be very high at 0.78. In conclusion, this model is not sufficiently effective to predict the music genre based the characterstics of the song.

4.6 Decision Tree

The last algorithm I implemented, like Naive
Bayes, for genre prediction was the Decision Tree which i thought it could create a good model for this purpose given its ability to handle redundant values, noise and the classify unknown records. First, I used the standard holdout method for splitting data into training and test sets and I normalized the data like i did before. Then, I imported *DecisionTreeClassifier* and *plot_tree* from *sklearn.tree* for the visualization of the decision tree. Looking at *Figure 21* and *Figure 22*, the validation metrics and confusion matrix indicate that the model performs exceptionally well on the training data but somewhat less effectively on unseen data. This situation is referred to as 'overfitting,' where the model fits the training data extremely well but lacks generalization, resulting in poorer performance on the test set. Overfitting can be mitigated through pruning. But first, I tried to change different parameters and finding the best value to improve the model even without pruning. I evaluated the performance across different values of depth, the minimum number of splits, and the minimum number of leaves by calculating the average accuracy and its variability using cross-validation. These results are very important as they reflect the model's complexity and its ability to generalize effectively: changing them could be a crucial step for the improvement of the perfomance.

Metric	Train	Test
Accuracy	1.0	0.0856
F1-score Class 1	1.0	0.0722
F1-score Class 2	1.0	0.0000
F1-score Class 3	1.0	0.1800
F1-score Class 4	1.0	0.0237
F1-score Class 5	1.0	0.0000

Figure 21: Train and Test Accuracy and F1-scores for each class

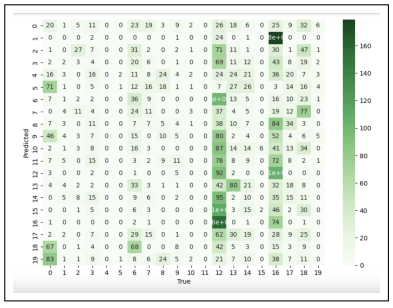


Figure 22: Confusion matrix of predicted data

I used the Randomized Search to find the best combination of parameters. The Randomized Search is very useful if the dataset is very large because it randomly selects a limited number of parameter combinations to test. The best parameters are:

- min_samples_split = 20
- min_samples_leaf = 10

- max_depth = 11
- criterion = gini

The score is 0.46 and the accuracy has improved: 0.15 but still very low. I also found out the most influential features in the model, and *popularity* is the most influential one with a score of 0.21, so basically it is the root node.

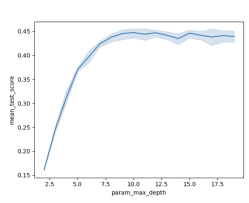


Figure 23: Best max-depth

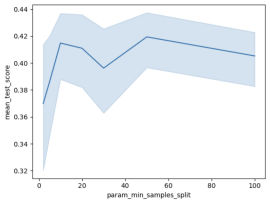


Figure 24: Best number of splits

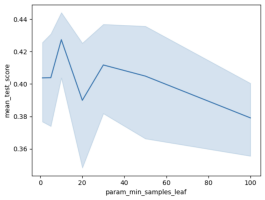


Figure 25: Best number of leaves

Here is how the tree looks after using the best parameters found with Randomized Search and the score of the importance of features:

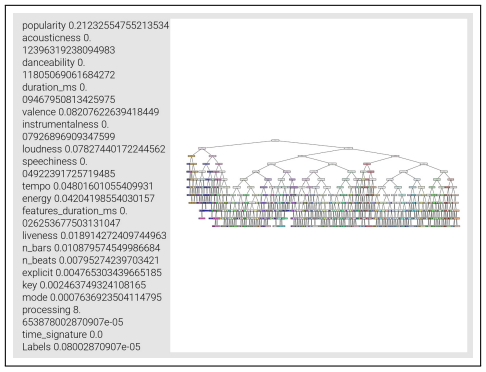


Figure 26: Caption1

4.7 Pruning:

Considering that the tree performes really good on training but bad on test we could talk of 'overfitting' phenoemenon. This problem can be mitigated with pruning. Pruning cut off useless nodes that can be the cause of overfitting. To do so i first detect those nodes, the impurities. This is part of the optimization process to avoid overfitting, which ultimately leads to better performance on unseen data. The parameter *alpha* is used to control pruning. It helps balance the trade-off between the complexity of the tree and the impurity of the nodes. A higher *alpha* value leads to more pruning, resulting in a simpler tree with higher impurity but potentially better generalization, while a lower value allows for maintaining greater complexity. Like seen in *Figure 27* the best value for alpha is 0.006. so i used it and also the best parameters found before to build again a new decision tree.

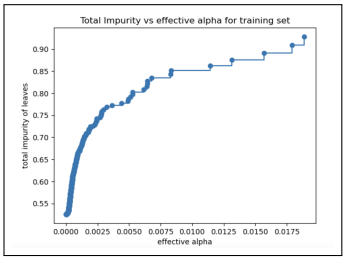


Figure 27: Best value of alpha

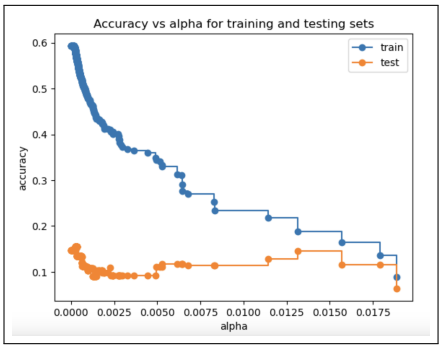
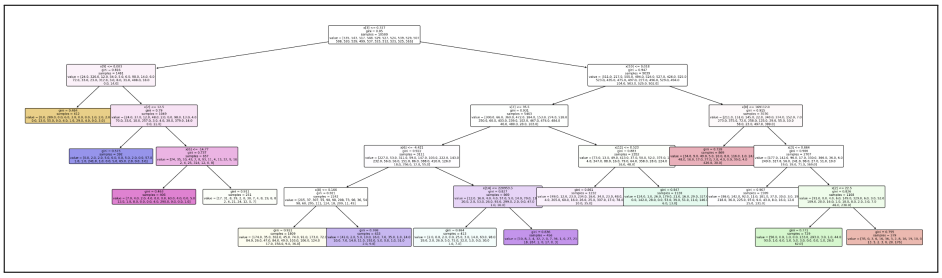


Figure 28: Training vs Test

Figure 29: Decision Tree



4.8 Conclusion

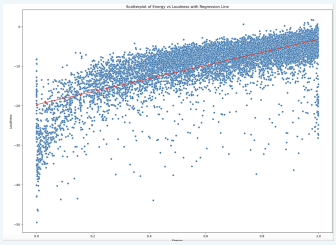
In conclusion, for the classification task of assigning songs to a genre, I compared two different methods: Gaussian Naive Bayes and Decision Tree. Gaussian Naive Bayes performed better with the highest accuracy score of 0.21, while the Decision Tree had a score of 0.12. However, neither model performed exceptionally well for this task. In contrast, K-Nearest Neighbors (K-NN) performed significantly better for assigning a song to a key, achieving an accuracy

score of 0.54. I explored different techniques to improve the parameter values for these algorithms. Specifically, for K-NN, I used hyperparameter tuning and grid search. Both methods provided good solutions, but hyperparameter tuning yielded the best results as it produced the highest accuracy with its recommended parameters.

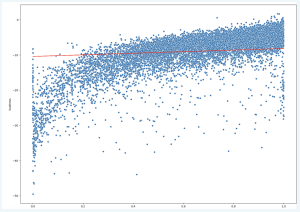
5 Regression

Regression is an analysis technique used to understand the relationships between a dependent (or target) variable and one or more independent (or predictor) variables. In this chapter, I will present my regression analysis, with the purpose of predicting one or more variables that describe the sound of a song. I started by importing LinearRegression, Ridge, Lasso, for linear regression and KNeighborsRegressor, DecisionTreeRegressor for different non linear regression, as well as *r2_score*, that shows how well the regression model explains the variance of the target variable, *mean_squared_error* measures the average squared difference between the predicted values and the actual values, and *mean_absolute_error* the average of the absolute differences between the predicted and actual values. Even for regression, I splitted the dataset in 2 parts for training and for testing.

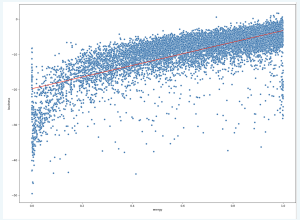
Linear regression: The first type of regression i implemented was linear regression, which uses two variables on the axes: *energy* and *loudness* in the scatter plot. From the graph, we can observe that the line follows the majority of the points, indicating that the model is reasonably good. However, these values not only represent the model's performance but also reflect the influence of a large number of outliers in the *loudness*variable, especially when considering the MSE. The metrics are as follows: R^2: 0.511, MSE: 16.991, MAE: 2.811.



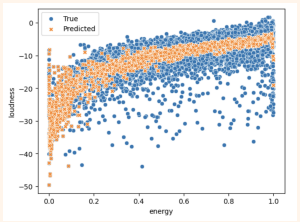
Lasso regression: (Least Absolute Shrinkage and Selection Operator) adds an L1 penalty to the loss function that generates the regression line. While this penalty helps to avoid overfitting, it can also reduce accuracy, leading to a higher MSE. Values R^2: 0.134, MSE: 30.108, MAE: 3.866.



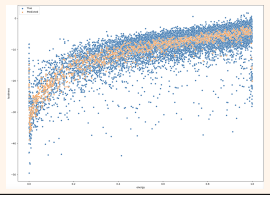
Ridge regression: adds an L2 penalty to the loss function. This penalty discourages large coefficients by shrinking them towards zero, which helps to mitigate overfitting and improve generalization. Values: R^2: 0.511, MSE: 16.990, MAE: 2.811.



Decision Tree Regressor: models the relationship between the dependent and independent variables as a non-linear function. In a DTR the leaves of the tree provide predicted values by averaging the target values of the data points that fall into each leaf. The picture shows that the model has some predictive power but may be only moderately effective. Values: R^2: 0.572, MSE: 14.860, MAE: 2.641.



K-Nearest Neighbors (KNN): predicts the value of a target variable based on the average value of its nearest neighbors in the features space. Values: R2: 0.561, MSE: 15.252, MAE: 2.705.



Multiple and Multivariate Regression: Multiple Regression: Multiple Regression uses two variables to predict one. In this case, I chose *loudness* to be predicted from *energy* and *instrumentalness*. The model performs quite well, with a high R^2 score indicating strong performance. The values for multiple regression are: R^2: 0.609, MSE: 13.592, MAE: 2.467. For multivariate regression I decided to predict *loudness* and *liveness* from *energy* and *instrumentalness*. Values of multivariate regression: R^2: -0.362, MSE: 6.912, MAE: 1.930.

Figure 30: Multiple regression

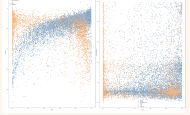


Figure 31: Multivariate regression

5.1 Conclusion

In conclusion, regression is an effective and efficient model for prediction. In particular, I used 7 different types of regression techniques, 3 linear and 4 non-linear. However, the model generated with Multiple regression is the best based on R^2, as it has the highest value (0.609), indicating the best fit to the data.

Model	R^2	MSE	MAE
Linear	0.511	16.991	2.811
Lasso	0.134	30.108	3.866
Ridge	0.511	16.990	2.811
DecisionTree	0.572	14.860	2.641
KNN	0.561	15.252	2.705
Multiple	0.609	13.592	2.467
Multivariate	-0.362	6.912	1.930

Table 8: Performance metrics for different models.

6 Pattern Mining

During the pattern mining phase, my goal was to create pattern mining rules using the entire dataset. I imported *Apriori*, *FP-Growth*, then the first step was to convert all the attributes into categorical strings by binning them. I chose to use 4 bins for all attributes except for *popularity*, which I split into 2 bins to classify each song as popular and non-popular. Additionally, I transformed the boolean value for *explicit* from 1 and 0 into Explicit and Non-Explicit, and the values for *mode* (1.0 and 0.0) into Major and Minor. By the end of this process, my dataset appeared as follows:

Table 9: First row of a portion of the dataset

explicit	key	mode	genre	DurBin	PopBin
Not Explicit	F4	Major	j-dance	(227826.0, 288903.0_duration]	(24.0, 94.0_popularity]

6.1 Apriori

Using the Apriori algorithm, I identified the most frequent itemsets. First, I set the support parameter to 20, so an item has to appear at least 20 times to be considered frequent, (because the dataset I am working with is very large) and the minimum number of items per itemset to 3. I found that (*(((Major, Not Explicit, (-0.001, 4.0]time_signature))*, is the most frequent itemset, with a support of 68.133, it means that is very frequent. The final dataset obteinaed contain 101 rows. Then, I searched for closed itemsets and maximal itemsets, and I observed that as the support increases, the number of closed and maximal itemsets decreases like shown in *Figure 32*. The reason for

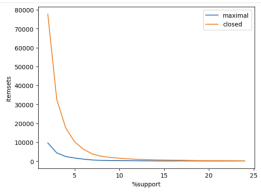


Figure 32: Support vs Itemsets

this is that with a higher support value, only the most common itemsets remain frequent. The dataset generated for closed itemsets consists of 99 rows. The algorithm for maximal itemsets searches for itemsets that are not subsets of any other frequent itemset, and it identified 47 such itemsets. The next step involves generating association rules to determine what attributes contribute to a song's popularity, specifically, I selected "popularityBin" as the consequent and generated rules. I used several parameters to create these rules: *conf* with value 60, support with the same value of 20 and minimum number of element with value 3, i also included *lift*, which measures the strength of the rule. I sorted the association rules by the highest lift. The algorithm produced all the rules where the result indicates a certain number of attributes that make a song popular.

Table 10: Dati della Regola

Consequent	(24.0, 94.0]_popularity
Antecedent	((-0.001, 0.00313]_instrumentalness, Major, No...
Absolute Support	3546
% Support	23.64
Confidence	0.672610
Lift	1.348096

6.2 Fp-growth

FP-Growth is a more efficient technique used to generate association rules compared to the Apriori algorithm. It is less expensive and faster because it identifies frequent itemsets directly without generating candidate sets. This efficiency is particularly beneficial because the dataset i very large. Using the same parameter values (but changing confidence with value 70 instead of 60), such as support and minimum number of elements for itemsets, the result obtained is *(((Major, Not Explicit, (-0.001, 4.0_time_signature))* with a support of 68.1333. This result is identical to what the Apriori algorithm provided, but achieved in a more efficient manner. These are the results obtained with FP-Growth:

Table 11: Detailed Rule Information

Field	**Value**
Consequent	(24.0, 94.0]_popolarità
Antecedent	((-0.001, 0.00313]_instrumentalness, Major, No Explicit)
Absolute Support	3546
% Support	23.64
Confidence	0.672610
Lift	1.348096

6.3 Conclusion

In this chapter, I used two methods for generating pattern mining rules. Both methods yielded the same results but differed significantly in terms of time and memory usage. Specifically, it is evident that the Apriori algorithm consumes much more memory compared to FP-Growth and is also slower. FP-Growth

Table 12: Performance Comparison of FP-Growth and Apriori Algorithms

Algorithm	**CPU Time**	**Wall Time**
FP-Growth	Total: 5 μs	7.87 μs
Apriori	Total: 8 μs	14.1 μs

execution is better in terms of both CPU time and wall time. Lower CPU time indicates less computational work was required, and lower wall time means the total time to complete the task was shorter.