FOM Hochschule für Oekonomie & Management

Hochschulzentrum Düsseldorf

Scientific Paper

part-time degree program

5th Semester

in the study course "Wirtschaftsinformatik"

as part of the course

Big Data & Data Science

on the subject

Predicting Music Genres based on Spotify Song Data using a Gradient Boosting Algorithm

by

Thomas Keiser

Martin Krüger

Luis Pflamminger

Jesper Wesemann

Advisor:	Prof. Dr. Adem Alparslan
Matriculation Number:	534320 (Keiser), 534306 (Krüger), 538276 (Pflamminger), 533882 (Wesemann),
Submission:	January 31st, 2022

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
URL	Uniform Resource Locator
JSON	JavaScript Object Notation
HTTP	Hyptertext Transfer Protocol
PCA	Principle Component Analysis
CART	Classification and Regression Tree
GBM	Gradient Boosting Machine
PR	Pseudo Residual
ROC	Reciver Operating Characteristic
AUC	Area Under the Curve
CRISP DM	Cross Industry Standard Process for Data Mining
CV	Cross Validation
BPM	Beats per Minute

1 Introduction

Spotify provides the world's leading streaming service. Since its launch in 2006, the company has gained over 365 million users, of which nearly 45 percent are subscribed to the chargeable premium service [1]. The ability to listen to almost any song a user might want is a great benefit streaming services have over regular music vendors like iTunes. On the other hand, users might quickly get lost or feel overwhelmed by such a large collection to choose from. To guide users and help them find the music they want to listen to in a certain situation, Spotify uses a number of methods like premade playlists or categories for specific moods and genres. The biggest benefit to the user experience might certainly be the personalized playlists, radios or artist and song recommendations. To be able to have such a robust recommendation system, Spotify needs to understand each user's listening behaviour and have methods to predict, which music a user might enjoy depending on their past usage. Not just user data is important to gain this knowledge, but Spotify also needs to understand how to categorize music itself and find ways to identify which songs are alike and how they relate to each other. Data Science is essential to achieve this goal. Machine learning is used to analyze the music in their catalogue and create characteristics about it [2]. Spotify offers this data to the public **SpotifyWebAPI**, which forms the foundation of the dataset.

1.1 Problem Definition and Goal

The paper is set in the context of Big Data Analytics and aims to explain and implement many of its concepts and processes. The basic goal of the project is to develop a machine learning model. During this process, general data mining stages, as layed out in the Cross Industry Standard Process for Data Mining (CRISP DM), must be theoretically explored, understood and practically implemented. This includes steps like data preparation, model creation and evaluation. Creating a model with a high accuracy is only a secondary concern in this project. Instead gaining a solid understanding of data mining concepts is paramount.

To achieve this goal Spotify song data is collected and an attempt is made to build a Gradient Boosting Algorithm, which is able to classify songs into different genres to a reasonable accuracy based on preprocessed audio features from Spotify.

1.2 Structure and Methodology

The paper is divided into four main sections, starting with the introduction which contains the problem statement, goal and overall structure.

The second section discusses fundamental concepts. This includes embedding this project into the larger context of Big Data, explaining the algorithms used on a theoretical level and introducing further concepts which are used for data preparation and modeling. Additionally, the CRISP DM model is explained, which is the basic structure that implementation is based on. The second section is concluded by an introduction to the use cases and funcionality of Web Application Programming Interfaces (APIs) and a discussion of basic concepts in music theory.

The third section begins with the data collection process and continues on with the practical implementation steps of understanding the dataset's features and labels in the context of Spotify's analysis, preparing and analyzing the data for modeling, creating a gradient boosting classifier and finally evaluating the project results.

The paper concludes with a summary of the insights gained and embedding the project into a broader context.

Facts presented as part of the paper's fundamental sections are derived from literature research using renowned book sources and scientific papers. Additionally online articles were used to round out the research. Most visualizations in this paper are made by hand using Python libraries such as *seaborn* and *matplotlib*. For data collection and understanding Spotify's developer resources are used as a reference. Documentation from Python libraries such as scikit learn is used extensively during the preparation, modeling and evaluation parts of the project.

2 Fundamentals

In this section the project is embedded into the larger context of Big Data, algorithms are explained on a theoretical level and further concepts for data preparation and modeling are introduced. Additionally, the CRISP DM model, Web APIs and basic concepts of music theory are explained.

2.1 Classification in the Context of Big Data

The project falls under the umbrella term Big Data. Big Data is defined as a large set of information which is stored and processed to create business value. The analysis represents a substantial part of Big Data and is referred to as Big Data Analytics [3, p.4].

The term analytics is used to describe a systematical analysis of data of any format. Analytics is about detecting hidden patterns, clusters and meaningful features ranging from simple detection to deep analysis and predictions [4, p.2]. The creation of value is closely linked to the use cases for which in-depth knowledge is required. Generally, literature distinguishes between four subsections of analytics by grouping them according to the methodology and their objectives [3, p.8f]. Descriptive analytics is the simplest form of analytics and is often the first step for further research. It gives information about what has happened in the past and is mostly used for reporting. Diagnostic analytics is based on descriptive analytics but serves a different goal, since its objective is to give insights on why something has happened. Diagnostic analytics evaluates the impact of features, detects correlations and dependencies. Predictive analytics takes this approach one step further as it predicts what is most likely going to happen in the future based on the knowledge gained from past data. It creates models with the help of algorithms to determine the probability of outcomes. The final step of analytics is prescriptive analytics which is again based on its predecessor. Prescriptive analytics predicts outcomes and recommends actions to avoid or support them respectively [3, p.8f].

This project contains elements of descriptive, diagnostics and predictive analytics. Using the CRISP DM process model, the dataset is analyzed first. Afterwards a predictive model is generated to classify new data based on known patterns.

Furthermore, Big Data Analytics covers a large variety of research areas that differenticate mostly in their input and learning methodology. The following sections briefly explains these concepts and classifies the project task accordingly.

Input data exists in various formats for which adequate storage solutions and machine learning algorithms are required. Conventional data is almost exclusively stored in structured data formats while modern developments increasingly rely on semi- and unstructured data. Unstructured data has no fixed format, scheme or structure. Structured data, on the other hand, has a fixed format and fits into a predefined data models. Semi-structured data is an intermediate stage and is not based on a strict standard. It often consists of predefined tags like JavaScript Object Notation (JSON) [4, p.2f]. This project utilizes semi-structured data during the collection-phases which is converted into a structured format for the modeling.

The fundamental learning methods are unsupervised, supervised and reinforced learning with semi-supervised learning as a subcategory. The methods differ in how the model learns and which input data is required [5, p. 4]. For supervised learning, a model is created, which maps input values, often refered to as targets, to a known label value. The model thus learns on already correct and complete data through the detection of hidden pattern and correlations. The target labels are either nominal or numerical which results in a classification or regression task [6, p. 46]. Unsupervised learning forms the opposite as its input data has no target labels. The algorithm is autonomous in deriving or recognizing common structures [7, p. 97]. Semi-supervised learning is set between both extremes and includes samples from both categories. Reinforcement learning is a fundamentally different learning approach with no training data being available at the beginning [5, p. 7]. The algorithm acquires data only by interacting with the environment. Models train by tackling challenging problems and having their decisions immediately rewarded if successful or punished if unsuccessful through feedback signals [7, p. 98]. This project uses a Gradient Boosting Algorithm, which belongs the category of supervised learning methods. Therefore, the dataset consists of features and labels.

2.2 Decision Trees

Decision trees are one of the most widely used Supervised Machine Learning Algorithms either as standalone solutions or in combination with enhancement approaches like boosting. They allow for a very flexible construction and can be utilized for various machine learning problems such as classification and regression.

Decision trees predict "the value of a target variable by learning simple decision rules inferred from the data features" [8] to reconstruct the dependence between the features and the respective labels for each sample. To perform classification or regression, Decision Trees rely on recursive splitting of the dataset into multiple subgroups. As the number of

iterations increases, the subgroups become more and more homogeneous [9, p.330]. The ideal result is that each subgroup is fully homogeneous and therefore only represents a single category (in case of classification). However, this is often only a theoretical best condition, as multiple risks, such as overfitting, are associated with the increasing depth of Decision Trees.

Trees consist out of four main components. A node is a discrete decision function that takes samples as its input and splits them based on features into subgroups. The aim of each split, as previously discussed, is to create a split that results in the overall most homogeneous distribution for all subgroups [10, p.6]. Nodes can be subclassified into three kinds. The top-node, from which the classification starts, is called a root node. Nodes that are located at the very end of a Decision Tree are referred to as leaves. Leaves do not split data any further and only mark the end of a decision tree. When reached, leaves categorize or predict a final output value depending on the prediction task. Nodes in between the root node and leaves are called internal nodes. Like the root, internal nodes are responsible for the recursive splitting of the data. Branches connect nodes with each other [10, p.4]. For classical trees, information only flows from top to bottom of the tree.

In practice, various implementations for computing decision trees exist. Each follows the same principle of regressively finding perfect splits to seperate the data, but uses different methods to find the ideal splitting criteria, which strongly influences the structure of the tree, its accuracy and performance. Thus, each implementation has benefits and constraints which have to be taken into account when determining the model. This project uses the Python library sklearn to implement a classification tree. Sklearn is based on the Classification and Regression Tree (CART) algorithm [8].

2.2.1 Decision Tree Algorithm

Initial Dataset:

To better visualize the procedure of the decision tree algorithm, a simplified dataset is used, on which the individual steps are explained. For this example, a classification problem is chosen. The dataset consists of actual features and labels from the project implementation phase. The features are *acousticness* and *danceability*. Both features are numerical with a value range in between 0 and 1. The classification problem is binary with *hiphop* and *jazz* representing the classes k for which the samples of the dataset are classified. Mathematically, the dataset is represented in the following form: x_i presents a set of explanatory features while y_i represents the corresponding label for one data point of the input dataset N with a total number of n samples.

Table 1: Input dataset

category	track	feature_danceability	feature_acousticness	label
hiphop	h1	0.949	0.132	1
hiphop	h2	0.743	0.234	1
hiphop	h3	0.913	0.394	1
hiphop	h4	0.810	0.504	1
hiphop	h5	0.434	0.198	1
jazz	j1	0.654	0.534	0
jazz	j2	0.593	0.312	0
jazz	j3	0.234	0.341	0

Recursive Tree Construction:

The decision tree algorithm splits the dataset recursively into subsets. At the beginning, the whole dataset is contained within the root node from which the division into subtrees starts. Each node is mathematically described as Q_m and contains a subgroup N_m of the input dataset. Nodes split their respective groups of data further or act as a final classifier in the form of a leaf. The split can be binary or multiway. CART only utilizes binary splits and divides the node Q_m into two repective subgroups labeled Q_m^{left} and Q_m^{right} [8].

The algorithm is greedy and thus determines an optimal division of the samples according to the impurity of both subgroups for each split. The goal for every split is to divide the group of data into two subgroups, which are more homogeneous than the origin and maximize the overall homogenity. The division of the data is always performed using a feature, on the basis of which the samples can be divided either by a threshold for nominal values or an is-equal-to query for categorical values. The split is mathematically described as the following equation (1). It consists of the the feature j and a criteria t_m [8].

$$\theta = (j, t_m) \tag{1}$$

Splitting Criteria and Information Gain:

The optimal split is determined by selecting the feature from which the most information can be gathered. There are multiple approaches for finding this split with the most common one being the information gain. Ideally the gain should be maximised. A simplification for the information gain is presented in (2). To receive information gain, $G(Q_m, \theta)$ must be subtracted from the entropy of the parent node [11, p.613f]. The simplification is to find the minimum $G(Q_m, \theta)$ as stated in (3). Both approaches lead to the same result [8].

$$G(Q_m, \theta) = \frac{N_m^{left}}{N_m} H(Q_m^{left}, \theta) + \frac{N_m^{right}}{N_m} H(Q_m^{right}, \theta) \tag{2}$$

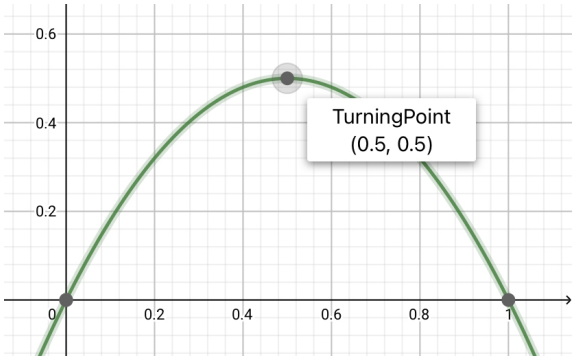
$$\theta^* = \arg\min_{\theta} G(Q_m, \theta) \tag{3}$$

To determine the overall best $G(Q_m, \theta)$ the respective impurities H for both subgroups Q_m^{left} and Q_m^{right} are required. Each impurity is weighted according to its relative size $\frac{N_m^{left}}{N_m}$.

The impurity H for CART is calculated by the gini index (4) [11, p.613f]. Gini index measures the probability that a sample does not belong to the category that represents the majority of the subgroup [9, p.335]. If both categories of a subgroup are identical in size, the gini index reaches its maximum point at $0,5$ (figure 1). The maximum of the gini index means the worst possible data constellation for a subgroup with maximum heterogeneity. Gini index equal to 0, on the other hand, represents the best possible result with the subgroup being fully homogenous. p_i represents the probability that a sample belongs to the class j [9, p.335].

$$Gini = 1 - \sum_{j=1}^{k} (p_j)^2 \tag{4}$$

Figure 1: Gini Graph



The optimal splits for the first iteration of the dataset look like the following. For numerical features the determination of the optimal split is more complex as for categorical ones because the calculation is not straight forward but must be computed for every value in the value range to find the best overall split for a single feature. The example only shows the best splitting criteria while a trial and error approach can be found in Appendix 4.

Danceability:

$$Gini^{left} = 1 - ((\frac{2}{5})^2 + (\frac{3}{5})^2) = 0,48$$
$$Gini^{right} = 1 - ((\frac{3}{3})^2 + (\frac{0}{3})^2) = 0$$
$$G(Q_0, \theta) = \frac{5}{8} * 0,48 + \frac{3}{8} * 0 = 0,30$$

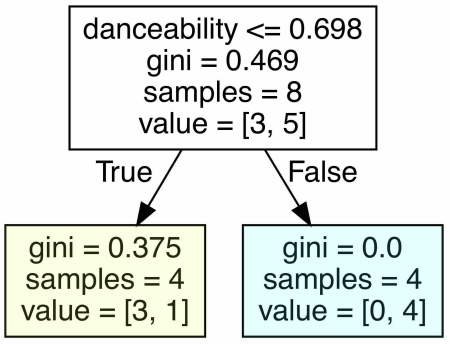
Acousticness:

$$Gini^{left} = 1 - ((\frac{4}{4})^2 + (\frac{0}{4})^2) = 0$$
$$Gini^{right} = 1 - ((\frac{1}{4})^2 + (\frac{3}{4})^2) = 0,36$$
$$G(Q_0, \theta) = \frac{4}{8} * 0 + \frac{4}{8} * 0,36 = 0,18$$

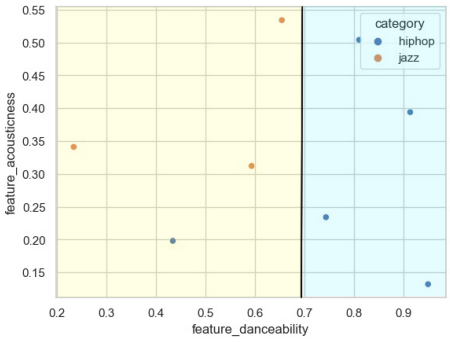
The conclusion is that the feature *danceability* creates a bettter initial prediction than *acousticness* does. It is thus determined to be the inital splitting criterion for the classification tree (figure 2). With the first iteration completed, the next iteration will start with each leaf as its input. The second iteration will split the nodes further if no stop criterion is met.

Figure 2: Decision tree first split

(a) Decision Tree



(b) Classification Tree



Stop Criteria:

The natural stop criterion is a completely homogeneous data-group for a node. If a node only contains one single class of samples, no further splitting is possible. This is the case for the right node of the example marked in blue. The samples are completely homogeneous and only consist of the category *hiphop*. The node therefore automatically becomes a leaf and represents the category of samples. Other stop criteria can be predefined depending on own preference. The most relevant criterion is the definition of a maximum depth of the decision tree. Maximum depth describes a maximum amount of recurrent

iterations before the algorithm automatically stops and treats the last nodes as leaves [10, p.7]. Maximum depth belongs to a set of hyperparameters that will be discussed in section 3.4.

Completion of the example:

The second iteration has both nodes created by the first iteration as its input but only the yellow node requires further splitting as the blue node is already fully classified. For the second iteration only the feature *acousticness* is relevant as *danceability* cannot split this subgroup of the dataset further. The input data for the second split looks like the following (table 2) and the same calculation takes place again.

Table 2: Input data of second split

category	track	feature_danceability	feature_acousticness	label
hiphop	h5	0.434	0.198	1
jazz	j1	0.654	0.534	0
jazz	j2	0.593	0.312	0
jazz	j3	0.234	0.341	0

Acousticness:

$$Gini^{left} = 1 - ((\frac{0}{1})^2 + (\frac{1}{1})^2) = 0$$
$$Gini^{right} = 1 - ((\frac{3}{3})^2 + (\frac{0}{3})^2) = 0$$
$$G(Q_1, \theta) = \frac{1}{4} * 0 + \frac{3}{4} * 0 = 0$$

This time *acousticness* leads to two fully homogeneous subgroups and therefore marks the end of the classification tree as all samples are classified correctly. The second split looks like the following with the total classification tree displayed in figure 3.

Figure 3: Decision Tree second Split

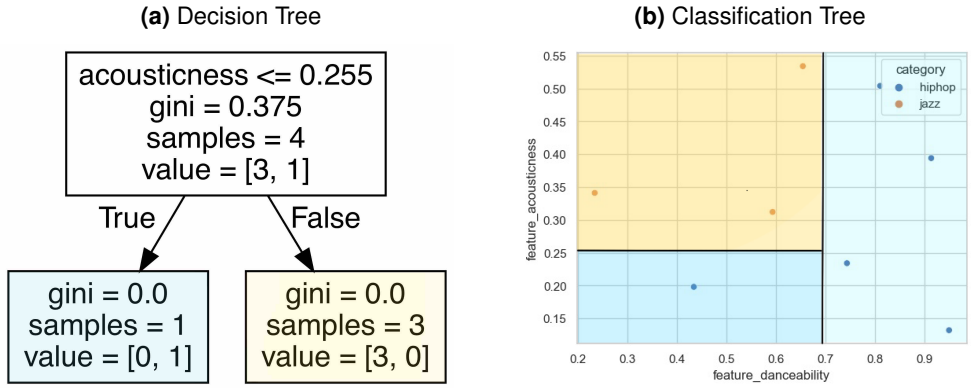
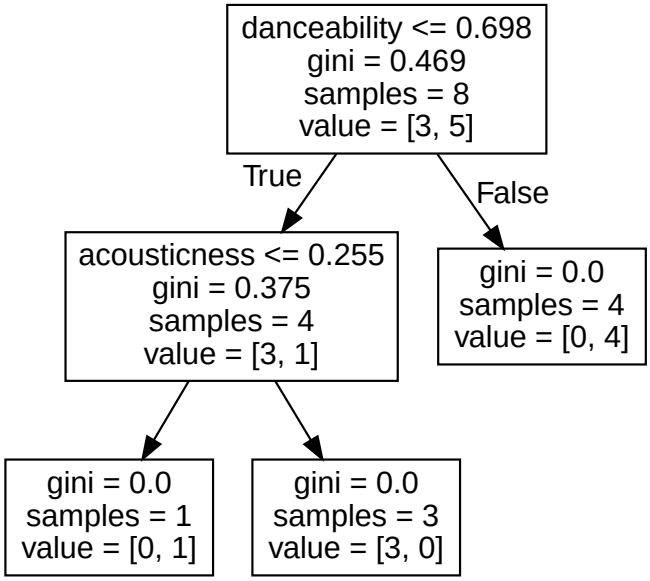


Figure 4: Final Classification Tree



2.2.2 Evaluation of Decision Trees

In conclusion, decision trees can be assessed as follows. Starting with the advantages, the main benefit is the overall simplicity of decision trees, both from a technical and business point of view [9, p.339]. For researchers and developers, trees are easy to construct,

require little to no data preparation, are almost universally applicable with the possibility of validation. However, the simplicity for business should not be underestimated either. When comparing machine learning algorithms, the main comparison is often the accuracy of a model. The areas in which decision trees stand out include visualization and comprehensibility. The decision tree algorithm is a white box model that allows complete transparency and explainability [8, p. 10.10.].

The disadvantages of decision trees are again closely related to their simplicity. Overfitting and the relative instability of decision trees are the main drawbacks and result in good memorization but a comparatively weak generalization ability [9, p.340] [8, p. 10.10.].

2.3 Gradient Boosting

The Gradient Boosting Algorithm is derived from Gradient Boosting Machine (GBM), which are a family of powerful machine learning algorithms with a certain procedure pattern for the creation of models. In general, GBMs are very flexible in their characteristics with the possibility of utilizing multiple different Machine Learning algorithms as their foundation [12].

Boosting differs from classical approaches as it does not consist out of a single predictive model but an ensemble approach. Ensemble algorithms contain multiple weak learners that form a committee to create a strong prediction. Weak learners are often very simple forms of traditional algorithms, like decision trees, and must just be able to predict parts of the dataset correctly. Only the combination of many weak learners allows the model to perform overall accurate predictions [13] [14, p.1f]. The most common form of ensemble algorithms are bagging algorithms with random forests as an example. Bagging, in essence, is the combination of multiple unique models. The prediction is formed by aggregating the outputs from all models into a single representative value. Typically, all models are derived from a single algorithm, like decision trees for random forests, but technically there is no limitation to aggregate outputs from different algorithms. Which is also true for other ensemble algorithms [14, p.2].

Boosting, on the other hand, follows a different principle and does not rely on independent models with an aggregation function. Boosting fits new models sequentially and can thereby use earlier acquired knowledge for further iterations. This allows a GBM to train specific areas of the dataset where it has previously performed poorly [14, p.11] [9, p.345f].

2.3.1 Gradient Boosting Algorithm

The generic Gradient Boosting Algorithm follows a sequence of three steps beginning with the initialization of the dataset, to the sequential training and ending with a final output. At the beginning, an additional initiation of the dataset and a loss function is necessary [15]. The mathematical representation of the dataset is like the one used for decision tees. A summary: x_i represent the explanatory features while y_i represents the corresponding label for one data point of the input dataset N with a total number of n samples.

The mathematical goal of the algorithm is to reconstruct the unknown functional dependance f between x_i and y_i with an estimate F^* for every data point, such that the specific loss function $L(y, \gamma)$ is minimized (5) [16, p.1189] [12, p. 2.1].

$$F^* = \arg\min_{\gamma} L(y, \gamma) \tag{5}$$

Loss Function and Weak Learners:

The loss function is an indicator for the quality of the model. A small loss for a data point means that the prediction is close or identical to the observed label and the model therefore categorizes the sample correctly whereas a high loss implies that the model could not predict the sample well. Given a particular learning task and dataset, different loss functions must be considered as loss functions are only suitable for specific data and task constellations. The most common loss function for binary classification is the so-called bernoulli loss (6). The bernoulli loss can be transformed into a $log(odds)$-prediction (7) as it is better suited for further calculations [12, p. 3.1] [17]. Variations of (7) will be used in the following section to demonstrate the gradient boosting procedure.

$$L(y_m, \gamma) = -log(likelihood) \tag{6}$$

$$L(y_m, \gamma) = -y_i * log(odds) - log(1 - p) \tag{7}$$

Additionally, a machine learning algorithm must be defined as a weak learner. For GBMs there are multiple learners to choose from. Again the choice mostly depends on the prediction task and available data [12, p. 3.2]. A classical approach is the use of Decision Trees, which was also chosen as the weak learner for this project. Decision Trees used for Gradient Boosting are always Regression Trees, regardless of whether they are used for regression or classification problems. The optimization parameters are almost identical

to the ones of standalone Decision Trees, but the trees often look very different because they are specifically created as weak learners. As a result, the Decision Trees often only consist of very few layers with only 8-32 leaves.

Initialization:

To showcase the Gradient Boosting Algorithm the same sample dataset is used as for Decision Trees. It again consists of eight samples with two features and two categories as target labels.

Table 3: Input dataset

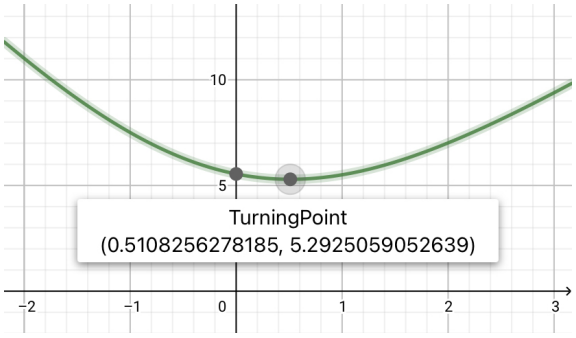
category	track	feature_danceability	feature_acousticness	label
hiphop	h1	0.949	0.132	1
hiphop	h2	0.743	0.234	1
hiphop	h3	0.913	0.394	1
hiphop	h4	0.810	0.504	1
hiphop	h5	0.434	0.198	1
jazz	j1	0.654	0.534	0
jazz	j2	0.593	0.312	0
jazz	j3	0.234	0.341	0

The first step is to set an initial prediction for all samples of the dataset. The initial prediction is not unique for individual samples but a uniform value. The optimal initial prediction can be calculated using the following equation (8) [18, p.361]. For $F_0(x)$, representing the initial prediction, a minimum of γ is searched for. The right-hand side of the equation only consists of a sum for each sample i (of the total dataset N) of the known loss function with the respective label y_i and γ as its input. To find the low point of the equation, the derivate of the loss function is required. The final calculation of the overall $log(odds)$ that a song is classified as *hiphop* is the $\log_e$ of the sum of songs of the category *hiphop* divided by the sum of the songs of the category *jazz*. The result can be checked graphically as is equal to the $x1$-coordinate value of the low point of the $log(odds)$-prediction (figure 5).

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma) \tag{8}$$

$$F_0(x) = \log_e(\frac{5}{3}) = 0.51$$

Figure 5: Gradient boosting minimum for first prediction



Sequential Processing:

With the completion of the initialization of the dataset the modeling can begin. The training is a sequential process of constructing regression trees with a total of M iterations. The first iteration starts with $m = 1$. The modelling consists out of four sub-steps which are numbered consecutively [16, p.1198].

First, the $log(odds)$-prediction must be converted back into a probability p with the help of a logistic function as probability is easier to use for classification (9). The result is that all songs have a 63% chance of belonging to the category *hiphop*.

$$p = \frac{e^{\log_e(odds)}}{1 + e^{\log_e(odds)}} \tag{9}$$

$$p = \frac{e^{\log_e(\frac{5}{3})}}{1 + e^{\log_e(\frac{5}{3})}} = 0,63$$

In the **first step** the Pseudo Residuals (PRs) r_{im} for each sample i of the dataset are created. The equation (10) for calculating the PR consists out of known fragments. For every sample i a PR r_{im} is calculated using the derivative of the $log(odds)$-prediction with the label y_i and the prediction of the last iteration $F = F_{m-1}$ as its input [18, p.361]. Again, the equation can be simplified greatly. For each sample the PR can be calculated by only subtracting the previously calculated probability p from the observed label y [15] [19]. Ideally, an additional column is created in which the PRs are temporarily stored (table 4).

$$\begin{aligned} r_{im} &= -(\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)})_{F=F_{m-1}} \\ r_{im} &= (y_i - p_i) \end{aligned} \tag{10}$$

This and every following calculation is explained for the track *h1* of the dataset while the result for every other sample will be shown in form of a table.

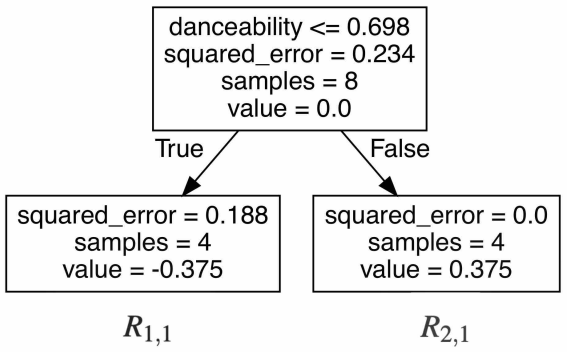
$$r_{1,1} = 1 - 0.625 = 0.375$$

Table 4: Pseudo Residuals for first Iteration

category	track	label	$F_0(x)$	probability $F_0(x)$	pseudo residuals
hiphop	h1	1	0.510826	0.624988	0.375012
hiphop	h2	1	0.510826	0.624988	0.375012
hiphop	h3	1	0.510826	0.624988	0.375012
hiphop	h4	1	0.510826	0.624988	0.375012
hiphop	h5	1	0.510826	0.624988	0.375012
jazz	j1	0	0.510826	0.624988	-0.624988
jazz	j2	0	0.510826	0.624988	-0.624988
jazz	j3	0	0.510826	0.624988	-0.624988

The **second step** constructs the a Regression Tree out of the features from the samples with the corresponding PR as the label. For this example, only tree stumps are created to simplify the implementation. The regression tree for the first iteration is shown in figure 6. After the completion of the tree, terminal regions R_{jm} must be defined for every leaf. j starts with 1 and is increased for every leaf [16, p.1195].

Figure 6: Regression tree for first iteration



In **step three**, following the completion of the regression tree, output values $\gamma_{j,m}$ are calculated by using the equation presented in (11). For each leaf in the tree, $\gamma_{j,m}$ is computed by finding $\gamma_{j,m}$ that minimizes the loss function (11) [18, p.361]. Like in the initialization step, the derivative has to be created and set equal to 0. Again, after a complicated transformation, a very simple equation remains (11). The $\gamma_{j,m}$ can be calculated using only the PR and the most recently predicted probabilities p for all samples in the leaf.

$$\begin{aligned} \gamma_{jm} &= \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma) \\ \gamma_{jm} &= \frac{\sum r_{im}}{\sum p_i * (1 - p_i)} \end{aligned} \tag{11}$$

For $R_{2,1}$, which contains track *h1*, the calculation looks like the following. Output values of the complete dataset are displayed in table 5.

$$\gamma_{2,1} = \frac{0.375 * 4}{(0.625 * (1 - 0.625)) * 4} = 1.6$$

Table 5: Output values for first iteration

category	track	label	$F_0(x)$	pseudo residuals	output value
hiphop	h1	1	0.510826	0.375012	1.600032
hiphop	h2	1	0.510826	0.375012	1.600032
hiphop	h3	1	0.510826	0.375012	1.600032
hiphop	h4	1	0.510826	0.375012	1.600032
hiphop	h5	1	0.510826	0.375012	-1.599926
jazz	j1	0	0.510826	-0.624988	-1.599926
jazz	j2	0	0.510826	-0.624988	-1.599926
jazz	j3	0	0.510826	-0.624988	-1.599926

Step four marks the end of the first iteration and creates a new prediction $F_m(x)$ for each sample. The new $log(odds)$-prediction is based on the last $log(odds)$-prediction plus the learning rate ν multiplied by the output values for the sample of the last regression tree (12) [16, p.1203]. Normally, there is only one output value for a sample which makes the summation sign obsolete. The learning rate ν is a hyperparameter for gradient boosting. For this example, a high ν of 0.6 is used to better visualize the changes. In practice a ν in the order of 0.1 is common as a small learning rate tends to give better results but also requires much more iterations [16, p.1206].

$$F_m(x) = F_m(x-1) + \nu * \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm}) \tag{12}$$

The prediction $F_1(x)$ for *h1* is shown below while the complete dataset is again displayed in table 6 with also the probabilites shown. As one can see, the probabilites for most samples got closer to the label. However, for track *h5* the prediction got worse. This is why multiple iterations are necessary to form a well-founded prediction.

$$F_1(x) = 0.51 + 0.6 * 1.6 = 1.47$$

Table 6: Gradient boosting complete

category	track	label	$F_0(x)$	probability $F_0(x)$	$F_1(x)$	probability $F_1(x)$
hiphop	h1	1	0.510826	0.624988	1.470845	0.813163
hiphop	h2	1	0.510826	0.624988	1.470845	0.813163
hiphop	h3	1	0.510826	0.624988	1.470845	0.813163
hiphop	h4	1	0.510826	0.624988	1.470845	0.813163
hiphop	h5	1	0.510826	0.624988	-0.449130	0.389579
jazz	j1	0	0.510826	0.624988	-0.449130	0.389579
jazz	j2	0	0.510826	0.624988	-0.449130	0.389579
jazz	j3	0	0.510826	0.624988	-0.449130	0.389579

The second iteration of the modelling phase follows the exact same principles and will again be showcased for track *h1*. It marks the end for this example.

- New probability:

$$p = \frac{e^{1.47}}{1 + e^{1.47}} = 0,81$$

- New pseudo residuals:

$$r_{1,2} = 1 - 0.81 = 0.19$$

- New Decision Tree with terminal regions (figure 7)
- New output value:

$$\gamma_{2,2} = \frac{0.98}{0.54} = 1.81$$

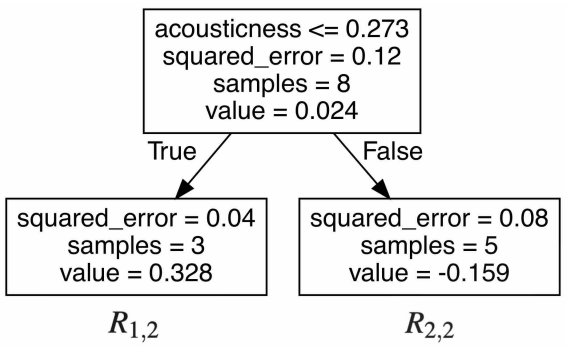
- New prediction:

$$F_2(x) = 1.47 + 0.6 * 1.81 = 2.56$$

- New probability:

$$p = \frac{e^{2.56}}{1 + e^{2.56}} = 0.93$$

Figure 7: Regression Tree for second Iteration



The iterative modeling is repeated until M is reached. M marks the completion of training for the Gradient Boosting model. $F_M(x)$ is the final prediction for every sample. Lastly, the predictions must again be transformed to probabilites like for the calculation of the PR. For the final probabilities, thresholds are used to compute the category to which the samples belong [16, p.1204]. A typical threshold for binary classification is 0.5 as it splits $F_M(x)$ into two equal classes.

Output:

Sample *h1* has a final probability 0.93 of belonging to the category *hiphop*. 0.93 is by far greater than the predefined threshold of 0.5 and thus *h1* is classified as a track that belongs to the category *hiphop*. Table 7 shows that all samples are classified correctly but some less clear than others. A real-world implementation would consist out of more iterative models with different regression trees and a different learning rate to enhance the classification potential. Also the dataset would be very different with more samples that consist out of more features.

Table 7: Gradient boosting Final Output

category	track	prob. $F_0(x)$	$F_1(x)$	prob. $F_1(x)$	$F_2(x)$	Output Probability
hiphop	h1	0.624988	1.470845	0.813163	2.560923	0.928286
hiphop	h2	0.624988	1.470845	0.813163	2.560923	0.928286
hiphop	h3	0.624988	1.470845	0.813163	1.001911	0.731414
hiphop	h4	0.624988	1.470845	0.813163	1.001911	0.731414
hiphop	h5	0.624988	-0.449130	0.389579	0.640948	0.654953
jazz	j1	0.624988	-0.449130	0.389579	-0.918064	0.285372
jazz	j2	0.624988	-0.449130	0.389579	-0.918064	0.285372
jazz	j3	0.624988	-0.449130	0.389579	-0.918064	0.285372

An unknown sample gets initialized by $F_0(x)$ and is sequentially routed through every model. For each iteration the prediction is updated using the output value of the regression tree. Finally, the last probability is used to classify the sample using the predefined threshold.

2.3.2 Evaluation of Gradient Boosting

Gradient Boosting is a very powerful method as it can effectively capture complex dependencies for various machine learning problems. GBMs improve many existing algorithms, such as decision trees, because many problems associated with single large models are (partially) solved by the iterative ensemble approach.

The main benefit over Decision Trees is the stability of Gradient Boosting. While large trees always have to make tradeoffs between detail and overcategorization, Gradient Boosting can successively get to deeper levels of detail thanks to small trees with overall better generalization. Furthermore the flexbility of Gradient Boosting and boosting in general is massive as it only represents the framework with many parameters to adapt the algorithm very specifically to the usecase [12, p. 7.2]

The drawbacks of Gradient Boosting often arise in practice. Gradient Boosting has a significantly higher memory consumption and build time, because the model must be constructed sequentially. Also the evalutation is more time consuming as the sample must be processed by each model. From a business perspective Gradient Boosting also has disadvantages. While the overall prediction is better, it is more complex to evaluate and explain [12, p. 7.2] [14, p.27].

2.4 Concepts for Data Preparation and Modeling

In this section, concepts and processes, which are applied during data preparation and modeling, are discussed theoretically. The practical implementation is shown in sections 3.3 and 3.4.

2.4.1 Mean Removal, Variance Scaling and Standardization

During the data preparation stage of model development, it can be helpful to transform the data into a different shape to improve the results of a model [20, p. 35]. This subsection explains some methods of data transformation, which are used in the implementation section.

Mean Removal is the process of shifting the data in a column, so that the mean $\bar{x}$ of all datapoints in the column is equal to 0 [21]. An example set of integers $a_1 = \{6, 7, 2, 1\}$ has a mean of $\overline{x_1} = 4$. Removing it's mean without distorting other properties of the data entails subtracting 4 from each member of the set, resulting in a new set $a_2 = \{2, 3, -2, -3\}$ with $\overline{x_2} = 0$. The dataset is now centered around 0.

Variance Scaling involves transforming the data in a way, that each column has unit variance, meaning variance $\sigma^2 = 1$ [21]. This is done by dividing each member of the column by the column's standard deviation. An example set of integers $b_1 = \{6, 7, 2, 1\}$ has standard deviation $\sigma_1 = 2.94$ and variance $\sigma_1^2 = 8.67$. Dividing each member of b_1 by σ_1 results in the set $b_2 = \{2.04, 2.38, 0.68, 0.34\}$ with variance $\sigma_2^2 = 1$.

Standardization involves performing both mean removal and variance scaling on a column [21]. Standardizing an example set $c_1 = \{6, 7, 2, 1\}$ would result in the following set $c_2 = \{0.68, 1.02, -0.68, 01.02\}$. Standardized features are normally distributed. Many Machine Learning Algorithms have improved results when receiving standardized input data. It has the additional benefit of equalizing the impact of features while keeping valuable information about the outliers and value range [20, p. 35].

2.4.2 Dimension Reduction

Dimension reduction refers to reducing the number of dimensions in the input data, while keeping the highest possible amount of information from the original dataset [20, p. 53]. The benefit of this is that some features might not contribute to a better model, but instead

promote bias and overfitting. There are multiple ways to reduce dimensions, such as feature selection, where some features, which don't contribute to a better model, are simply removed [20, p. 55]. New features can be constructed by using data from different original features and combining it into one, also reducing dimensionality. Another approach is Principle Component Analysis (PCA). Here, features are constructed by finding principal components in a dataset [20, 62f]. This is done by projecting the data into m-dimensional space, where each feature is contained in one spatial dimension. The first principal component is found by rotating the feature space to find the direction which offers the highest variance in the whole dataset. The second component must be orthogonal to the first one and again have the highest variance of all possible directions. This can be repeated to create as many principal components as are benefitial to the model [20, 62f]. Dimension reduction using PCA was attempted as part of this project, but is not explained deeper as it didn't yield great result for the dataset.

2.4.3 Hyperparameter Optimization

Machine learning models have two types of parameters: model parameters and hyperparameters. Model parameters are values, which are optimized using training data during the training process. They are not specified in advance [22, p. 21]. Hyperparameters, on the other hand, are not optimized using data, but are specified before training. They are like "settings" for the estimator **scikit-hyperparameters**.

Even though hyperparameters are not optimized during training, there are multiple ways to find the best set of hyperparameters. In this project, grid search is used, which takes a parameter grid containing multiple values for each hyperparameter as input **scikit-hyperparameters**. A model is trained using every possible combination of all given values on the grid and a score is calculated for each model. The best combination of hyperparameters are the ones used on the model with the highest score. This might not be the optimal set of hyperparameters though, as only values in the search grid are considered.

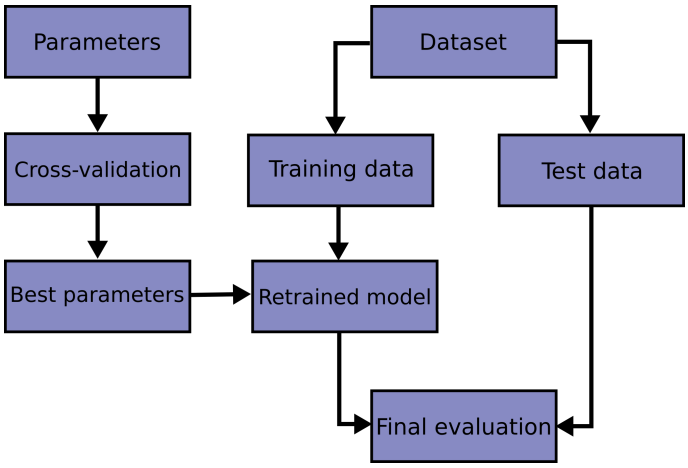
2.4.4 K-fold Cross Validation

When training a model, it is necessary to split the data into train and test samples. Using the same data for training and calculating the accuracy score of a model would result in a model, that is fitted exactly to the training data. It will perform very well on that data, but will fail to make predictions on any data it has not yet seen. This is called overfitting [22, p. 7].

When optimizing the hyperparameters of an estimator, one could use the test set to score the model and find the best parameters. A problem with this approach is, that now test data, which should be completely independent from the training process, is involved in the optimization process, which could lead to the hyperparameters being "fitted" to the test data **scikit-cross-validation**.

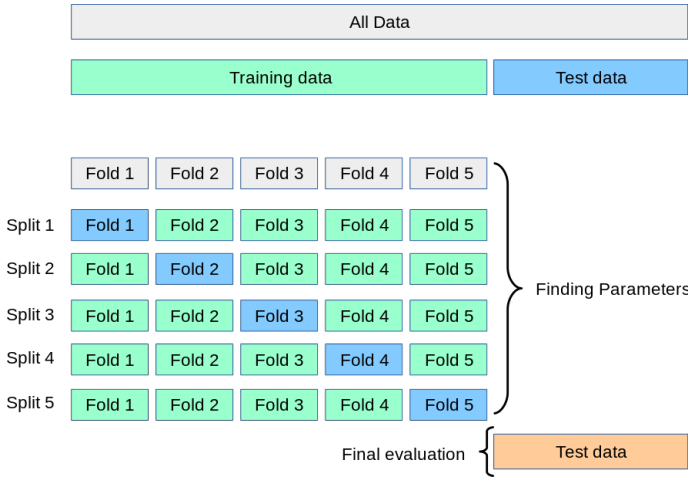
Cross Validation (CV) is a solution to this problem. During optimization, the training set is split into k groups, named folds. A given estimator is trained on $k-1$ folds and its accuracy is calculated using the samples in the remaining folds. This is done k times, using a different fold for scoring each time. The average of all scores is the cross-validation score, giving a good representation of the overall performance of the estimator. The benefit of this is, that test data is kept completely seperate from training data, but there is still a good way of evaluating the performance of an estimator trained with a given set of hyperparameters. This is crucial during hyperparameter optimization, as the best parameters can not be found without a good way to score each model [22, 24f] **scikit-cross-validation**. Figure 8 shows a typical workflow when using cross validation and hyperparameter tuning to create a model. Figure 9 shows, how the data is segmented when using 5-fold cross validation on the training set.

Figure 8: Typical Cross Validation Workflow



Source: **scikit-cross-validation**

Figure 9: Data Segmentation for Cross Validation



Source: **scikit-cross-validation**

2.5 Cross Industry Standard Process for Data Mining

CRISP DM is an organizational model, which provides an overview of the life cycle of data mining projects. This cycle is divided into six phases: business understanding, data understanding, data preparation, modeling, evaluation and deployment. The order of the phases is not strictly adhered to. In certain projects, the outcome of each phase determines which phase or which specific task in a phase must be performed next [23, p. 528]. However, data mining is not necessarily complete once a solution is implemented. After deployment, new insights are gained from the models results and additional data collection, which can be used to further improve the implementation [23, p. 529].

The process starts with business understanding, which is an in-depth analysis of business objectives and requirements. These insights allow an assessment of the current situation and the definition of goals [24]. With business understanding complete, data is collected during the data understanding process. The collected data could come in the form of existing database entries, answers from questionnaires or machine log entries, to name a few examples. The gathered data is aggregated and examined for usability. Insufficient or bad data can cause a model to produce unsatisfying or even misleading results [25]. In the data preparation step, the data is selected, cleaned, formatted and preprocessed to ensure a high quality dataset for modeling. This is usually the most costly and time intensive phase of the project, but is crucial for its overall success [25].

In the next stage, an appropriate model must be developed to generate a result, which satisfies the requirements defined during business understanding. During modeling, it

is important to constantly present interim results to the management using meaningful visualizations and subsequently improve the model using further input.

During the step of evaluation, the results of the final model are evaluated in the context of business intentions. This might lead to new insights for the organization and in turn alter the goal. This is an iterative process ending in a final decision to deploy a certain model [23, p. 530]. Depending on the requirements, the deployment phase could involve creating simple reports, altering systems or complex data visualizations with a constant reiteration of the data mining process [23, p. 532].

2.6 Web Application Programming Interfaces

This section gives an overview over the basic concepts and technologies behind Web APIs.

2.6.1 Use Cases and Types of Web APIs

An API in general is an interface between two pieces of software[26, S.1]. It abstracts complex functionality away and provides a simple interface to interact with to retrieve data and services[27]. Web APIs are a specific form of API that can be accessed over the internet using the HTTP protocol.[28] They are used to power desktop, web and mobile applications. pass data between different services, systems and devices. They make automated data access and retrieval possible and enable the integration of internal and external systems into processes and data flows.[29]

There are different types of Web APIs, mainly Open/Public APIs, Internal APIs and Partner APIs[28]. Open APIs are provided by a company, government or other organization and make it possible for any developer to access that organizations data or services[28]. Internal APIs are only used to share resources within one organization, while partner APIs are exposed to the public, but are only open to specific users, which might pay for the right to use the API[28].

Spotify offers a Public Web API, which is used in this project in the context of data collection (section 3.1). In order to understand the specifics of data collection, some concepts regarding the Hyptertext Transfer Protocol (HTTP) protocol are explained in the following section.

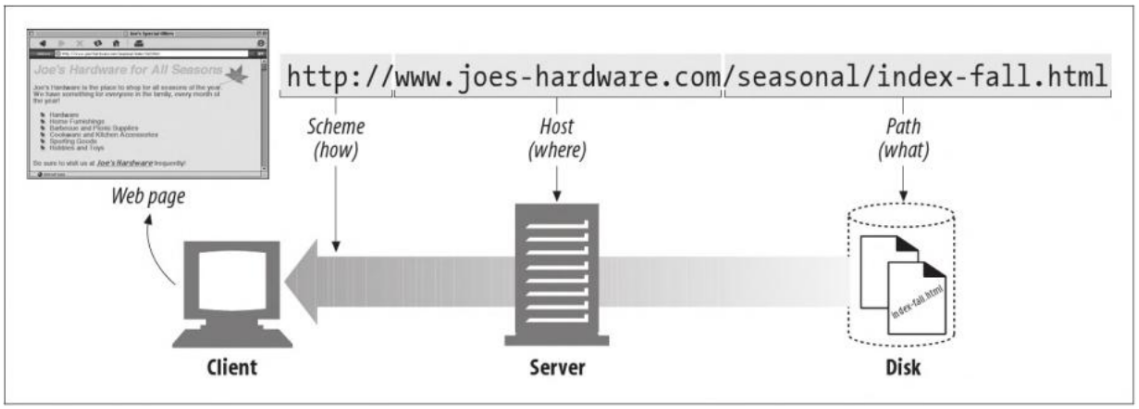
2.6.2 The HTTP Protocol

All communication on the world wide web is conducted through HTTP because of its reliability and guarantee for data integrity[30, 3f.].

Most communication on the Web is conducted between a client and a server. The client usually requests a resource from the server and the server responds with that resource, which the client can then use[30, p. 4].

To tell the server, which specific resource it wants, the client uses a Uniform Resource Locator (URL), which uniquely identifies a resource on the web. Figure 10 shows how URLs are structured. If the server knows where to find the resource and the client is authorized to see it, the server will send it to the client. In the context of APIs, a URL pointing to a specific resource is also called an "API endpoint".[31] This interaction between client and server is usually done in an HTTP transaction.

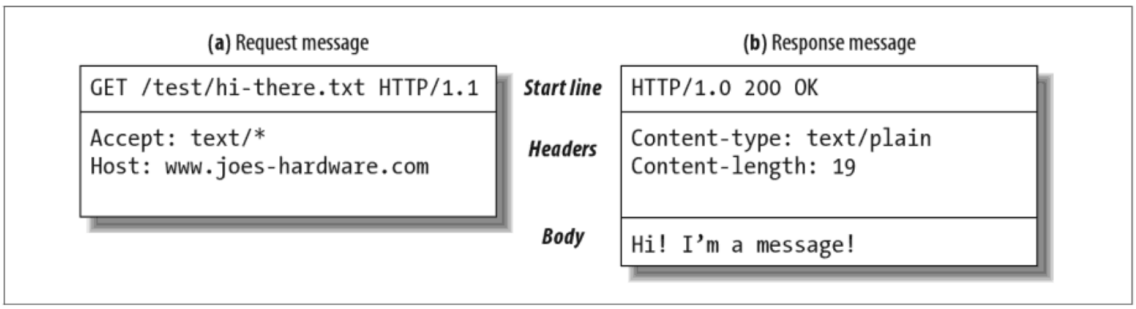
Figure 10: Description of different parts of a URL



Source: [30, p. 24]

A transaction consists of a request (sent from client to server) and a response in which the server answers the clients request in some way.[30, p. 8] In the context of Web APIs, a request made to an API is also called "API call".[28] Figure 11 shows the structure of an HTTP request and response message.

Figure 11: Example HTTP request and response messages



Source: [30, p. 47]

Every HTTP request message starts with a method. There are many HTTP methods defined, that can be used to delete a resource, store data on the server, get header information for a document and more.[30, p. 48] In this paper, only two HTTP methods are used, GET and POST. A GET request is used to ask the server to send a resource to the client. It does not contain a body.[30, p. 48] A POST request message is used to send data to the server for processing. The data to be processed is sent in the request body.[30, p. 48]

A request also includes the URL, protocol information and a set of headers, delivering additional information about the client [30, p. 47].

Response messages contain a status code instead of a method or URL.

Every HTTP request message has a start line. It contains a method, which tells the server what action to perform on the resource, the URL and some protocol information.[30, p. 47] After the start line, some headers are given. These contain information about the request, e.g. what types of data the client can accept or credentials for authorization. Some requests also contain a body to give some data to the server. [30, p. 52]

A response message does not contain a method or URL Instead it contains protocol information and status information in the form of a numerical status code and status text.[30, p. 48] A response also contains headers and might contain a body.[30, p. 52]

Status codes tell the client if the transaction was successful or what type of problem occured[30, p. 49]. There are many status codes defined in the HTTP standard, some of the most common ones are shown in table 8.

Table 8: Common status codes

Status code	Reason phrase	Meaning
200	OK	Successful transaction. Any requested data is in the response body
401	Unauthorized	The response needsto contain credentials to access the requested resource.
404	Not Found	The server cannot find a resource for the requested URL.

Source: [30, p. 50]

2.6.3 JSON

The Spotify API uses the JSON data format to transfer data to the client**SpotifyWebAPI**.

Most modern APIs transmit data to the client using JSON. It is a text-based format able to represent semi-structured data. [32] JSON objects are simple, have good readability and are easily processed by computers, which explains the wide usage of the data format[33]. JSON is tightly integrated in the JavaScript programming language and can be used without any parsing or serialization.[33] This is beneficial, as JavaScript is widely used both on the client and server side of web applications. JSON can also be used with many other programming languages. [34]

JSON supports only six very basic datatypes: String, Number, Boolean, Null, Object and Array.[33] The following code snippet shows how each of the datatypes are defined in JSON.

```
{
    "string": "Martin",
    "number": 200,
    "number2": 300,
    "boolean": true,
    "booleanFalse": false,
    "nullValue": null,
    "object": {
        "objectAttribute": "string",
        "objectAttribute2": "string2"
    },
    "array": [
        "entry1",
        "entry2",
        {
            "objectInArray": true
        }
    ]
}
```

The code snippet shows, that the whole JSON object is wrapped in curly braces. Each attribute is defined as an attribute-value-pair, where the attribute name is defined in double quotes, followed by a colon and the value definition. String values are defined in double quotes, numbers and booleans without. Objects are defined in curly braces and can again contain any datatype. Arrays are defined using brackets. They can contain any number of entries of any type. Types can also be mixed within the same array. All attributes-value-pairs or array items are seperated by commas. Indentation and newline characters are not parsed in JSON as all syntax is given with braces, colons, commas and quotation marks [34].

2.7 Basic Concepts of Music Theory

Music itself is a form of art that combines either vocal or instrumental sounds, sometimes both to express an emotion or convey an idea [35]. But despite being able to define music it can be quite difficult even for artists itself to describe it and they struggle to put their perception into words. Famous Tenor and Saxophonist of the 1950s and 1960s John Coltrane described Music as follows. "My music is the spiritual expression of what I am my faith, my knowledge, my being. When you begin to see the possibilities of music, your desire to do something really good for people, to help humanity free itself from its hang ups. I want to speak to their souls" [36].

2.7.1 Melodical Components of Music

Based on John Coltrane's quote, music could colloquially be described as something, "that is used to trigger human sense" [36]. To achieve this music consists of various elements that work together harmoniously to form a pleasant-sounding melody. Many factors are highly interdependent and correlate with each other.

- **Pitch and Melody**: Pitch is the word used to describe the highness or lowness of a musical sound. A series of pitches, also described as scales, is used to create a melody. Melodies can be derived from various scales for e.g., the traditional major and minor scales of music or even more unusual ones like the whole tone scale [37, p. 86]. Furthermore melodies can be described in two ways. Conjunct melodies are smooth and easy to sing or play. Disjunct melodies however represent the opposite. They are ragged or jumpy and difficult to sing or play.

- **Harmony and Chords**: Harmony can be described as the sound created when two or more pitches are performed at the same time to form a chord. Chord is simply the definition for three or more notes sounding at once. Harmony can be split up into two categories taking the sound of pitches into consideration. Harmony can sound consonant, which means the pitches sound pleasant together or dissonant, meaning the pitches sound unpleasant together [37, p. 94].
- **Rhythm**: Rhythm refers to the recurrence of notes and silences in time. A rhythmic pattern is formed by a series of notes and silence repeats. In addition to signify when notes are played, rhythm also defines how long notes are played and with what intensity. The difference in time creates varying note durations as well as different type of accents [38].
- **Texture**: Texture indicates the number of instruments or voices that are used to contribute to the density of the music. Texture can be split up into 3 types of monophonies, homophony and polyphony. Monophony describes a single layer of sound e.g. a solo voice. Homophony is a melody with an accompaniment which can be a lead singer with a band or a solo singer an a guitar or piano. Polyphony is a form of texture which consists of two or more independent voices. One voice forms the melody and the other forms a support role. This could be for example a lead singer with a choir [39].
- **Timbre**: Timbre, also known as tone colour or tone quality, can be described as the specific tone or quality an instrument or a voice has. Timbre helps to distinguish instruments from each other when playing the same melody or simple notes. For example, a “C” note on the piano and a sung “C” have the same pitch but different sound quality which gets differentiated by timbre. Timbre usually can be described with adjectives used to describe color, temperature, or the human voice e.g., warm, cold, metallic, harsh, dry [37, p. 94].

2.7.2 Lyrics and Instruments

Lyrics are the definition for the linguistic part of a song. They usually consist of verses and choruses and can be implicit or explicit. More or less every song uses unique lyrics but nevertheless we can categorize lyrics by topics they address [40]. Statistics have shown that the most common themes are growing up, friendship, atatements of discontent, heartbreak or death [41]. Another factor that must be taken into consideration when characterizing music are instruments. In the previous chapter we already talked about the fact that timbre is used to distinguish instruments from each other. Timbre focuses mainly

on the sound of the instruments to distinguish them. However, we can distinguish instruments not only by the way they sound but also by how the sound is produced in the first place. This is done with a classification of the instruments into 5 categories [42].

- Idiophones: Sound gets produced by the body of the instrument vibrating e.g., xylophones
- Membranophones: Sound produced by vibration of a tightly stretched membrane such as drums
- Chordophones: Sound produced by vibrating strings e.g., piano or cello
- Aerophones: Produce sound by vibrating columns of air such as the pipe organ or oboe
- Electrophones: Produce sound by electronic means e.g., synthesizers or electronic instruments.

2.7.3 Genres

One potential classification criterion would be the chronological categorization with the help of epochs. Due to the fact that modern music is diverse, epochs would be an insufficient solution. As a result genres were developed which include both chronolocial apects and elements of classical music theory. Genres combine songs that sound the same or emit the same feelings as well as songs that are based on the same characteristics of lyrics and instruments in songs. To further specify songs, genres again can be split into more homogeneous sub-genres [43].

Hip Hop

Although hiphop is often widely considered as a synonym for rap music it can rather be defined as somewhat of a cultural movement or a form of lifestyle that includes a music genre. Hiphop culture consists of various elements which are united under this umbrella term. Elements are djing/turntablism, rapping, beatboxing, breakdancing, and graffiti/visual art [44]. These typical street style elements or activities created a cultural revolution and shaped music styles, fashion, technology, art, and more [45]. Even to this day hiphop continues to develop new art forms that impact the lives of young and old generations. When it comes to the history of hiphop the culture has its origins in the 1970s where it first appeared in African American ghettos. Because of this heritage, hiphop still sees itself as street culture, and musicians who practice this form of music often have close

ties to gangs, clans, and poverty [46]. Basic elements of hiphop come from the genres of funk and soul, which are also influenced by African Americans. Especially because of the distinctive rhythms, these genres offered themselves very well as a basis. Through the development of hiphop, however, it is also becoming more and more unique in type and form. The main characteristic of the music of hiphop is the interaction between a rapper and a beat. Rarely real instruments are used for the music and beats are usually created by electronic instruments or samples, which resembles the use of already finished or existing sound/music recordings. Beats can be very diverse and can represent many moods, e.g., aggressive, relaxed, harsh, etc [44]. But the main focus in rap is usually on the rapper himself and even more on the lyrics. Rappers use rhythm, lyrics, and the timbre of their voice to express themselves. The artists use their voice as an instrument and more importantly different voice pitches to adapt to the intention of the lyrics. Rappers are often measured by the scene on their so-called "flow", the ability how quickly and how smoothly they can perform chants without errors. The themes of hiphop songs are often poverty, drugs, violence, struggles, or righteousness. Since many hip-hoppers come from ethnic and racial minorities, they often use personal life experiences for their lyrics. In addition, many songs also want to convey a message that often revolves around necessary political changes, social justice, or grievances in society [47].

Jazz

Jazz is a musical style in which improvisation plays a central role. In many jazz performances, artists often play pieces they just made up from their heads and perform them on spot. Jazz has its historical roots in the early 19th century America more specifically in New Orleans. The city was an ideal breeding ground for jazz music because of given cultural diversity. The percentage of ethnic minorities was much higher in this city than elsewhere in America, which is why it was often called a melting pot of cultures [48]. The biggest influence on jazz had the Afro-American culture as the music somewhat reflected the breaking away from previous rules of slavery and oppression. The improvisation of songs should act symbolically as a counter to the rules they had to deal with for a long time. Since its inception, jazz has gone through many different phases. The beginnings of jazz, from the 1910s to the 1920s, were characterized by small bands, often consisting of only a frontman and a few accompanists. The frontman often improvised pieces using a cornet or a trumpet, while the accompanists supported him with clarinets or trombones. In addition, often instruments like the banjo, piano, double bass, or drums were used to create a rhythm. In the 1930s and 1940s, the Swing and Big Band era emerged. Now, for the first time, singers appeared before big bands and bandleaders, and the clarinet was largely replaced by the saxophone. Moreover, the jazz epicenter shifted from New Orleans further and further to New York [49]. The 1950s and 1960s then introduced laid-back cool

jazz in contrast to the more fast-paced songs of the previous decades. Here a jazz quartet often played soothing and slow songs. With the introduction of electronic instruments in jazz from 1970 onwards, many subgenres were formed, such as jazz-rock. Until today, there are many forms of jazz, which are mainly oriented to the New Orleans origins and the 3 mentioned eras. Jazz has its main musical roots in the blues, but there are also elements of rock and classical music in it [50]. A distinctive rhythm is a key characteristic of jazz music, these are created mainly by "swinging" eighth notes. The so-called swinging is created by emphasizing one note of the eighth note pair while the second note is lighter and "swings" to the next note. Jazz is also very polyphonic, which means that many sounds are played simultaneously and as a result, various layers of harmony are laid over an initial basic melody. In addition to the use of classical European instruments such as the saxophone or trumpet, instruments such as drums, bass, keyboard, guitar, and trombone are often used. Some types of jazz also have front singers, but often pieces are played without vocals [51]. As already mentioned before the main characteristic of jazz is the spirit of improvisation. This unifies almost all forms of jazz music. Jazz artists attach great importance to imprinting their own sound and style on the music, so they usually even play their own songs slightly modified or with a distinct style. This leads to the fact that thousands of jazz recordings can be found for the same song, but they all sound different. Finally, it is important to mention that jazz can also reflect many different emotions. Everything from pain to joy is possible. For many People of Colour, it resembles the feeling of freedom, because for them, as mentioned above, jazz represents a strong voice against suffering, oppression, and injustice [52].

Rock

The musical genre rock is a popular music genre that combines elements of rhythm and blues, jazz and country music while adding electric instruments. It originated as Rock ‘n Roll in the late 1940’s and early 1950’s and of course also is constantly changing and evolving. The basis of rock formed the music genres blues, gospel and country. Early Rock ’N’ Roll came from cities like Memphis, Chicago, New Orleans or St. Louis. However, the genre spread very quickly throughout Western culture, and so it was mainly the British who liked the genre very much and who, in turn, took it to new heights [53]. British bands like the Beatles or the Rolling Stones emerged and became very popular in the USA as well. Rock dominated the music with its loud, dynamic, energetic and intense style for almost over 50 years until it was replaced by hiphop as the most mainstream genre. One of the main characteristics of rock is the infectious beat and rhythm. The music was primarily designed for dancing and was a clear distinction from other music genres popular at the time, such as jazz or Swing. Thats why from its start in the 50s the genre was very popular among young people as they felt they could break out of rigid traditions. Furthermore it

is also important to mention that rock, like hiphop later on, not only had an impact on music but also a cultural influence on the generations from the 1950s to the 1990s [53]. Rock especially supported the sense of rebellion and social justice in the western world of the 1960s and influenced clothing style, hair style and even attitude of its listeners for over 40 years. The older, more conservative generation at the time, which favored more quiet songs, rather detested rock. The energy already mentioned above is a unique characteristic that sets rock apart from many other genres. Rock music is often very wild, impulsive and driving. But what defines rock the most is the use of electric instruments and especially the use of the electric guitar [54]. The indispensable electric guitar makes rock probably the genre that is most influenced by a single instrument. Pioneers of rock like Elvis Presley, Jimi Hendrix or Chuck Berry experimented a lot with the electric guitar and used the unique sound to their advantage. Amplifiers allowed the artists to reach new melodic aspects and pitches that are not achievable with the use of pure acoustic instruments [54]. Rock music is typically performed only in bands with a lead singer. He usually plays an electric guitar himself and is accompanied either by other electric guitars, normal guitars, other electric instruments and a drum kit. Lyrical texts of rock are also very diverse due to its division into many subgenres. Lyrics may not be very profound other lyrics by artists like Bob Dylan, however, are considered comparable to fine poetry. Rock music subgenres are very different and vary greatly in terms of rhythm and tempo. For example, the music genre of heavy metal is almost incomparable with soft rock [55].

3 Implementation

3.1 Data Collection

In this section the approach and implementation of data collection for this project is examined. First, the basic requirements for the dataset are given and some existing datasets are evaluated. Then the resources used are presented and the general approach to data collection is explained, including authorization and how features and labels are collected. Lastly, the concrete implementation in Python is explained.

3.1.1 Requirements for the Dataset

Basic requirements the dataset should fulfill are:

- **Includes Spotify song features**

  Spotify provides a set of song features that were generated using their own models. The dataset should include these features, as they are needed to train the model.

- **Includes genre as label**

  The dataset needs to include the genre of the track to use as a label for the classifier.

- **Has sufficient sample size per genre**

  In order to train the model well, a sufficient sample size is needed per genre. It was not known before collecting the data, how many samples are enough to get a decent result, which is why the dataset with the highest sample size is preferred, without compromising too much on the other criteria.

- **Song and Artist name**

  The best way to filter out duplicates is to use the song and artist names. Spotify does provide a track id for each song, however, if a song is released twice (e.g. as a single and later in an album), these track ids will differ which will lead to a duplicate entry.

Additional fields are not going to be used in this analysis, but might still be collected in order to publish the dataset and enable others to use it for different applications.

3.1.2 Ressources and Approach

Spotify provides extensive documentation for developers on their developer website [56]. This includes a developer dashboard and the Web API documentation, which is the main resource for data collection from Spotify.

There are a number of pre-made datasets available online containing song data from Spotify, which were examined for this project. In conclusion, none of them were able to fulfill all of the requirements listed above, be it because of unclear methodology or an insufficient sample size. To get around this, a dataset was specifically created for this paper using the Spotify Web API.

The API offers a number of endpoints, which return JSON metadata directly from the Spotify data catalogue **SpotifyWebAPI**. Requests are made via HTTP GET or POST methods. The API can be used by anyone, but authorization via the OAuth protocol is required to access data. To explore the API and find endpoints to use, Spotify provides a developer console, which can be used to send requests and see what kind of responses come back [57]. This is not suitable for saving the data or making multiple requests programmatically, but is helpful for API exploration. As there is not one single endpoint that delivers all required fields, multiple queries that build on top of each other have to be created.

The approach began with using the API reference to get an overview over the endpoints and their responses. The specific endpoints that might return interesting data were queried using the Spotify web console to see a response with live data and which exact fields are returned. Once exploration was complete, a data collection workflow was implemented in Python 3, mainly using the libraries *http*, *json* and *requests*. The final result was saved as a CSV file to be used for data exploration and further processing.

Many API endpoints give the option to specify a country and language the results should be returned for [58]. This was set to english and the United States wherever possible, to get a dataset able to be used in applications beyond this project.

3.1.3 Getting Features

In order to predict the genre of a track based on audio features, these features have to be requested for every track. Spotify provides an endpoint to get audio features for up to 99 tracks at a time. The typical request/response pattern for the audio-feature request endpoint is shown in figure 12.

Figure 12: Audio Feature Request

GET **https://api.spotify.com/v1/audio-features?ids={ids}**
request audio features for id

Parameter

ids comma seperated list of up to 99 song ids

Response application/json

200 ok


```
{
    "audio_features": [
        {
            "danceability": 0.677,
            "energy": 0.638,
            "key": 8,
            "loudness": -8.631,
            "mode": 1,
            "speechiness": 0.333,
            "acousticness": 0.589,
            "instrumentalness": 0,
            "liveness": 0.193,
            "valence": 0.435,
            "tempo": 82.810,
            "type": "audio_features",
            "id": "2e3Ea0o24lReQFR4FA7yXH",
            "uri": "spotify:track:2e3Ea0o24lReQFR4FA7yXH
                ",
            "track_href": "https://api.spotify.com/v1/
                tracks/2e3Ea0o24lReQFR4FA7yXH",
            "analysis_url": "https://api.spotify.com/v1/
                audio-analysis/2e3Ea0o24lReQFR4FA7yXH",
            "duration_ms": 211497,
            "time_signature": 4
        },
        ...
    ]
}
```


With the exception of *type*, *id*, *uri*, *track_href* and *analysis_url*, all of the fields included in this response can be used as features in the dataset. However, this api call expects a *track id*, which needs to be retrieved using other api calls first. This could be a search endpoint, getting all tracks in a playlist, etc. Also, it does not give the track or artist names and doesn't include a genre.

3.1.4 Getting Track IDs and Labels

There is no simple endpoint that takes one or more *track ids* and returns a "genre" field in its response. The exploration of the API revealed two ways of getting the genre of a track.

The first way is using the artist of a track. Given a *track id*, the artists of the track and their corresponding ids can be requested by using the */tracks/id* endpoint. Then, using the *artist id*, the *genres* that an artist is known for are returned, as shown in figure 13.

Figure 13: Artist Request

GET **https://api.spotify.com/v1/artists/{id}**
request information about an artist by their id

Parameter

id id of the artist

Response application/json

200 ok

```
{
    "external_urls": {
        "spotify": "link to resource.."
    },
    "followers": {
        "href": null,
        "total": 15554811
    },
    "genres": [
        "conscious hip hop",
        "hip hop",
        "north carolina hip hop",
        "rap"
    ],
    "href": "link to resource...",
    "id": "6l3HvQ5sa6mXTsMTB19rO5",
    "images": [ links to album covers... ],
    "name": "J. Cole",
    "popularity": 89,
    "type": "artist",
    "uri": "spotify:artist:6l3Hv..."
}
```

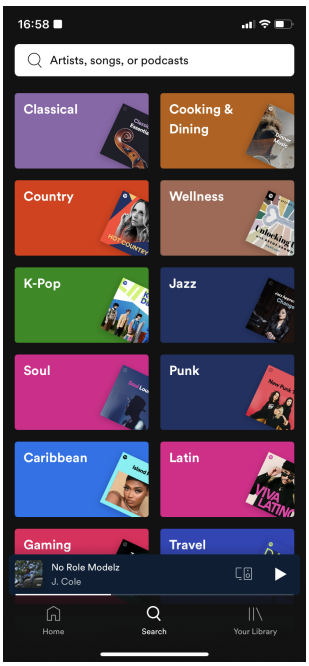
The examplary API response shows a problem with this approach. One artist can be sorted into multiple genres. A given track might be associated with either of the artist's

genres, but the data does not show, which one exactly. Additionally a track might have multiple artists which further complicates this. Given these circumstances, this approach is problematic.

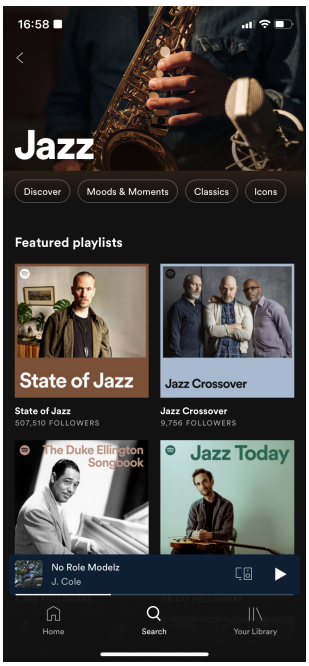
The second approach is possible via Spotifys "categories" feature. The app's search tab provides a number of categories that a user can browse to find new music in their preferred genre or style. In figure 14a an overview over some of the categories that are available in the Spotify app is shown. There are categories of multiple types like places and activities, but also for all major genres. In the app screenshot for example "Classical", "Jazz or "Soul". These categories can be used to get tracks that belong in each specific category. When a user taps on one of the categories, playlists that contain tracks of the respective category are shown to the user (figure 14b).

Figure 14: Categories and Playlists in Spotify App

(a) Category Overview



(b) Playlists in category



The API mirrors the app's behaviour and provides an endpoint to get a list of categories and their ids, one to get all playlists and *playlist ids* in a category, and one to get all tracks and *track ids* in a playlist. This chain of API calls is used to request every track in every playlist in a certain category.

3.1.5 Python Implementation

This section describes, how the previously explained API calls are used to request data from the API and save the complete dataset as a CSV file. As explaining the Python code in detail is beyond the scope of this paper, only the general approach to the application is explained here. A well documented Jupyter Notebook containing all of the code is contained in Appendix 2 and should be refered to for further details.

The script was implemented in a way that lets anyone with a Spotify developer account run it to gather their own data. It supports filtering for certain genres to speed up the process and exporting the current data after each step to be able to pause the data collection and check the current results or continue later.

The Python *requests* library is used to execute the requests, as it supports the features needed to send the GET and POST requests needed without having to write complicated code. The *json* library is used for transforming Python dictionaries into json data, *math* for access to rounding functions, *os* for managing system paths and access to files, *http* to define retry and timeout behaviour and *csv* to finally transform the JSON data into a CSV file.

After importing all necessary libraries, the credentials are read and a method is defined for authenticating to the API. The *http* library is configured to minimize errors and exceptions due to late responses or failed requests. A filter is implemented to limit the categories the data should be retrieved for using *ategory ids*. Next, the four main steps of data collection are executed:

1. Getting categories
2. Getting playlists
3. Getting tracks
4. Getting audio features

For each step, a method was implemented, which takes data from the previous step and uses it to request further data from the API. *Category ids* are used to get a playlists category, *playlist ids* are used to get a playlists tracks and finally *track ids* are used to request audio features.

After going through all stages of data retrieval, the following JSON datastructure has been built:

```
{
  "categories": [
    {
      "id": "hiphop",
      "name": "Hip-Hop",
      "playlists": [
        {
          "id": "37i9dQZF1DX0XUsuxWHRQd",
          "name": "RapCaviar",
          "tracks": [
            {
              "id": "2AaJeBEq3WLcfFW1y8svDf",
              "name": "By Your Side",
              "album": {
                "id": "2RrZgDND03MLu6pRJdTkz5",
                "name": "By Your Side"
              },
              "artists": [
                {
                  "id": "45TgXXqMDdF8BkjA83OM7z",
                  "name": "Rod Wave"
                }
              ],
              "features": {
                "danceability": 0.649,
                "energy": 0.508,
                "key": 8,
                "loudness": -10.232,
                "mode": 1,
                "speechiness": 0.0959,
                "acousticness": 0.0345,
                "instrumentalness": 3.59e-05,
                "liveness": 0.0736,
                "valence": 0.405,
                "tempo": 157.975,
                "duration_ms": 194051,
                "time_signature": 4
              }
            },
            ...
```


This JSON contains all the necessary fields needed for continuing with the CRISP DM process. To prepare the dataset for data analysis, the JSON data is transformed into a flat structure and stored in a CSV file. The final structure of the data is shown in table 9.

Table 9: Data Structure after Data Collection

Column Name	Type
categories.id	String
categories.name	String
categories.playlists.id	String
categories.playlists.name	String
categories.playlists.tracks.id	String
categories.playlists.tracks.name	String
categories.playlists.tracks.album.id	String
categories.playlists.tracks.album.name	String
categories.playlists.tracks.artists	String
categories.playlists.tracks.features.danceability	Float
categories.playlists.tracks.features.energy	Float
categories.playlists.tracks.features.key	Integer
categories.playlists.tracks.features.loudness	Float
categories.playlists.tracks.features.mode	Integer
categories.playlists.tracks.features.speechiness	Float
categories.playlists.tracks.features.acousticness	Float
categories.playlists.tracks.features.instrumentalness	Float
categories.playlists.tracks.features.liveness	Float
categories.playlists.tracks.features.valence	Float
categories.playlists.tracks.features.tempo	Float
categories.playlists.tracks.features.duration_ms	Integer
categories.playlists.tracks.features.time_signature	Integer

3.2 Data Understanding

In this chapter, the collected data is analyzed in detail. This process starts at the data source, meaning how and why spotify collects the data. Then the features, the Spotify algorithm uses, are covered in detail. This includes a theoretical explanation of each feature and its format and numerical representation. The most important features are finally compared visually using appropriate graphs and plots.

3.2.1 The Streaming Service Spotify

As stated in the introduction, Spotify is the leading streaming service. Their extensive catalogue of about 70 million tracks from over $1,2$ million Artists necessitates Spotify to focus on Big Data Analytics to generate business value and a competitive advantage. The main objective here is to understand the music itself and their user's behavior and interests, in order to offer personalized recommendations and curated playlists. The company evaluates millions of songs and and uses AI to independently recognize patterns and structures within different genres. A good example for personalized content is the "Discover Weekly" playlist, which recommends 30 new songs to each user on a weekly basis.

At the moment, the Spotify algorithm, which goes by the name Echo Nest, consists of three main components [59]. First of all, it creates an individual taste profile for each individual user. Data is collected about which artists, songs, albums, playlists, podcast and audio books the respective user listens to. The listening duration, frequency and listening location is also recorded. Different users can then be compared against each other using these metrics, to find matching taste profiles among them. For example, if *user a* has a large overlap in musical taste with *user b*'s playlists, songs from *user b*'s playlists will end up in *a*'s weekly recommendation. Furthermore, popular playlists with a large amount of followers are considered more often for recommendation puposes [59].

In addition, the algorithm analyzes the individual components of the music itself. It breaks down each song by tempo, instruments, duration, highs and lows, timbre, and other metrics. Based on the large amount of data, the Spotify algorithm divides the songs into more than 1500 unique genres. In addition to many regional differences in music genres, the AI also comes up with "nonsensical" genres. While it might be able to imagine something under "deep power pop punk", genres like "djent" are not really close to reality. [60]

This analysis step also includes the classification into emotional categories, for example happy, sad or melancholic. However, this categorization is controversial among experts, as it is not based on predefined concepts in music theory and is highly subjective. To support this classification, the AI reads the titles of the users' playlists using natural language processing algorithms. If a song is often put into playlists with titles like "love songs" or "romantic music", the AI predicts a song be related to "romance". [60]

Additionally, the Echo Nest algorithm scrapes publicly available information from websites like blogs, social media platforms, and news sites to gather data from user comments, videos, articles or posts. It evaluates the results and uses them to understand the public opinion of the song, and how it is received overall. This quickly creates a specific rating

for each track, which in turn influences the discover function and other recommendation formats within the platform. [60]

3.2.2 Feature Analysis

In the course of the analysis carried out here, three different genres are selected in order to attempt to assign songs to the correct genre on the basis of the underlying data. The genres selected here are *rock*, *hiphop* and *jazz*. The selection was made under the subjective assumption that these genres differ particularly strongly in their characteristics and style. This assumption was primarily focused on characteristics such as vocals, the general use of instruments, the instruments used, or the energy of the songs of the respective genre.

Some important insights for further consideration of the analysis are as follows. Each song feature that is evaluated does not reflect metrics found in classical music theory, but is exclusively based on the results of Spotify's algorithms. As Spotify uses their own metrics to define the very categories in which songs are classified, the analysis could sort a song, which would be considered to belong into the genre *hiphop* by theoretical standards, into the category *rock*. As the labels in this dataset come directly from Spotify, the implemented model presented in this paper will make the same assumptions.

As already mentioned, every song consists of thirteen features, which capture musical attributes and translate them into numbers. To understand the data, it is crucial to analyze these features. For a better presentation, they are divided into four clusters below.

Musical Standards

This cluster includes the features, that capture and reflect the standard properties of music. These features are *duration*, *key*, *mode* and *time signature*. The feature *duration* contains information about the track's length measured in seconds. The feature *key* is measured using integers between -1 and 11 and holds information about the key the track is in. Integers are mapped to pitches using standard pitch class notation, e.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on. If no key was detected, the value is -1. *mode* is a boolean (either 0 or 1) and indicates the modality (major or minor) of a track. Major is represented by 1 and minor is 0. The feature *time signature* is measured in an integer with a value range of 3 to 7. It is a notational convention to specify how many beats are in each bar (or measure). A value of 3 indicates a time signatures of "3/4" for example [61].

Mood

This cluster lists features that measure the emotional aspects of songs, i.e. whether the song encourages dancing or spreads a positive or negative mood. This is captured with the features *danceability*, *valence*, *energy* and *tempo*.

All of these features except *tempo* are measured in a float between 0 and 1. *danceability* describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 represents "least danceable" while 1.0 stands for "most danceable". *valence* describes the musical positiveness conveyed by a track. Tracks with high valence sound more happy, cheerful and euphoric while tracks with low valence sound sad, depressed or angry. *energy* represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. A value near 1.0 indicates high energy, while tracks near 0.0 are calm. *tempo* holds information about the overall estimated tempo of a track in Beats per Minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and is derived directly from the average beat duration [61]

Properties

This cluster bundles all features that capture the musical characteristics of the tracks. It includes *loudness*, *speechiness* and *instrumentalness*.

loudness is measures the average loudness of a track in decibels. This is represented as a float value between -60 and 0. The value is useful for comparing relative loudness of tracks. *speechiness*, is again represented as a float value between 0 and 1 and detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values below 0.33 most likely represent music and other non-speech-like tracks. *instrumentalness* plays the counterpart to *speechiness* and is represented in the same way. It predicts whether a track contains no vocals at all. Sounds like "ooh" and "aah" are treated as instrumental in this context. The closer the *instrumentalness* value is to 1.0, the more likely the track contains no vocal content. Values above 0.5 are intended to represent instrumental tracks, but confidence becomes higher as the value approaches 1.0. [61]

Context

The cluster context groups the features *liveness* and *accousticness*. These capture the setting of a song, for example if it is played in front of a live audience.

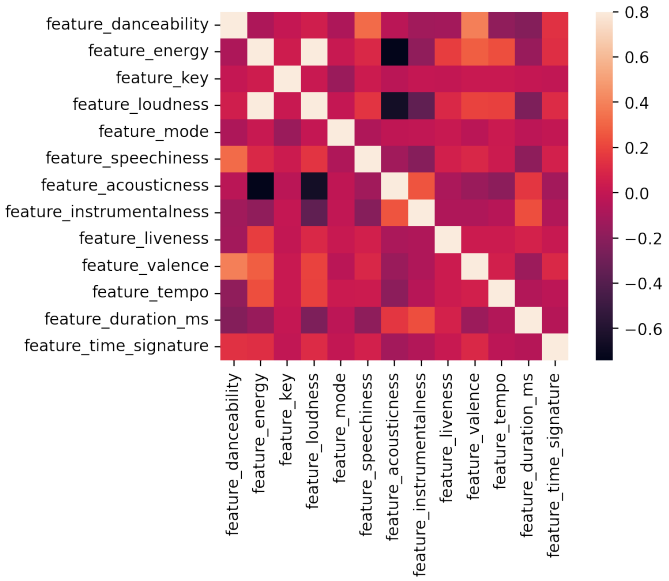
Both features are represented as float values between 0.0 and 1.0. *accousticness* indicates whether a track is acoustic. 1.0 represents high confidence that the track is acous-

tic. *liveness* detects the presence of an audience in the recording. Higher *liveness* values (above 0.8) indicate an increased probability that the track was performed live.

3.2.3 Correlation between Features

Next, an overall correlation of the different features is examined. A correlation matrix is used for this purpose. It shows how features correlate with each other on a scale between 1 and -1. Since every song has a value for every feature, a high correlation here does not mean that the features often appear together, but how similar the values per song are. To start with an overall look over the data, figure 15 shows a genre independent matrix. This means that the features of all songs that could be gathered in the step of data collection are analyzed to create this figure [61].

Figure 15: Category independent Correlation of Features

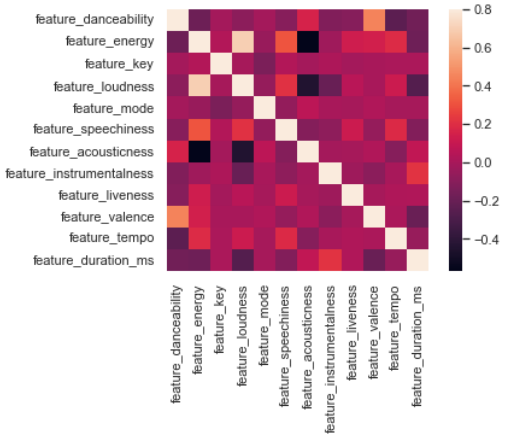


Generally, the features in the cluster "musical standards" can be ignored in this representation, as they do not measure but only reflect characteristics (e.g., time signature or key). It can be observed that *energy* and *loudness* have the highest correlation of the features, with a value of approx. 0.65. This means that energetic songs have a tendency to also be loud. The correlation matches the definition of these features as energetic tracks are defined to feel fast, loud, and noisy. In addition, there is a comparatively high correlation between *valence* and *danceability* of about 0.5, which means that positive songs encourage dancing. The correlation between *energy* and *accousticness* is particularly negative, with a value of approximately -0.7. Acoustic tracks therefore often do not seem to be very

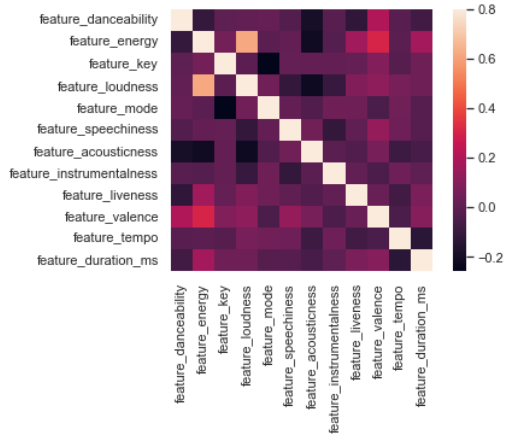
energetic and vice versa. The same can be said for *accousticness* and *loudness*, with a correlation value of about -0.5. Outside of these extreme values, the remaining correlation values settle between -0.2 and 0.2. Worth mentioning here are the correlations between *instrumentalness* and *loudness* at approximately -0.3 as well as *instrumentalness* and *valence* with around -0.3. Furthermore, a positive correlation between *speechiness* and *energy* can be observed at a value of around 0.3.

Figure 16: Correlations within Categories

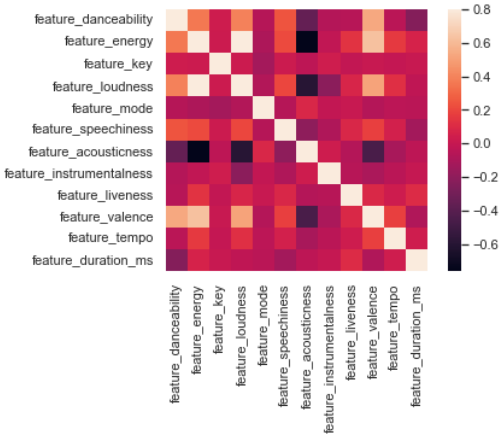
(a) Correlation within *rock*



(b) Correlation within *hiphop*



(c) Correlation within *jazz*



The correlations in figure 16 show the matrices on a per category basis. The patterns described above can also be observed here. However, the average correlation changes depending on the category. For example, *hiphop* shows the same characteristics as the category independent matrix but the correlations are fundamentally more negative.

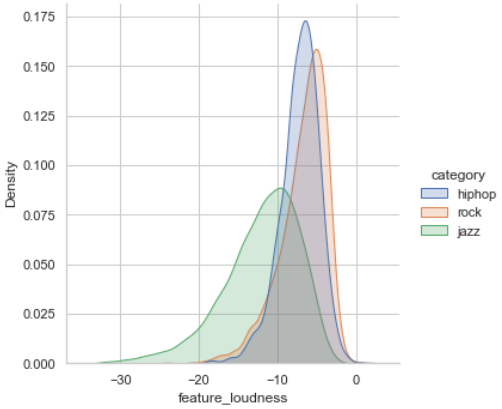
3.2.4 Comparison of Music Theory to Audio Features

An equally important step for data understanding is to compare features to the characteristics of genres as described in music theory. Also a comparison between the categories itself is carried out. This is necessary to develop a deeper understanding of the data. As mentioned in the beginning of this chapter, theory and Spotify's findings are not always necessarily congruent, which is why a clear delineation is important here. For this comparison, three features are chosen and discussed more deeply. Subsequently, a better picture can be obtained with the help of further representations.

According to music theory a distinct property for rock songs is that they are very loud. The characteristic is also present in the features created by Spotify in the feature *loudness* which is discussed here.

Figure 17: A Comparison of *loudness* between the Categories

(a) Distribution Plot *loudness*



(b) Boxplot *loudness*

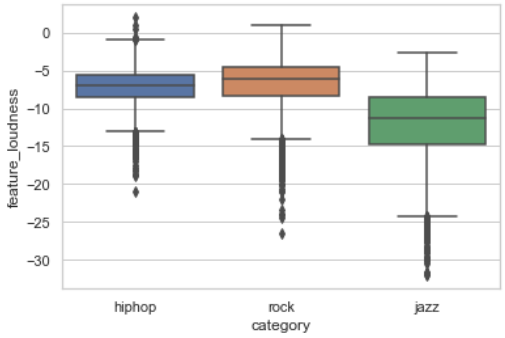


Figure 17a shows a distribution plot. The horizontal axis contains the value range of the feature *loudness*. The vertical axis shows the relative density of samples in a category at the respective loudness. There are density curves for each of the categories, labeled accordingly. Figure 17b shows a box plot, where the box represents values contained between the 25th ($Q1$) and 75th ($Q2$) percentile and is called Interquartile Range (IQR). The center line marks the median. The lines outside of the box mark minimum ($Q1 - 1.5 * IQR$) and maximum ($Q3 + 1.5 * IQR$) values, with the points outside of them representing outliers [62].

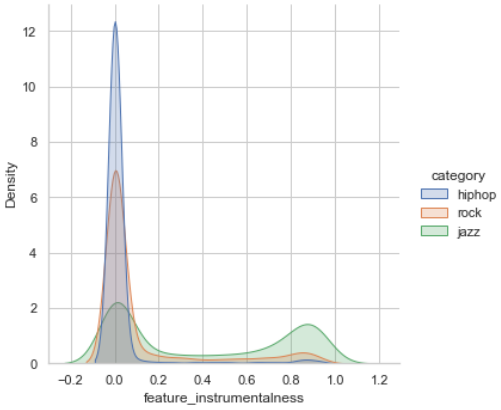
These visualizations show, that *hiphop* and *rock* are more tightly distributed than *jazz*. *jazz* spreads its songs over an area from -32.06 to -2.695 but locates most songs at about -10. *hiphop* and *rock* are distributed less widely and have a density which doubles

that of *jazz* at its highest point. *rock* places its point furthest to the right of the scale. All of the features have a clear maximum density, which shows that a majority of tracks are equally loud. These observations confirm, that *rock* is indeed the loudest category, closely followed by *hiphop*. This mimics the characteristics described in music theory. *jazz* can also have loud tracks, but the value range is far wider and it is generally more quiet.

The next feature discussed here is *instrumentalness*. Because the categories differ greatly in the amount of vocals used, differences can be expected here. One would expect *jazz* to have higher values, while vocals are an integral part of *hiphop* and therefore the feature might be concentrated at lower value ranges. The overall average for this feature is 0.174. The lowest value overall is 0.0 while the highest goes up to 0.989.

Figure 18: A Comparison of *instrumentalness* between the Categories

(a) Distribution Plot *instrumentalness*



(b) Boxplot *instrumentalness*

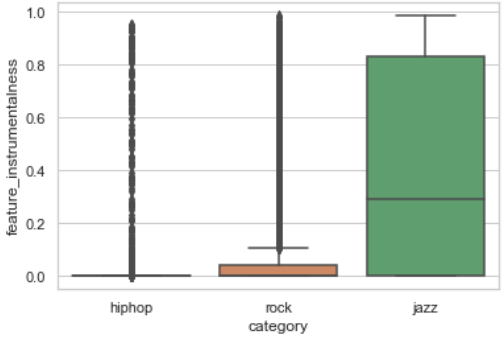
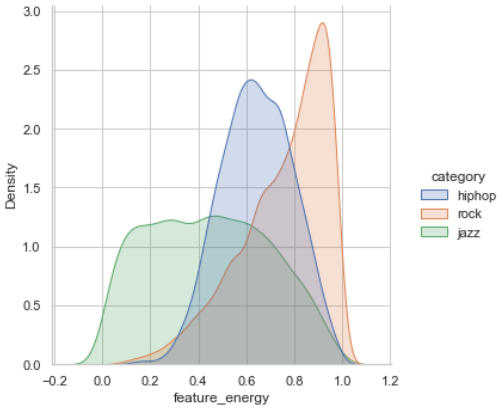


Figure 18 again shows a distribution plot and boxplot for *instrumentalness* It shows that both *hiphop* and *rock* are almost exclusively concentrated in the 0.0 to 0.1 range. This means that almost all songs in these categories contain vocals. However, slight increases in the range between 0.8 and 1.0 can also be seen. *jazz*, on the other hand, is grouped into two different maxima. One in the range around 0.0, the other between 0.8 and 1.0. Since *jazz* often contains only musical tracks without any vocals, this reflects the reality quite well. Here again, *rock* and *hiphop* are similar in their high points, while *jazz* differs. The distribution is more distributed and heterogeneous.

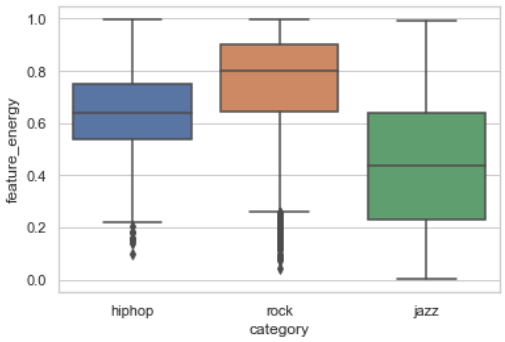
The last feature discussed in detail is *energy*. As explored in section 3.2.3 it correlates highly with the feature *loudness*. Therefore, one would expect similar results to the plots shown above.

Figure 19: A comparison of *energy* between the Categories

(a) Distribution Plot *energy*



(b) Boxplot *energy*

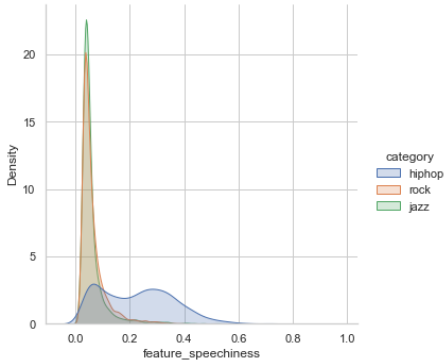


The plots (figure 19) support the initial predictions, as *rock* is again highly concentrated at the top of the value range, followed by *hiphop*. *jazz* is again distributed very widely and seems to generally be less energetic than the other categories. There is also not a clear maximum, indicating that there does not seem to be a default energy level for *jazz* artists. Instead, artists seem to approach the genre from many different angles. This is seen in music theory, as *jazz* is divided into a number of subgenres with different properties. In contrast to the distribution seen in *loudness*, *hiphop* is fairly broadly positioned. It is a multi-faceted genre as well, which is supported by the data.

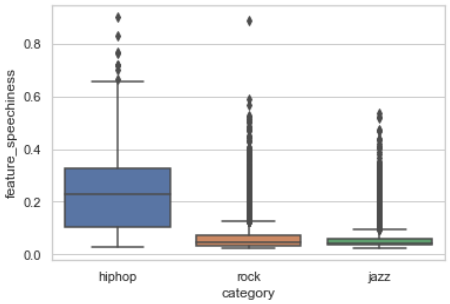
After these three sample features, additional Distribution- and Boxplots are visualized to give further insights about the characteristics of the individual genres.

Figure 20: Different Features visualized in Distribution- and Boxplots

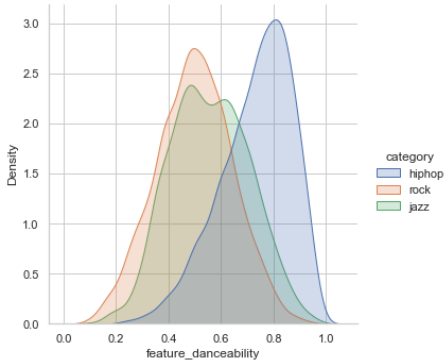
(a) Distribution Plot *Speechiness*



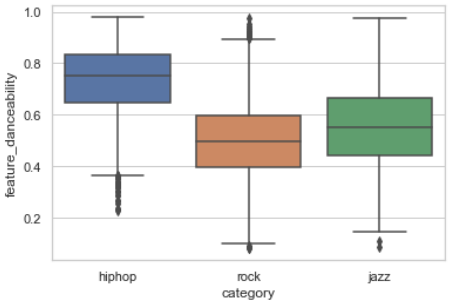
(b) Boxplot *Speechiness*



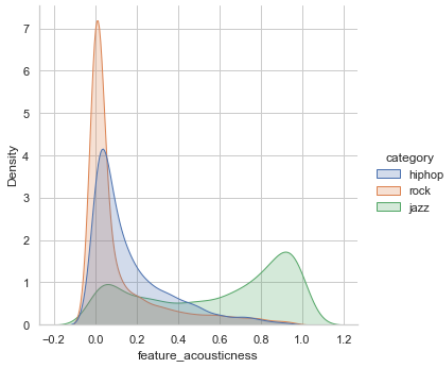
(c) Distribution Plot *danceability*



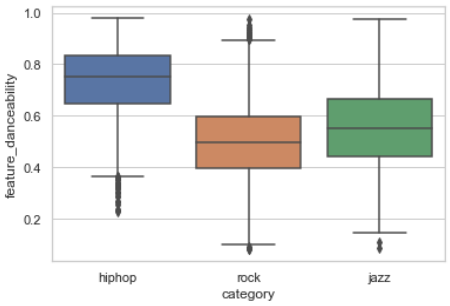
(d) Boxplot *danceability*



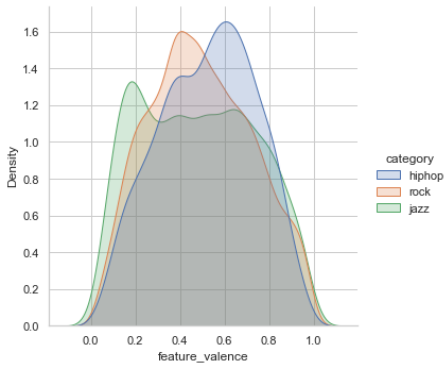
(e) Distribution Plot *accousticness*



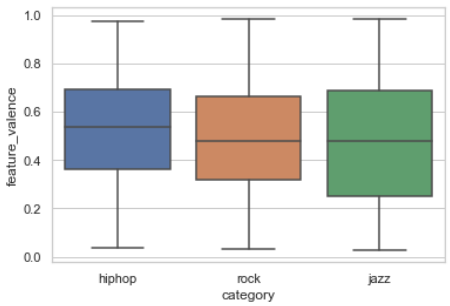
(f) Boxplot *accousticness*



(g) Distribution Plot *valence*



(h) Boxplot *valence*



3.3 Data Preparation

In the following section the data is prepared and cleaned up for training the model.

Data preparation is done in Python using multiple libraries. The same libraries are also used in the modeling and evaluation stages of this project. *pandas* is used for storing and manipulating data in dataframes. *sklearn* is used for splitting data, creating validation curves, transforming data and training the Gradient Boosting model. *matplotlib* and *seaborn* are used for data visualization and plotting. Additionaly *numpy* is used for other operations on the data.

As with data collection, the complete code with documentation is attached in Appendix 3.

3.3.1 Import and Cleaning

The CSV file is first imported and converted into a *pandas* dataframe.

```
df_import = pd.read_csv(os.path.join('..', 'data_collection', 'final_result.csv'))
```

Next, duplicates are removed. If there are two or more datapoints, which have the same value in their *artist* and *name* column, only the first one is kept and all subsequent entries are removed. This deduplication could also be done by using *track ids*. The drawback of this method is, that many artists release their tracks multiple times, e.g. as a single and later in an album. These duplicates would not be caught using the id, as different releases have different id values. As the artist and title doesn't change, almost all duplicates are caught using artist and name.

```
df_dedup = df_import.drop_duplicates(subset=['categories.playlists.tracks.artists'
    , 'categories.playlists.tracks.name'])
```

Next, all genres that are not to be used for training the model are filtered out.

```
genre_filter = ['hiphop', 'jazz', 'rock']
df_filtered = df_dedup[df_dedup['categories.id'].isin(genre_filter)]
```

Then columns that are not required for training are removed, including category name, all playlist information and all track and album information. The *category id* is kept, as it will serve as the label. The remaining columns are renamed, as the long JSON tree names are no longer needed. The field *category.id* is renamed to *category* and each audio feature is renamed for example the danceability column is now called *feature_danceability*. With unnecessary columns removed, a check is done to show any remaining null values.

```
df.isnull().sum()
```

In this dataset, there are no null values present.

The *GradientBoostingClassifier* used for modeling only supports integer values as label input. The category data is therefore encoded with integers using a custom function *encode_target*. It takes a dataframe and the label column's name, than adds a "target" column with corresponding integer mappings.

```
    def encode_target(df, target_column):

        df_mod = df.copy()
        map_to_int = {name: n for n, name in enumerate(df_mod["category"].unique())}
        df_mod["target"] = df_mod[target_column].replace(map_to_int)

        return (df_mod)

    df_target = encode_target(df, "category")
```

The head of the resulting dataframe is shown in table 10. The category to target integer mapping is shown in table 11.

Table 10: Dataframe after Cleanup

row	0	1	2	...
category	hiphop	hiphop	hiphop	
target	0	0	0	
feature_danceability	0.649	0.849	0.793	
feature_energy	0.508	0.631	0.481	
feature_key	8	3	9	
feature_loudness	-10.232	-4.241	-9.258	
feature_mode	1	0	1	
feature_speechiness	0.0959	0.0637	0.1240	
feature_acousticness	0.03450	0.17100	0.01900	
feature_instrumentalness	0.000036	0.000000	0.000001	
feature_liveness	0.0736	0.1490	0.1390	
feature_valence	0.405	0.550	0.395	
feature_tempo	157.975	135.997	132.202	
feature_duration_ms	194051	215304	2046526	
feature_time_signature	4	4	4	

Table 11: Categories mapped to Integer Targets

Category	Integer Target
hiphop	0
rock	1
jazz	2

Looking at the number of samples per category (table 12) reveals, that the dataset is very uneven, with more than half of the samples falling into the category *rock*.

Table 12: Number of Samples per Category after Cleanup

Category	Number of Samples
hiphop	2694
rock	7252
jazz	3431

As explained in section 2.4.1 and 2.4.2 methods like standardization and PCA can have a positive impact on the models accuracy. Before finding the best model using hyperparameter tuning, the best form of input data is evaluated by transforming the data in different ways and training a Gradient Boosting model using the default parameters specified in *sklearn*. This way, the best form of input data can be found without the overhead of resource intensive grid search.

To be able to easily train models using different forms of input data, a method *eval_prep* was created, which takes a dataframe, a list of all feature column names and the label column name. It then splits the dataframe into train and test sets, trains a *GradientBoostingClassifier* and returns a simple accuracy score. This process is explained in depth in the modeling section.

First, the regular dataframe is used as input data, resulting in an accuracy score of 0.8505.

```
1    score = eval_prep(df, features, "target")
```

3.3.2 Data Transformation

Next, mean removal, variance scaling and standardisation are attempted, by using the *StandardScaler* class from *sklearn* to transform the data. The code for standardization is shown below. Mean removal and variance scaling use the same class with additional parameters.

```
1    X_s = df[features]
2    y = df["target"]
3
4    X_s = StandardScaler().fit_transform(X_s)
5
6    df_s = pd.DataFrame(data=X_s)
7    df_s.insert(0, "target", y)
```

```
8     df_s.columns = ["target"] + features
9
10    score = eval_prep(df_s, features, "target")
```

This results in the following scores, beating the untransformed data in all cases.

- Mean Removal: 0.8509
- Variance Scaling: 0.8520
- Standardization: 0.8523

3.3.3 Principal Component Analysis

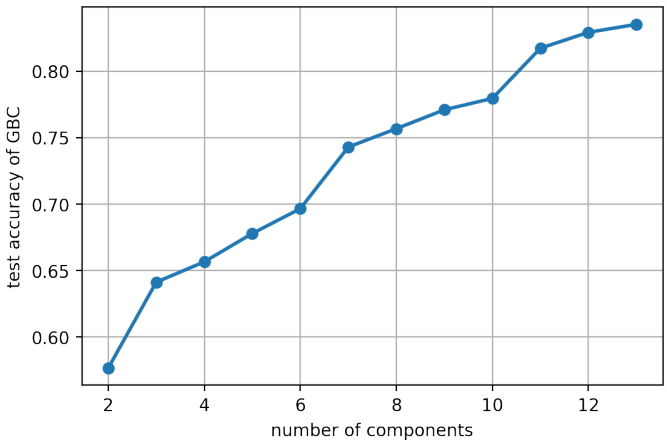
Next, PCA is attempted using *sklearns PCA* class. It is able to take any dataset and reduce its dimensionality to any number of dimensions. The dimensionality is reduced to all dimensions between 2 and 13 to see, which dimensionality results in the best score. PCA was done both on the raw dataset and in combination with standardized data (both before and afterwards).

PCA does not yield good results with this dataset. The score for all of the combinations is always worse than using untransformed data. The PCA score was best when keeping 13 dimensions, declining further when removing more dimensions. These are the scores using each combination with PCA and keeping 13 dimensions:

- PCA only with 13 components: 0.8352
- Standardization after PCA with 13 components: 0.8348
- Standardization before PCA with 13 components: 0.8277

Figure 21 shows the test accuracy using PCA on the untransformed data for each number of components.

Figure 21: Test accuracy for each dimensionality after PCA



As the results using PCA do not improve the score, the code is shown in the appendix. Testing data transformation and dimension reduction with this dataset and a default *GradientBoostingClassifier* shows, that the model benefits slightly from standardized data, improving the overall score compared to the raw dataset by 0.19%. Therefore, standardized data is used going forward.

3.4 Modeling

3.4.1 The Classifier

For training the Gradient Boosting Algorithm, the *GradientBoostingClassifier* class from the *scikit* learn Python library is used. It is part of a group of classes offering different ensemble methods. As explained in section 2.3, ensemble methods combine the predictions of several weak learners to generate a more robust model and reduce overfitting issues. *sklearn* supports averaging methods like Bagging and Random Forests, which take the average of each learners prediction as their output. It also supports Boosting methods like AdaBoost or Gradient Boosting, which build base estimators sequentially, improving the output for each iteration. *sklearn* also provides a classifier and regressor model for each method.

The Gradient Boosting classifier supports binary and multi-class classification and uses 20 hyperparameters to control the size of each regression tree, the number of trees, the learning rate and many more. As explaining and tuning all 20 hyperparameters of the classifier would be beyond the scope of this paper, three important parameters are explained and optimized.

- **learning_rate**

  As explained in section 2.3.1, the learning rate is used to control how much each tree contributes to the result, by multiplying it with the output values of the previous tree. The default value here is 0.1.

- **n_estimators**

  The number of weak learners (here regression trees) to be used while boosting. The default is 100.

- **max_depth**

  The maximum depth of each regression tree. This also impacts its number of nodes. The default value is 3.

3.4.2 Validation Curves

During hyperparameter tuning, grid search is used to find the optimal value for each parameter. As the computation is very resource intensive, a sensible range of values to search in must be found beforehand. Validation curves can be used to observe how a model's score changes when modulating a single hyperparameter. Here, an important distinction between the training and cross-validation score must be made. The training score is the accuracy the model achieves when being scored on the same data it was trained on. A very high training score could be a sign of overfitting. The cross-validation score (cv score) is the mean accuracy achieved during 5-fold cross validation and is a good measure of the actual performance of the model.

Figure 22 shows the curves for each hyperparameter. The shaded areas around the graph are the standard deviation, while the graph itself represents the mean score between 5 folds.

Figure 22a shows the best cv score with a learning rate between 0 and 0.2, declining as it approaches one. The training score rises as the learning rate increases, which indicates that the model becomes less generalizable and more overfitted as the learning rate increases. Because of these results, only values between 0.01 and 0.25 are considered during hyperparameter tuning.

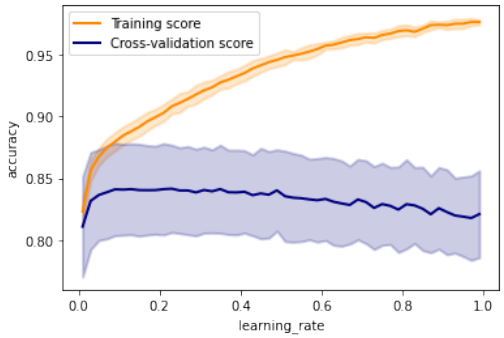
The cv score stays relatively flat as the number of estimators goes beyond 25 (figure 22). The training score increases slightly. This indicates that a training score close to, but above

25 might be optimal. As the results are not very clear, a range between 50 and 200 is used for optimization.

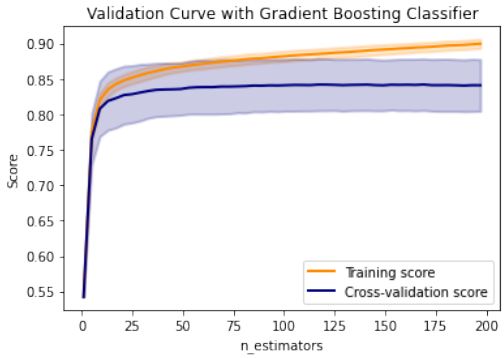
For *max_depth*, the cv score stays relatively flat between 2 and 13 (figure 22). The training score, on the other hand, increases rapidly from 0 to 7.5, where it reaches its maximum. Because of this, a range between 2 and 6 is used for hyperparameter tuning.

Figure 22: Validation Curves modulatng 3 hyperparameters

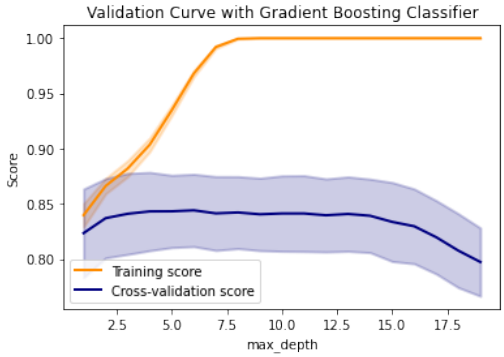
(a) learning_rate



(b) n_estimators



(c) max_depth



3.4.3 Hyperparameter Tuning and Fitting a Model

The parameter grid is given as a Python dictionary containing the parameter name as keys and a list of values for each key. All undefined hyperparameters are kept at default. The grid used for training with the value ranges found above looks like this:

```
param_grid = {
    "learning_rate": [0.01, 0.04, 0.07, 0.1, 0.13, 0.16, 0.19, 0.22, 0.25],
    "n_estimators": [50, 75, 100, 125, 150, 175, 200],
    "max_depth": [2, 3, 4, 5, 6]
}
```

The grid and a *GradientBoostingClassifier* object are passed into a *GridSearchCV* object which executes the grid search using 5-fold CV.

```
    gbc = GradientBoostingClassifier(random_state=45)
    search = GridSearchCV(gbc, param_grid,
        n_jobs=-1,
        error_score="raise",
        verbose=1)
```

The search object's fit method combines cross-validation, hyperparameter tuning and classifier fitting to find the best possible combination of hyperparameters to the estimator for the given dataset. This is done in the following way:

1. An estimator is created using the first combination of hyperparameters.
2. The training set is split into five folds and each fold is used once for calculating accuracy, with the other four used for training. The average of the five scores is used as the final score for this specific set of parameters.
3. The score is saved together with the parameters
4. These steps are repeated with every possible combination of parameters from the parameter grid. This results in a score for every set of parameters.
5. As every model was fitted using the same method, it is clear that the model with the highest score is using the best hyperparameters. A new model is trained without cross-validation using all available training data, to create the final model.

The complete standardized dataset is split in 80% train and 20% test data. The samples are shuffled before splitting to ensure an even distribution of data. Then features and labels are seperated for input into the classifier.

```
    train, test = train_test_split(df_s, test_size=0.2, random_state=45, shuffle=True)

    X_train = train[features]
    X_test = test[features]
    y_train = train["target"]
    y_test = test["target"]
```

Then hyperparameter tuning and estimator fitting is started using the training features and labels.

```
    search.fit(X_train, y_train)
```

The parameter grid above has 315 possible combinations. As each is fitted on five sets of folds, there are 1575 models fitted in total during this process.

The resulting model with the best cross-validation score used the following parameters:

- learning_rate: 0.1
- n_estimators: 100
- max_depth: 3

Calculating the accuracy of the resulting model using the test set resulted in an accuracy score of 0.8591.

A weakpoint of the parameter grid used is that the values for *n_estimators* are spread by increments of 25, leaving possible room for improvement. To check whether the model could be improved another parameter grid was used:

```
param_grid = {
    "learning_rate": [0.01, 0.04, 0.07, 0.1, 0.13, 0.16, 0.19, 0.22, 0.25],
    "n_estimators": [97, 98, 99, 100, 101, 102, 103],
    "max_depth": [2, 3, 4, 5, 6]
}
```

This did not deliver a better result.

Even though a broad spectrum of values was given in the search grid, the optimal values are equal to the default parameters for the estimator set by *sklearn*. This might suggest that the developers set the default values using a similar dataset to the one used in this project.

Although many combinations were evaluated, there is still a possibility, that the best result is still only a local maximum and a set of hyperparameters, that is far different from the ones tested here, would result in a global maximum.

3.5 Evaluation

After the completion of the modelling phase, the next step is to evaluate the results. The evaluation is split into the evaluation of the model itself followed by the evaluation of the overall process.

3.5.1 Gradient Boosting Evaluation

In this section the Gradient Boosting algorithm will observed in more detail by the most common metrics. Additionally, it also will be compared with the much simpler Classification Tree algorithm to highlight commonalities and different approaches between the two algorithms and create an all-encompassing picture for Gradient Boosting.

The overall accuracy measured is around 86%. This result is already positive as the Gradient Boosting Algorithm outperforms the Classification Tree, which achieves an also respectable accuracy of 80%. However, the accuracy score can only be used as a fundamental basis for evaluating the overall performance with many unknowns that need to be worked through.

The first step of a deeper analysis is to create a confusion matrix for all categories. The confusion matrix is an approach of visualizing the performance by clustering the the output. The $x1$-axis represents the predicted values for the classes while the $x2$-axis stands of actual (correct) values [63, p.235]. The confusion matrix reveals four combinations of predicted and actual values (13). For Gradient Boosting the confusion matrix is displayed in table 14.

Table 13: Confusion matrix

predicted actual	class 1	class 2
class 1	true positive	false negative
class 2	false positive	true negative

Table 14: Gradient Boosting Confusion Matrix

predicted actual	hiphop (1)	rock (2)	jazz (3)	all
hiohop	438	68	19	525
rock	41	1307	122	1470
jazz	30	115	536	681
all	509	1490	677	2676

Visually, the highest misclassification takes place between classes *jazz* and *rock* while the lowest missclassification occurs between *hiphop* and *jazz*. This result however has little significance, since the number of *rock* tracks, with a total of over 1400, is significantly larger than the amount of both *hiphop* and *jazz* tracks. An evaluation based on absolute numbers would lead to misleading results partly due to the imbalance of the dataset.

A better approach is to look at the following metrics which can be derived form the confusion matrix for every category [63, p.235] [64, p.862](15).

$$precision = \frac{true\,positive}{true\,positive\ +\ false\,positive}$$
$$recall = \frac{true\,positive}{true\,positive\ +\ false\,negative}$$
$$f1-score = 2 * \frac{precision * recall}{precision\ +\ recall}$$

Table 15: Gradient Boosting Classification Report

classes	precision	recall	f1-score	support
hiphop	0.86	0.83	0.85	525
rock	0.88	0.89	0.88	1470
jazz	0.84	0.79	0.79	681

Precision and recall are both performance metrics with different objectives. While precision is a measure of how many of the predicted elements for a class were correct, recall measures how many elements of a category were detected. It can be observed that the precision is high across all categories with *rock* being classified best with 12% false-positive predictions while the false-positive rate for *jazz* was worst with over 21%. Interestingly the results for recall are very similar with the model performing best for the category *rock* with an recall of 0.89 and again worst for *jazz* with 0.79. For both metrics *Hiphop* is in between of both extremes at around 0.85.

The f1-score is a combination out of precision and recall and an attempt of capturing both metrics in a single value. Therefore, the results are not surpising. *Jazz* performed worst with only 0.79 while *rock* was classified best with 0.88 according to the f1-score.

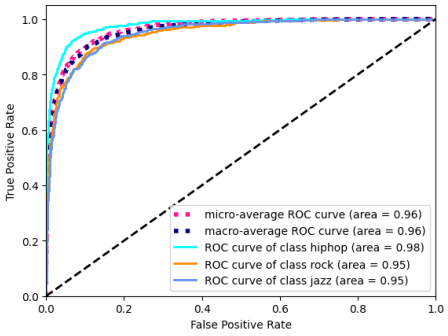
In conclusion all scores can be evaluated as positive outputs without any negative and unexplainable abnormalities. Also, in comparison to decision trees, a clear improvement of the results can be seen. The reason that *jazz* is ranked worst while *rock* reaches the highest values may be due to several reasons. One possible explanation can be found by reviewing the findings from the data understanding chapter. It is recognizable that the value-ranges for the features of *jazz* were significantly more distributed than for both *rock* and *hiphop* and often without any high points. In addition, the value ranges of *rock* were often different from those of hip-hop and jazz, which simplifies the respective classification.

Another measure to evaluate a models performance is to plot its Reciver Operating Characteristic (ROC). The ROC is a plot of the true-positive rate for the $x1$-axis and the false-positive rate on the $x2$-axis for every possible threshold. The benefit of ROC is that it plots

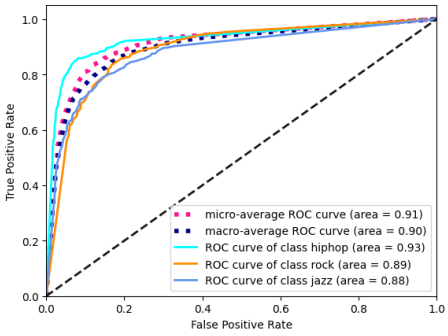
the misclassification for every threshold while other metrics rely on a single threshold. A model whose results are close to the diagonal classifies data worse than a model whose curve is as close as possible to the point $(0/1)$ in the coordinate system. This performance is often measured by means of the Area Under the Curve (AUC), which is directly derived from the ROC. With help of the AUC comparability between models is possible [64, p.862f] [65]. The ROC and AUC for the project are shown in figure 23.

Figure 23: Comparison of ROC and AUC

(a) Gradient Boosting



(b) Classification Tree

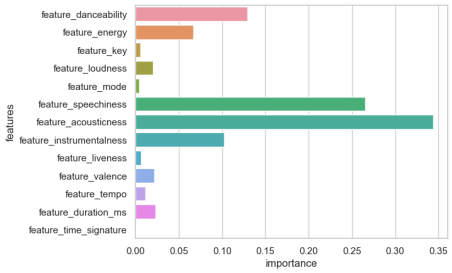


The following matrix shows that the model works well for all classes. Both ROC and the associated AUC are convincing in all cases. Interesting is the fact that *rock* has the worst AUC while being classified best according to the previous metrics. *Hiphop*, on the other hand, has the significantly best AUC followed by *jazz*. An explanation for this result is complicated, with a possible reason again being the imbalance of the dataset. The model is focussed classifing *rock* correctly as it constitutes to a large part of the dataset. Therefore, the optimum for both *jazz* and *hiphop* are exchanged for an overall optimum. It should be noted that a well-founded explanation would explanation would require a more in-depth analysis. Regardless, the result is worth including in the analysis and leaves room for further research.

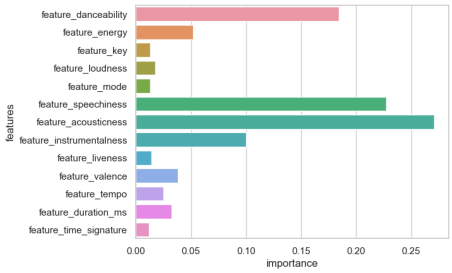
In addition, a very different evaluation can be performed. It is also interesting to take a closer look on the input data with help of the so-called feature importance. The feature importance is a measure of how much impact a feature had for the classification of the dataset and is presented in the following figure 24 for both Gradient Boosting and the Classification Tree algorithm [66].

Figure 24: Comparison of Feature Importance

(a) Gradient Boosting



(b) Classification Tree



It is clearly visible that both Gradient Boosting and Decision Trees relied on similiar features to similar extends for the classification task with minor differences for features such as *danceability*. Noticeable is that Gradient Boosting is heavily focused on only a few features while the Classification Tree makes more use out of features like *duration*, *valence* and *mode*. One possibility for the difference in weighting could be the overall construction of the algorithms. Gradient Boosting consists out of multiple weak learners while the Classification Tree algorithm froms a widely branched and deep tree. For feature importance, an analogy can again be made with data understanding and music theory. The features that theoretically distinguish the genres from another play the most important role for the modeling while musical standards and more subjective featues only play a subordinate role. This result is positive both for theory and the machine learning model as it states that music theory can be confirmed by real-world examples while giving credibility and plausibility to the model.

3.5.2 Process Evaluation

The evaluation of the overall process is complicated as there are many unknowns starting with the dataset itself. The data used is preprocessed by Spotify without detailed definition and theoretical basis. The features are furthermore heavily subjective, which adds another layer of complexity. Also the data collection is non-trivial and error-prone. Both the collection process and the collection approach would have to be revised for a productive use case. On the other hand, the data quality is decent with high correlation to the music theory. In addition, the dataset convinces with its uniqueness and novelty.

The model in combination with preprocessing and evaluation also leaves room for further research. The use of only two very similar machine learning algorithms allows no comparison to completely differnt approaches based on other algorithms and concepts. It can

therefore not be ruled out that even better results can be achieved. The pre-processing is intensive with many constallations tested. A possible improvement would be to apply the hyperparametric optimization not only on the standardized data but on all approaches to ensure an overall best result. In addition, there are further hyperparameter constellations which were not tested. The evaluation may also have reached incorrect conclusions and assumptions for a variety of reasons.

The overall highest risk is the fact that all participants of the group are not familiar with machine learning and lack experience. For this reason, the result can be considered promising, as both algorithms provide good overall results.

4 Conclusion

The project had the purpose to classify a dataset with the help of a machine learning algorithm. With no restrictions on the choice of dataset and algorithm given, a classification of genres based on Spotify Music data was selected as a project task. After an extended search, no sufficient existing dataset was found and thus a new dataset had to be created. Even though the data collection was complex at times, it proofed to be a good decision and convinces with its novelty. After testing and debating over various algorithms, Gradient Boosting was selected as the machine learning algorithm. Additionally, a classification tree was implemented as a reference point.

The overall result was successful. The Gradient Boosting model recieved an accuracy fo around 86% with additional in-dept evaluation metrics supporting the positive performance. It is worth mentioning that despite high efforts of optimization only standardization has lead to a better classification result as other models did not lead to an improvement. This indicates that either the models were implemented incorrectly or that the overall quality of the dataset was excellent from the beginning. The implementation itself, despite all the encuraging signs and positive, still leaves room for additional research. It is furthermore very pleasing to see that the model relies similar features that are also used in music theory for differentiation between genres and allows affirmatory assumptions to be made on Spotifys data quality, the gathered dataset and the implementation of the Gradient Boosting algorithm.

Also from a personal perspective there are several lessons learned during this project. The first lesson is that, despite the choice of a comparatively simple machine learning algorithm in Gradient Boosting, limitations of the project framework were reached. Both in terms of

scope and size of the paper and from an implementations perspective, as multiple optimization possibilities and design choices were discovered during the implementation and required further attention. The project was not straight-forward with multiple modifications and additions to be made along the way. However, all changes should in no way be seen as negative, as they have led to the desired result and are the result of a steep learning process. No participant of this group had prior experiences in machine learning. New discoveries had to be gathered and processed to form conclusions and drive the project forward. It is remarkable how uncomplicated the creation of simple models is with the help of machine learning libraries. Nevertheless, the final classification model is convincing and meets the predefined objectives. The result is a team effort and was only possible through good collaboration.

In conclusion, Spotify is just one example of many modern applications that utilize big data analytics to attempt to caputure and reflect the oftentimes subjective worlds in numbers. Particullary intresting is the fact to what extent music, which is believed to be subjective, can be processed by Spotify to give recommendations, cluster songs in genres or feelings.

Appendix

Appendix 1: Table of Contributions

Section Name	Keiser	Krüger	Pflamminger	Wesemann
Introduction	x	x	x	x
Classification in the Context of Big Data	x	x		
Decision Trees		x		
Gradient Boosting		x		
Concepts for Data Prep...			x	
CRISP DM				x
Web APIs			x	
Basic Concepts of Music Theory	x			
Data Collection			x	x
Data Understanding		x		x
Data Preparation			x	
Modeling			x	
Evaluation	x	x		
Conclusion	x	x	x	x

Appendix 2: Data Collection Coding

This is the complete documented code for data collection.

data_collection

January 6, 2022

1 Data Collection from Spotify Web API

1.0.1 Method

This code is used for all data collection. Data is collected using the following method: 1. Authentication and http client setup 2. Use the 'categories' endpoint to get the id and name of all categories.
The category is used to identify the genre of a track. 3. Use the {category_id}/playlists endpoint to get the name and id of each playlist in the category 4. Use the playlists/{playlist_id}/tracks endpoint to get - Track id and name - album information - names of artists 5. Use the audio-features endpoint to get all available audio features for each track
At this point we have all data stored in a nested json file, which needs to be flattened, in order to create a dataframe that can be used for further processing and data exploration 6. Flatten json and create dataframe ### Running this code - Place a .env file in the project root directory, which contains the following variables

```
CLIENT_ID=
CLIENT_SECRET=
```

To obtain these credentials - create a Spotify account - go to the developer dashboard - create an app - copy the apps credentials into the .env file

1.0.2 Filtering

You might not want to get data for all categories, as this needs several thousand calls to the api. We implemented a category filter system which you can use to filter by category id.

1.0.3 Continuing with data from file

Each step of the method can be run on its own and the data used as input can be given in form of a json file.
If no file is used, results from the previous step is directly passed in.
Alternatively, each step can output its result to a file. The path is specified in the function arguments.
`(write_to_file=True, path_to_file=%path%)`

How to use input file: - uncomment code to load file into data variable - run all the setup code up to and including the point where data is loaded from file - run blocks from the step you want to continue at

1.1 Imports, authentification and http setup

[ ]:
```
# imports
import http
from dotenv import load_dotenv
import os
import json
import requests
import math
from copy import copy
import csv

#Get environment variables from ".env" file and read credentials
load_dotenv('.env')
client_id = os.environ.get('CLIENT_ID')
client_secret = os.environ.get('CLIENT_SECRET')

# Authenticate and get an API Token from Spotify using a Client ID and secret
def getAuthTokenFromCredentials(id, secret):

    url = "https://accounts.spotify.com/api/token"

    payload =␣
 ↪f'grant_type=client_credentials&client_id={id}&client_secret={secret}'
    headers = {
    'Content-Type': 'application/x-www-form-urlencoded',
    }

    response = requests.request("POST", url, headers=headers, data=payload)

    return response.json()["access_token"]

auth_token = getAuthTokenFromCredentials(client_id, client_secret)
```

1.1.1 HTTP Setup

[ ]:
```
from requests.adapters import HTTPAdapter
from urllib3.util import Retry

DEFAULT_TIMEOUT = 10 # seconds

# This is used to configure timeouts and retries if the API takes a long time␣
 ↪to respond to the call
# It's crucial that theres some room for slow responses, as one failed request␣
 ↪will exit out of the whole function
class TimeoutHTTPAdapter(HTTPAdapter):
    def __init__(self, *args, **kwargs):
```

```
        self.timeout = DEFAULT_TIMEOUT
        if "timeout" in kwargs:
            self.timeout = kwargs["timeout"]
            del kwargs["timeout"]
        super().__init__(*args, **kwargs)

    def send(self, request, **kwargs):
        timeout = kwargs.get("timeout")
        if timeout is None:
            kwargs["timeout"] = self.timeout
        return super().send(request, **kwargs)

def setupRequestsSession():
    http = requests.Session()
    assert_status_hook = lambda response, *args, **kwargs: response.
 ↪raise_for_status()
    http.hooks["response"] = [assert_status_hook]

    retries = Retry(total=5, backoff_factor=1, status_forcelist=[429, 500, 502,␣
 ↪503, 504])
    adapter = TimeoutHTTPAdapter(max_retries=retries)
    http.mount("https://", adapter)
    http.mount("http://", adapter)

    return http
```

[ ]:
```
# get http session
http = setupRequestsSession()
```

1.2 Filter

[ ]:
```
category_filter = None

# comment this out if you don't want to set a filter
category_filter = ["hiphop", "pop", "country", "rock", "latin", "rnb", "mood",␣
 ↪"indie_alt",
                   "regional_mexican", "edm_dance", "inspirational", "chill",␣
 ↪"party", "roots",
                   "kpop", "instrumental", "ambient", "alternative",␣
 ↪"classical", "jazz", "soul",
                   "punk", "blues", "arab", "afro", "metal", "caribbean",␣
 ↪"funk"]

# tells data collection functions to not use filter if it's not set
if category_filter is not None:
    use_filter = True
```

```
else:
    use_filter = False
```

1.3 Data from file

[ ]:
```
data = {}

#To load data_object from file instead of rerunning the scripts, uncomment this:

#file = open(os.path.join("data_collection", "json", "tracks_full.json"))
#data = json.load(file)
```

1.4 Get Categories

[ ]:
```
# function definition

def getAllCategories(requests_session, auth_token, data_object,␣
 ↪use_category_filter=False, category_filter=None, write_to_file=False,␣
 ↪path_to_file=''):

    # Establishing the requests session
    http = requests_session

    # Establishing given data object
    data = data_object

    # First API call used to get the total amount of categories
    headers = { 'Authorization': f'Bearer {auth_token}' }
    url = "https://api.spotify.com/v1/browse/categories?
 ↪country=US&locale=en_US&limit=1"

    try:
        response = http.request("GET", url, headers=headers, data={})
        if response.status_code != requests.codes.ok:
            raise Exception
    except Exception as e:
        raise SystemExit(e)

    response.raise_for_status()
    categoryAmount = response.json()["categories"]["total"]

    # API only returns 50 items at a time. Offset can be used to gradually get␣
 ↪all items
    # Calculate number of pages with 50 items
    pages = int(math.ceil(categoryAmount/50))
    data = {"categories": []}
```

```
    # Second call gets all categories
    for x in range(pages):
        url = f"https://api.spotify.com/v1/browse/categories?
↪country=US&locale=en_US&limit=50&offset={x * 50}"

        try:
            response = http.request("GET", url, headers=headers, data={})
            if response.status_code != requests.codes.ok:
                raise Exception
        except Exception as e:
            raise SystemExit(e)

        # categories are stored in the data dictionary
        for el in response.json()["categories"]["items"]:
            if (use_category_filter == True and el["id"] in category_filter) or␣
↪use_category_filter == False:
                data["categories"].append({
                    "id": el["id"],
                    "name": el["name"]
                })

    if write_to_file == True:
        with open(path_to_file, 'w') as outfile:
            json.dump(data, outfile, indent=2)

    return data
```

[ ]:
```
# execution
data = getAllCategories(
    requests_session=http,
    auth_token=auth_token,
    data_object=data,
    use_category_filter=use_filter,
    category_filter=category_filter,
    write_to_file=True,
    path_to_file=os.path.join("json", "01_categories.json"))

print("got categories")
```

```
got categories
```

1.5 Get Playlists

[ ]:
```
# function definition

def getPlaylistsForCategories(requests_session, auth_token, data_object,␣
↪write_to_file=False, path_to_file=''):

    # Establishing the requests session
    http = requests_session

    # Establishing given object
    data = data_object

    for category in data["categories"]:

        category_id = category["id"]
        url = f"https://api.spotify.com/v1/browse/categories/{category_id}/
↪playlists?country=US&limit=1&offset=0"
        headers = { 'Authorization': f'Bearer {auth_token}' }

        try:
            response = http.request("GET", url, headers=headers, data={})
            if response.status_code != requests.codes.ok:
                raise Exception
        except Exception as e:
            raise SystemExit(e)

        categoryAmount = response.json()["playlists"]["total"]

        #Calculate number of pages with 50 items
        pages = int(math.ceil(categoryAmount/50))

        #Initialize playlist attribute
        category["playlists"] = []

        # Get 50 playlists at a time and increase offset by 50
        for page in range(pages):
            url = f"https://api.spotify.com/v1/browse/categories/{category_id}/
↪playlists?country=US&limit=50&offset={page * 50}"
            headers = { 'Authorization': f'Bearer {auth_token}' }
            try:
                response = http.request("GET", url, headers=headers, data={})
                if response.status_code != requests.codes.ok:
                    raise Exception
            except Exception as e:
                raise SystemExit(e)
```

```
            # Store playlists for each category
            i = 0
            for playlist in response.json()["playlists"]["items"]:
                if playlist["type"] == "playlist":
                    category["playlists"].append({
                        "id": playlist["id"],
                        "name": playlist["name"]
                    })
                i += 1

        if write_to_file == True:
            with open(path_to_file, 'w') as outfile:
                json.dump(data, outfile, indent=2)

    return data
```

[ ]:
```
# execution

data = getPlaylistsForCategories(
    requests_session=http,
    auth_token=auth_token,
    data_object=data,
    write_to_file=True,
    path_to_file=os.path.join("json", "02_playlists.json"))

print("got playlists")
```

```
got playlists
```

1.6 Get Tracks

[ ]:
```
# function definition

def getTracksOfPlaylists(requests_session, auth_token, data_object,␣
 ↪write_to_file=False, path_to_file=''):

    # Establishing the requests session
    http = requests_session

    # Establishing given object
    data = data_object

    for category in data["categories"]:

        # Get all tracks for all playlists in all categories
        for playlist in category["playlists"]:
```

```
            # First we call the API once to learn how many tracks are in the␣
↪playlist. This is indicated in the field "total"
            playlist_id = playlist["id"]
            url = f"https://api.spotify.com/v1/playlists/{playlist_id}/tracks?
↪market=US&limit=2&offset=0&fields=items(track(name,id,album(name,id),artists)),total&additi
            headers = { 'Authorization': f'Bearer {auth_token}' }

            try:
                response = http.request("GET", url, headers=headers, data={})
                if response.status_code != requests.codes.ok:
                    raise Exception
            except Exception as e:
                raise SystemExit(e)

            trackAmount = response.json()["total"]

            #Calculate number of pages with 50 items based on the number of␣
↪total tracks
            pages = int(math.ceil(trackAmount/50))

            #Initialize playlist attribute
            playlist["tracks"] = []

            # Get tracks on each page
            for page in range(pages):
                url = f"https://api.spotify.com/v1/playlists/{playlist_id}/
↪tracks?market=US&limit=50&offset={page * 50}&fields=items(track(name, id,␣
↪album(name, id), artists)), total&additional_types=track"
                headers = { 'Authorization': f'Bearer {auth_token}' }

                try:
                    response = http.request("GET", url, headers=headers,␣
↪data={})
                    if response.status_code != requests.codes.ok:
                        raise Exception
                except Exception as e:
                    raise SystemExit(e)

                i = 0
                for item in response.json()['items']:
                    track = item["track"]

                    # Some track elements will have value null, this throws␣
↪exception
                    if track is None:
                        continue
```

```
                    artists = []
                    # Contains all artists and their ids for each track
                    for artist in track["artists"]:
                        artists.append({
                            "id" : artist["id"],
                            "name" : artist["name"]
                        })

                    # This is all of the metadata saved for each track
                    playlist["tracks"].append({
                        "id": track["id"],
                        "name": track["name"],
                        "album" : {
                            "id" : track["album"]["id"],
                            "name" : track["album"]["name"]
                            },
                        "artists" : artists
                     })
                    i += 1

        if write_to_file == True:
            with open(path_to_file, 'w') as outfile:
                json.dump(data, outfile, indent=2)

    return data
```

[ ]:
```
# execution

data = getTracksOfPlaylists(
    requests_session=http,
    auth_token=auth_token,
    data_object=data,
    write_to_file=True,
    path_to_file=os.path.join("json", "03_tracks.json"))

print("got tracks")
```

```
got tracks
```

1.7 Get Features

[ ]:
```
# function definition

def getFeaturesOfTracks(requests_session, auth_token, data_object,␣
 ↪write_to_file=False, path_to_file=''):
```

```
    #Establishing the requests session
    http = requests_session

    #Establishing given object
    data = data_object

    for category in data["categories"]:

        for playlist in category["playlists"]:

            track_ids = [[]]

            # API endpoint is called using all track ids seperated by comma
            # This creates arrays of arrays containing 99 track ids
            # This is done because a maximum of 99 tracks can be requested at a␣
↪time
            for track in playlist["tracks"]:

                track_ids[len(track_ids)-1].append(track["id"])

                if len(track_ids[len(track_ids)-1]) > 99:
                    track_ids.append([])

            all_track_features = []

            # Get all features for all tracks for all playlists in all␣
↪categories
            for page in track_ids :
                # Each subarray is joined by comma and used for a get request
                comma_seperated_ids = ",".join(page)

                url = f"https://api.spotify.com/v1/audio-features?
↪ids={comma_seperated_ids}"
                headers = { 'Authorization': f'Bearer {auth_token}' }

                try:
                    response = http.request("GET", url, headers=headers,␣
↪data={})
                    if response.status_code != requests.codes.ok:
                        raise Exception
                except Exception as e:
                    raise SystemExit(e)

                # Combine lists
                all_track_features = all_track_features + response.
↪json()["audio_features"]
```

```
            i = 0
            for track in playlist["tracks"]:

                # Removing unneeded features to save a bit of space
                for entry in all_track_features:
                    if entry is not None and track["id"] == entry["id"] and␣
↪entry["type"] == "audio_features":
                        track["features"] = copy(entry)
                        del track["features"]["id"]
                        del track["features"]["type"]
                        del track["features"]["uri"]
                        del track["features"]["track_href"]
                        del track["features"]["analysis_url"]
                        break
                i += 1

        if write_to_file == True:
            with open(path_to_file, 'w') as outfile:
                json.dump(data, outfile, indent=2)

    return data
```

[ ]:
```
# execution

data = getFeaturesOfTracks(
    requests_session=http,
    auth_token=auth_token,
    data_object=data,
    write_to_file=True,
    path_to_file=os.path.join("json", "04_features.json"))

print("got features")
```

```
got features
```

1.8 Flatten JSON

This does not mutate any of the data collected except artist information.
To maintain one table row per track and remove redundancy, we collapse multiple artists into a comma seperated list.
This does not matter, as we won't use "artist" as a feature

[ ]:
```
# function definition

def flatten_json(data_object, write_to_file=True, path_to_file=''):

    data = data_object
```

```
    # variables
    temp = {}
    result = []

    # loop over categories
    for category in data["categories"]:
        path = "categories"

        for item in category:
                if item != "playlists":
                    key     = f"{path}.{item}"
                    value   = f"{category[item]}"
                    temp[key] = value
                else:
                    # loop over playlist
                    for playlist in category["playlists"]:
                        path = "categories.playlists"

                        for item in playlist:
                            if item != "tracks":
                                key     = f"{path}.{item}"
                                value   = f"{playlist[item]}"
                                temp[key] = value
                            else:
                                # loop over tracks
                                for track in playlist["tracks"]:
                                    path = "categories.playlists.tracks"
                                    for item in track:
                                        if item != "album" and item !=␣
↪"artists" and item != "features":
                                            key     = f"{path}.{item}"
                                            value   = f"{track[item]}"
                                            temp[key] = value

                                        # album data
                                        elif item == "album":
                                            for album in track["album"]:
                                                key     = f"{path}.album.
↪{album}"
                                                value   =␣
↪f"{track['album'][album]}"
                                                temp[key] = value

                                        # artist data (just name)
                                        # at this point, multiple datafields␣
↪are collapsed into a comma seperated list to maintain one table row per␣
↪track and remove redundancy
```

```
                                        # this does not matter for our␣
↪purposes, as we are not using artist names as features
                                        elif item == "artists":
                                            value = ""

                                            for artist in track["artists"]:
                                                if not value:
                                                    value = artist["name"]
                                                else:
                                                    value =␣
↪f"{value},{artist['name']}"

                                            key = f"{path}.artists"
                                            temp[key] = value

                                        # track features
                                        elif item == "features":

                                            for feature in track["features"]:
                                                key     = f"{path}.features.
↪{feature}"
                                                value   =␣
↪f"{track['features'][feature]}"
                                                temp[key] = value

                                    # At this point, temp contains a flat␣
↪dictionary with all nested fields (which represents one row in a table)
                                    # This is appended to the result, which is␣
↪an array containing all table rows
                                    result.append(copy(temp))

    if write_to_file == True:
        with open(path_to_file, 'w') as outfile:
            json.dump(result, outfile, indent=2)

    return result
```

[ ]:
```
# execution

print("flattening json...")
data = flatten_json(data, True,  os.path.join("json", "05_flat_data.json"))
print("done flattening json")
```

```
flattening json…
done flattening json
```

1.9 Create CSV from flat JSON data

[ ]:
```
# function definition

def json_to_csv(data_object, path_to_file=''):

    # open file to write to
    f = open(path_to_file, 'w')

    # create the csv writer object
    csv_writer = csv.writer(f)

    count = 0
    for line in data_object:
        if count == 0:

            # Writing headers of CSV file
            header = line.keys()
            csv_writer.writerow(header)
            count += 1

        # Writing data of CSV file
        csv_writer.writerow(line.values())

    f.close()
```

[ ]:
```
# execution

json_to_csv(data, os.path.join("final_result.csv"))
print("created csv")
```

```
created csv
```

Appendix 3: Coding for CRISP DM Process

This includes all code for the implementation of data understanding, preparation, modeling and evaluation.

crisp_dm_process

January 31, 2022

1 Imports

[174]:
```
import os
import pandas as pd
import seaborn as sns
import matplotlib as plt
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import validation_curve
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
from itertools import cycle
```

2 Preprocessing

2.0.1 Importing CSV from Data Collection

[175]:
```
df_import = pd.read_csv(os.path.join('data_collection', 'final_result.csv'))
df_import
```

[175]:

	categories.id	categories.name	categories.playlists.id	\
0	hiphop	Hip-Hop	37i9dQZF1DX0XUsuxWHRQd	
1	hiphop	Hip-Hop	37i9dQZF1DX0XUsuxWHRQd	
2	hiphop	Hip-Hop	37i9dQZF1DX0XUsuxWHRQd	
3	hiphop	Hip-Hop	37i9dQZF1DX0XUsuxWHRQd	
4	hiphop	Hip-Hop	37i9dQZF1DX0XUsuxWHRQd	
…	…	…	…	
158317	funk	Funk & Disco	37i9dQZF1DX0oFpWfPwcGv	
158318	funk	Funk & Disco	37i9dQZF1DX0oFpWfPwcGv	

```
158319          funk    Funk & Disco  37i9dQZF1DX0oFpWfPwcGv
158320          funk    Funk & Disco  37i9dQZF1DX0oFpWfPwcGv
158321          funk    Funk & Disco  37i9dQZF1DX0oFpWfPwcGv

       categories.playlists.name categories.playlists.tracks.id  \
0                      RapCaviar         2AaJeBEq3WLcfFW1y8svDf
1                      RapCaviar         7uLFOXgLrS90tEYPO1DGXy
2                      RapCaviar         2lUDBd7JrgAMltcp6dcd7D
3                      RapCaviar         0qHPxjC83zQYcxe39xSShx
4                      RapCaviar         2QIBJFl8DJR1mDh9GwfZef
…                            …                              …
158317           Disco Decadence         00Zum0eGUHgcE5l8MoNuUG
158318           Disco Decadence         2vLaES21zwbX1Rnmj56Bbb
158319           Disco Decadence         3eudp9ZxZAGaDBOuWGrW2D
158320           Disco Decadence         3YJx77Xx8JSwEoxqrkQO5c
158321           Disco Decadence         0Dj3iMM3fvHMOqWsqS64Fu

              categories.playlists.tracks.name  \
0                                  By Your Side
1                              Man in the Mirror
2                                    25 million
3                                      thailand
4                      Don’t Play (with Lil Baby)
…                                              …
158317     Turn the Beat Around - 7" Single Edit
158318                             I'll Be Around
158319           Got to Be Real - Single Version
158320                              Knock on Wood
158321  White Lines (Long Version) [Re-Recorded]

       categories.playlists.tracks.album.id  \
0                    2RrZgDND03MLu6pRJdTkz5
1                    1VxVQAgekwkFo8yoXvFZ8o
2                    1eVrpJbHRLBbioB9sb5b94
3                    1eVrpJbHRLBbioB9sb5b94
4                    2rLqUcipEjIKK9rma5OTN8
…                                        …
158317               7vlYGZ9hnsuC57PUiqy0WC
158318               6QVemXFGMR4OLvlXvtQVjg
158319               059jmsqbxhu2n78LMS0H3P
158320               07ojYfe9B08p7nmOL2kgNF
158321               5bl0ggdoV1nNEGuVkUG9c3

                  categories.playlists.tracks.album.name  \
0                                          By Your Side
1                                                B4 AVA
2                                        LIVE LIFE FAST
```

```
3                                        LIVE LIFE FAST
4                                      Hall of Fame 2.0
…                                                    …
158317          Never Gonna Let You Go (Expanded Edition)
158318                                          Spinners
158319                     Cheryl Lynn (Expanded Edition)
158320                                       Knock On Wood
158321  Hip Hop Soundtrack To The Concrete Jungle (Re-…

         categories.playlists.tracks.artists  \
0                                  Rod Wave
1                       A Boogie Wit da Hoodie
2                                Roddy Ricch
3                                Roddy Ricch
4                             Polo G,Lil Baby
…                                        …
158317                     Vicki Sue Robinson
158318                           The Spinners
158319                            Cheryl Lynn
158320                            Eddie Floyd
158321  Grandmaster Flash & The Furious Five

        categories.playlists.tracks.features.danceability  …  \
0                                                    0.649  …
1                                                    0.849  …
2                                                    0.793  …
3                                                    0.875  …
4                                                    0.684  …
…                                                        …  …
158317                                               0.707  …
158318                                               0.593  …
158319                                               0.830  …
158320                                               0.864  …
158321                                               0.814  …

        categories.playlists.tracks.features.loudness  \
0                                             -10.232
1                                              -4.241
2                                              -9.258
3                                             -10.562
4                                              -7.414
…                                                 …
158317                                         -7.540
158318                                         -8.698
158319                                         -7.462
158320                                        -12.918
158321                                         -7.525
```

```
        categories.playlists.tracks.features.mode  \
0                                               1
1                                               0
2                                               1
3                                               1
4                                               0
…                                             …
158317                                          0
158318                                          0
158319                                          1
158320                                          1
158321                                          1

        categories.playlists.tracks.features.speechiness  \
0                                                  0.0959
1                                                  0.0637
2                                                  0.1240
3                                                  0.2180
4                                                  0.3470
…                                                   …
158317                                             0.0720
158318                                             0.0680
158319                                             0.0448
158320                                             0.0365
158321                                             0.0376

        categories.playlists.tracks.features.acousticness  \
0                                                  0.03450
1                                                  0.17100
2                                                  0.01900
3                                                  0.00717
4                                                  0.23900
…                                                    …
158317                                             0.44500
158318                                             0.17500
158319                                             0.20200
158320                                             0.27700
158321                                             0.01030

        categories.playlists.tracks.features.instrumentalness  \
0                                                     0.000036
1                                                     0.000000
2                                                     0.000001
3                                                     0.000000
4                                                     0.000000
…                                                       …
```

```
158317                                              0.000000
158318                                              0.000000
158319                                              0.044800
158320                                              0.005210
158321                                              0.007750

        categories.playlists.tracks.features.liveness  \
0                                               0.0736
1                                               0.1490
2                                               0.1390
3                                               0.1470
4                                               0.1120
…                                                …
158317                                          0.3660
158318                                          0.0976
158319                                          0.1370
158320                                          0.0514
158321                                          0.1320

        categories.playlists.tracks.features.valence  \
0                                               0.405
1                                               0.550
2                                               0.395
3                                               0.409
4                                               0.708
…                                               …
158317                                          0.822
158318                                          0.630
158319                                          0.901
158320                                          0.964
158321                                          0.786

        categories.playlists.tracks.features.tempo  \
0                                          157.975
1                                          135.997
2                                          132.202
3                                          128.990
4                                          146.925
…                                            …
158317                                     131.242
158318                                     112.295
158319                                     114.646
158320                                     105.164
158321                                     114.998

        categories.playlists.tracks.features.duration_ms  \
0                                                194051
```

```
1                                                215304
2                                                204626
3                                                200959
4                                                156735
…                                                   …
158317                                           204653
158318                                           188800
158319                                           223173
158320                                           189840
158321                                           465048

        categories.playlists.tracks.features.time_signature
0                                                        4
1                                                        4
2                                                        4
3                                                        4
4                                                        4
…                                                      …
158317                                                   4
158318                                                   4
158319                                                   4
158320                                                   4
158321                                                   4

[158322 rows x 22 columns]
```

[176]:
```
df_import.dtypes
```

[176]:
```
categories.id                                        object
categories.name                                      object
categories.playlists.id                              object
categories.playlists.name                            object
categories.playlists.tracks.id                       object
categories.playlists.tracks.name                     object
categories.playlists.tracks.album.id                 object
categories.playlists.tracks.album.name               object
categories.playlists.tracks.artists                  object
categories.playlists.tracks.features.danceability   float64
categories.playlists.tracks.features.energy         float64
categories.playlists.tracks.features.key              int64
categories.playlists.tracks.features.loudness       float64
categories.playlists.tracks.features.mode             int64
categories.playlists.tracks.features.speechiness    float64
categories.playlists.tracks.features.acousticness   float64
categories.playlists.tracks.features.instrumentalness float64
categories.playlists.tracks.features.liveness       float64
categories.playlists.tracks.features.valence        float64
```

```
categories.playlists.tracks.features.tempo               float64
categories.playlists.tracks.features.duration_ms           int64
categories.playlists.tracks.features.time_signature        int64
dtype: object
```

2.0.2 Removing duplicates

Duplicates are removed by artist and track names. This ensures that tracks that were released twice (e.g. single before album) are still deduplicated. Using the track id as a deduplication criterion wouldn't achieve this.

[177]:
```
df_dedup = df_import.drop_duplicates(subset=['categories.playlists.tracks.
↪artists', 'categories.playlists.tracks.name'])
```

2.0.3 List all genre names

[178]:
```
genres = df_dedup['categories.name'].unique()
print(genres)
```

```
['Hip-Hop' 'Pop' 'Country' 'Rock' 'Latin' 'R&B' 'Mood' 'Indie'
 'Regional Mexican' 'Dance/Electronic' 'Christian & Gospel' 'Chill'
 'Party' 'Folk & Acoustic' 'K-Pop' 'Instrumental' 'Ambient' 'Alternative'
 'Classical' 'Jazz' 'Soul' 'Punk' 'Blues' 'Arab' 'Metal' 'Caribbean'
 'Funk & Disco']
```

2.0.4 Filter genres

[179]:
```
genre_filter = ['hiphop', 'jazz', 'rock']
df_filtered = df_dedup[df_dedup['categories.id'].isin(genre_filter)]
```

2.0.5 Drop unneeded columns and rename

[180]:
```
columns_to_drop = [
    "categories.name",
    "categories.playlists.id",
    "categories.playlists.name",
    "categories.playlists.tracks.id",
    "categories.playlists.tracks.name",
    "categories.playlists.tracks.album.id",
    "categories.playlists.tracks.album.name",
    "categories.playlists.tracks.artists"
    ]
df_dropped = df_filtered.drop(columns=columns_to_drop)

df_dropped = df_dropped.rename(columns={
    "categories.id": "category",
    "categories.playlists.tracks.features.danceability": "feature_danceability",
```

```
    "categories.playlists.tracks.features.energy": "feature_energy",
    "categories.playlists.tracks.features.key": "feature_key",
    "categories.playlists.tracks.features.loudness": "feature_loudness",
    "categories.playlists.tracks.features.mode": "feature_mode",
    "categories.playlists.tracks.features.speechiness": "feature_speechiness",
    "categories.playlists.tracks.features.acousticness": "feature_acousticness",
    "categories.playlists.tracks.features.instrumentalness":␣
 ↪"feature_instrumentalness",
    "categories.playlists.tracks.features.liveness": "feature_liveness",
    "categories.playlists.tracks.features.valence": "feature_valence",
    "categories.playlists.tracks.features.tempo": "feature_tempo",
    "categories.playlists.tracks.features.duration_ms": "feature_duration_ms",
    "categories.playlists.tracks.features.time_signature":␣
 ↪"feature_time_signature"
})
df_dropped = df_dropped.reset_index(drop=True)
df_dropped
```

[180]:

	category	feature_danceability	feature_energy	feature_key	\
0	hiphop	0.649	0.5080	8	
1	hiphop	0.849	0.6310	3	
2	hiphop	0.793	0.4810	9	
3	hiphop	0.875	0.4780	7	
4	hiphop	0.684	0.6240	2	
…	…	…	…	…	
13372	jazz	0.421	0.0952	6	
13373	jazz	0.503	0.4910	0	
13374	jazz	0.644	0.5940	5	
13375	jazz	0.462	0.2110	0	
13376	jazz	0.309	0.8370	2	

	feature_loudness	feature_mode	feature_speechiness	\
0	-10.232	1	0.0959	
1	-4.241	0	0.0637	
2	-9.258	1	0.1240	
3	-10.562	1	0.2180	
4	-7.414	0	0.3470	
…	…	…	…	
13372	-12.561	1	0.0479	
13373	-12.020	1	0.0295	
13374	-9.965	1	0.1170	
13375	-13.396	1	0.0586	
13376	-8.135	1	0.1310	

	feature_acousticness	feature_instrumentalness	feature_liveness	\
0	0.03450	0.000036	0.0736	
1	0.17100	0.000000	0.1490	

```
2                   0.01900                  0.000001            0.1390
3                   0.00717                  0.000000            0.1470
4                   0.23900                  0.000000            0.1120
…                     …                        …                 …
13372               0.93100                  0.000201            0.1260
13373               0.04120                  0.922000            0.0965
13374               0.75100                  0.224000            0.1070
13375               0.66500                  0.946000            0.1140
13376               0.08500                  0.621000            0.1430

       feature_valence  feature_tempo  feature_duration_ms  \
0               0.4050        157.975               194051
1               0.5500        135.997               215304
2               0.3950        132.202               204626
3               0.4090        128.990               200959
4               0.7080        146.925               156735
…                 …             …                    …
13372           0.0773        109.698               177922
13373           0.4890        166.105               263447
13374           0.6320         90.564               494467
13375           0.4260        179.658                77190
13376           0.3880        114.757               810322

       feature_time_signature
0                           4
1                           4
2                           4
3                           4
4                           4
…                         …
13372                       4
13373                       4
13374                       4
13375                       3
13376                       4

[13377 rows x 14 columns]
```

2.0.6 See, if data contains any null or NA values

[181]:
```
df_dropped.isnull().sum()
```

[181]:
```
category                  0
feature_danceability      0
feature_energy            0
feature_key               0
feature_loudness          0
```

```
feature_mode                0
feature_speechiness         0
feature_acousticness        0
feature_instrumentalness    0
feature_liveness            0
feature_valence             0
feature_tempo               0
feature_duration_ms         0
feature_time_signature      0
dtype: int64
```

3 Data Understanding

3.1 Basic Understanding

Show table head

[182]:
```
df_und = df_dropped
df_und.head()
```

[182]:

	category	feature_danceability	feature_energy	feature_key	feature_loudness	feature_mode	feature_speechiness	feature_acousticness	feature_instrumentalness	feature_liveness	feature_valence	feature_tempo	feature_duration_ms	feature_time_signature
0	hiphop	0.649	0.508	8	-10.232	1	0.0959	0.03450	0.000036	0.0736	0.405	157.975	194051	4
1	hiphop	0.849	0.631	3	-4.241	0	0.0637	0.17100	0.000000	0.1490	0.550	135.997	215304	4
2	hiphop	0.793	0.481	9	-9.258	1	0.1240	0.01900	0.000001	0.1390	0.395	132.202	204626	4
3	hiphop	0.875	0.478	7	-10.562	1	0.2180	0.00717	0.000000	0.1470	0.409	128.990	200959	4
4	hiphop	0.684	0.624	2	-7.414	0	0.3470	0.23900	0.000000	0.1120	0.708	146.925	156735	4

Show shape of data and samples per category

[183]:
```
# shape (rows, columns)
print('basic shape:  ', df_und.shape)

# amount of each category
print('amount of samples for ...')
print('hiphop:  ', df_und.loc[df_und['category'] == 'hiphop'].shape[0])
print('rock:   ' , df_und.loc[df_und['category'] == 'rock'].shape[0])
print('jazz:    ', df_und.loc[df_und['category'] == 'jazz'].shape[0])
```

```
basic shape:   (13377, 14)
amount of samples for …
hiphop:   2694
rock:    7252
jazz:     3431
```

3.2 Analysis of a single feature

Change the following variable feature to select which features should be analysed in this section.

[184]:
```
feature = 'feature_energy'
```

3.2.1 Basic feature information

[185]:
```
# overall basic feature information
print('overall:')
print(df_und[feature].describe())
#skewness and kurtosis

# skewness and kurtosis
print('\nskewness: %f' % df_und[feature].skew())
print('kurtosis: %f'   % df_und[feature].kurt())
```

```
overall:
count    13377.000000
mean         0.650739
std          0.235224
min          0.001530
25%          0.505000
50%          0.687000
75%          0.845000
max          0.999000
Name: feature_energy, dtype: float64

skewness: -0.689911
kurtosis: -0.254297
```

3.2.2 Category specific feature information

[186]:
```
df_hiphop   = df_und.loc[df['category'] == 'hiphop']
df_rock     = df_und.loc[df['category'] == 'rock']
df_jazz     = df_und.loc[df['category'] == 'jazz']
```

Hip-Hop

[187]:
```
print('hiphop:')
print(df_hiphop[feature].describe())

# skewness and kurtosis
print('\nskewness: %f' % df_hiphop[feature].skew())
print('kurtosis: %f'   % df_hiphop[feature].kurt())
```

```
hiphop:
count    2694.000000
mean        0.641614
std         0.149233
min         0.097600
25%         0.536000
50%         0.641000
75%         0.750000
max         0.995000
Name: feature_energy, dtype: float64

skewness: -0.118530
kurtosis: -0.365503
```

Jazz

[188]:
```
print('jazz:')
print(df_jazz[feature].describe())

# skewness and kurtosis
print('\nskewness: %f' % df_jazz[feature].skew())
print('kurtosis: %f'   % df_jazz[feature].kurt())
```

```
jazz:
count    3431.000000
mean        0.441448
std         0.248839
min         0.001530
25%         0.231500
50%         0.436000
75%         0.638000
max         0.992000
Name: feature_energy, dtype: float64
```

```
skewness: 0.126479
kurtosis: -1.005078
```

Rock

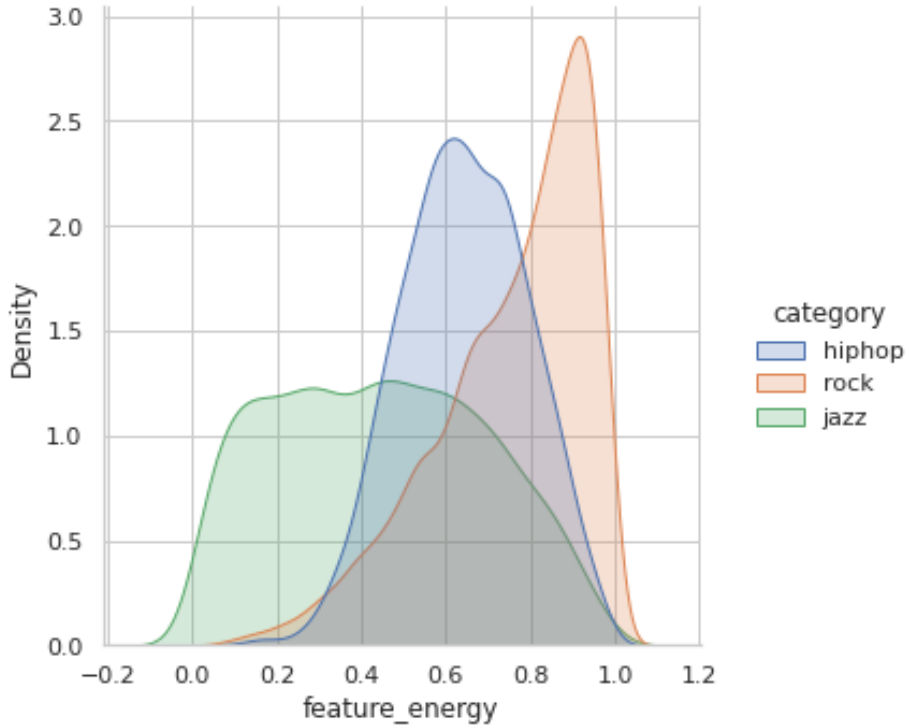
[189]:
```
print('rock:')
print(df_rock[feature].describe())

# skewness and kurtosis
print('\nskewness: %f' % df_rock[feature].skew())
print('kurtosis: %f'   % df_rock[feature].kurt())
```

```
rock:
count    7252.000000
mean        0.753147
std         0.182375
min         0.045700
25%         0.645000
50%         0.798500
75%         0.901000
max         0.999000
Name: feature_energy, dtype: float64

skewness: -0.935620
kurtosis: 0.327018
```

3.2.3 Graphical analysis of the selected feature

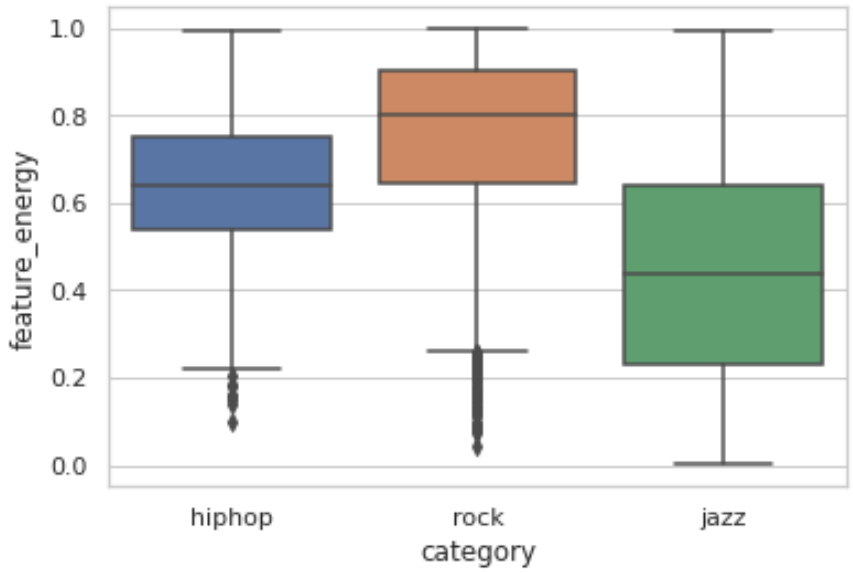
Distribution Plot

[190]:
```
sns.set(style="whitegrid")
ax = sns.displot(data = df_und, x = feature, hue = df_und['category'],␣
 ↪kind="kde",fill=True,common_norm=False)
```



Boxplot

[191]:
```
ax = sns.boxplot(data = df_und, x = 'category', y = feature)
```

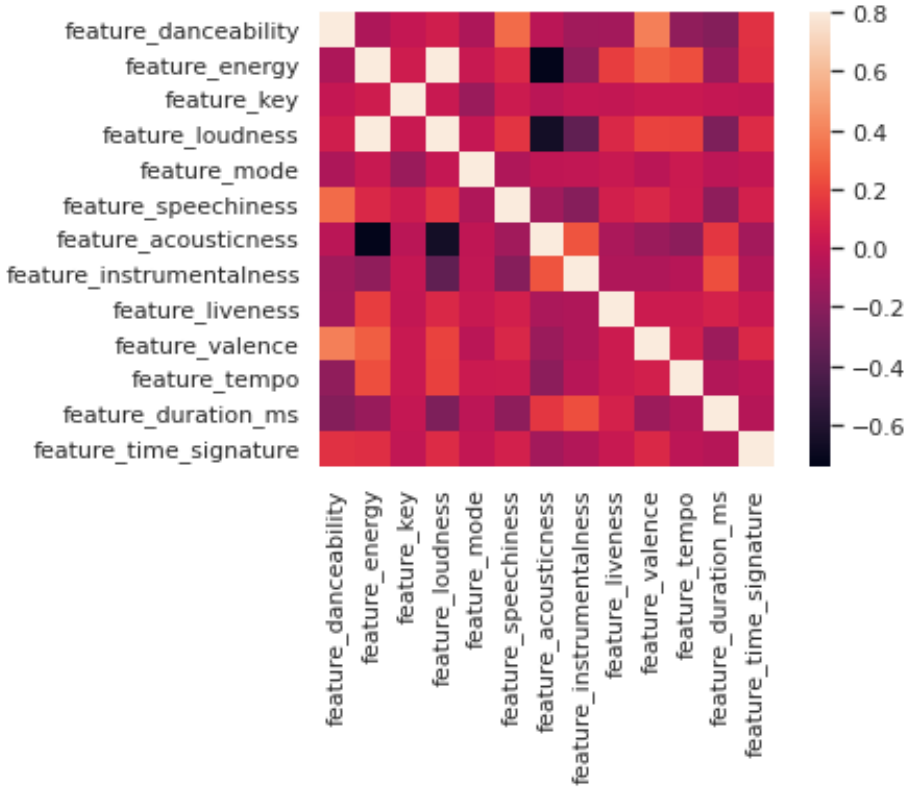


3.3 Correlation between all features

Overall Correlation

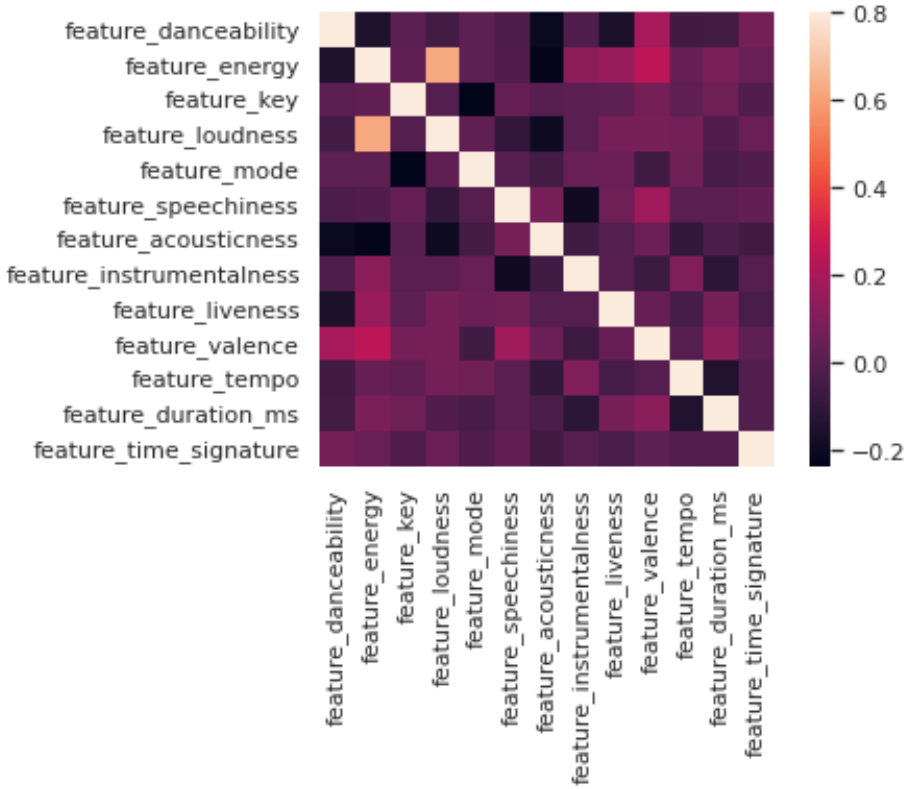
[192]:
```
corrmat = df_und.corr()
sns.heatmap(corrmat, vmax=.8, square=True)
```

[192]: <AxesSubplot:>



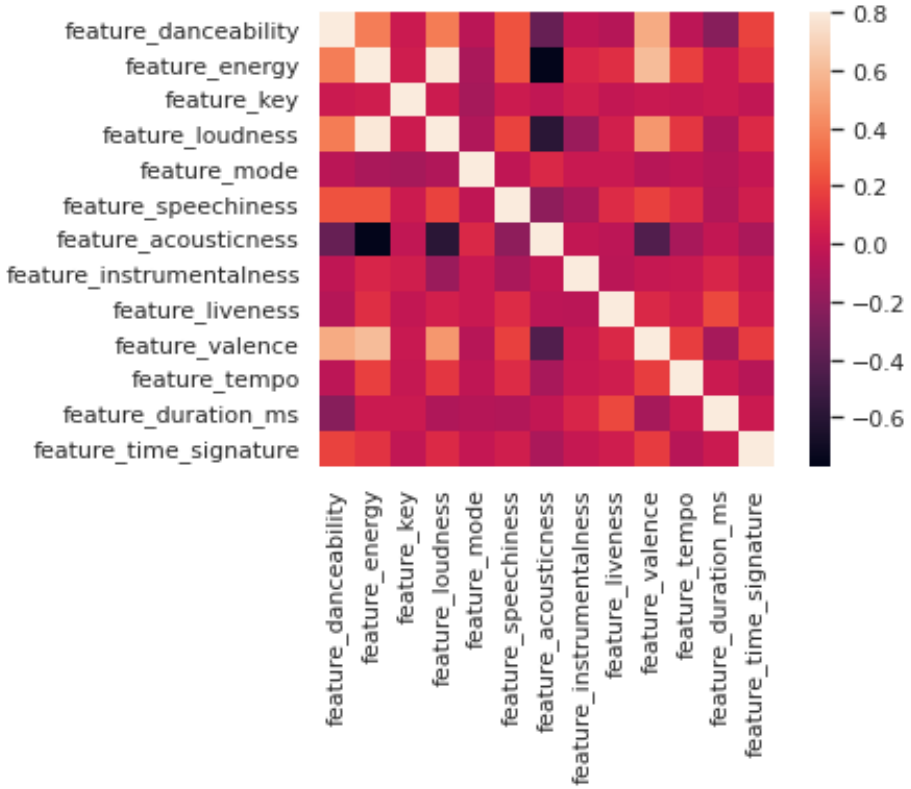
Correlations of Hip-Hop samples

[193]:
```
corrmat = df_hiphop.corr()
sns.heatmap(corrmat, vmax=.8, square=True);
```



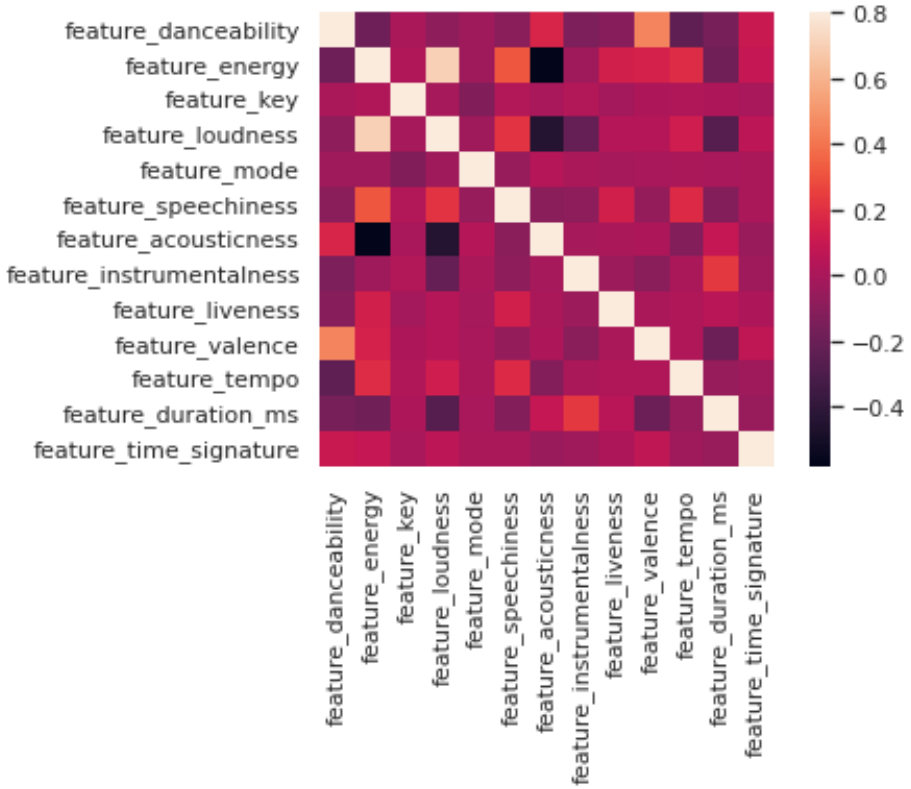
Correlations of Jazz samples

[194]:
```
corrmat = df_jazz.corr()
sns.heatmap(corrmat, vmax=.8, square=True);
```



Correlations of Rock samples

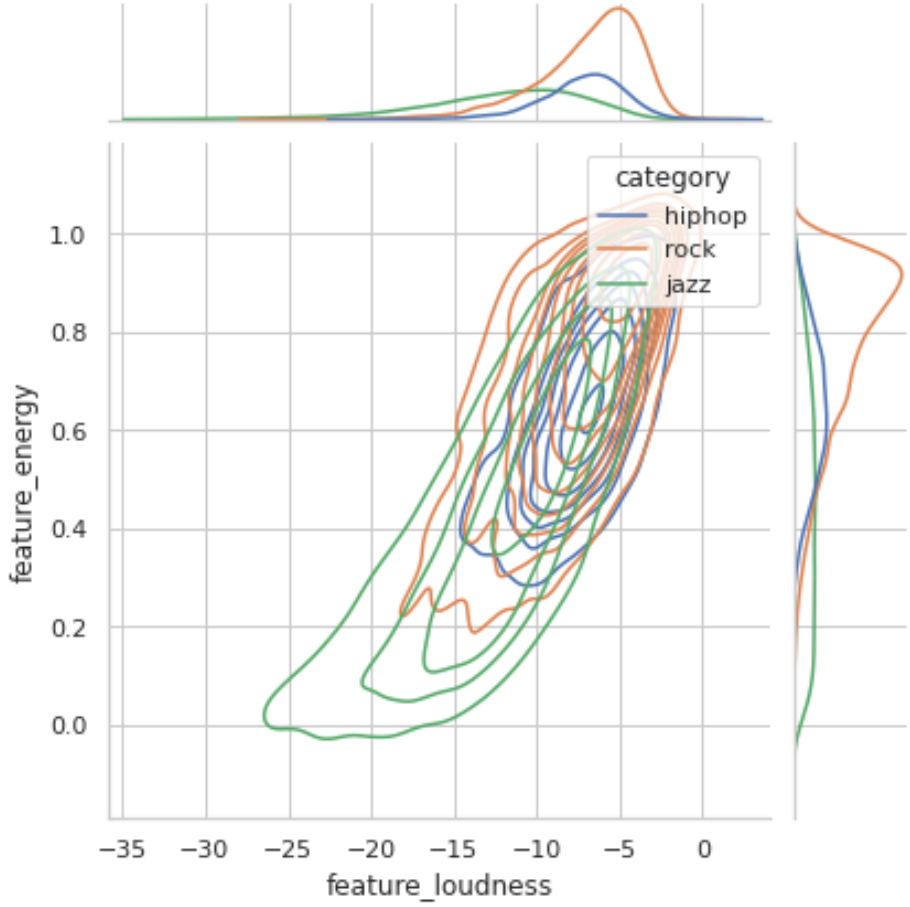
[195]:
```
corrmat = df_rock.corr()
sns.heatmap(corrmat, vmax=.8, square=True);
```



3.4 Correlation between two features

[196]:
```
feature_x = 'feature_loudness'
feature_y = 'feature_energy'

sns.jointplot(
    data = df_und,
    x    = feature_x,
    y    = feature_y,
    hue  = "category",
    kind = "kde")
```

[196]: <seaborn.axisgrid.JointGrid at 0x7f2eb5247490>



4 Data Preparation for Modeling

4.0.1 Map categories to integers

[197]:
```
def encode_target(df, target_column):

    df_mod = df.copy()
    map_to_int = {name: n for n, name in enumerate(df_mod["category"].unique())}
    df_mod.insert(1, "target", df_mod[target_column].replace(map_to_int))

    return df_mod
```

```
df = encode_target(df_dropped, "category")
df
```

[197]:

	category	target	feature_danceability	feature_energy	feature_key	\
0	hiphop	0	0.649	0.5080	8	
1	hiphop	0	0.849	0.6310	3	
2	hiphop	0	0.793	0.4810	9	
3	hiphop	0	0.875	0.4780	7	
4	hiphop	0	0.684	0.6240	2	
…	…	…	…	…	…	
13372	jazz	2	0.421	0.0952	6	
13373	jazz	2	0.503	0.4910	0	
13374	jazz	2	0.644	0.5940	5	
13375	jazz	2	0.462	0.2110	0	
13376	jazz	2	0.309	0.8370	2	

	feature_loudness	feature_mode	feature_speechiness	\
0	-10.232	1	0.0959	
1	-4.241	0	0.0637	
2	-9.258	1	0.1240	
3	-10.562	1	0.2180	
4	-7.414	0	0.3470	
…	…	…	…	
13372	-12.561	1	0.0479	
13373	-12.020	1	0.0295	
13374	-9.965	1	0.1170	
13375	-13.396	1	0.0586	
13376	-8.135	1	0.1310	

	feature_acousticness	feature_instrumentalness	feature_liveness	\
0	0.03450	0.000036	0.0736	
1	0.17100	0.000000	0.1490	
2	0.01900	0.000001	0.1390	
3	0.00717	0.000000	0.1470	
4	0.23900	0.000000	0.1120	
…	…	…	…	
13372	0.93100	0.000201	0.1260	
13373	0.04120	0.922000	0.0965	
13374	0.75100	0.224000	0.1070	
13375	0.66500	0.946000	0.1140	
13376	0.08500	0.621000	0.1430	

	feature_valence	feature_tempo	feature_duration_ms	\
0	0.4050	157.975	194051	
1	0.5500	135.997	215304	
2	0.3950	132.202	204626	
3	0.4090	128.990	200959	

```
4               0.7080        146.925               156735
…                …            …                  …
13372           0.0773        109.698               177922
13373           0.4890        166.105               263447
13374           0.6320         90.564               494467
13375           0.4260        179.658                77190
13376           0.3880        114.757               810322

       feature_time_signature
0                           4
1                           4
2                           4
3                           4
4                           4
…                         …
13372                       4
13373                       4
13374                       4
13375                       3
13376                       4

[13377 rows x 15 columns]
```

4.0.2 Show final integer mapping

[198]:
```
df[["target", "category"]].drop_duplicates(subset=['target', 'category'])
```

[198]:
```
      target category
0          0   hiphop
2694       1     rock
9946       2     jazz
```

4.1 Data Transformation and Dimension Reduction

Overview over all scores is given at the end of this section

Function for shuffling, splitting, fitting and scoring a simple model

[199]:
```
def eval_prep(df, feature_column_names, label_column_name):
    train, test = train_test_split(df, test_size=0.2, random_state=45,␣
 ↪shuffle=True)

    X_train = train[feature_column_names]
    X_test = test[feature_column_names]

    y_train = train[label_column_name]
    y_test = test[label_column_name]
```

```
    # Create gradient boosting classifier object with default values
    gbc = GradientBoostingClassifier(random_state=45)

    gbc.fit(X_train, y_train)
    score = gbc.score(X_test, y_test)

    return score

score_overview = []
```

4.1.1 Simple regular Gradient Boosting Model for reference

[200]:
```
features = list(df.columns[2:])

score = eval_prep(df, features, "target")
score_overview.append({"score": score, "desc": "Regular Gradient Boosting"})
```

4.1.2 Mean Removal

[201]:
```
X_mr = df[features]
y = df["target"]

X_mr = StandardScaler(with_std=False).fit_transform(X_mr)

df_mr = pd.DataFrame(data=X_mr)
df_mr.insert(0, "target", y)
df_mr.columns = ["target"] + features

score = eval_prep(df_mr, features, "target")

score_overview.append({"score": score, "desc": "Mean Removal"})
```

4.1.3 Variance Scaling

[202]:
```
X_vs = df[features]
y = df["target"]

X_vs = StandardScaler(with_mean=False).fit_transform(X_vs)

df_vs = pd.DataFrame(data=X_vs)
df_vs.insert(0, "target", y)
df_vs.columns = ["target"] + features

score = eval_prep(df_vs, features, "target")
```

```
score_overview.append({"score": score, "desc": "Variance Scaling"})
```

4.1.4 Standardization

[203]:
```
X_s = df[features]
y = df["target"]

X_s = StandardScaler().fit_transform(X_s)

df_s = pd.DataFrame(data=X_s)
df_s.insert(0, "target", y)
df_s.columns = ["target"] + features

score = eval_prep(df_s, features, "target")

score_overview.append({"score": score, "desc": "Standardization"})
```

4.1.5 PCA

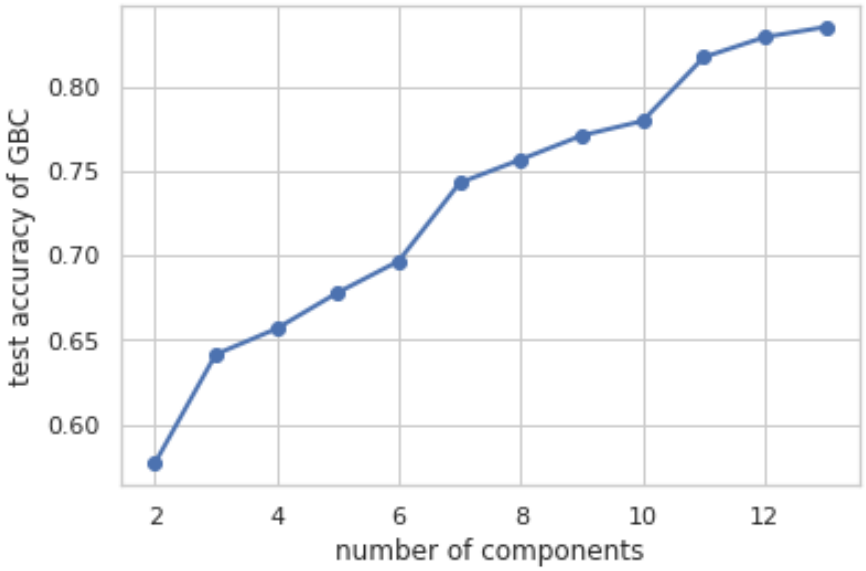
[204]:
```
scores = []

for n_components in range(2,14):

    X_pca = df[features]
    y = df["target"]

    # column names
    column_names = ["target"]
    for n in range(n_components):
        column_names.append(f"pc{n}")

    pca = PCA(n_components=n_components)

    X_pca = pca.fit_transform(X_pca)

    df_pca = pd.DataFrame(data=X_pca)

    df_pca.insert(0, "target", y)
    df_pca.columns = column_names

    score = eval_prep(df_pca, column_names[1:], "target")

    scores.append({"components": n_components, "score": score})

highscore = 0
optimal_components = 0
for entry in scores:
```

```
    if entry["score"] > highscore:
        highscore = entry["score"]
        optimal_components = entry["components"]

print(f"PCA with {optimal_components} components yielded optimal score:␣
 ↪{highscore}")

score_overview.append({"score": highscore, "desc": "Regular PCA"})
```

```
PCA with 13 components yielded optimal score: 0.8352017937219731
```

Plotting score for each dimensionality

[205]:
```
fig, ax = plt.subplots()

plt_scores = []
plt_comps = []
for e in scores:
    plt_scores.append(e['score'])
    plt_comps.append(e['components'])
ax.plot(plt_comps, plt_scores, linewidth=2.0, marker='o')
ax.set_xlabel('number of components')
ax.set_ylabel('test accuracy of GBC')
ax.grid(True)
```



4.1.6 Standardization after PCA

[206]:
```
scores = []

for n_components in range(2,14):

    X_pca_s = df[features]
    y = df["target"]

    # column names
    column_names = ["target"]
    for n in range(n_components):
        column_names.append(f"pc{n}")

    pca = PCA(n_components=n_components)

    X_pca_s = pca.fit_transform(X_pca_s)
    X_pca_s = StandardScaler().fit_transform(X_pca_s)

    df_pca_s = pd.DataFrame(data=X_pca_s)

    df_pca_s.insert(0, "target", y)
    df_pca_s.columns = column_names

    score = eval_prep(df_pca_s, column_names[1:], "target")

    scores.append({"components": n_components, "score": score})

highscore = 0
optimal_components = 0
for entry in scores:
    if entry["score"] > highscore:
        highscore = entry["score"]
        optimal_components = entry["components"]

print(f"PCA with {optimal_components} components yielded optimal score:␣
 ↪{highscore}")

score_overview.append({"score": highscore, "desc": "Standardization after PCA"})
```

```
PCA with 13 components yielded optimal score: 0.8348281016442451
```

4.1.7 Standardization before PCA

[207]:
```
scores = []

for n_components in range(2,14):
```

```
    X_pca_s = df[features]
    y = df["target"]

    # column names
    column_names = ["target"]
    for n in range(n_components):
        column_names.append(f"pc{n}")

    pca = PCA(n_components=n_components)

    X_pca_s = StandardScaler().fit_transform(X_pca_s)
    X_pca_s = pca.fit_transform(X_pca_s)

    df_pca_s = pd.DataFrame(data=X_pca_s)
    df_pca_s.insert(0, "target", y)
    df_pca_s.columns = column_names

    score = eval_prep(df_pca_s, column_names[1:], "target")

    scores.append({"components": n_components, "score": score})

highscore = 0
optimal_components = 0
for entry in scores:
    if entry["score"] > highscore:
        highscore = entry["score"]
        optimal_components = entry["components"]

print(f"PCA with {optimal_components} components yielded optimal score:␣
↪{highscore}")

score_overview.append({"score": highscore, "desc": "Standardization before␣
↪PCA"})
```

```
PCA with 13 components yielded optimal score: 0.827727952167414
```

4.1.8 Score Overview

[208]:
```
print("Score overview:")
highscore = 0
best_desc = ""
for entry in score_overview:
    if entry['score'] > highscore:
        highscore = entry['score']
        best_desc = entry['desc']
    print(f"{entry['desc']}: {entry['score']}")
```

```
print("\nMethod with highest score: " + best_desc)
```

```
Score overview:
Regular Gradient Boosting: 0.8505231689088192
Mean Removal: 0.850896860986547
Variance Scaling: 0.852017937219731
Standardization: 0.8523916292974589
Regular PCA: 0.8352017937219731
Standardization after PCA: 0.8348281016442451
Standardization before PCA: 0.827727952167414

Method with highest score: Standardization
```

The best score was achieved using the standardized dataset (df_s), so this will be used in the modeling stage.

5 Modeling

5.0.1 Basic examination of the standardized dataset

[209]:
```
df_s
```

[209]:

	target	feature_danceability	feature_energy	feature_key	\
0	0	0.525952	-0.606845	0.791667	
1	0	1.699036	-0.083920	-0.611631	
2	0	1.370572	-0.721633	1.072326	
3	0	1.851536	-0.734387	0.511007	
4	0	0.731242	-0.113680	-0.892290	
…	…	…	…	…	
13372	2	-0.811363	-2.361830	0.230348	
13373	2	-0.330399	-0.679119	-1.453609	
13374	2	0.496625	-0.241223	-0.050312	
13375	2	-0.570881	-1.869516	-1.453609	
13376	2	-1.468290	0.791872	-0.892290	

	feature_loudness	feature_mode	feature_speechiness	\
0	-0.469262	0.790819	-0.001432	
1	0.952395	-1.264512	-0.317468	
2	-0.238133	0.790819	0.274363	
3	-0.547571	0.790819	1.196953	
4	0.199446	-1.264512	2.463060	
…	…	…	…	
13372	-1.021931	0.790819	-0.472542	
13373	-0.893553	0.790819	-0.653134	
13374	-0.405903	0.790819	0.205660	
13375	-1.220076	0.790819	-0.367524	
13376	0.028354	0.790819	0.343067	

```
       feature_acousticness  feature_instrumentalness  feature_liveness  \
0                 -0.677224                 -0.560184         -0.743415
1                 -0.240457                 -0.560299         -0.296830
2                 -0.726820                 -0.560296         -0.356059
3                 -0.764673                 -0.560299         -0.308675
4                 -0.022873                 -0.560299         -0.515977
…                       …                         …                 …
13372              2.191359                 -0.559652         -0.433056
13373             -0.655785                  2.407617         -0.607781
13374              1.615403                  0.160756         -0.545591
13375              1.340224                  2.484873         -0.504131
13376             -0.515636                  1.438699         -0.332367

       feature_valence  feature_tempo  feature_duration_ms  \
0            -0.387093       1.175112            -0.468065
1             0.235137       0.449933            -0.270797
2            -0.430006       0.324715            -0.369909
3            -0.369929       0.218733            -0.403946
4             0.913153       0.810510            -0.814428
…                  …              …                    …
13372        -1.793333      -0.417819            -0.617773
13373        -0.026629       1.443366             0.176060
13374         0.587018      -1.049158             2.320361
13375        -0.296977       1.890556            -1.552755
13376        -0.460045      -0.250894             5.252090

       feature_time_signature
0                    0.189922
1                    0.189922
2                    0.189922
3                    0.189922
4                    0.189922
…                          …
13372                0.189922
13373                0.189922
13374                0.189922
13375               -2.667886
13376                0.189922

[13377 rows x 14 columns]
```

[210]:
```
df_s.describe()
```

[210]:
```
              target  feature_danceability  feature_energy   feature_key  \
count  13377.000000          1.337700e+04    1.337700e+04  1.337700e+04
mean       1.055095         -5.269181e-16    6.119049e-16 -1.699736e-17
std        0.674444          1.000037e+00    1.000037e+00  1.000037e+00
```

```
min        0.000000         -2.805019e+00   -2.760061e+00 -1.453609e+00
25%        1.000000         -7.116513e-01   -6.195990e-01 -8.922902e-01
50%        1.000000         -4.885906e-02    1.541591e-01 -5.031183e-02
75%        2.000000          7.077799e-01    8.258832e-01  7.916665e-01
max        2.000000          2.473271e+00    1.480602e+00  1.633645e+00

       feature_loudness  feature_mode  feature_speechiness  \
count      1.337700e+04  1.337700e+04         1.337700e+04
mean      -1.359789e-16 -2.124670e-18        -1.359789e-16
std        1.000037e+00  1.000037e+00         1.000037e+00
min       -5.649022e+00 -1.264512e+00        -7.228188e-01
25%       -4.571599e-01 -1.264512e+00        -5.844304e-01
50%        2.248371e-01  7.908191e-01        -4.391716e-01
75%        7.034689e-01  7.908191e-01         4.862320e-02
max        2.443819e+00  7.908191e-01         7.910265e+00

       feature_acousticness  feature_instrumentalness  feature_liveness  \
count          13377.000000              1.337700e+04      1.337700e+04
mean               0.000000              6.798944e-17     -6.374010e-18
std                1.000037              1.000037e+00      1.000037e+00
min               -0.787612             -5.602992e-01     -1.098789e+00
25%               -0.765761             -5.602992e-01     -6.012662e-01
50%               -0.526515             -5.588410e-01     -4.152875e-01
75%                0.527485             -2.272643e-02      3.132278e-01
max                2.399344              2.623290e+00      4.725779e+00

       feature_valence  feature_tempo  feature_duration_ms  \
count     1.337700e+04   1.337700e+04         1.337700e+04
mean     -3.399472e-17  -1.784723e-16         1.189815e-16
std       1.000037e+00   1.000037e+00         1.000037e+00
min      -2.015620e+00  -2.826733e+00        -2.052770e+00
25%      -7.904702e-01  -8.080589e-01        -5.583408e-01
50%      -1.804667e-02  -7.793114e-02        -1.942868e-01
75%       7.758331e-01   7.131066e-01         2.929470e-01
max       2.101827e+00   3.226484e+00         2.161469e+01

       feature_time_signature
count            1.337700e+04
mean            -1.019842e-16
std              1.000037e+00
min             -8.383503e+00
25%              1.899224e-01
50%              1.899224e-01
75%              1.899224e-01
max              3.047731e+00
```

5.1 Decision Tree Classifier for Comparison

5.1.1 Preparation

Splitting and shuffling the dataset

[211]:
```
train, test = train_test_split(df_s, test_size=0.2, random_state=45,␣
↪shuffle=True)
```

Seperating labels and features

[212]:
```
X_train = train[features]
X_test = test[features]

y_train = train["target"]
y_test = test["target"]
```

5.1.2 Hyperparameter Tuning

[213]:
```
dt = DecisionTreeClassifier(random_state=45)

param_grid = {
    "criterion": ["gini", "entropy"],
    "splitter": ["best", "random"],
    "max_depth": [30, 50, 75, None],
    "min_samples_split": [2, 5, 10, 15, 20],
    "min_samples_leaf": [1, 3, 5, 10],
    "min_samples_leaf": [1],
    "max_features": ["auto", "sqrt", "log2"],
    "random_state": [42],
    "max_features": [None]
}

# Grid search object
search_dt = GridSearchCV(dt, param_grid, n_jobs=-1, cv=5)

# Fitting model
search_dt.fit(X_train, y_train)

score_dt = search_dt.score(X_test, y_test)
```

5.2 Validation Curves

5.2.1 Function to create validation curves

This takes the hyperparameter that should be modulated and a range to use for plotting.

[214]:
```
def val_curve(param_name, param_range):
```

```
    X = df_s[features]
    y = df_s["target"]

    gbc = GradientBoostingClassifier(random_state=45)
    train_scores, test_scores = validation_curve(gbc, X, y,
        param_name=param_name,
        param_range=param_range,
        n_jobs=-1)

    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    #plt.title("Validation Curve with Gradient Boosting Classifier")
    plt.xlabel("learning_rate")
    plt.ylabel("accuracy")
    #plt.ylim(0.0, 1.1)
    lw = 2
    plt.plot(
        param_range, train_scores_mean, label="Training score",␣
↪color="darkorange", lw=lw
    )
    plt.fill_between(
        param_range,
        train_scores_mean - train_scores_std,
        train_scores_mean + train_scores_std,
        alpha=0.2,
        color="darkorange",
        lw=lw,
    )
    plt.plot(
        param_range, test_scores_mean, label="Cross-validation score",␣
↪color="navy", lw=lw
    )
    plt.fill_between(
        param_range,
        test_scores_mean - test_scores_std,
        test_scores_mean + test_scores_std,
        alpha=0.2,
        color="navy",
        lw=lw,
    )
    plt.legend(loc="best")
    plt.figure(dpi=200)
    plt.show()
```

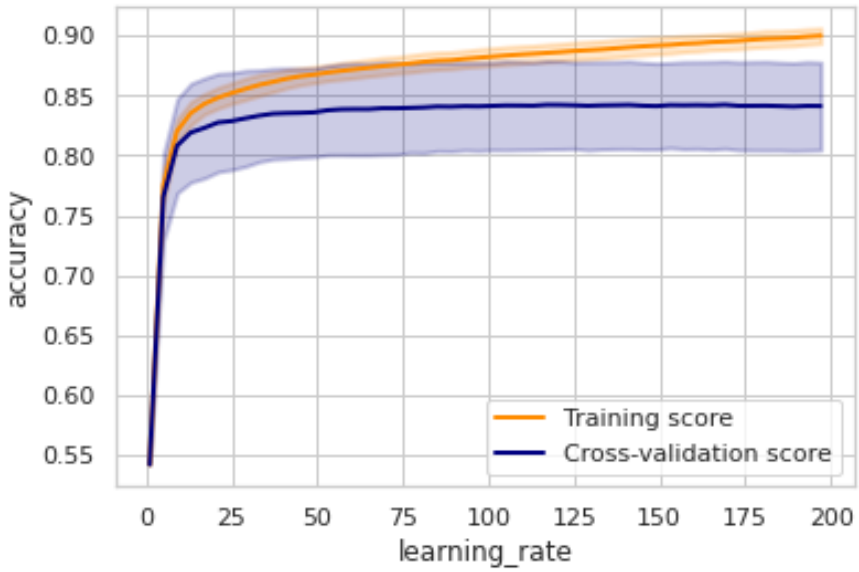
5.2.2 Validation curve learning_rate

[215]:
```
val_curve("learning_rate", np.linspace(0.01,0.99,50))
```


```
<Figure size 1200x800 with 0 Axes>
```

5.2.3 Validation curve n_estimators

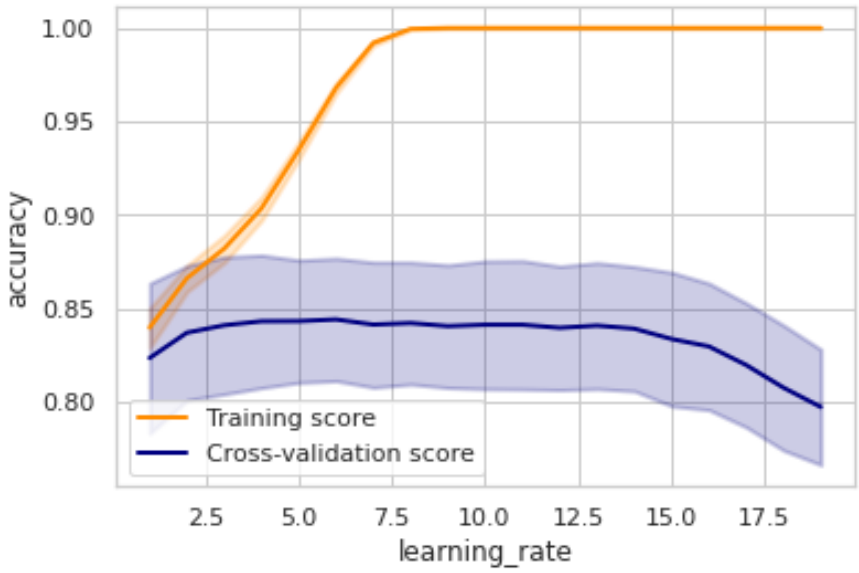
[216]:
```
val_curve("n_estimators", range(1,199,4))
```




```
<Figure size 1200x800 with 0 Axes>
```

5.2.4 Validation curve max_depth

[217]:
```
val_curve("max_depth", range(1,20,1))
```




```
<Figure size 1200x800 with 0 Axes>
```

5.3 Gradient Boosting Classifier

5.3.1 Preparation

Splitting and shuffling the dataset

[218]:
```
train, test = train_test_split(df_s, test_size=0.2, random_state=45,␣
↪shuffle=True)
```

Seperating labels and features

[219]:
```
X_train = train[features]
X_test = test[features]

y_train = train["target"]
y_test = test["target"]
```

5.3.2 Hyperparameter Tuning

Specifying parameter ranges for hyperparameter tuning

[220]:
```
learning_rate_values = np.linspace(0.01,0.25,9).round(2)
n_estimators_values = np.linspace(50,200,7).astype(int)
max_depth_values = np.linspace(2,6,5).astype(int)
```

```
param_grid = {
    "n_estimators": n_estimators_values,
    "learning_rate": learning_rate_values,
    "max_depth": max_depth_values
}

print("Values for learning_rate in grid search:")
print(learning_rate_values)
print("Values for n_estimators in grid search:")
print(n_estimators_values)
print("Values for max_depth in grid search:")
print(max_depth_values)
```

```
Values for learning_rate in grid search:
[0.01 0.04 0.07 0.1  0.13 0.16 0.19 0.22 0.25]
Values for n_estimators in grid search:
[ 50  75 100 125 150 175 200]
Values for max_depth in grid search:
[2 3 4 5 6]
```

Creating the Classifier and Grid Search objects

[221]:
```
gbc = GradientBoostingClassifier(random_state=45)

search = GridSearchCV(gbc, param_grid,
    n_jobs=-1,
    error_score="raise",
    verbose=1)
```

Fitting the first Grid Search model on the training data and calculating an accuracy score using the test set

[222]:
```
search.fit(X_train, y_train)

print("Accuracy score:")
print(search.score(X_test, y_test))
```

```
Fitting 5 folds for each of 315 candidates, totalling 1575 fits
Accuracy score:
0.859118086696562
```

Parameters found through Grid Search

[223]:
```
search.get_params()
```

[223]:
```
{'cv': None,
 'error_score': 'raise',
 'estimator__ccp_alpha': 0.0,
```

```
 'estimator__criterion': 'friedman_mse',
 'estimator__init': None,
 'estimator__learning_rate': 0.1,
 'estimator__loss': 'deviance',
 'estimator__max_depth': 3,
 'estimator__max_features': None,
 'estimator__max_leaf_nodes': None,
 'estimator__min_impurity_decrease': 0.0,
 'estimator__min_samples_leaf': 1,
 'estimator__min_samples_split': 2,
 'estimator__min_weight_fraction_leaf': 0.0,
 'estimator__n_estimators': 100,
 'estimator__n_iter_no_change': None,
 'estimator__random_state': 45,
 'estimator__subsample': 1.0,
 'estimator__tol': 0.0001,
 'estimator__validation_fraction': 0.1,
 'estimator__verbose': 0,
 'estimator__warm_start': False,
 'estimator': GradientBoostingClassifier(random_state=45),
 'n_jobs': -1,
 'param_grid': {'n_estimators': array([ 50,  75, 100, 125, 150, 175, 200]),
  'learning_rate': array([0.01, 0.04, 0.07, 0.1 , 0.13, 0.16, 0.19, 0.22,
0.25]),
  'max_depth': array([2, 3, 4, 5, 6])},
 'pre_dispatch': '2*n_jobs',
 'refit': True,
 'return_train_score': False,
 'scoring': None,
 'verbose': 1}
```

Fitting the second Grid Search Model This model uses a tighter range for the n_estimators parameter to try and further optimize the results

[236]:
```
param_grid_2 = {
    "n_estimators": [97,98,99,100,101,102,103],
    "learning_rate": learning_rate_values,
    "max_depth": max_depth_values
}

gbc2 = GradientBoostingClassifier(random_state=45)

search2 = GridSearchCV(gbc2, param_grid_2,
    n_jobs=-1,
    error_score="raise",
    verbose=1)
```

```
search2.fit(X_train, y_train)

print("Accuracy score:")
score_gb = search2.score(X_test, y_test)
print(score_gb)
```

```
Fitting 5 folds for each of 315 candidates, totalling 1575 fits
Accuracy score:
0.8568759342301944
```

Parameters of the final model

[237]:
```
search2.get_params()
```

[237]:
```
{'cv': None,
 'error_score': 'raise',
 'estimator__ccp_alpha': 0.0,
 'estimator__criterion': 'friedman_mse',
 'estimator__init': None,
 'estimator__learning_rate': 0.1,
 'estimator__loss': 'deviance',
 'estimator__max_depth': 3,
 'estimator__max_features': None,
 'estimator__max_leaf_nodes': None,
 'estimator__min_impurity_decrease': 0.0,
 'estimator__min_samples_leaf': 1,
 'estimator__min_samples_split': 2,
 'estimator__min_weight_fraction_leaf': 0.0,
 'estimator__n_estimators': 100,
 'estimator__n_iter_no_change': None,
 'estimator__random_state': 45,
 'estimator__subsample': 1.0,
 'estimator__tol': 0.0001,
 'estimator__validation_fraction': 0.1,
 'estimator__verbose': 0,
 'estimator__warm_start': False,
 'estimator': GradientBoostingClassifier(random_state=45),
 'n_jobs': -1,
 'param_grid': {'n_estimators': [97, 98, 99, 100, 101, 102, 103],
  'learning_rate': array([0.01, 0.04, 0.07, 0.1 , 0.13, 0.16, 0.19, 0.22,
0.25]),
  'max_depth': array([2, 3, 4, 5, 6])},
 'pre_dispatch': '2*n_jobs',
 'refit': True,
 'return_train_score': False,
 'scoring': None,
 'verbose': 1}
```

6 Evaluation

General sources for evaluation:
https://www.kaggle.com/vikumsw/guide-for-comprehensive-data-exploration-in-python/notebook
https://www.kaggle.com/pmarcelino/comprehensive-data-exploration-with-python

6.1 Evaluating the Decision Tree

6.1.1 Classification Report

[238]:
```
print(classification_report(y_test,search_dt.predict(X_test)))
```

```
              precision    recall  f1-score   support

           0       0.77      0.78      0.78       525
           1       0.85      0.85      0.85      1470
           2       0.74      0.73      0.74       681

    accuracy                           0.81      2676
   macro avg       0.79      0.79      0.79      2676
weighted avg       0.81      0.81      0.81      2676
```

6.1.2 Confusion Matrix

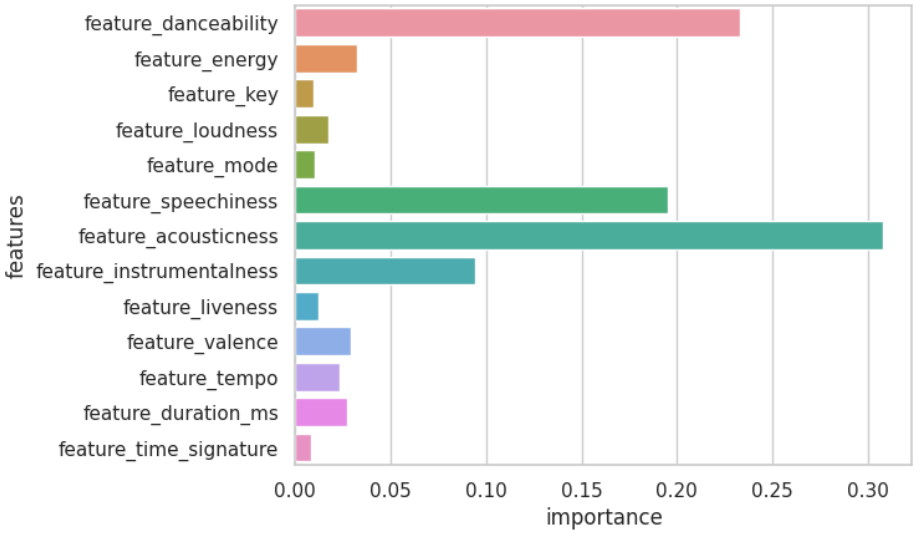
[239]:
```
df_confusion_dt = pd.crosstab(y_test, search_dt.predict(X_test),␣
↪rownames=['Actual'], colnames=['Predicted'], margins=True)
df_confusion_dt
```

[239]:
```
Predicted    0     1    2   All
Actual
0          408    84   33   525
1           77  1252  141  1470
2           42   139  500   681
All        527  1475  674  2676
```

6.1.3 Feature Importance

Source: https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html

[240]:
```
plt.style.use("default")
f = {'features' : features}
feature_importance = pd.DataFrame(f)
feature_importance['importance'] = search_dt.best_estimator_.
↪feature_importances_
sns.set_theme(style="whitegrid")
ax = sns.barplot(x="importance", y="features", data=feature_importance)
```



6.1.4 ROC

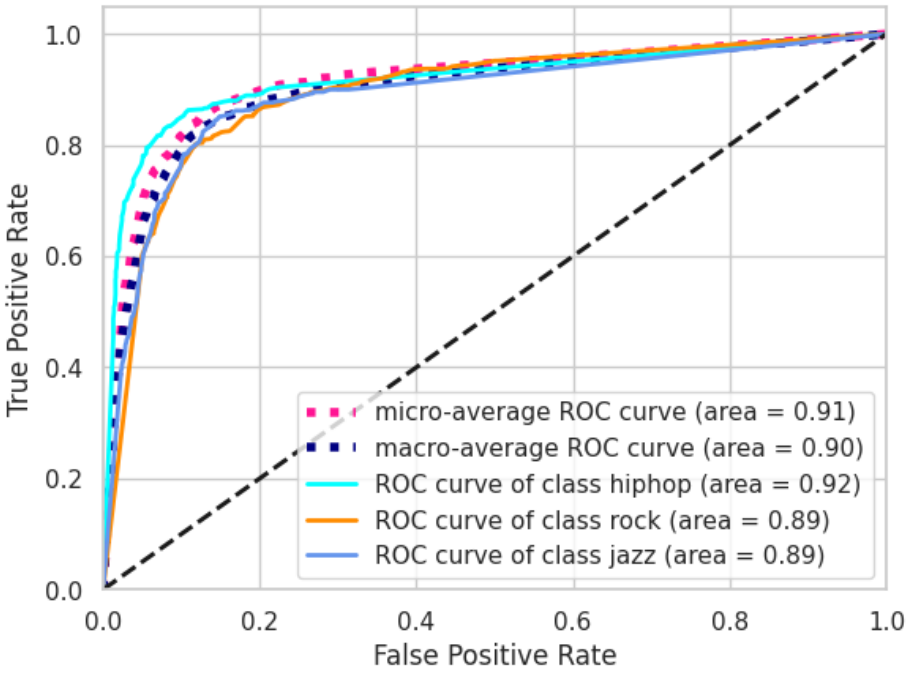
Source: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html

[241]:
```
y = label_binarize(y_test, classes=[0, 1, 2])
n_classes = y.shape[1]

# create x_score
y_test_new = []

for i in y_test:
    if i == 0:
        y_test_new.append([1, 0, 0])
    elif i == 1:
        y_test_new.append([0, 1, 0])
    elif i == 2:
        y_test_new.append([0, 0, 1])

# test input
y_test_np = np.asarray(y_test_new)

# score input
y_score = search_dt.predict_proba(X_test)

# Compute ROC curve and ROC area for each class
fpr = dict()
```

```
tpr = dict()
roc_auc = dict()

for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_np[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_np.ravel(), y_score.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

plt.figure()

lw = 2
# First aggregate all false positive rates
all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

# Then interpolate all ROC curves at this points
mean_tpr = np.zeros_like(all_fpr)
for i in range(n_classes):
    mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])

# Finally average it and compute AUC
mean_tpr /= n_classes

fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

# Plot all ROC curves
plt.figure()
plt.plot(
    fpr["micro"],
    tpr["micro"],
    label="micro-average ROC curve (area = {0:0.2f})".format(roc_auc["micro"]),
    color="deeppink",
    linestyle=":",
    linewidth=4,
)

plt.plot(
    fpr["macro"],
    tpr["macro"],
    label="macro-average ROC curve (area = {0:0.2f})".format(roc_auc["macro"]),
    color="navy",
    linestyle=":",
    linewidth=4,
```

```
)

colors = cycle(["aqua", "darkorange", "cornflowerblue"])
for i, color in zip(range(n_classes), colors):
    if i == 0:
        cat = 'hiphop'
    elif i == 1:
        cat = 'rock'
    elif i == 2:
        cat = 'jazz'

    plt.plot(
        fpr[i],
        tpr[i],
        color=color,
        lw=lw,
        label="ROC curve of class {0} (area = {1:0.2f})".format(cat,␣
 ↪roc_auc[i]),
    )

plt.plot([0, 1], [0, 1], "k--", lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend(loc="lower right")
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



6.2 Evaluating the Gradient Boosting Model

6.2.1 Classification Report

[242]:
```
print(classification_report(y_test,search2.predict(X_test)))
```

```
              precision    recall  f1-score   support

           0       0.85      0.84      0.85       525
           1       0.89      0.89      0.89      1470
           2       0.79      0.80      0.80       681

    accuracy                           0.86      2676
   macro avg       0.84      0.84      0.84      2676
weighted avg       0.86      0.86      0.86      2676
```

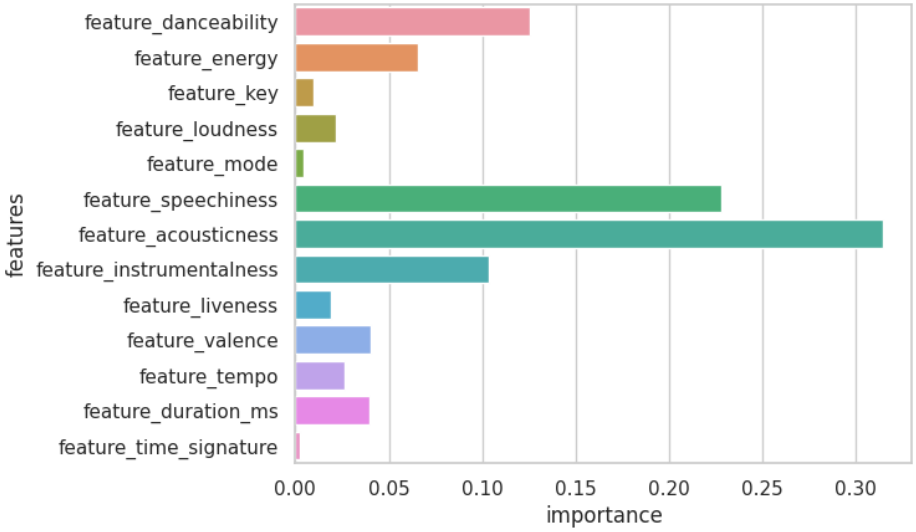
6.2.2 Confusion Matrix

[243]:
```
df_confusion_gb = pd.crosstab(y_test, search.predict(X_test),␣
↪rownames=['Actual'], colnames=['Predicted'], margins=True)
df_confusion_gb
```

[243]:
```
Predicted    0     1    2   All
Actual
0          438    62   25   525
1           43  1310  117  1470
2           31    99  551   681
All        512  1471  693  2676
```

6.2.3 Feature Importance

[244]:
```
f = {'features' : features}
feature_importance = pd.DataFrame(f)
feature_importance['importance'] = search2.best_estimator_.feature_importances_

sns.set_theme(style="whitegrid")
ax = sns.barplot(x="importance", y="features", data=feature_importance)
```



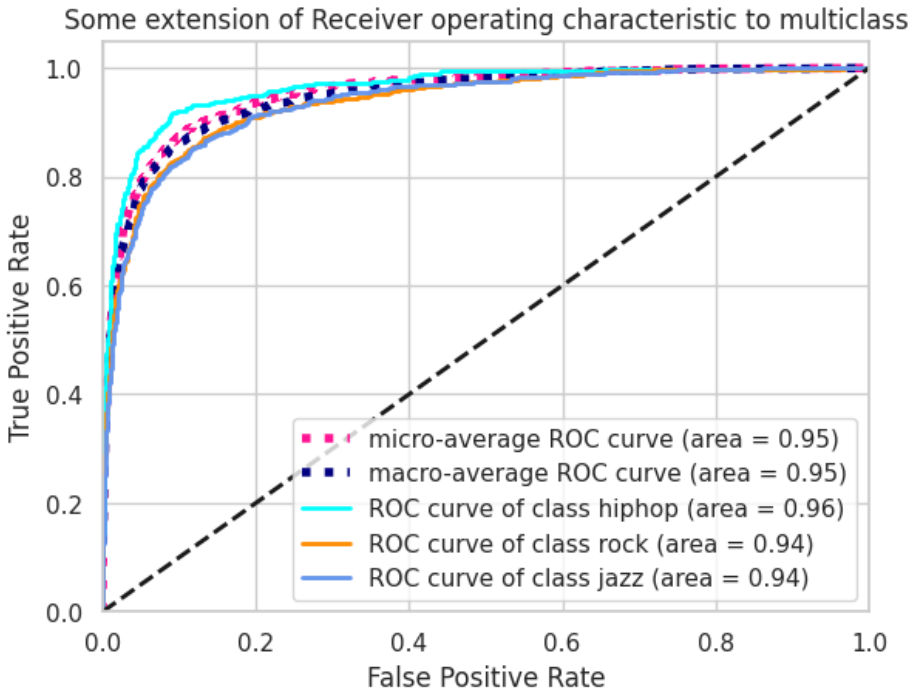
6.2.4 ROC

[246]:
```
y = label_binarize(y_test, classes=[0, 1, 2])
n_classes = y.shape[1]

n_classes

# create x_score
y_test_new = []

for i in y_test:
    if i == 0:
        y_test_new.append([1, 0, 0])
    elif i == 1:
        y_test_new.append([0, 1, 0])
    elif i == 2:
        y_test_new.append([0, 0, 1])

# test input
y_test_np = np.asarray(y_test_new)

# score input
y_score = search2.decision_function(X_test)


# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_np[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_np.ravel(), y_score.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

plt.figure()
lw = 2
# First aggregate all false positive rates
all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

# Then interpolate all ROC curves at this points
mean_tpr = np.zeros_like(all_fpr)
for i in range(n_classes):
    mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])
```

```
# Finally average it and compute AUC
mean_tpr /= n_classes

fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

# Plot all ROC curves
plt.figure()
plt.plot(
    fpr["micro"],
    tpr["micro"],
    label="micro-average ROC curve (area = {0:0.2f})".format(roc_auc["micro"]),
    color="deeppink",
    linestyle=":",
    linewidth=4,
)

plt.plot(
    fpr["macro"],
    tpr["macro"],
    label="macro-average ROC curve (area = {0:0.2f})".format(roc_auc["macro"]),
    color="navy",
    linestyle=":",
    linewidth=4,
)

colors = cycle(["aqua", "darkorange", "cornflowerblue"])

for i, color in zip(range(n_classes), colors):
    if i == 0:
        cat = 'hiphop'
    elif i == 1:
        cat = 'rock'
    elif i == 2:
        cat = 'jazz'

    plt.plot(
        fpr[i],
        tpr[i],
        color=color,
        lw=lw,
        label="ROC curve of class {0} (area = {1:0.2f})".format(cat,␣
 ↪roc_auc[i]),
    )

plt.plot([0, 1], [0, 1], "k--", lw=lw)
```

```
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Some extension of Receiver operating characteristic to multiclass")
plt.legend(loc="lower right")
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



Appendix 4: Coding for Decision Tree Theory

This is the code used to create the figures and tables for section 2.2

dt_algorithm

January 31, 2022

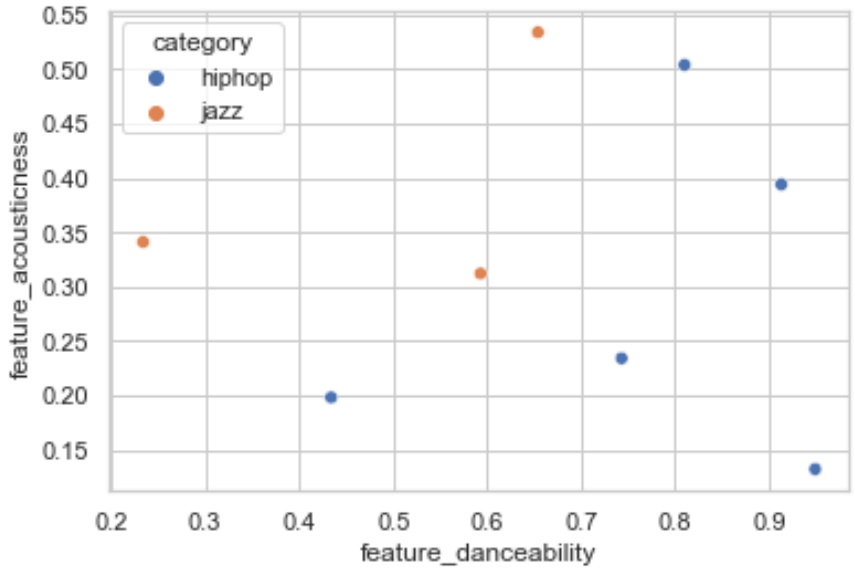
[64]:
```
import pandas as pd
import seaborn as sns
from sklearn.tree import DecisionTreeClassifier
import graphviz
from sklearn import tree
```

[65]:
```
d = {'category': ['hiphop', 'hiphop', 'hiphop', 'hiphop', 'hiphop', 'jazz',␣
↪'jazz', 'jazz'],
    'track':    ['h1', 'h2', 'h3', 'h4', 'h5', 'j1', 'j2', 'j3'],
    'feature_danceability' : [0.949, 0.743, 0.913, 0.810, 0.434, 0.654, 0.593,␣
↪0.234],
    'feature_acousticness' : [0.132, 0.234, 0.394, 0.504, 0.198, 0.534, 0.312,␣
↪0.341],
    'label'                : [1, 1, 1, 1, 1, 0, 0, 0]}
df = pd.DataFrame(data=d)

df
```

[65]:

	category	track	feature_danceability	feature_acousticness	label
0	hiphop	h1	0.949	0.132	1
1	hiphop	h2	0.743	0.234	1
2	hiphop	h3	0.913	0.394	1
3	hiphop	h4	0.810	0.504	1
4	hiphop	h5	0.434	0.198	1
5	jazz	j1	0.654	0.534	0
6	jazz	j2	0.593	0.312	0
7	jazz	j3	0.234	0.341	0

[67]:
```
sns.set(style="whitegrid")
sns.scatterplot(data = df, x = "feature_danceability", y =␣
↪"feature_acousticness", hue="category");
```



First Iteration

[68]:
```
# best possible Gini for danceability

threshold_array = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
#threshold_array = [0.4]
hiphop          = 0
jazz            = 0

# best split
best_treshold = 0
best_gain     = 1
best_left     = 0
best_right    = 0

for t in threshold_array:

    left  = df.loc[df['feature_danceability'] < t]
    right = df.loc[df['feature_danceability'] > t]

    if len(left.index) == 0 or len(right.index) == 0:
        continue

    hiphop = 0
    jazz   = 0
```

```
    # gini left
    for i, row in enumerate(left.values):
        # hiphop
        if row[4] == 1:
            hiphop = hiphop + 1
        # jazz
        else:
            jazz   = jazz + 1

    gini_left = 1 - pow((hiphop/len(left.index)), 2) - pow((jazz/len(left.
↪index)), 2)

    hiphop = 0
    jazz   = 0

    # gini right
    for i, row in enumerate(right.values):
        # hiphop
        if row[4] == 1:
            hiphop = hiphop + 1
        # jazz
        else:
            jazz   = jazz + 1

    gini_right = 1 - pow((hiphop/len(right.index)), 2) - pow((jazz/len(right.
↪index)), 2)

    gain = (len(left.index)/len(df.index)) * gini_left + (len(right.index)/
↪len(df.index)) * gini_right

    if gain < best_gain:
        best_gain       = gain
        best_treshold   = t
        best_left       = gini_left
        best_right      = gini_right

print(best_gain)
print(best_treshold)
```

```
0.1875
0.7
```

[69]:
```
# best possible Gini for danceability

threshold_array = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
#threshold_array = [0.4]
```

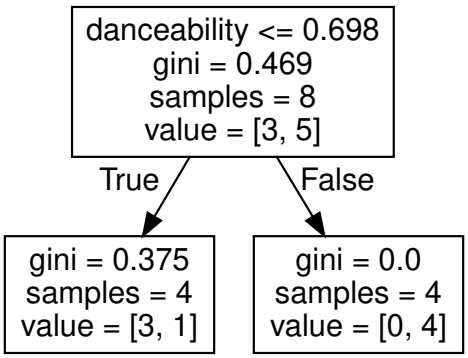
```
hiphop          = 0
jazz            = 0

# best split
best_treshold = 0
best_gain     = 1
best_left     = 0
best_right    = 0

for t in threshold_array:

    left  = df.loc[df['feature_acousticness'] < t]
    right = df.loc[df['feature_acousticness'] > t]

    if len(left.index) == 0 or len(right.index) == 0:
        continue

    hiphop = 0
    jazz   = 0

    # gini left
    for i, row in enumerate(left.values):
        # hiphop
        if row[4] == 1:
            hiphop = hiphop + 1
        # jazz
        else:
            jazz   = jazz + 1

    gini_left = 1 - pow((hiphop/len(left.index)), 2) - pow((jazz/len(left.
↪index)), 2)

    hiphop = 0
    jazz   = 0

    # gini right
    for i, row in enumerate(right.values):
        # hiphop
        if row[4] == 1:
            hiphop = hiphop + 1
        # jazz
        else:
            jazz   = jazz + 1

    gini_right = 1 - pow((hiphop/len(right.index)), 2) - pow((jazz/len(right.
↪index)), 2)
```

```
    gain = (len(left.index)/len(df.index)) * gini_left + (len(right.index)/
↪len(df.index)) * gini_right

    if gain < best_gain:
        best_gain       = gain
        best_treshold   = t
        best_left       = gini_left
        best_right      = gini_right

print(best_gain)
print(best_treshold)
```

```
0.3
0.3
```

[70]:
```
# build regression tree

# input values
features = ['feature_danceability']
Y = df['label']
X = df[features]

# build tree
dt = DecisionTreeClassifier(max_depth = 1)
dt.fit(X, Y)

# display
dot_data = tree.export_graphviz(
    dt,
    feature_names = ['danceability']
    )
graph = graphviz.Source(dot_data)

graph
```

[70]:



Second Iteration

[71]:
```
# right leaf is homogenous

# -> no further actions required
```

[72]:
```
# left leaf is not homogenous and has to be processed further

df_second_iteration = df.loc[df['feature_danceability'] < 0.700]

df_second_iteration
```

[72]:

	category	track	feature_danceability	feature_acousticness	label
4	hiphop	h5	0.434	0.198	1
5	jazz	j1	0.654	0.534	0
6	jazz	j2	0.593	0.312	0
7	jazz	j3	0.234	0.341	0

[73]:
```
# best possible Gini for danceability

threshold_array = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
#threshold_array = [0.4]
hiphop        = 0
jazz          = 0

# best split
best2_treshold = 0
best2_gain     = 1
best2_left     = 0
best2_right    = 0

for t in threshold_array:

    left  = df_second_iteration.loc[df['feature_acousticness'] < t]
    right = df_second_iteration.loc[df['feature_acousticness'] > t]

    if len(left.index) == 0 or len(right.index) == 0:
        continue

    hiphop = 0
    jazz   = 0

    # gini left
    for i, row in enumerate(left.values):
        # hiphop
```

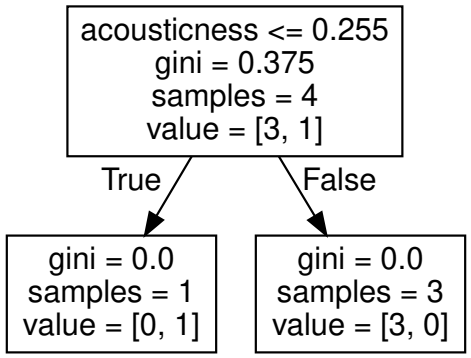
```
        if row[4] == 1:
            hiphop = hiphop + 1
        # jazz
        else:
            jazz   = jazz + 1

    gini_left = 1 - pow((hiphop/len(left.index)), 2) - pow((jazz/len(left.
↪index)), 2)

    hiphop = 0
    jazz   = 0

    # gini right
    for i, row in enumerate(right.values):
        # hiphop
        if row[4] == 1:
            hiphop = hiphop + 1
        # jazz
        else:
            jazz   = jazz + 1

    gini_right = 1 - pow((hiphop/len(right.index)), 2) - pow((jazz/len(right.
↪index)), 2)

    gain = (len(left.index)/len(df_second_iteration.index)) * gini_left +␣
↪(len(right.index)/len(df_second_iteration.index)) * gini_right

    if gain < best_gain:
        best2_gain       = gain
        best2_treshold   = t
        best2_left       = gini_left
        best2_right      = gini_right

print(best2_gain)
print(best2_treshold)
```

```
0.0
0.3
```

[74]:
```
# build regression tree

# input values
features = ['feature_acousticness']
Y = df_second_iteration['label']
X = df_second_iteration[features]

# build tree
```

```
dt2 = DecisionTreeClassifier(max_depth = 1)
dt2.fit(X, Y)

# display
dot_data = tree.export_graphviz(
    dt2,
    feature_names = ['acousticness']
    )
graph = graphviz.Source(dot_data)

graph
```

[74]:

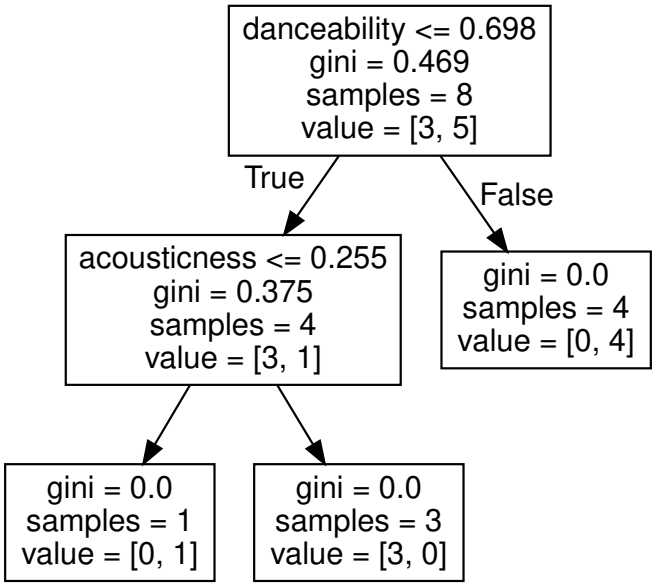


[75]:

```
# overall calculation

# build regression tree

# input values
features = ['feature_danceability', 'feature_acousticness']
Y = df['label']
X = df[features]

# build tree
dt3 = DecisionTreeClassifier(max_depth = 3)
dt3.fit(X, Y)

# display
dot_data = tree.export_graphviz(
    dt3,
    feature_names = ['danceability', 'acousticness']
    )
graph = graphviz.Source(dot_data)
```

```
graph
```

[75]:


danceability <= 0.698
gini = 0.469
samples = 8
value = [3, 5]
True
False
acousticness <= 0.255
gini = 0.375
samples = 4
value = [3, 1]
gini = 0.0
samples = 4
value = [0, 4]
gini = 0.0
samples = 1
value = [0, 1]
gini = 0.0
samples = 3
value = [3, 0]

Appendix 5: Coding for Gradient Boosting Theory

This is the code used to create the figures and tables from section 2.3

gb_algorithm

January 31, 2022

[2]:
```
import pandas as pd
import math
from sklearn.tree import DecisionTreeRegressor
import graphviz
from sklearn import tree
```

Dataset:

[3]:
```
d = {'category': ['hiphop', 'hiphop', 'hiphop', 'hiphop', 'hiphop', 'jazz',␣
↪'jazz', 'jazz'],
     'track':    ['h1', 'h2', 'h3', 'h4', 'h5', 'j1', 'j2', 'j3'],
     'feature_danceability' : [0.949, 0.743, 0.913, 0.810, 0.434, 0.654, 0.593,␣
↪0.234],
     'feature_acousticness' : [0.132, 0.234, 0.394, 0.504, 0.198, 0.534, 0.312,␣
↪0.341],
     'label'                : [1, 1, 1, 1, 1, 0, 0, 0]}
df = pd.DataFrame(data=d)

df
```

[3]:

	category	track	feature_danceability	feature_acousticness	label
0	hiphop	h1	0.949	0.132	1
1	hiphop	h2	0.743	0.234	1
2	hiphop	h3	0.913	0.394	1
3	hiphop	h4	0.810	0.504	1
4	hiphop	h5	0.434	0.198	1
5	jazz	j1	0.654	0.534	0
6	jazz	j2	0.593	0.312	0
7	jazz	j3	0.234	0.341	0

Initial Prediction:

[4]:
```
hiphop  = 0
jazz    = 0

for i, row in enumerate(df.values):
    # hiphop
    if row[4] == 1:
```

```
        hiphop  = hiphop + 1
    # jazz
    else:
        jazz    = jazz + 1

# calculate initial predictions

f0x = math.log(hiphop/jazz)

f0x
```

[4]: 0.5108256237659907

[5]:
```
# append initial prediction to dataset (equal for every sample)
f0x_array = []

for i, row in enumerate(df.values):
    f0x_array.append (f0x)

df['f0x'] = f0x_array

df
```

[5]:

	category	track	feature_danceability	feature_acousticness	label	f0x
0	hiphop	h1	0.949	0.132	1	0.510826
1	hiphop	h2	0.743	0.234	1	0.510826
2	hiphop	h3	0.913	0.394	1	0.510826
3	hiphop	h4	0.810	0.504	1	0.510826
4	hiphop	h5	0.434	0.198	1	0.510826
5	jazz	j1	0.654	0.534	0	0.510826
6	jazz	j2	0.593	0.312	0	0.510826
7	jazz	j3	0.234	0.341	0	0.510826

Iterative Process:

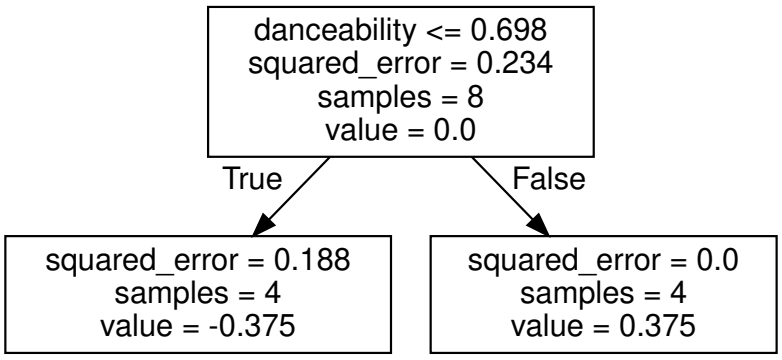
First Iteration:

[6]:
```
# calculate probabilites
prob_array = []
e = 2.718

for i, row in enumerate(df.values):
    prob = math.pow(e, row[5]) / (1 + math.pow(e, row[5]))
    prob_array.append(prob)

df['probability_f0x'] = prob_array
```

[7]:
```
# calculate pseudo residuals
residual_array = []

for i, row in enumerate(df.values):
    residual = row[4] - row[6]
    residual_array.append(residual)

df['pseudo_residuals_1'] = residual_array
```

[8]:
```
# build regression tree

# input values
features = ['feature_danceability']
Y = df['pseudo_residuals_1']
X = df[features]

# build tree
dt = DecisionTreeRegressor(max_depth = 1)
dt.fit(X, Y)

# display
dot_data = tree.export_graphviz(
    dt,
    feature_names = ['danceability']
    )
graph = graphviz.Source(dot_data)

graph
```

[8]:



[9]:
```
# calculate output values

sum_pr    = 0
```

```
sum_prob = 0

# leafs
left = df.loc[df['feature_danceability'] < 0.698]
right = df.loc[df['feature_danceability'] > 0.698]

# left leaf
for i, row in enumerate(left.values):
    sum_pr   = sum_pr + row[7]
    sum_prob = sum_prob + (row[6] * (1 - row[6]))

ov_left = sum_pr / sum_prob

sum_pr   = 0
sum_prob = 0

for i, row in enumerate(right.values):
    sum_pr   = sum_pr + row[7]
    sum_prob = sum_prob + (row[6] * (1 - row[6]))

ov_right = sum_pr / sum_prob

# fill table
ov_array = []

for i, row in enumerate(df.values):
    if row[2] < 0.698:
        ov = ov_left
    else:
        ov = ov_right
    ov_array.append(ov)

print(ov_left)
print(ov_right)

df['output_value_1'] = ov_array
```

```
-1.599925851127446
1.6000317795910484
```

[10]:
```
# new predictions
learning_rate = 0.6

f1x_array = []

for i, row in enumerate(df.values):
    f1x = row[5] + (learning_rate * row[8])
```

```
    f1x_array.append(f1x)

df['f1x'] = f1x_array
```

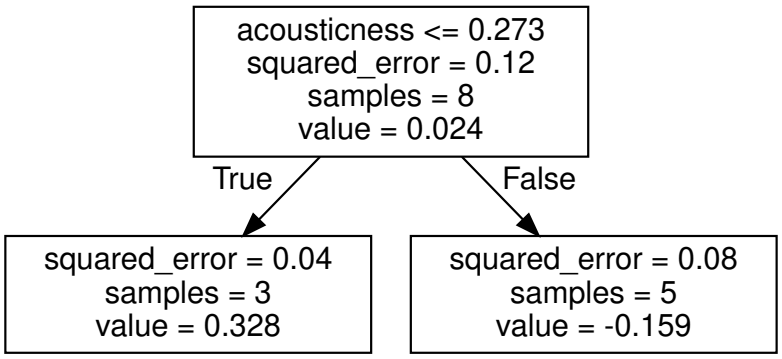
Second Iteration:

[11]:
```
# calculate probabilites
prob_array = []

for i, row in enumerate(df.values):
    prob = math.pow(e, row[9]) / (1 + math.pow(e, row[9]))
    prob_array.append(prob)

df['probability_f1x'] = prob_array
```

[12]:
```
# calculate pseudo residuals
residual_array = []

for i, row in enumerate(df.values):
    residual = row[4] - row[10]
    residual_array.append(residual)

df['pseudo_residuals_2'] = residual_array
```

[13]:
```
# build regression tree

# input values
features = ['feature_acousticness']
Y = df['pseudo_residuals_2']
X = df[features]

# build tree
dt2 = DecisionTreeRegressor(max_depth = 1)
dt2.fit(X, Y)

# display
dot_data = tree.export_graphviz(
    dt2,
    feature_names = ['acousticness']
    )
graph = graphviz.Source(dot_data)

graph
```

[13]:



[14]:
```
# calculate output values

sum_pr   = 0
sum_prob = 0

# leafs
left = df.loc[df['feature_acousticness'] < 0.273]
right = df.loc[df['feature_acousticness'] > 0.273]

# left leaf
for i, row in enumerate(left.values):
    sum_pr   = sum_pr + row[11]
    sum_prob = sum_prob + (row[10] * (1 - row[10]))

ov_left = sum_pr / sum_prob

print(sum_pr)
print(sum_prob)

sum_pr   = 0
sum_prob = 0

for i, row in enumerate(right.values):
    sum_pr   = sum_pr + row[11]
    sum_prob = sum_prob + (row[10] * (1 - row[10]))

ov_right = sum_pr / sum_prob

# fill table
ov_array = []

for i, row in enumerate(df.values):
```

```
    if row[3] < 0.273:
        ov = ov_left
    else:
        ov = ov_right
    ov_array.append(ov)

print(ov_left)
print(ov_right)

df['output_value_2'] = ov_array
```

```
0.9840961179667522
0.5416655512088253
1.8167965745108998
-0.7815561900329465
```

[15]:
```
# new predictions
f2x_array = []

for i, row in enumerate(df.values):
    f2x = row[9] + (learning_rate * row[12])
    f2x_array.append(f2x)

df['f2x'] = f2x_array
```

Final Calculation Predictions for samples

[16]:
```
# calculate probabilites
prob_array = []

for i, row in enumerate(df.values):
    prob = math.pow(e, row[13]) / (1 + math.pow(e, row[13]))
    prob_array.append(prob)

prob_array

df['Final Classification Probability'] = prob_array

df

# probability that song belongs to category hiphop
```

[16]:

	category	track	feature_danceability	feature_acousticness	label	f0x	\
0	hiphop	h1	0.949	0.132	1	0.510826	
1	hiphop	h2	0.743	0.234	1	0.510826	
2	hiphop	h3	0.913	0.394	1	0.510826	
3	hiphop	h4	0.810	0.504	1	0.510826	

```
4   hiphop    h5                0.434                0.198      1  0.510826
5     jazz    j1                0.654                0.534      0  0.510826
6     jazz    j2                0.593                0.312      0  0.510826
7     jazz    j3                0.234                0.341      0  0.510826

   probability_f0x  pseudo_residuals_1  output_value_1       f1x  \
0         0.624988            0.375012        1.600032  1.470845
1         0.624988            0.375012        1.600032  1.470845
2         0.624988            0.375012        1.600032  1.470845
3         0.624988            0.375012        1.600032  1.470845
4         0.624988            0.375012       -1.599926 -0.449130
5         0.624988           -0.624988       -1.599926 -0.449130
6         0.624988           -0.624988       -1.599926 -0.449130
7         0.624988           -0.624988       -1.599926 -0.449130

   probability_f1x  pseudo_residuals_2  output_value_2       f2x  \
0         0.813163            0.186837        1.816797  2.560923
1         0.813163            0.186837        1.816797  2.560923
2         0.813163            0.186837       -0.781556  1.001911
3         0.813163            0.186837       -0.781556  1.001911
4         0.389579            0.610421        1.816797  0.640948
5         0.389579           -0.389579       -0.781556 -0.918064
6         0.389579           -0.389579       -0.781556 -0.918064
7         0.389579           -0.389579       -0.781556 -0.918064

   Final Classification Probability
0                          0.928286
1                          0.928286
2                          0.731414
3                          0.731414
4                          0.654953
5                          0.285372
6                          0.285372
7                          0.285372
```

Bibliography

[3] A. Meier, 'Rundgang big data analytics – hard & soft data mining', in *Big Data Analytics*, Springer Fachmedien Wiesbaden, 2021, pp. 3–23. DOI: 10.1007/978-3-658-32236-6_1.

[4] M. Tanwar, R. Duggal, and S. K. Khatri, 'Unravelling unstructured data: A wealth of information in big data', in *2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions)*, IEEE, 2015-09. DOI: 10.1109/icrito.2015.7359270.

[5] 'Quick guide: Machine learning im maschinen und anlagenbau', VDMA Software und Digitalisierung, 2018.

[6] G. Paaß and D. Hecker, *Künstliche Intelligenz*. Springer Fachmedien Wiesbaden, 2020. DOI: 10.1007/978-3-658-30211-5.

[7] S. Schacht and C. Lanquillon, *Blockchain und maschinelles Lernen: Wie das maschinelle Lernen und die Distributed-Ledger-Technologie voneinander profitieren*. Springer Berlin Heidelberg, 2019, ISBN: 9783662604083. [Online]. Available: https://books.google.de/books?id=nL%5C_ADwAAQBAJ.

[9] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*, 2nd ed. Springer US, 2021. DOI: 10.1007/978-1-0716-1418-1.

[10] R. J. Lewis, 'An introduction to classification and regression tree (cart) analysis', in *Annual meeting of the society for academic emergency medicine in San Francisco, California*, Citeseer, vol. 14, 2000.

[11] S. Tangirala, 'Evaluating the impact of gini index and information gain on classification using decision tree classifier algorithm', *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 2, pp. 612–619, 2020.

[12] A. Natekin and A. Knoll, 'Gradient boosting machines, a tutorial', *Frontiers in Neurorobotics*, vol. 7, 2013. DOI: 10.3389/fnbot.2013.00021.

[13] T. Parr and J. Howard, *How to explain gradient boosting*, Gradient Boosting: Distance to target, https://explained.ai/gradient-boosting/L2-loss.html, Accessed: 2022-01-29, 2022.

[14] P. Bühlmann, 'Bagging, boosting and ensemble methods', Berlin, Papers 2004,31, 2004. [Online]. Available: http://hdl.handle.net/10419/22204.

[15] T. Parr and J. Howard, *How to explain gradient boosting*, Gradient boosting: Heading in the right direction, https://explained.ai/gradient-boosting/L1-loss.html, Accessed: 2022-01-29, 2022.

[16] J. H. Friedman, 'Greedy function approximation: A gradient boosting machine', *The Annals of Statistics*, vol. 29, no. 5, 2001-10. DOI: 10.1214/aos/1013203451.

[17] B. Bischl, *Introduction to machine learning (i2ml)*, Chapter 10.7: Bernoulli Loss, https://introduction-to-machine-learning.netlify.app, Accessed: 2022-01-29, 2022.

[18] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer New York, 2009. DOI: 10.1007/978-0-387-84858-7.

[19] T. Parr and J. Howard, *How to explain gradient boosting*, Gradient boosting performs gradient descent, https://explained.ai/gradient-boosting/descent.html, Accessed: 2022-01-29, 2022.

[20] A. Subasi, *Practical machine learning for data analysis using python*. London: Academic Press, 2020, ISBN: 9780128213797.

[22] S. Raschka, 'Model evaluation, model selection, and algorithm selection in machine learning', 2018-11-13. arXiv: 1811.12808 [cs.LG].

[23] J. M. G. Christoph Schroer Felix Kruse, *A Systematic Literature Review on Applying CRISP-DM Process Model*. 2021.

[24] C. P. López, *DATA MINING. The CRISP-DM METHODOLOGY. The CLEM Language and IBM SPSS MODELER*. 2021-03-28.

[25] R. N. Gary D Miner Ken Yale, *Handbook of Statistical Analysis and Data Mining Applications*. 2017-11-09.

[26] M. Reddy, *API Design for C++*. Elsevier Science, 2011, ISBN: 9780123850041. [Online]. Available: https://books.google.de/books?id=IY29LylT85wC.

[29] K. Lane, 'Intro to apis: What is an api?', 2019-10-05. [Online]. Available: https://blog.postman.com/intro-to-apis-what-is-an-api/ (visited on 2022-01-13).

[30] D. Gourley, B. Totty, M. Sayer, A. Aggarwal, and S. Reddy, *HTTP: The Definitive Guide* (Definitive Guides). O'Reilly Media, Incorporated, 2002, ISBN: 9781565925090.

[37] J. Hemming, *Methoden der Erforschung populärer Musik*. Gabler, Betriebswirt.-Vlg, 2015-10-28, 514 pp., ISBN: 3658114967. [Online]. Available: https://www.ebook.de/de/product/25204785/jan_hemming_methoden_der_erforschung_populaerer_musik.html.

[63] J. Davis and M. Goadrich, 'The relationship between precision-recall and ROC curves', in *Proceedings of the 23rd international conference on Machine learning - ICML '06*, ACM Press, 2006. DOI: 10.1145/1143844.1143874.

[64] T. Fawcett, 'An introduction to roc analysis', *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

Internet sources

[1] L.Rabe. ‘Spotify: Daten und fakten zum schwedischen musikstreaming-anbieter’. (2021-10-14), [Online]. Available: https://de.statista.com/themen/1894/spotify/#dossierKeyfigures (visited on 2022-01-31).

[2] M. Rangaiah. ‘How spotify is using big data to enhance customer experience’, Analytics Steps Infomedia LLP. (2021-01-07), [Online]. Available: https://www.analyticssteps.com/blogs/how-spotify-using-big-data (visited on 2022-01-31).

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. ‘Scikit-learn: Machine learning in Python’, Feature importances with a forest of trees. Accessed: 2022-01-29. (2011).

[21] ‘Scikit learn user guide, 6.3. preprocessing data’, scikit-learn developers. (), [Online]. Available: https://scikit-learn.org/stable/modules/preprocessing.html#standardization-or-mean-removal-and-variance-scaling (visited on 2022-01-31).

[27] ‘Introduction to web apis’, Mozilla. (), [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction (visited on 2022-01-31).

[28] ‘Types of apis & popular rest api protocol’, Stoplight. (), [Online]. Available: https://stoplight.io/api-types/ (visited on 2022-01-24).

[31] B. Cooksey. ‘An introduction to apis’. B. Landers and D. Schreiber, Eds., Zapier, Inc. (2014-04-23), [Online]. Available: https://zapier.com/learn/apis/chapter-1-introduction-to-apis/ (visited on 2022-01-31).

[32] ‘Working with json’, Mozilla. (), [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON (visited on 2022-01-31).

[33] ‘Json definiert’, Oracle. (), [Online]. Available: https://www.oracle.com/de/database/what-is-json/ (visited on 2022-01-31).

[34] ‘Introducing json’. (), [Online]. Available: https://www.json.org/json-en.html (visited on 2022-01-13).

[35] R. Becker. ‘What is music?’, Study.com. (2021-10-29), [Online]. Available: https://study.com/learn/lesson/what-is-music-characteristics-of-music-how-music-is-made.html (visited on 2022-01-31).

[36] R. Havers. ‘John coltrane quotes: The iconic saxophonist in his own words’, UDiscoverMusic. (2021-09-23), [Online]. Available: https://www.udiscovermusic.com/stories/john-coltrane-in-20-quotes/ (visited on 2022-01-31).

[38] ‘Chapter 2: Music: Fundamentals and educational roots in the u.s.’ (), [Online]. Available: https://milnepublishing.geneseo.edu/music-and-the-child/chapter/chapter-2/ (visited on 2022-01-31).

[39] ‘Music theory: 5 fundamentals that you should know’, Shaw Academy. (2019-04), [Online]. Available: https://www.shawacademy.com/blog/music-theory-5-fundamentals-that-you-should-know/ (visited on 2022-01-31).

[40] M. Shipman. ‘Analysis of 50 years of hit songs yields tips for advertisers’, NC State University. (2014-03-18), [Online]. Available: https://news.ncsu.edu/2014/03/wms-henard-hits2014/ (visited on 2022-01-31).

[41] ‘What are we listening to? | the most popular themes in songs’, Atlanta Institute of Music and Media. (2020-02-14), [Online]. Available: https://www.aimm.edu/blog/most-popular-themes-in-songs (visited on 2022-01-31).

[42] ‘Instrument classification’, Goshen College. (), [Online]. Available: https://www.goshen.edu/academics/music/mary-k-oyer-african-music-archive/instrument-classification/ (visited on 2022-01-31).

[43] ‘Musikrichtungen’, MUSICFLX. (), [Online]. Available: https://www.musicflx.de/musikrichtungen/ (visited on 2022-01-31).

[44] ‘What is hip-hop?’, Musical Dictionary. (), [Online]. Available: https://musicaldictionary.com/what-is-hip-hop/ (visited on 2022-01-31).

[45] G. Tate. ‘Hip-hop music and cultural movement’, Encyclopædia Britannica, Inc. (), [Online]. Available: https://www.britannica.com/art/hip-hop (visited on 2022-01-31).

[46] R. PQ. ‘Hip hop history: From the streets to the mainstream’, Icon Collective. (2019-11-13), [Online]. Available: https://iconcollective.edu/hip-hop-history/ (visited on 2022-01-31).

[47] K. Goodrich. ‘What is hip-hop?’ (2017-04-25), [Online]. Available: https://medium.com/@katiegoodrich/what-is-hip-hop-9f9abfffb25e (visited on 2022-01-31).

[48] M. Beek. ‘What is jazz music?’, BBC Music Magazine. (2021-07-15), [Online]. Available: https://www.classical-music.com/features/articles/jazz-music-what-it-is-and-how-it-evolved/ (visited on 2022-01-31).

[49] D. J. Wildridge. ‘Characteristics of jazz: An introduction’, CMUSE. (2020-01-15), [Online]. Available: BBC%20Music%20Magazine (visited on 2022-01-31).

[50] ‘What is jazz?’, National Museum of American History. (), [Online]. Available: https://americanhistory.si.edu/smithsonian-jazz/education/what-jazz (visited on 2022-01-31).

[51] ‘What is jazz? a guide to the history and sound of jazz’, MasterClass. (2020-11-08), [Online]. Available: https://www.masterclass.com/articles/what-is-jazz#what-is-jazz-music (visited on 2022-01-31).

[52] ‘What is jazz?’, Musical Dictionary. (), [Online]. Available: https://musicaldictionary.com/what-is-jazz/ (visited on 2022-01-31).

[53] ‘Rock ’n’ roll music guide: 4 characteristics of rock ’n’ roll’, MasterClass. (2021-10-25), [Online]. Available: https://www.masterclass.com/articles/rock-n-roll-music-guide (visited on 2022-01-31).

[54] ‘What is rock music?’, Musical Dictionary. (), [Online]. Available: https://musicaldictionary.com/what-is-rock-music/ (visited on 2022-01-31).

[55] B. Clark. ‘30 different types of rock music (rock subgenres)’, MusicianWave.com. (2021-08-19), [Online]. Available: https://www.musicianwave.com/types-of-rock-music-subgenres/ (visited on 2022-01-31).

[56] ‘Spotify authorization documentation’, Spotify AB. (), [Online]. Available: https://developer.spotify.com/ (visited on 2022-01-31).

[57] ‘Spotify developer console’, Spotify AB. (), [Online]. Available: https://developer.spotify.com/console/ (visited on 2022-01-31).

[58] ‘Spotify web api reference’, Spotify AB. (), [Online]. Available: https://developer.spotify.com/documentation/web-api/reference/#/ (visited on 2022-01-31).

[59] B. Stephenson. ‘What is spotify and how does it work?’ (2021-04-28), [Online]. Available: https://www.lifewire.com/what-is-spotify-4685829 (visited on 2022-01-31).

[60] C. Boyd. ‘How spotify recommends your new favorite artist’. (2019-11-11), [Online]. Available: https://towardsdatascience.com/how-spotify-recommends-your-new-favorite-artist-8c1850512af0 (visited on 2022-01-31).

[61] G. T. A. Features, Ed. (), [Online]. Available: https://developer.spotify.com/documentation/web-api/reference/#/operations/get-audio-features (visited on 2022-01-31).

[62] M. Galarnyk. ‘Understanding boxplots’, Towards Data Science. (2018-09-12), [Online]. Available: https://towardsdatascience.com/understanding-boxplots-5e2df7bcbd51 (visited on 2022-01-31).

[65] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. ‘Scikit-learn: Machine learning in Python’, Feature importances with a forest of trees. Accessed: 2022-01-29. (2011).

[66] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. ‘Scikit-learn: Machine learning in Python’, Feature importances with a forest of trees. Accessed: 2022-01-29. (2011).

Declaration in lieu of oath

I hereby declare that I produced the submitted paper with no assistance from any other party and without the use of any unauthorized aids and, in particular, that I have marked as quotations all passages which are reproduced verbatim or near-verbatim from publications. Also, I declare that the submitted print version of this thesis is identical with its digital version. Further, I declare that this thesis has never been submitted before to any examination board in either its present form or in any other similar version. I herewith agree that this thesis may be published. I herewith consent that this thesis may be uploaded to the server of external contractors for the purpose of submitting it to the contractors' plagiarism detection systems. Uploading this thesis for the purpose of submitting it to plagiarism detection systems is not a form of publication.

Düsseldorf, 31.1.2022

(Location, Date)

Thomas Keiser
Martin Krüger
Luis Pflamminger
Jesper Wesemann

(handwritten signature)