



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

Jesper Wesemann

Luis Pflamminger

Advisor: Prof. Dr. Adem Alparslan

Matriculation Number: 123456 (Krüger), 123456 (Keiser), 123456 (Wesemann), 123456 (Pflamminger)

Submission: January 31st, 2022

Contents

List of Figures	IV
List of Tables	V
List of Abbreviations	VI
List of Symbols	VII
1 Einleitung	1
1.1 Problem Definition	1
1.2 Goal	1
1.3 Structure	1
2 Fundamentals	2
2.1 Basic Concepts of Big Data	2
2.1.1 Relevance of Data	2
2.1.2 The 5V Matrix for Big Data	3
2.1.3 Reinforcement Learning	3
2.1.4 Machine Learning Algorithms	3
2.2 Gradient Boosting	3
2.3 Cross Industry Standard Process for Data Mining	3
2.4 Application Programming Interfaces	3
2.4.1 Purpose and Usage	3
2.5 Basic Concepts of Music Theory	3
3 Implementation	4
3.1 Data Collection	4
3.1.1 Requirements for the dataset	4
3.1.2 Existing Datasets	5
3.1.3 Ressources and Approach	5
3.1.4 Authorization	6
3.1.5 Getting Features	8
3.1.6 Getting Track IDs and Labels	10
3.1.7 Python Implementation	12
3.2 Data Understanding	25
3.3 Data Preparation	25
3.4 Modeling	25

3.5 Evaluation	25
4 Fazit	25
Appendix	26
Bibliography	27

List of Figures

Figure 1: Spotify Authorization Flow	7
Figure 2: Access Token Request	8
Figure 3: Audio Feature Request	9
Figure 4: Artist Request	11
Figure 5: Categories and Playlists in Spotify App	12
Figure 6: Categories Request	15
Figure 7: Get Category's Playlists Request and Response	20
Figure 8: Get Playlist's Tracks	22

List of Tables

List of Abbreviations

API	Application Programming Interface
REST	Representational State Transfer

List of Symbols

1 Einleitung

Dies soll eine \LaTeX -Vorlage für den persönlichen Gebrauch werden. Sie hat weder einen Anspruch auf Richtigkeit, noch auf Vollständigkeit. Die Quellen liegen auf Github zur allgemeinen Verwendung. Verbesserungen sind jederzeit willkommen.

1.1 Problem Definition

1.2 Goal

1.3 Structure

2 Fundamentals

2.1 Basic Concepts of Big Data

Big Data is an umbrella term used to describe various technological but also organizational developments. Originally, Big Data refers to large sets of structured and unstructured data which must be stored and processed to gain business value. Today, Big Data is also often used as buzzword to outline various modern use cases that deal with large amounts of data. Big Data is therefore often used in conjunction with other buzzwords like automatization, personalization or monitoring. This chapter presents the foundation of Big Data in its technical implementation and combines the topics with business cases.

2.1.1 Relevance of Data

Data in combination with Business Intelligence become increasingly important over the past decades and is closely associated with the advances of the internet itself. Looking back, Business Intelligence can be divided into three sub-categories, which follow another linearly. The first phase is centered around getting critical insights into operations from structured data gathered while running the business and interacting with customers. Examples would be transactions and sales. The second phase focuses increasingly on data mining and gathering customer-specific data. These insights can be used to identify customer needs, opinions and interests. The third phase, often referred as Big Data, enhances the focus set in phase two by more features and much deeper analysis possibilities. It allows to gain critical information such as location, person, context often through mobile and sensor-based context.

In conclusion, organizations require Business Intelligence as it allows them to gain crucial insights which is needed to run the business and achieve an advantage over the competition. It is important to minimize the uncertainty of decisions and maximize the knowledge about the opportunity costs and derive their intended impacts. It is clearly noticeable that the insights and analysis possibilities become progressively deeper and much more detailed. Along this trend the amount of data required becomes larger and larger with increasingly complex data structures. Size, complexity of data and deep analysis form the foundation of Big Data and can be found again in the 5V matrix of Big Data.

2.1.2 The 5V Matrix for Big Data

When describing Data, a reference is often made to the five Vs, which highlight its main characteristics. The previous aspects of Big Data can again be recognized in averted form.

Volume: The size of the datasets is in the range of tera- and zettabyte . This massive volume is not a challenge for storing but also extracting relevant information from the mass of data.

2.1.3 Reinforcement Learning

2.1.4 Machine Learning Algorithms

2.2 Gradient Boosting

2.3 Cross Industry Standard Process for Data Mining

2.4 Application Programming Interfaces

This section gives an overview over the basic concepts[1, S.1] and technologies behind Application Programming Interfaces (APIs).

2.4.1 Purpose and Usage

An APIs is an interface between two pieces of software.[1, S.1] These might run on the same machine and communicate locally, in the case of a desktop application for example, or on separate machines that are connected via some network, e.g. in a client/server application.

APIs provide a readily implemented solution to a problem in programming and can be reused , how the API can be used to solve it. APIs such a problem might be finding some value in an array, fetching a file from a hard drive, or getting the latest weather data from a weather service. \@expl@@@filehook@file@pop@assign@@nnnn sections/02_Fundamentals05_api.texsections/02_Fundamentals05_api.tex

2.5 Basic Concepts of Music Theory

3 Implementation

3.1 Data Collection

In this section the approach and implementation of data collection for this project is examined.

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 were enough.

- **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 Existing Datasets

As this paper examines creating a model specifically on Spotify Song Data, a search on the internet was conducted first, to find a potential pre-made dataset, pulled from the Spotify API, which could be used. Kaggle ¹ lists an extensive catalogue of community provided datasets, so the main sources of this search were Kaggle and Google search for the term "Spotify Song Data". Kaggle lists a couple of datasets that could be applicable to the research question in this paper. Some examples of datasets listed are given and explained, why they could not be used for this project.

"Spotify music analysis" by user Aeryan ² is a dataset of 2017 rows, which includes musical features like acousticness and tempo, the song title and artist, but lacks a genre field. Because of the small sample size and the missing genre field, this dataset could not be used.

There are multiple datasets which include songs that were featured in Spotify's "Top 50" Playlists, charts, or year in review, recorded at a single point in time or historically. ^{3 4} These could not be used, as the sample size is again too small and the focus is specifically on the most popular tracks and not a wide variety of music in a genre.

"Dataset of songs in Spotify" ⁵ is the most promising dataset examined, as it has a big sample size and includes genre data. However, the methodology of how the data was collected is not included and there could be multiple ways of how genre data for a given song is collected, as is explained later. Also the genres are limited to very specific directions of Electronic Dance Music and Hip-Hop.

As no optimal dataset for this research paper could be found using our search criteria, a dataset was specifically created for this paper using the Spotify Web API.

3.1.3 Ressources and Approach

Spotify provides extensive documentation for developers on their developer website ⁶. This includes development and design guidelines for teams, that want to integrate Spotify's service into their own apps, documentation on IOS and Android development a community forum, a developer dashboard and the Web API documentation, which is the main ressource for data collection from Spotify.

¹ Kaggle Website: <https://www.kaggle.com/>

² <https://www.kaggle.com/aeryan/spotify-music-analysis>

³ <https://www.kaggle.com/nadintamer/top-spotify-tracks-of-2018>

⁴ <https://www.kaggle.com/leonardopena/top50spotify2019>

⁵ <https://www.kaggle.com/mrmorj/dataset-of-songs-in-spotify>

⁶ <https://developer.spotify.com/>

The API is based on the Representational State Transfer (REST) architecture. The different endpoints return JSON metadata directly from the Spotify Data Catalogue [2]. There are also features to query for user related data using an authorization flow with the users Spotify account, but this is not relevant in this context. [2] Requests to the API are made via HTTPS GET or POST methods. The API can be used by anyone, but authorization via the OAuth protocol is required to access data from the API. 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. 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 made.

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. Beyond the Web Console, the tool "Postman" was used to explore the API. It is a platform that can be used to make HTTP requests to an API and store these requests in a collaborative environment. [3] It supports the required authorization workflows and enabled the research team to explore endpoints together, easily make API calls without having to authenticate by hand, and save the endpoints and required input parameters in a shared workspace. Once exploration was complete, the complete data collection 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.

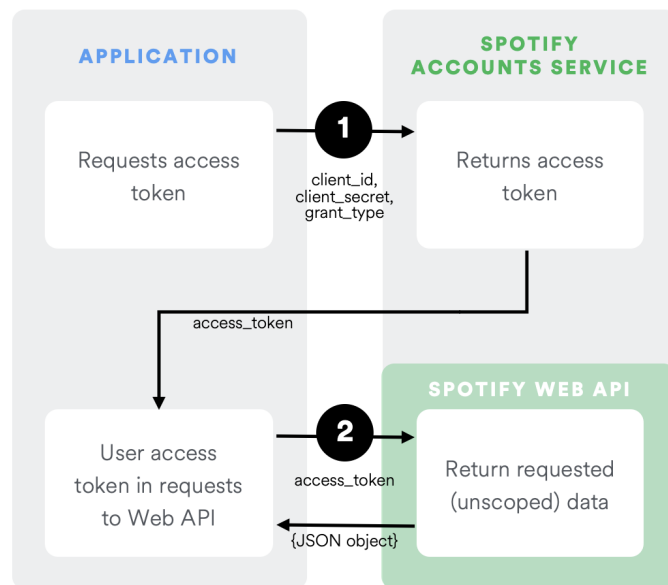
Many API endpoints give the option to specify a country and locale that the results should apply to. Whenever possible these values were specified to eliminate the risk of receiving data for multiple different countries or languages, which would have made the dataset incoherent. For country, we used the country tag "US" which means that the API replies only with datapoints that are applicable to users in the United States of America. For locale we used `en_US` in order to receive the data in the english language. These two values differ in that country changes the actual entries from the database that are selected for the response and sent to the users. Locale defines in which language these selected entries are returned in.

3.1.4 Authorization

Using the Spotify documentation for authorization workflows [4], authorization was first tested using Postman and then implemented in Python. The workflow is explained using

the Python code. Using a Spotify account to log in, the developer dashboard can be accessed. Here an application was registered with Spotify for the research project. Spotify tracks API usage per application and can recognize if the API is being abused or too many requests are sent, which will result in rate limitation or blocking. On the API Dashboard, a "Client ID" and "Client Secret" can be retrieved. These credentials are used to start the authorization flow, as described by Spotify in Figure 1.

Figure 1: Spotify Authorization Flow



Source: [4]

Step one is a post request to the Spotify Account Service with the client id and client secret from the application dashboard, which returns an access token, that is valid for one hour. This token can be used to access any endpoint of the actual API that does not require user specific data. When the token expires, a new one has to be requested before querying the API again. Figure 2 shows the full request and response to acquire the token.

Figure 2: Access Token Request

POST	https://accounts.spotify.com/api/token <i>request access token</i>
Body	application/x-www-form-urlencoded
<pre> 1 { 2 "grant_type": "client_credentials", 3 "client_id": client id from application dashboard, 4 "client_secret": client secret from dashboard 5 } </pre>	
Response	application/json
200 ok	<pre> 1 { 2 "access_token": "BQDgQCSx-tIMDo9LfVeZxm6Ym12p_WbEU3Q 3 9ENsVl7e--6d_vockTsfzMVUhPWihSSnFUuHvm_9POA1kYEw" 4 , 5 "token_type": "Bearer", 6 "expires_in": 3600 7 } </pre>

3.1.5 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 a single track or up to 99 tracks at a time. The latter is used in the Python implementation as it reduces the number of requests to be made. The typical request/response pattern for the audio-feature request endpoint is shown in figure 3.

Figure 3: Audio Feature Request

GET	https://api.spotify.com/v1/audio-features?ids={ids} <i>request audio features for id</i>
Parameter	
ids	comma seperated list of up to 99 song ids
Response	
application/json	
200	ok
<pre> 1 { 2 "audio_features": [3 { 4 "danceability": 0.677, 5 "energy": 0.638, 6 "key": 8, 7 "loudness": -8.631, 8 "mode": 1, 9 "speechiness": 0.333, 10 "acousticness": 0.589, 11 "instrumentalness": 0, 12 "liveness": 0.193, 13 "valence": 0.435, 14 "tempo": 82.810, 15 "type": "audio_features", 16 "id": "2e3Ea0o24lReQFR4FA7yXH", 17 "uri": "spotify:track:2e3Ea0o24lReQFR4FA7yXH", 18 "track_href": "https://api.spotify.com/v1/tracks/2e3Ea0o24lReQFR4FA7yXH", 19 "analysis_url": "https://api.spotify.com/v1/audio-analysis/2e3Ea0o24lReQFR4FA7yXH", 20 "duration_ms": 211497, 21 "time_signature": 4 22 }, 23 ... 24] 25 }</pre>	

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 we need to get 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.

3.1.6 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 using the reference, web console and Postman only 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 can be seen in figure 4.

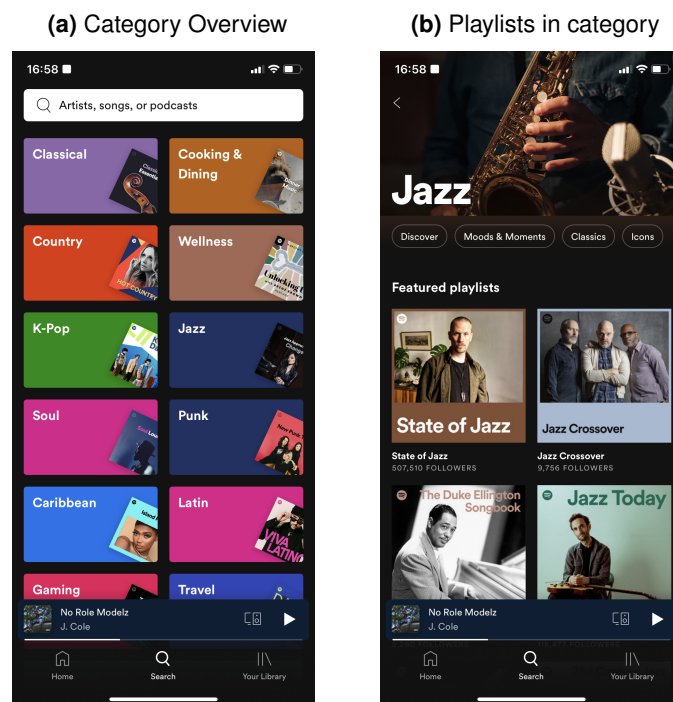
Figure 4: Artist Request

GET	https://api.spotify.com/v1/artists/{id} <i>request information about an artist by their id</i>
Parameter	
id	id of the artist
Response	
application/json	
200	ok
<pre> 1 { 2 "external_urls": { 3 "spotify": "https://open.spotify.com/artist/6l3HvQ5sa6mXTsMTB19rO5" 4 }, 5 "followers": { 6 "href": null, 7 "total": 15554811 8 }, 9 "genres": [10 "conscious hip hop", 11 "hip hop", 12 "north carolina hip hop", 13 "rap" 14], 15 "href": "https://api.spotify.com/v1/artists/6l3HvQ5sa6mXTsMTB19rO5", 16 "id": "6l3HvQ5sa6mXTsMTB19rO5", 17 "images": [18 ... 19], 20 "name": "J. Cole", 21 "popularity": 89, 22 "type": "artist", 23 "uri": "spotify:artist:6l3HvQ5sa6mXTsMTB19rO5" 24 }</pre>	

This exemplary 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 artists 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 way is Spotify's "categories" feature. The app's search tab provides a number of categories that a user can browse through to find new music in their preferred genre or style. In figure 5a an overview over some of the categories that are available in the Spotify App is shown. There are categories of multiple types, e.g. specific activities, like Cooking or Gaming, or places, like "At Home" or "In the Car". But there are also categories for nearly all major genres. In the app screenshot there is 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, like shown in figure 5b.

Figure 5: Categories and Playlists in Spotify App



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.

3.1.7 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. The script was implemented in

a way that lets anyone with a registered Spotify developer application 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 details of these functions are explained in this section using code snippets.

The Python requests library is used to execute the requests, as it supports the features needed to send the GET and POST requests we need without having to write complicated code. The dotenv library is used to read the client id and secret from a separate .env file, rather than writing it into the code. This prevents these sensitive credentials from being committed into version control, which is hosted in a public GitHub repository and would therefore make the credentials public. 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 of the API-Calls and "csv" to finally transform the json data into a CSV file.

First, the credentials are read using "load_dotenv". Then a method is defined which takes the client id and secret as input, executes a POST request to the accounts API, as shown in 2 and return the access token from the "access_token" field in the response.

```
1  #Get environment variables from ".env" file and read credentials
2  load_dotenv('.env')
3  client_id = os.environ.get('CLIENT_ID')
4  client_secret = os.environ.get('CLIENT_SECRET')
5
6  # Authenticate and get an API Token from Spotify using a Client ID and secret
7  def getAuthTokenFromCredentials(id, secret):
8
9      url = "https://accounts.spotify.com/api/token"
10
11     payload = f'grant_type=client_credentials&client_id={id}&client_secret={secret}'
12     headers = {
13         'Content-Type': 'application/x-www-form-urlencoded',
14     }
15
16     response = requests.request("POST", url, headers=headers, data=payload)
17
18     return response.json()["access_token"]
19
20 auth_token = getAuthTokenFromCredentials(client_id, client_secret)
```

Next the http library is configured to use a ten second timeout and attempt five retries. If a request to the API is made and no response is given from Spotify's servers after ten seconds, the request will be sent again. If there is still no answer from the server after five attempts, an Exception is thrown and the script ends. Timeouts are crucial as short

disconnections from the Internet could result in the whole data collection step failing. The http setup code is found in appendix ??.

The next part of the script implements the filter mechanism

```
1  category_filter = None
2
3  # comment this out if you don't want to set a filter
4  category_filter = ["hiphop", "pop", "country", "rock", "latin", "rnb", "mood", "
    indie_alt",
5                      "regional_mexican", "edm_dance", "inspirational", "chill", "
    party", "roots",
6                      "kpop", "instrumental", "ambient", "alternative", "classical",
    "jazz", "soul",
7                      "punk", "blues", "arab", "afro", "metal", "caribbean", "funk"]
8
9  # tells data collection functions to not use filter if it's not set
10 if category_filter is not None:
11     use_filter = True
12 else:
13     use_filter = False
```

The option to filter the categories that are to be taken into account when collecting the data is given. Possible values are all valid category ids, which can be requested using the API call shown in figure 6

Figure 6: Categories Request

GET	<code>https://api.spotify.com/v1/categories?country={country}&locale={locale}&limit={limit}&offset={offset}</code> <i>Get a list of categories used on Spotifys browse tab</i>
Parameter	
id	id of the artist
country	only display categories available in a specific country
locale	the language in which the categories should be returned
limit	the maximum number of items to be returned
offset	index of the first item to return
Response	application/json
200 ok	<pre> 1 { 2 "categories": { 3 "href": "...", 4 "items": [5 { 6 "href": "...", 7 "icons": [8 { 9 "height": 274, 10 "url": "...", 11 "width": 274 12 } 13], 14 "id": "hiphop", 15 "name": "Hip-Hop" 16 }, 17 { 18 "href": "...", 19 "icons": ... 20 "id": "pop", 21 "name": "Pop" 22 }, 23 ... 24], 25 "limit": 10, 26 "next": "...", 27 "offset": 0, 28 "previous": "...", 29 "total": 58 30 } 31 }</pre>

The response shows an id field for each item in the items array. These ids can be used in the filter list. If the category_filter variable is undefined, no filter is set.

Next, the option to import existing data from a file is given. If a user already has the json data containing all playlists for all categories they want to use, they can load the json file here and skip executing the collection of categories and playlists and continue right at the collection of tracks for each playlist.

```
1 data = {}
2
3 #To load data_object from file instead of rerunning the scripts, uncomment this:
4
5 file = open(os.path.join("data_collection", "json", "tracks_full.json"))
6 data = json.load(file)
```

Next, the four main steps of data collection will be defined and executed:

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

Each step is defined as a function with some input parameters and the resulting json data as output, which can then be used as input for the next step.

The function definition for getting the categories looks like this:

```
1 def getAllCategories(
2     requests_session, # takes the requests session, which was set up in the
3     beginning
4     auth_token, # this is the token which was retrieved in the authorization step
5     use_category_filter=False, # set to True if the category_filter should be used
6     category_filter=None, # this takes the list which is used as a category
7     filter
8     write_to_file=False, # set to True if the result of this step should be
9     written into a json file
10    path_to_file=''): # give the file path at which the json file should be stored
11
12    ...
```

The function first makes a simple API call with limit=1 to get the total amount of categories available. This is necessary, because the API returns a maximum number of 50 entries per request, so if the total number of items exceeds 50, multiple calls have to be made.

```

1  # Establishing the requests session
2  http = requests_session
3
4  # First API call used to get the total amount of categories
5  headers = { 'Authorization': f'Bearer {auth_token}' }
6  url = "https://api.spotify.com/v1/browse/categories?country=US&locale=en_US&limit=1"
7
8  try:
9      response = http.request("GET", url, headers=headers, data={})
10     if response.status_code != requests.codes.ok:
11         raise Exception
12 except Exception as e:
13     raise SystemExit(e)

```

The `requests_session` is saved in the `http` variable. Then the `auth_token` is wrapped in an Authorization header conforming to the OAuth Bearer Token standard, which is used by the API. The url is given with the right country and locale settings and the request is sent inside of a try block. If the response doesn't contain an HTTP status code 200, an Exception is raised and the program is stopped, as there is possibly an error in the request or response which could mean corrupted data.

Next, the total amount of categories is extracted from the response and the number of pages is calculated.

```

1  categoryAmount = response.json()["categories"]["total"]
2
3  # API only returns 50 items at a time. Offset can be used to gradually get all
   items
4  # Calculate number of pages with 50 items
5  pages = int(math.ceil(categoryAmount/50))
6  data = {"categories": []}

```

By dividing the category amount by 50 and rounding up to the next integer, the number of API calls necessary to get all entries is calculated. We call this pages, like pages in a book. Page one contains items 0 to 49 and is retrieved by using offset 0 and limit 50. Page two contains items 50 to 99 and is retrieved by using offset 50 and limit 50. This is done as many times as required. the last page will not necessarily contain 50 items but less, as those are left over.

A for loop is used to execute the request for every page as shown before. The offset is increased by 50 with each run. Each category's id and name are read from the response and appended to the data object:

```

1  for x in range(pages):
2      url = f"https://api.spotify.com/v1/browse/categories?country=US&locale=en_US&limit=50&offset={x * 50}"
3

```



```

4         try:
5             response = http.request("GET", url, headers=headers, data={})
6             if response.status_code != requests.codes.ok:
7                 raise Exception
8         except Exception as e:
9             raise SystemExit(e)
10
11         # categories are stored in the data dictionary
12         for el in response.json()["categories"]["items"]:
13             if (use_category_filter == True and el["id"] in category_filter) or
14                 use_category_filter == False:
15                 data["categories"].append({
16                     "id": el["id"],
17                     "name": el["name"]
18                 })

```

Outside of the for loop, the resulting data is written to a file if `write_to_file` was set to `True` and the data object is returned:

```

1     if write_to_file == True:
2         with open(path_to_file, 'w') as outfile:
3             json.dump(data, outfile, indent=2)
4
5     return data

```

The data collected after this step looks like this:

```

1  {
2      "categories": [
3          {
4              "id": "hiphop",
5              "name": "Hip-Hop"
6          },
7          {
8              "id": "pop",
9              "name": "Pop"
10         },
11         ...
12     ]
13 }

```

This concludes the function `getAllCategories`. The data retrieved from this step is used for the next step, getting the playlists. The function definition here is slightly different:

```

1     def getPlaylistsForCategories(
2         requests_session,
3         auth_token,
4         data_object,
5         write_to_file=False,
6         path_to_file=''):

```

Instead of the filtering options, which only apply to the categories, this function takes the `data_object` from the previous step. This could be passed in directly from the previous

function or read from a file. The function implements a nested for loop. The outer loop iterates over all categories in the data object. The inner loop uses an endpoint to get each category's playlists. The API request/response is shown in figure ??.

Figure 7: Get Category's Playlists Request and Response

GET	<code>https://api.spotify.com/v1/categories/{category_id}/playlists?country={country}&limit={limit}&offset={offset}</code> <i>Get Category's Playlists</i>
Parameter	
category_id	category id
country	only display categories available in a specific country
limit	the maximum number of items to be returned
offset	index of the first item to return
Response	application/json
200 ok	
	<pre> 1 { 2 "playlists": { 3 "href": "...", 4 "items": [5 { 6 "collaborative": false, 7 "description": "New music from YoungBoy 8 Never Broke Again, DaBaby and 2 9 Chainz. ", 10 "external_urls": ... 11 "href": "...", 12 "id": "37i9dQZF1DX0XUsuxWHRQd", 13 "images": ... 14 "name": "RapCaviar", 15 "owner": ... 16 "primary_color": null, 17 "public": null, 18 "snapshot_id": "...", 19 "tracks": { 20 "href": "https://api.spotify.com/v1/ 21 playlists/37i9dQZF1DX0XUsuxWHRQd/ 22 tracks", 23 "total": 50 24 }, 25 "type": "playlist", 26 "uri": "spotify:playlist:37i9dQZF1DX0 27 XUsuxWHRQd" 28 }, 29 ... 30], 31 "limit": 50, 32 "next": "...", 33 "offset": 0, 34 "previous": null, 35 "total": 50 36 } 37 }</pre>

As there is an upper limit of 50 playlists per request, the same paging method is used as in the category function. It is checked, if each items "type" field contains the value "playlist", to filter out any items that don't fit the schema. Then each playlists name and id is stored in the data object, which is returned from the function. The json data after this step looks like this:

```

1 {
2   "categories": [
3     {
4       "id": "hiphop",
5       "name": "Hip-Hop",
6       "playlists": [
7         {
8           "id": "37i9dQZF1DX0XUsuxWHRQd",
9           "name": "RapCaviar"
10        },
11        {
12          "id": "37i9dQZF1DX6GwdWRQMqpq",
13          "name": "Feelin' Myself"
14        },
15        ...
16      ],
17      ...
18    }
19  ]

```

The third step gets all tracks in each playlist. The function definition is the same as before, as this function also takes data from the previous step. This time, there are three nested for loops. One for looping through the categories, then another one for looping through the playlists. In this one there is again calculated how many pages of 50 tracks need to be requested and the third loop requests the pages and appends the tracks to the playlist data. The request parameters for this request differ from the other steps. Instead of country, there is a market parameter, which essentially does the same thing. There is also a fields parameter, which takes a string that can be used to define which fields the API should return. This is done to save bandwidth, as the response of this endpoint is very large and most of the datapoints might not be needed by the client. In this case, the value "items(track(name, id, album(name, id), artists)), total" is used to return only items of type track, their id and name, their albums name and id, and their artist information. Also the total amount of tracks in the playlist is returned. There is also an "additional_types" parameter, which can be used to specify if only music tracks, or also podcast episodes should be returned. In this case, only tracks are returned as podcasts are irrelevant to our analysis. An exemplary API request/response using the described "fields" and "additional_types" parameters is shown in figure 8. As a track can have multiple artists, each artist's name and id are extracted and saved with the track. As this function is very similar to the previous one, the code is not shown here. It can be found in the appendix.

Figure 8: Get Playlist's Tracks

GET	https://api.spotify.com/v1/playlists/{playlist_id}/tracks?market={market}&fields={fields}&additional_types={additional_types}&limit={limit}&offset={offset} Get Playlist's Tracks	
Parameter		
playlist_id	playlist id	
market	only display items available in this market	
fields	a string describing which fields to return	
additional_types	supported item types. valid types are "track" and "episode"	
limit	the maximum number of items to be returned	
offset	index of the first item to return	
Response		application/json
200 ok		
<pre>1 { 2 "items": [3 { 4 "track": { 5 "album": { 6 "id": "4oxmme6i4mypSt2DDzPTsW", 7 "name": "DS4EVER" 8 }, 9 "artists": [10 { 11 "external_urls": { 12 "spotify": "...", 13 }, 14 "href": "...", 15 "id": "2hlmm7s2ICUX0LVihVFlzQ", 16 "name": "Gunna", 17 "type": "artist", 18 "uri": "spotify:artist:2hlmm7s2ICUX0LVihVFlzQ" 19 }, 20 ... 21] 22 }, 23 }, 24 ... 25], 26 "total": 50 27 }</pre>		

The data collected after this step looks like this:

```

1 {
2   "categories": [
3     {
4       "id": "hiphop",
5       "name": "Hip-Hop",
6       "playlists": [
7         {
8           "id": "37i9dQZF1DX0XUsuxWHRQd",
9           "name": "RapCaviar",
10          "tracks": [
11            {
12              "id": "2AaJeBEq3WLcfFWly8svDf",
13              "name": "By Your Side",
14              "album": {
15                "id": "2RrZgDND03MLu6pRJdTz5",
16                "name": "By Your Side"
17              },
18              "artists": [
19                {
20                  "id": "45TgXXqMDdF8BkjA83OM7z",
21                  "name": "Rod Wave"
22                }
23              ]
24            },
25            ...
26          ]
27        },
28        ...
29      ],
30      ...
31    },
32    ...
33  ]
34 }

```

The fourth step is getting the audio features for each track. The request that is used was already shown in 3 The function used has the same definition as before but implements one more for loop to iterate through the tracks. Again, the code for this step is found in the appendix.

The data collected after this step looks like this:

```

1 {
2   "categories": [
3     {
4       "id": "hiphop",
5       "name": "Hip-Hop",
6       "playlists": [
7         {
8           "id": "37i9dQZF1DX0XUsuxWHRQd",
9           "name": "RapCaviar",
10          "tracks": [
11            {

```

```

12         "id": "2AaJeBEq3WLcfFWly8svDf",
13         "name": "By Your Side",
14         "album": {
15             "id": "2RrZgDND03MLu6pRJdTzk5",
16             "name": "By Your Side"
17         },
18         "artists": [
19             {
20                 "id": "45TgXXqMDdF8BkjA83OM7z",
21                 "name": "Rod Wave"
22             }
23         ],
24         "features": {
25             "danceability": 0.649,
26             "energy": 0.508,
27             "key": 8,
28             "loudness": -10.232,
29             "mode": 1,
30             "speechiness": 0.0959,
31             "acousticness": 0.0345,
32             "instrumentalness": 3.59e-05,
33             "liveness": 0.0736,
34             "valence": 0.405,
35             "tempo": 157.975,
36             "duration_ms": 194051,
37             "time_signature": 4
38         }
39     },
40     ...

```

This JSON contains all the necessary fields that are needed for continueing with the CRISP DM process. To prepare the dataset for data analysis, the JSON data needs to be transformed into a flat structure to be stored as a CSV file. Another Python function is used to flatten the data. It takes the resulting data from the last step and transforms it into the following format.

```

1  [
2      {
3          "categories.id": "hiphop",
4          "categories.name": "Hip-Hop",
5          "categories.playlists.id": "37i9dQZF1DX0XUsuxWHRQd",
6          "categories.playlists.name": "RapCaviar",
7          "categories.playlists.tracks.id": "2AaJeBEq3WLcfFWly8svDf",
8          "categories.playlists.tracks.name": "By Your Side",
9          "categories.playlists.tracks.album.id": "2RrZgDND03MLu6pRJdTzk5",
10         "categories.playlists.tracks.album.name": "By Your Side",
11         "categories.playlists.tracks.artists": "Rod Wave",
12         "categories.playlists.tracks.features.danceability": "0.649",
13         "categories.playlists.tracks.features.energy": "0.508",
14         "categories.playlists.tracks.features.key": "8",
15         "categories.playlists.tracks.features.loudness": "-10.232",
16         "categories.playlists.tracks.features.mode": "1",
17         "categories.playlists.tracks.features.speechiness": "0.0959",
18         "categories.playlists.tracks.features.acousticness": "0.0345",

```

```
19         "categories.playlists.tracks.features.instrumentalness": "3.59e-05",
20         "categories.playlists.tracks.features.liveness": "0.0736",
21         "categories.playlists.tracks.features.valence": "0.405",
22         "categories.playlists.tracks.features.tempo": "157.975",
23         "categories.playlists.tracks.features.duration_ms": "194051",
24         "categories.playlists.tracks.features.time_signature": "4"
25     },
26     ...
27 ]
```

Notice, that the nested fields and arrays have been flattened to represent a two dimensional data structure, containing only one array to represent rows which contains json objects with 22 fields. Each field represents a column. This flattened JSON structure is then converted to a CSV file and saved for further processing.

3.2 Data Understanding

3.3 Data Preparation

3.4 Modeling

3.5 Evaluation

4 Fazit

Appendix

Appendix 1: Beispielanhang

Dieser Abschnitt dient nur dazu zu demonstrieren, wie ein Anhang aufgebaut sein kann.







Appendix 1.1: Weitere Gliederungsebene

Auch eine zweite Gliederungsebene ist möglich.

Appendix 2: Bilder

Auch mit Bildern. Diese tauchen nicht im Abbildungsverzeichnis auf.

Figure 9: Beispielbild

Name	Änderungsdatum	Typ	Größe
 abbildungen	29.08.2013 01:25	Dateiordner	
 kapitel	29.08.2013 00:55	Dateiordner	
 literatur	31.08.2013 18:17	Dateiordner	
 skripte	01.09.2013 00:10	Dateiordner	
 compile.bat	31.08.2013 20:11	Windows-Batchda...	1 KB
 thesis_main.tex	01.09.2013 00:25	LaTeX Document	5 KB

Bibliography

- [1] M. Reddy, *API Design for C++*. Elsevier Science, 2011, ISBN: 9780123850041. [Online]. Available: <https://books.google.de/books?id=IY29LyIT85wC>.

Internet sources

- [2] 'Spotify web api documentation,' Spotify AB. (), [Online]. Available: <https://developer.spotify.com/documentation/web-api/>.
- [3] 'What is postman?' Postman, Inc. (), [Online]. Available: <https://www.postman.com/product/what-is-postman/> (visited on 2022-01-16).
- [4] 'Spotify authorization documentation,' Spotify AB. (), [Online]. Available: <https://developer.spotify.com/documentation/general/guides/authorization/>.

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/disagree** 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, 21.1.2022

(Location, Date)

A handwritten signature in black ink, consisting of a large, stylized 'H' followed by a series of loops and a final flourish.

(handwritten signature)