Lucerne University of Applied Sciences and Arts – Business

Master of Science in Applied Information and Data Science

Music Recommender for Deezer

Recommender Systems

22.10.2021

Lecturer:

Dr Guang Lu

Project Team:

Malik Sogukoglu
Marius Gisler
Remo Oehninger

Table of Content

List of Figures

List of Tables

1. Introduction

1.1. Context

Everyone is used to getting new recommendations for consuming the next content right after opening the apps for Amazon, YouTube, Spotify, Facebook, etc. Indeed, recommender systems are today unavoidable in our daily online journey from e-commerce to online advertisement and music streaming platforms.

For the mentioned platforms, offering tailored recommendations is very important to bring consumers and content producers closer together - in times when consumers are confronted with so much content that they are less able to show explicit interest or disinterest with ratings or comments.

This makes it a big challenge to offer them user-centric recommendations by considering implicit behavioural patterns such as the time spent on a video, the time of day a certain content is consumed, the location-based trends, etc. Furthermore, these recommendations need to be delivered within seconds by having analyzed the user's dataset, including its most recent activities.

This project describes the challenge of how to recommend the best song to an individual user from Deezer. Deezer is a French online music-streaming app and available on the web, where users can listen to over 73 million licensed songs, 30’000 radio channels and 100 million playlists. Deezer has 16 million active users and is available in more than 180 countries, through a free limited service and a premium offer («Deezer», 2021).

The music-streaming platform has its music recommender system called Flow, which is using collaborative filtering to provide a user with the song he wants to listen to at the time he wants. If the user does not want to listen to some specific songs, he can skip them, and the algorithm should detect them quickly. In this context, getting the first song recommendation right is important (*Deezer -Über uns*, 2021).

1.2. Goal

Deezer floated a challenge in 2017 on Kaggle (2021) with the goal to predict whether the users of the test dataset listened to the first song Flow proposed to them or not. To tackle this challenge, Deezer provided a train and a test dataset. The test dataset consists of a list of the first recommended songs on Flow for several users. Each row represents one user. The training dataset was generated using the listening history of these Deezer users for one month. In that dataset, each row represents one listened song. The list of distinct users in the training dataset matches exactly with the test dataset's one.

The authors from this report will describe how they tackle this challenge from Deezer, how their solution looks and what the difference between the collaborative filtering method from Deezer and the new solution from this report is. For that, the authors will compare different models to predict the best song for the individual users, by means of a comparison based on a receiver-operating characteristic (ROC) curve.

Moreover, regarding data preprocessing, the authors have addressed the challenge of dealing with large datasets and the need to process them in the shortest possible time, which arises for modern recommendation systems. Hereby, it will be presented how modern big data frameworks such as Spark and NVTabular, which are designed to deal with these problems, were used throughout this project work.

As an introduction to the entire topic, the next chapter describes the preliminary data exploration and analysis of the available data.

2. Data Analysis and Visualization

The first chapter of this project describes the data to get an overview of how they look. This is important for the data preprocessing part in the next chapter and the models for the recommender system, which are to be applied to the data.

2.1. Data Description

As mentioned before, Deezer provided a training and a test dataset. The training set consists of 15 columns, 7’558’834 entries and 19918 different user identities. Each row contains the following attributes:

- media_id - identifier of the song listened by the user
- album_id - identifier of the album of the song
- media_duration - duration of the song
- user_gender - gender of the user
- user_id - anonymized identifier of the user
- context_type - type of content where the song was listened: playlist, album ...
- release_date - release date of the song with the format YYYYMMDD
- ts_listen - timestamp of the listening in UNIX time
- platform_name - type of os
- platform_family - type of device
- user_age - age of the user
- listen_type - if the songs were listened in a flow or not
- artist_id - identifier of the artist of the song
- genre_id - identifier of the genre of the song
- is_listened - 1 if the song was listened, 0 otherwise

The test dataset looks very similar to the training dataset. The only difference is that the test dataset has a sample identity for the consecutive numbering and the attribute “is_listened” is not available.

2.2. Data Visualization

The user age is between 18 and 30 years and the distribution of the age is well distributed, as can be seen in the bar plot in figure 1. The box plot in figure 2 shows that the gender distribution is well balanced. For this reason, there is no need to make any special adjustments on the corresponding columns in the data preparation. Furthermore, there are no empty values in the training data set, so no additional enrichment or deletion of data is needed here either.

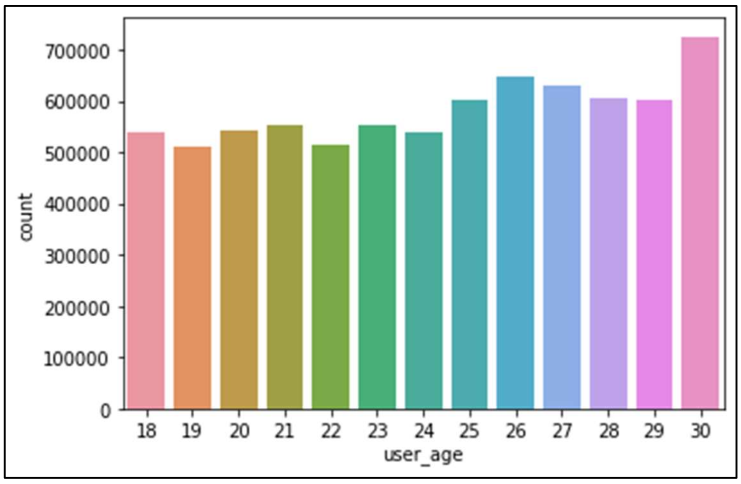


Figure 1. Distribution of the predictor “user_age”

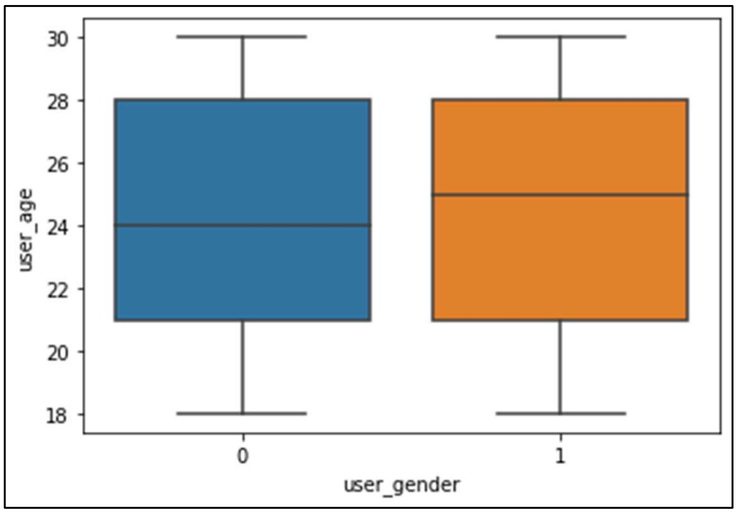


Figure 2. Box plot of the gender distribution comparing to the "user_age"

The column "is_listened" shows if the corresponding user has listened to the song for more than 30 seconds (is_listened = 1) or not (is_listened = 0) and is later also the target variable to be predicted by the models. The next plot shows the distribution of the target column values. The bar plot shows that more than twice as many entries are marked as listened. The training dataset looks very clean; therefore, the assumption is that the distribution from the target values is real and does not need further data preparation.

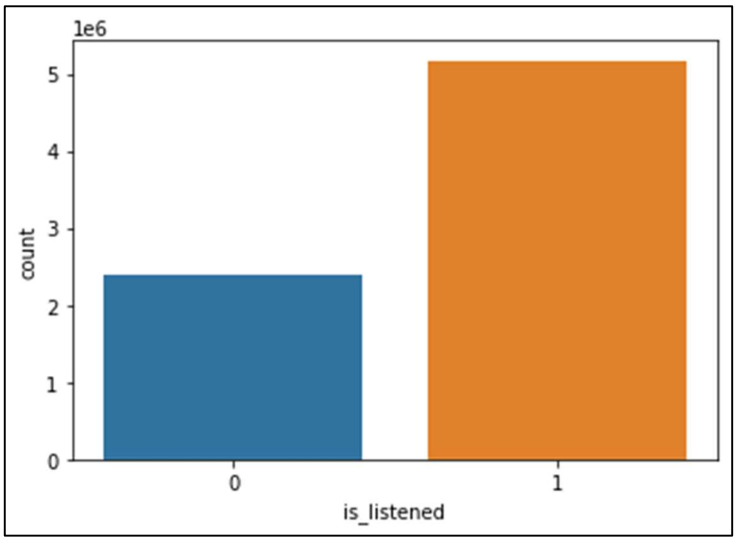


Figure 3: Bar plot of the target values. 0 means the song was not listened more the 30 seconds

The following correlation heatmap compares all attributes from the training dataset. The heatmap shows that there is no strong collinearity between the predictors. Therefore, there are no predictors to drop from the training dataset concerning to the collinearity.

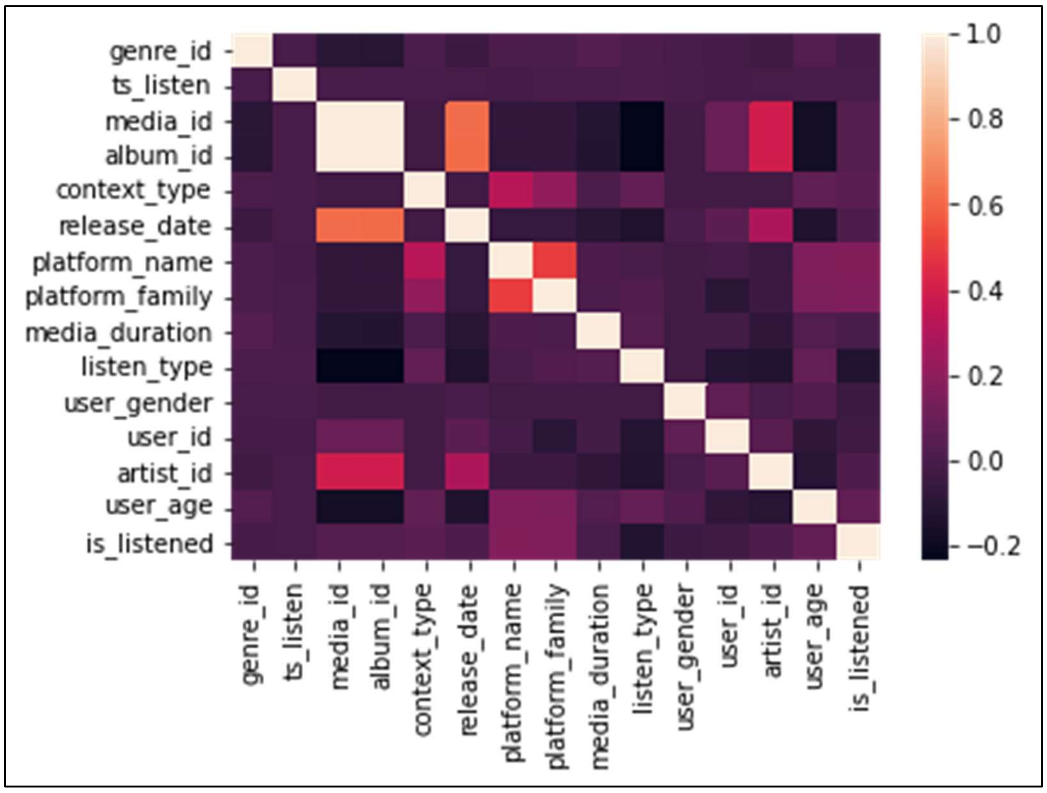


Figure 4. Correlation matrix from all attributes from the training set

For further data analysis the predictor variable “ts_listen” was examined. For the authors it seemed interesting whether the time of the day could have an influence on the target variable “is_listened”. Thus, the predictor was converted to hours number from 1 to 24.

Also, the predictor “release_date” was examined to check if there is an influence on the target variable. The release date was given in the format “YYYYMMDD” and was converted to the predictors “release_date_year” and “release_date_month”.

After creating the three new predictors, the target variable was plotted against the new predictors in figure 5. As can be seen, the histogram of the predictor “ts_listen_hour” is slightly right skewed. The most songs, which the user was starting to be listening is between 10am and 7pm. It is not clearly evident if there is an influence on the target variable.

The histogram of the year of the release is clearly right skewed and has maybe a small influence on the target variable. That could have an impact in the recommender models. Also, the histogram of the release month looks interesting, and it is right skewed as well. Most of the songs were released in December, which can be explained by the fact that Christmas is the time of year when albums and singles sell the most. The influence of the target variable is not clear but the fact of the distribution of the histogram could be interesting or the recommender models.

The authors assume that the new predictors are better suited for the recommender models, since the influence on the target variable will be stronger due to the covert of the previous variable. Therefore, this is also better described and applied in the next chapter “3. Data Preprocessing”.

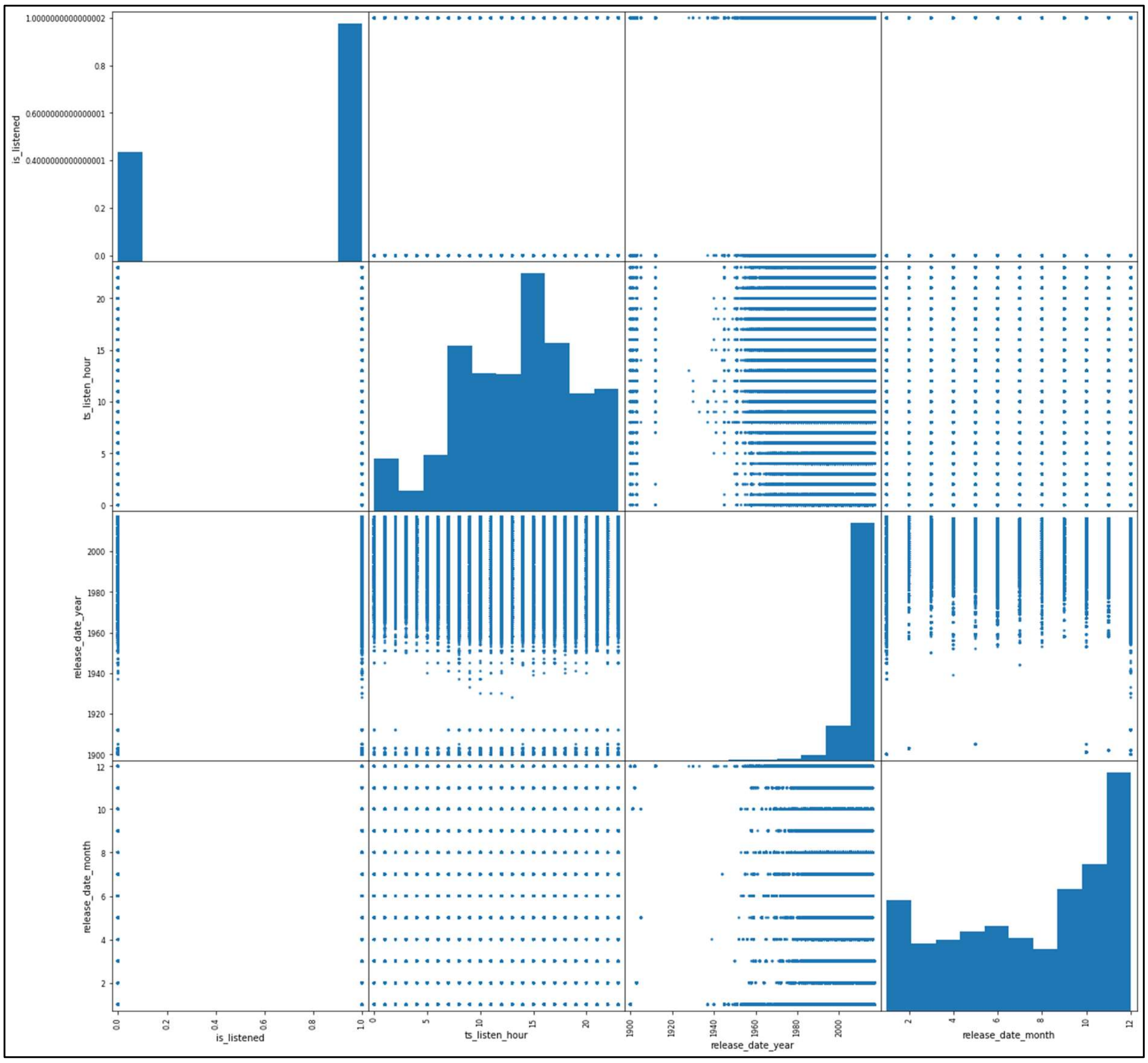


Figure 5. Scatter plot

3. Data Preprocessing

3.1. Converting Dataset to Parquet

The dataset provided to the authors is a CSV file of about 500 MB. Loading and processing CSV files can be very time consuming at this size. Since the authors set efficient data processing for the recommender system as a central point in this project, the authors intended to convert the data into a format that allows faster processing of the data by minimizing the I/O. Parquet, which stores the data in a column-based manner and allows for partitioning and compression of the data (Thomas, 2019), was an obvious choice. The authors of this work have saved the entire Deezer dataset in Parquet-format even before preprocessing.

3.2. Splitting the Dataset into Train, Validation and Test Dataset

As mentioned in the chapter "2. Data Analysis and Visualization", the available test dataset does not have the binary target values "is_listened". However, this makes it unusable for validating the models to be developed.

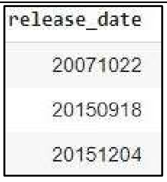
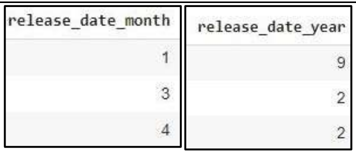
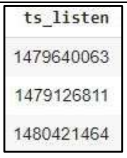
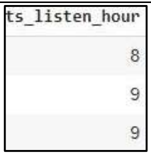
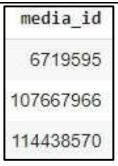
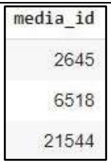
Therefore, this test dataset was ignored, and the available **training dataset** was taken and split as follows:

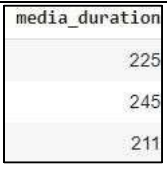
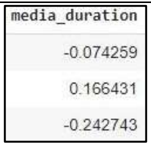
- The dataset was shuffled to avoid temporal bias.
- Training dataset got 70% of the data
- Validation dataset got 15% of the data
- Test dataset got 15% of the data

3.3. Data Preprocessing Steps

This section provides a brief overview of the way data was preprocessed for upcoming machine learning tasks. All columns in the dataset given by the lecturer were restructured during an initial preprocessing. Continuous values were scaled, and categorical values were replaced by smaller digits. This processing is intended to prevent columns with high range values from receiving more worth than columns with small range values, in the later execution of machine learning tasks. In other words, this step is taken to prevent bias. Specifically, the following adjustments were made to the dataset during preprocessing:

Table 1: Description of preprocessing steps in the training dataset

Column	Example Before	Example After
Release_date was split into the columns **release_month** and **release_year**, on the assumption that trends could emerge in the information on months and years. The resulting columns where then "categorized", i.e., the values for the years and months are no longer the same as before, but represent the category to which they are assigned with smaller digits.	 release_date: 20071022; 20150918; 20151204	 release_date_month: 1; 3; 4 — release_date_year: 9; 2; 2
Ts_listen which records the timestamps for the data entry of the song, was converted from UNIX time to readable format and reduced only to the information on hour (column: "ts_listen_hour"). In this way, the authors aim to be able to identify trends over the course of the day.	 ts_listen: 1479640063; 1479126811; 1480421464	 ts_listen_hour: 8; 9; 9
Media_id, user_id, genre_id album_id, artist_id: Ids (Identifier) can also have a high range if counting has not started from 0 or if some ids have been deleted and left a gap in the course	 media_id: 6719595; 107667966; 114438570	 media_id: 2645; 6518; 21544

of time. By "categorizing" these ids, they are also brought into a smaller range with smaller digits.		
The columns **media_duration** and **user_age** have not been "categorized" but "normalized" by scaling and mean-centring the data. Here, too, the aim was to reduce the range so that the corresponding columns are is not overweighted.	 media_duration 225 245 211	 media_duration -0.074259 0.166431 -0.242743

Finally, all **null values were deleted** from the dataset to avoid error messages during the subsequent execution of machine learning tasks.

The graphic in figure 6 shows the entire preprocessing workflow mentioned above.

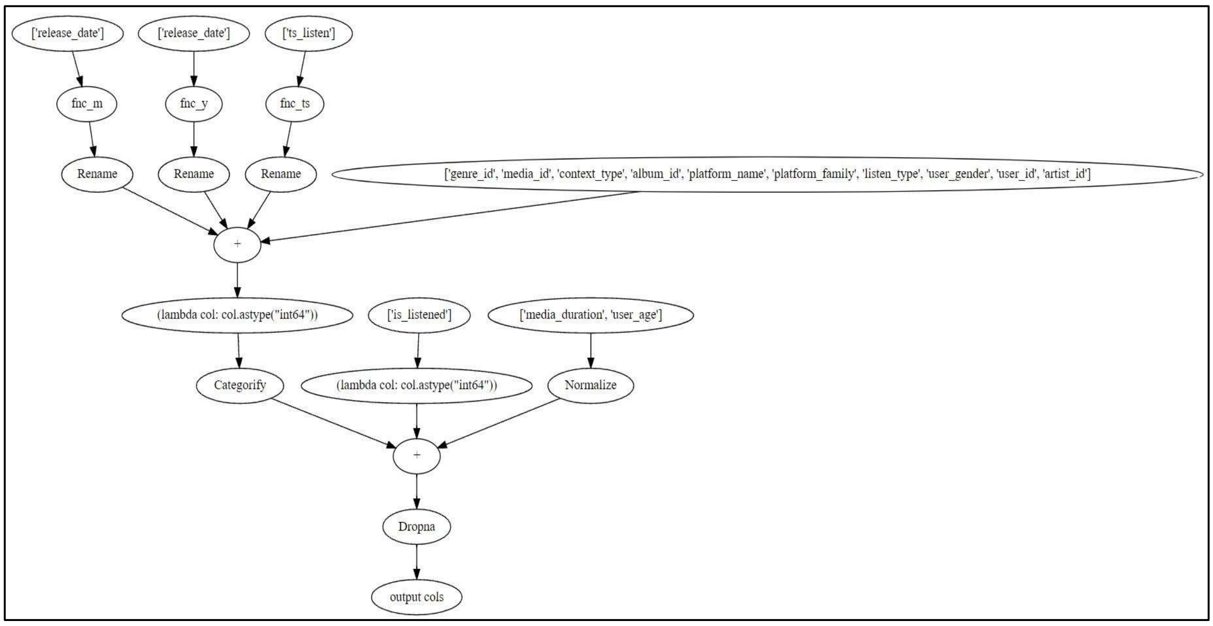

['release_date']
['release_date']
['ts_listen']
fnc_m
fnc_y
fnc_ts
Rename
Rename
Rename
['genre_id', 'media_id', 'context_type', 'album_id', 'platform_name', 'platform_family', 'listen_type', 'user_gender', 'user_id', 'artist_id']
+
(lambda col: col.astype("int64"))
['is_listened']
['media_duration', 'user_age']
Categorify
(lambda col: col.astype("int64"))
Normalize
+
Dropna
output cols

Figure 6. Flow chart preprocessing

For this preprocessing, **NVTabular** was used. NVTabular is a tool from Nvidia that can process data in graphics processing units (**GPUs)** and thus offers a way to more efficiently mine large amounts of data for recommendation tasks and therefore fits perfectly with the topic of the authors' work (jperrez99, 2020/2021; Nguyen et al., 2020).

For more details on this section, see Jupyter file “01_Preprocessing/prep_nvtabular.ipynb”

Once the pipeline for efficient data processing was established, the authors of this paper sought to develop accurately working recommender systems. These are discussed in the following part of the paper.

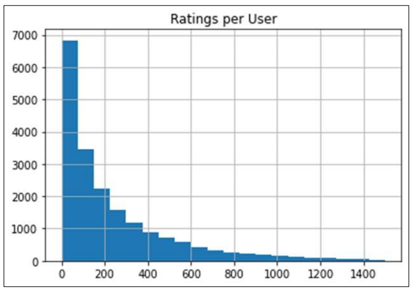
4. Applying Collaborative Filtering Models

4.1. Simple Collaborative Filtering Model

To start working with a recommender model, a user-based collaborative filtering recommender engine was created. The goal in building the model was to explore the possibilities of a user-based collaborative model build on scratch with the help of the python scikit-learn library. The approach is based on two articles from towardsdatascience.com. The authors GreekDataGruy (2019) and Liao (2018) describe how they created a collaborative filtering recommendation engine. Taking these articles into consideration, the authors of this paper have implemented the collaborative user-based model the in the following way:

4.1.1. Configuring the Dataset

Firstly, several calculations to further explore the data were undertaken to further assess the dataset for the specific model. The average number of ratings per user is around 267. The average number of ratings per song is around 14. The variable "is_listened" serves as the target variable for the recommender engine. In this work the target variable "is_listened" is referred as "rating".



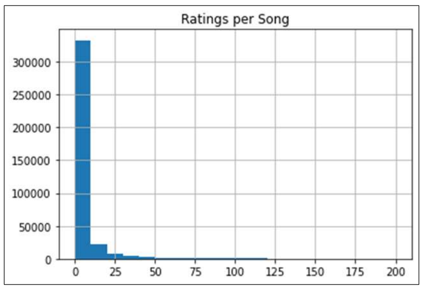
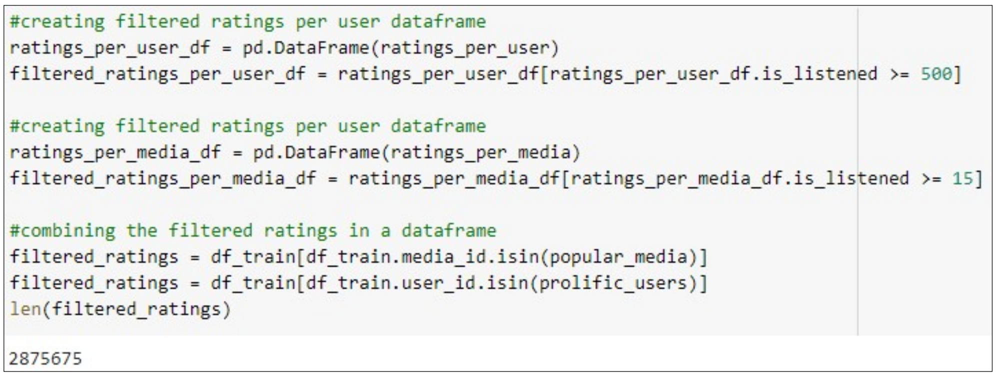


Figure 7. Ratings per user

Figure 8. Ratings per song

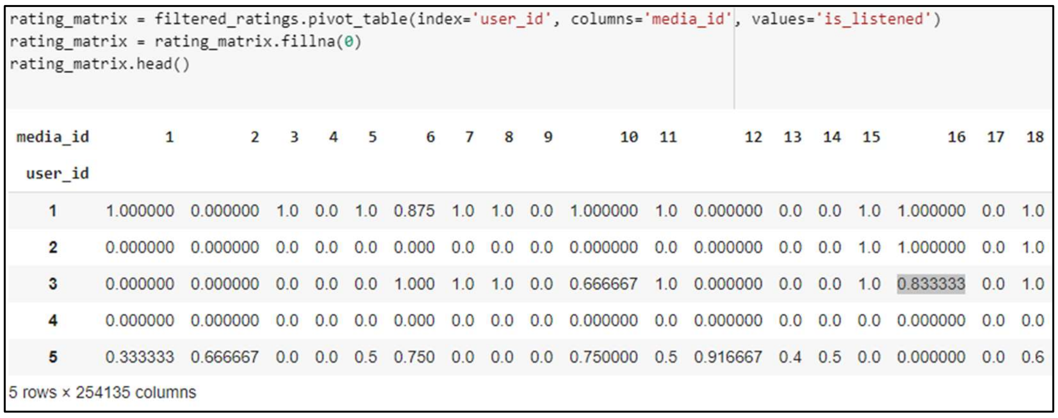
Only a small number of songs is rated by users. Hereby, especially, the number of ratings per users and the ratings per song provided valuable insights. This distribution phenomenon is described as long-tails in recommender systems (Yasar, 2018). This insight led to that the data for building the model was filtered by a threshold of ratings per song and a threshold of ratings per user. Creating the ratings matrix, the authors encountered performance issues, due to the size of the data. The authors tackled this issue with filtering the data to a greater extend. This process was characterised by a trial-and-error approach. The filtered data served as the basis for the development of the ratings matrix. In figure 9 the filtering process is depicted.



```
#creating filtered ratings per user dataframe
ratings_per_user_df = pd.DataFrame(ratings_per_user)
filtered_ratings_per_user_df = ratings_per_user_df[ratings_per_user_df.is_listened >= 500]

#creating filtered ratings per user dataframe
ratings_per_media_df = pd.DataFrame(ratings_per_media)
filtered_ratings_per_media_df = ratings_per_media_df[ratings_per_media_df.is_listened >= 15]

#combining the filtered ratings in a dataframe
filtered_ratings = df_train[df_train.media_id.isin(popular_media)]
filtered_ratings = df_train[df_train.user_id.isin(prolific_users)]
len(filtered_ratings)

2875675
```

Figure 9. Filtering process

The resulting rating matrix is sparse and contains many missing data points. These missing datapoints were each filled with a 0. The ratings matrix is depicted in figure 10.



```
rating_matrix = filtered_ratings.pivot_table(index='user_id', columns='media_id', values='is_listened')
rating_matrix = rating_matrix.fillna(0)
rating_matrix.head()
```

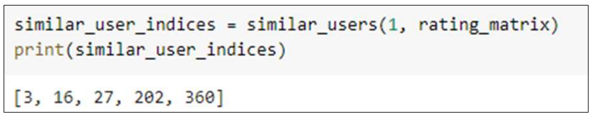
media_id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
user_id																		
1	1.000000	0.000000	1.0	0.0	1.0	0.875	1.0	1.0	0.0	1.000000	1.0	0.000000	0.0	0.0	1.0	1.000000	0.0	1.0
2	0.000000	0.000000	0.0	0.0	0.0	0.000	0.0	0.0	0.0	0.000000	0.0	0.000000	0.0	0.0	1.0	1.000000	0.0	1.0
3	0.000000	0.000000	0.0	0.0	0.0	1.000	1.0	1.0	0.0	0.666667	1.0	0.000000	0.0	0.0	1.0	0.833333	0.0	1.0
4	0.000000	0.000000	0.0	0.0	0.0	0.000	0.0	0.0	0.0	0.000000	0.0	0.000000	0.0	0.0	0.0	0.000000	0.0	0.0
5	0.333333	0.666667	0.0	0.0	0.5	0.750	0.0	0.0	0.0	0.750000	0.5	0.916667	0.4	0.5	0.0	0.000000	0.0	0.6

5 rows × 254135 columns

Figure 10. Creating Ratings Matrix

4.1.2. Creating the Recommender Engine

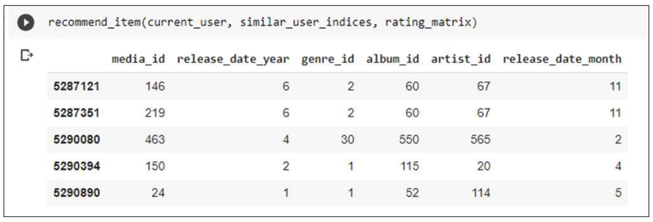
After creating the ratings matrix, a function generating a list of similar users for a specific user was applied (similar_users()). The function uses the cosine similarity function from the of the scikit learn library to create a list of similar users by using near neighbour search. The figure 11 shows the most similar users for the user with the id = 1.



```
similar_user_indices = similar_users(1, rating_matrix)
print(similar_user_indices)

[3, 16, 27, 202, 360]
```

Figure 11. Calling function similar_users()

Secondly, a function recommending five top items was established. To create a list of recommended songs, a separate dataframe consisting only of the unique songs with their specific datapoint describing the song (media_id, release_date_year, genre_id, album_id, artist_id and release_date_month) was used. The recommended items were than mapped to this separate dataframe. Calling the function recommend_item() the output for the user with the id=1 looks as follows (figure 12).



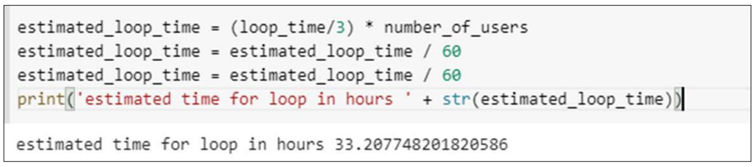
```
recommend_item(current_user, similar_user_indices, rating_matrix)
```

	media_id	release_date_year	genre_id	album_id	artist_id	release_date_month
5287121	146	6	2	60	67	11
5287351	219	6	2	60	67	11
5290080	463	4	30	550	565	2
5290394	150	2	1	115	20	4
5290890	24	1	1	52	114	5

Figure 12. Calling function recommend_item()

4.1.3. Results and Implications

The recommendation engine works in creating recommendations for specific users. However, trying to apply the engine to all users, the authors realised that the engine lacks scalability functionality. By using a for loop the estimated time for creating recommendations for all datapoints resulted in 33 hours (figure 13).



```
estimated_loop_time = (loop_time/3) * number_of_users
estimated_loop_time = estimated_loop_time / 60
estimated_loop_time = estimated_loop_time / 60
print('estimated time for loop in hours ' + str(estimated_loop_time))

estimated time for loop in hours 33.207748201820586
```

Figure 13. Result of assessment of estimated time to train the engine for all datapoints

Moreover, due to the long tail caused by the rating frequency, the engine is prone to the popularity bias (Catalogue of Bias Collaboration, 2018). Furthermore, due to the lack of insufficient datapoints the recommender faces the cold-start problem (Lendave, 2021). Since the engine is not likely to recommend less rated items, the engine lacks in diversity for recommended songs too (Saúl Vargas Sandoval, 2012). Therefore, more diverse approaches to conquer the recommender challenge were investigated.

For more details on this simple collaborative filtering model, see Jupyter file:

“02_Models/Simple_Collaborative_Filtering_Model.ipynb”.

4.2. Introduction to Matrix Factorization

To tackle the issues of the simple collaborative filtering model described in the previous subchapter, the authors have studied matrix factorization models. Matrix factorization is considered a well-known method that solves common issues related to recommender systems such as popularity bias and the cold-start problem.

The input for this algorithm is a rating matrix, that contains the information of which user rated which item. The items in the case of this project are the songs. By matrix factorization the matrix is split into two lower dimensional matrices: a user matrix and an item matrix, each expressed by a latent feature representation consisting of several layers.

Latent features are the features in the new dimension-reduced matrices, that are generated by the model, with the goal of imitating the real user-item interactions as good as possible. To reach this goal, the model is trained to reduce the loss between the ratings in the real world and the predicted ones in the latent representation.

The final output matrices are then used to make accurate recommendations. Since latent feature representation allows hidden information between user-item interactions to emerge, personal preferences get a much higher importance than the general popularity of individual songs with it. Considering the hidden information also solves the cold-start problem - the fact that there is little explicit information about the behaviour of new, as yet unknown, users. This all is a significant benefit for providing personalized recommendation (Liao, 2018b). In the following chapters, the matrix factorization models which were developed throughout this project will be discussed.

5. Matrix Factorization with Alternating Least Squares Method (ALS)

To apply matrix factorization, the authors have first developed an ALS matrix factorization model by utilizing **Spark**. Spark is a big data tool that is built to process large amounts of data within a short period of time. It runs data processing and machine learning operations **in-memory**, and thus saves unnecessary batch processes (load and save processes), as would be the case with Hadoop (Apache Spark, 2021a; Apache Spark, 2021b).

In addition to all the advantages of matrix factorization mentioned in the chapter "4.2. Introduction to Matrix Factorization", the use of ALS further solves the problem of the otherwise low scalability of matrix factorization tasks (Liao, 2018b). Since normally with matrix factorization, the data entries of the users with all data entries of the items can form a large set, the matrix calculations to be performed can become accordingly slow. To prevent this, ALS is used as an alternative cost function, which only brings a linear increase into the complexity of the learning process, as the amount of data increases (Kriplani, 2020). Thus, this model is also a solution approach for the problem with the "Simple Collaborative Filtering Model", which is mentioned in chapter 4.1.

These efficiency thoughts are the same as for preprocessing with NVTabular: Recommender systems not only need to work accurately but should also be able to compute the recommendations in the shortest possible time, in order to be useful in everyday applications like Deezer.

5.1. Implementation

The authors of this project have implemented ALS in Spark as follows:

- Column "user_id" was used to represent the users.
- Column "media_id" was used to represent the songs.
- Column "is_listened" was used to represent the ratings. The authors are aware that the information about "is_listened" does not directly represent a rating. However, it shows certain likes and dislikes, thus forming interaction between users and items.
- The amount of latent feature layers was set to 20. Therewith both, the user-matrix and item-matrix got 20 layers to represent the latent features.
- Maximal number of iterations of the algorithm was set to 10.

Furthermore, the authors have made configurations to Spark, which allows parallelized query processing to increase the efficiency even more.

Applied efficiency considerations are as follows:

- The users and items (songs) matrices were partitioned into blocks of size 1024MB in order to parallelize computation.
- Each Spark executor[1] was allocated a maximum of 6GB of random access memory (RAM),
  - [1]Spark executor: Executors are the individual work instances that divide incoming queries among themselves to enable parallelized processing (Hussain, 2019).
- 6GB of RAM has been allocated to the Spark driver[2].
  - [2] Spark driver: The spark driver is the place that orchestrates the work of the executors (Hussain, 2019).
- Each Spark executor was allocated up to 10 central processing unit (CPU) cores.
- Filter pushdown is enabled so that only the data needed for the various analyses is loaded into the executors.
- The loaded data were "cached in memory" to avoid multiple loading for the different analyses to be performed.

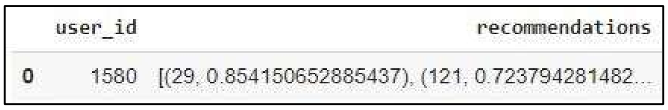
5.2. Evaluation

In the context of evaluating the predictions of this model, it achieved a root-mean squared error (RMSE) rate of about **0.43**, which is basically positive. In the upcoming chapter "8. Evaluation and Comparison of the Models", a further evaluation procedure will be discussed.

5.3. Real world application example

To test the model in its applicability for real-world uses, songs were recommended for users, and matching users were found for songs. The figures below 14 and 15 show the results, with the recommendations listed in order of score.

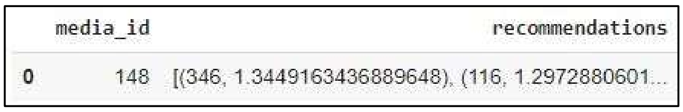
User 1580 gets the song no. 29 recommended to him with a score of 0.85:



	user_id	recommendations
0	1580	[(29, 0.854150652885437), (121, 0.723794281482...

Figure 14. Example user-based recommendations

On the other hand, the song no. 148 has been seen as most suitable for user 116:



	media_id	recommendations
0	148	[(346, 1.3449163436889648), (116, 1.2972880601...

Figure 15. Example item-based recommendations

For more details on this model, see the Jupyter file “02_Models/Matrix Factorization with Pyspark.ipynb”

6. Deep Neural Network Model with TensorFlow and NVTabular

NVTabular's support for fast GPU-based data pipelines is not limited to preprocessing, but also extends to the execution of deep neural networks in TensorFlow.

In its blog post, Nvidia claims that they have found that the data loader is a bottleneck in deep learning recommendation systems when training deep learning data pipelines with TensorFlow. They claim that the data loader cannot prepare the next batch fast enough and therefore the GPU is underutilized.

The authors of this project work have used the framework from Nvidia that provides a solution for this problem. This framework brings a new data loader with it, which can accelerate TensorFlow pipelines up to 9x according to Nvidia. The capabilities of NVTabular data loader are as follows:

- removes bottleneck of item-by-item data loading
- enables larger than memory dataset by streaming from disk
- reads data directly into GPU memory and removes CPU-GPU communication
- prepares batch asynchronously in GPU to avoid CPU-GPU communication
- supports commonly used Parquet-format
- easy integration into existing TensorFlow pipelines by using similar API
- works with tf.keras models

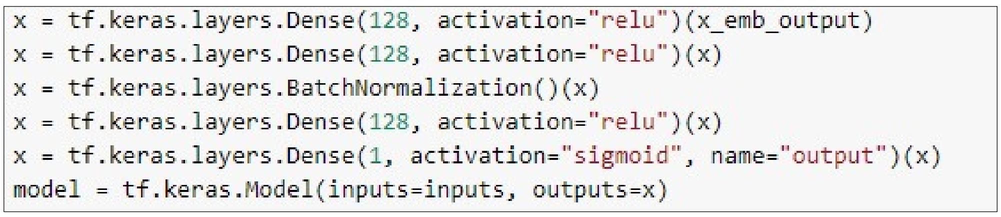
For more information in the blog mentioned, please refer to Oldridge, (2021) article "Training Deep Learning Based Recommender Systems 9x Faster with TensorFlow" .

6.1. Implementation of the model

This interaction between the data loader NVTabular and TensorFlow was taken by the authors of this work as the basis for the goal of creating an efficient data pipeline towards an accurately working deep neural network model. In this model, the authors intended to set all previously preprocessed features for the development of the recommender task as inputs. This is to detect all linear and hidden interactions, from the predictor columns to the binary target column "is_listened" (0 or 1) and the interactions of the predictor columns with each other.

The neural network model referred to has **three feed forward dense layers** with a **unit-size of 128 each**, that transfer the input features through latent features into a final one of unit-size one. There, on **sigmoid** basis the score is calculated and brought in a 0-1 range for each song to be treated, judging the probability that it will be listened. The activation functions in the previous layers are set on a **rectified linear unit (ReLU**) basis to avoid "squeezing" the latent features into the [0,1] value range. After the authors noticed that the accuracy remains in the 0.69 area over several epochs, they added a **batch normalization** right after the second layer and set the **learning rate to 0.01**. Therewith, the learning improved enormously. In this model, **stochastic gradient descent (SGD)** was used as the cost function, in place of ALS.

In figure 16, the process flow of the deep neural network model is shown on a code basis. For more details on this model, see the Jupyter file "02_Models/Training_with_TF.ipynb".



```
x = tf.keras.layers.Dense(128, activation="relu")(x_emb_output)
x = tf.keras.layers.Dense(128, activation="relu")(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.Dense(128, activation="relu")(x)
x = tf.keras.layers.Dense(1, activation="sigmoid", name="output")(x)
model = tf.keras.Model(inputs=inputs, outputs=x)
```

Figure 16. Deep neural network model

6.2. Training

The number of epochs to train the model was chosen so as to get past a plateau of learning. Ultimately, the model was trained over 500 epochs.

The figures 17 and 18 show how the accuracy and the loss has developed throughout the learning process. It can be seen that the model has learned well up to the 300 epochs, before the learning curve has flattened. The overshoots around the 250th epoch have had no effect on further learning. Plot 18 also shows that the validation loss has essentially steadily decreased, so that there was no overfitting of the model in the later epochs.

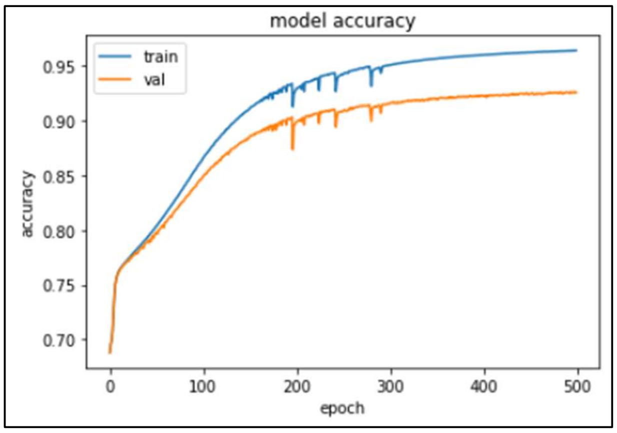


Figure 17. Development of accuracy of deep neural network throughout the epochs

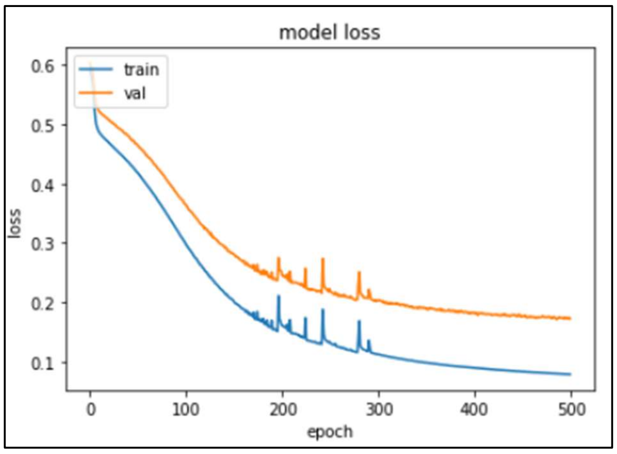


Figure 18. Development of loss of deep neural network throughout the epochs

7. Simplified Deep Neural Network Model with TensorFlow and NVTabular

Through the review of the previous model, it was supposed that perhaps the interactions of all the different predictor columns could have led to a degradation of the trained features and therewith, for example caused the overshoots of the learning around the 250th epoch.

7.1. Implementation

Therefore, a new model was created based on the same architecture, except the difference that the new model only considers the essential columns for recommender systems: the Id of the users (user_id) and the Id of the songs (media_id) as well as the information whether a user has listened to a song or not (is_listened). This made the model more similar to the matrix factorization models.

In figure 19 is a graphic representation of the architecture of the model.

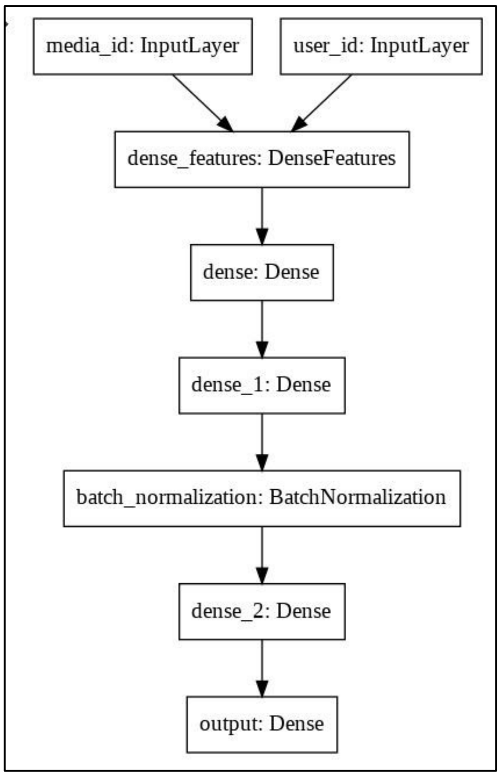


Figure 19. Flow chart simplified deep neural network model

7.2. Training

This model was trained over 200 epochs to have overcome the plateau of learning. The figures 20 and 21 show how the accuracy and the loss has developed for this simplified deep neural network model throughout the learning process. The model has learned well up to the 75 epochs, before the learning curve has flattened. This model has also not undergone any overfitting during learning, as can be seen from the continual decrease in the loss of validation over the entire course of the epoch (see figure 21).

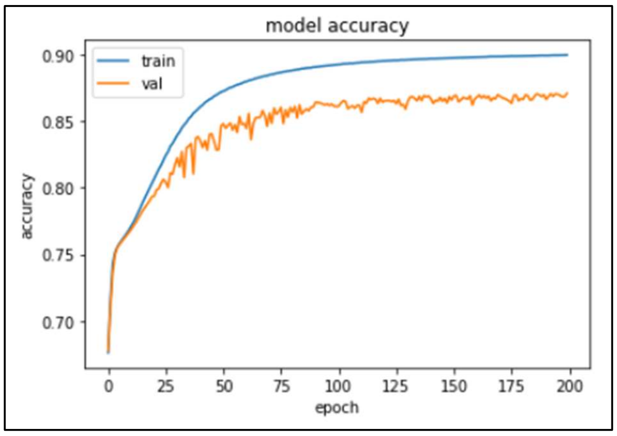


Figure 20. Development of accuracy of simplified deep neural network throughout the epochs

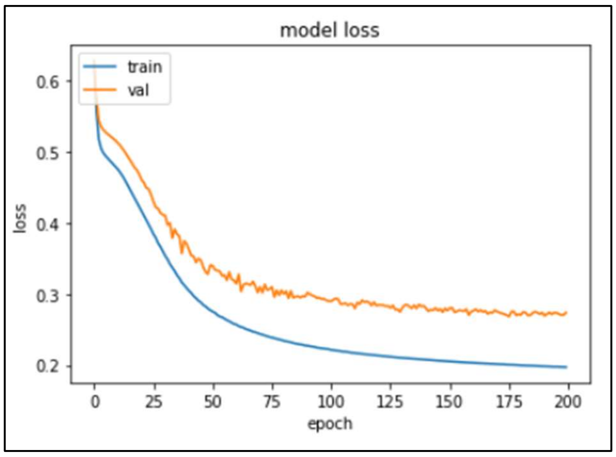


Figure 21. Development of loss of simplified deep neural network throughout the epochs

Ultimately, this simplified model, with its approximate 0.9 training accuracy and 0.85 validation accuracy score, fell short of the original deep neural network model, which achieved a training accuracy of approximate 0.95 and a validation accuracy of 0.9. This denies the above-mentioned claim, that the many input features in the original deep neural network model could have led to a degradation of the learning. On the contrary, the features helped the model to make more accurate predictions. More details on the testing of model performance are discussed in the chapter "8. Evaluation and Comparison of the Models".

For more details on the implementation of this model, see the Jupyter file:

“02_Models/Training_with_TF_simplified.ipynb”.

8. Evaluation and Comparison of the Models

In this chapter, the results of the evaluation of the models are discussed. For the evaluation, the same dataset was used for the different models, namely the previously untouched **test dataset**. Thus, the evaluation provides the basis for a comparison of the various models.

The three models, with which predictions could be made across the entire test dataset, were evaluated. These are the following ones:

- ALS Matrix Factorization Model with Spark (from chapter 5)
- Deep Neural Network Model with TensorFlow and NVTabular (from chapter 6)
- Simplified Deep Neural Network Model with TensorFlow and NVTabular (from chapter 7)

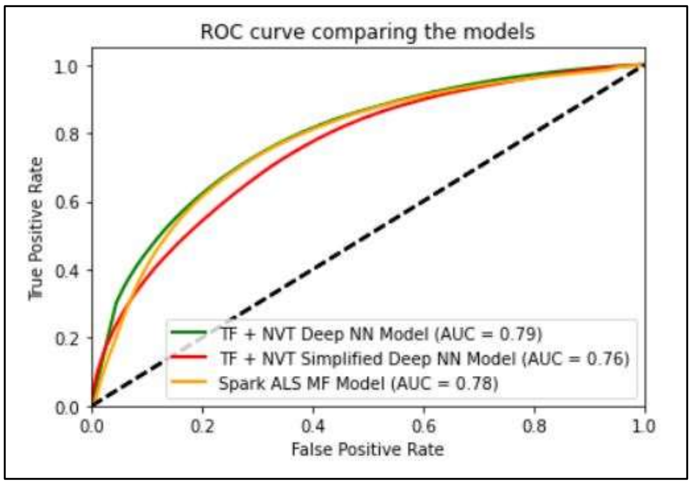


Figure 22. ROC-curve for evaluation of models

The evaluation was done by means of a ROC-curve plot shown in figure 22. The model "ALS Matrix Factorization Model with Spark" is shown with an **orange line** and achieved an area under curve (AUC) score of 0.78 and is therefore the second-best model. The best model is the "Deep Neural Network Model with TensorFlow and NVTabular" (**green line** in figure 22) with an AUC score of 0.79. In contrast, the "Simplified Deep Neural Network Model with TensorFlow and NVTabular" (**red line** in figure 22) achieved an AUC score of 0.76 and therewith became the worst of the three. This confirms once again that the original deep neural network model performed better than the simplified one, most probably because of the many more input features it has. The result of the AUC values is also affirmed by the way the visualization emerged.

If the aspect of efficiency were to be emphasized again, it is worth mentioning that the two neural network models went through 500 respectively 200 epochs for their trainings, while the ALS model only went through a maximum of 10 epochs. In this context, a runtime measurement of the training of the models up to their relatively high degree of learning or their learning plateau was also performed, which resulted in the following:

- **ALS Matrix Factorization Model with Spark** reached a relatively high degree of learning in **8min 34s.**
- Deep Neural Network Model with TensorFlow and NVTabular reached the plateau in 1h 52min 59s.
- Simplified Deep Neural Network Model with TensorFlow and NVTabular reached the plateau in 15min 12s.

To sum up, it can be said that **all three models work very accurately** overall and are on a par with each other, hence, the **ALS model** proves to be the **most efficient** of the three models.

For more details on the implementation of the evaluation, see the Jupyter file “03_Evaluation/Evaluation.ipynb”. To see the runtime measurements of the trainings, see the notebooks of the respective models mentioned.

9. Conclusion

To conclude the project, this chapter serves as a conclusion and wrap-up of the applied steps, models and discusses the steps. The process of the whole project is depicted in figure 23.

1. *Data Analysis and Visualisation*

In-depth, the raw data were examined and the target variable as well as the predictors were analysed and visualized. This provided the authors with an overview of the data. In addition, the time when the song was played, and the release date were converted in new predictors to find further insights in the raw data.

2. *Data Preprocessing*

To preprocess the enormous amount of data with modern approaches, the authors firstly converted the data into Parquet files to enhance load speed. Moreover, the data was divided into train, validation, and a test dataset. NVTabular as a modern big data tool was applied in the preprocessing steps, to meet the high requirements of the recommendation systems regarding efficient data processing.

3. *3.1. Simple Collaborative Filtering Model*

Applying a simple collaborative filtering model, the engine could create quick results, however, it lacked in scalability, was exposed to the cold-start problem, had a high probability to lack in diverse results and it faced the popularity bias. This led to applying matrix factorization models.

3.2. ALS Matrix Factorization with Spark

As a first approach to solve the problems of scalability, cold-start, diversity and popularity bias, the ALS method of matrix factorization with Spark was applied. In this way, an accurately working recommendation model was generated in a relatively short training period. The use of the ALS approach and the big data tool Spark particularly characterizes the solution for scalability.

3.3. Deep Neural Network with TensorFlow and NVTabular

In another process, a deep learning-based approach was applied using TensorFlow, with NVTabular accelerating the feature loading process. Hereby all predictor columns available in the dataset were loaded into the deep neural network model. When comparing the performance, this model turned out to be on par with the ALS model.

3.4. Simplified Deep Neural Network Model with TensorFlow and NVTabular

In order to bring the deep learning approach mentioned closer to matrix factorization, a similar model was created, but wherein only the most essential features representing the user-song interaction were loaded into the model. The aim was to find out whether such a deep learning model would perform better than the above-mentioned one, which takes all features into account.

4. *Evaluation and Comparison of the Models*

When all models were benchmarked, the deep neural network model, which considers all predictor columns, was found to be the most accurate, and the ALS model, the most efficient.

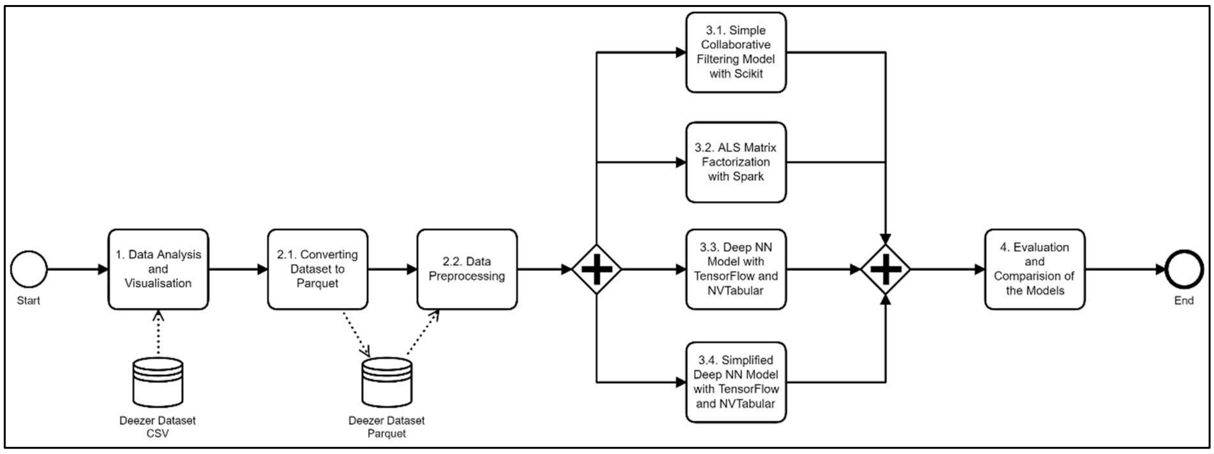

Start
1. Data Analysis and Visualisation
2.1. Converting Dataset to Parquet
2.2. Data Preprocessing
3.1. Simple Collaborative Filtering Model with Scikit
3.2. ALS Matrix Factorization with Spark
3.3. Deep NN Model with TensorFlow and NVTabular
3.4. Simplified Deep NN Model with TensorFlow and NVTabular
4. Evaluation and Comparision of the Models
End
Deezer Dataset CSV
Deezer Dataset Parquet

Figure 23. Process creating the recommender system

10. Literature

Apache Spark. (2021, Oktober 19). *Overview—Spark 3.2.0 Documentation*. https://spark.apache.org/docs/latest/

Catalogue of Bias Collaboration. (2018, April 27). Popularity bias. *Catalog of Bias*. https://catalogofbias.org/biases/popularity-bias/

Deezer. (2021). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Deezer&oldid=1049946394

Deezer -Über uns. (2021, Oktober 19). Deezer. https://www.deezer.com/de/company

GreekDataGruy. (2019, Oktober 22). Build a user-based collaborative filtering recommendation engine for Anime | by GreekDataGuy | Towards Data Science. *Towardsdatascience*. https://towardsdatascience.com/build-a-user-based-collaborative-filtering-recommendation-engine-for-anime-92d35921f304

Hussain, S. (2019, Dezember 27). Understanding the working of Spark Driver and Executor—Knoldus Blogs. *Knoldus*. https://blog.knoldus.com/understanding-the-working-of-spark-driver-and-executor/

jperrez99. (2021). *NVIDIA-Merlin/NVTabular* [Python; GitHub]. NVIDIA-Merlin. https://github.com/NVIDIA-Merlin/NVTabular (Original work published 2020)

Kaggle—DSG17 Online Phase. (2021, Oktober 19). https://kaggle.com/c/dsg17-online-phase

Kriplani, H. (2020, August 4). Alternating Least Square for Implicit Dataset with code. *Medium*. https://towardsdatascience.com/alternating-least-square-for-implicit-dataset-with-code-8e7999277f4b

Lendave, V. (2021, September 26). Cold-Start Problem in Recommender Systems and its Mitigation Techniques. *Analytics India Magazine*. https://analyticsindiamag.com/cold-start-problem-in-recommender-systems-and-its-mitigation-techniques/

Liao, K. (2018a, November 10). Prototyping a Recommender System Step by Step Part 1: KNN Item-Based Collaborative Filtering | by Kevin Liao | Towards Data Science. *Towardsdatascience*. https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-1-knn-item-based-collaborative-filtering-637969614ea

Liao, K. (2018b, November 19). Prototyping a Recommender System Step by Step Part 2: Alternating Least Square (ALS) Matrix…. *Medium*. https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-2-alternating-least-square-als-matrix-4a76c58714a1

Nguyen, V., Oldridge, E., & Huang, M. (2020, Juli 16). Accelerating ETL for Recommender Systems on NVIDIA GPUs with NVTabular. *NVIDIA Developer Blog*. https://developer.nvidia.com/blog/accelerating-etl-for-recsys-on-gpus-with-nvtabular/

Oldridge, E. (2021, Januar 6). Training Deep Learning Based Recommender Systems 9x Faster with TensorFlow | by Even Oldridge | NVIDIA Merlin | Medium. *NVIDIA Merlin*. https://medium.com/nvidia-merlin/training-deep-learning-based-recommender-systems-9x-faster-with-tensorflow-cc5a2572ea49

Saúl Vargas Sandoval. (2012). *Novelty and diversity enhancement and evaluation in recommender systems and information retrieval | Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval* [Universidad Autónoma de Madrid]. https://dl.acm.org/doi/10.1145/2600428.2610382

Thomas, S. (2019, Mai 28). Apache Parquet vs. CSV Files—DZone Database. *Dzone.Com*. https://dzone.com/articles/how-to-be-a-hero-with-powerful-parquet-google-and

Yasar, K. (2018, Juli 16). Recommender Systems: What Long-Tail tells ? *Medium*. https://medium.com/@kyasar.mail/recommender-systems-what-long-tail-tells-91680f10a5b2