UNIVERSITÀ DI PISA

MASTER'S DEGREE IN ARTIFICIAL INTELLIGENCE AND DATA ENGINEERING

DATA MINING AND MACHINE LEARNING

SYP 2.0 –SHARE YOUR PLAYLIST ENHANCED

PROJECT DEVELOPED BY

ANTONIO PATIMO, FABIO BUCHIGNANI

ACADEMIC YEAR 2021/2022

TABLE OF CONTENTS

INTRODUCTION AND REQUIREMENTS

Share Your Playlist is an application shaped as a social network about songs and playlists. Users can log into the platform to follow other users, review songs and rate them, create and share their own playlists and follow others' ones. The application offers the possibility of searching for a specific user, playlist or song using a search bar and then seeing their details. In order to improve the ease of browsing, each page containing the details of a specific user, playlists or songs shows also links to correlated pages, such as the owned playlists for a user or the contained songs for a certain playlist. In this way the user is encouraged in exploring the social network and being part of it. In addition to these core functionalities the application records statistics to be accessed by the users themselves or by administrators. As an example, an user can browse the most popular users or the most reviewed songs, while information collected specifically to help administrators is about how well the application is going in each social and non-social aspect.

MAIN ACTORS

The application interacts with an actor called "User". The user is a generalization of three types of users: Registered user, Unregistered user and Admin user.

- Unregistered user: An unregistered user can register to the application and use the "search" functionalities, like browse songs, but he can't use the social functions of the application;

- Registered user: A registered user can use all the basic functionalities of the application;

- Admin user: An admin user can use all the basic functionalities, view the statistical information and make important changes to the application.

FUNCTIONAL REQUIREMENTS

An unregistered user can register to the application and use the "search" functionalities:

- Register to the application;
- Look up for a user, playlist and song by name;
- Visualize information about a user, playlist and song;
- Visualize the most popular songs;
- Visualize the most followed playlists;
- Visualize the most followed users;
- Visualize the comments of a song.

A registered user is allowed to use all the search functionalities as the unregistered users and moreover to:

- Login and Logout from the application;
- Follow/Unfollow a user;
- Like/Unlike a playlist;
- Create, modify and delete his playlists;
- Modify and Delete his account;
- Comment a song;
- Modify or delete his comments;
- See suggestions on playlists based on the users that he follows.

An admin user is allowed to do all the registered user functionalities and moreover to:

- Change the privileges of a user;
- Add, delete and modify a song;
- Find the top k users that has created the highest number of playlists;
- Find the top k users that have added to their playlists the highest number of song of a specific artist;
- Find the users that follows at least k same playlists of the user provided in input;
- Find the k songs that has the highest number of comments;
- View statistics about the application;

NON FUNCTIONAL REQUIREMENTS

- The application must be always available and provide high quality of service.

- The application must react with low latency to the users activities.

- The application must be maintainable, in order to add new features and adapt to the future evolution of social networks and to the user's feedback.

- The application shall be user-friendly, since a user shall interact with an intuitive graphical interface.

SPECIFICATION

USE CASES DIAGRAM

The use cases diagram of the application is shown below:

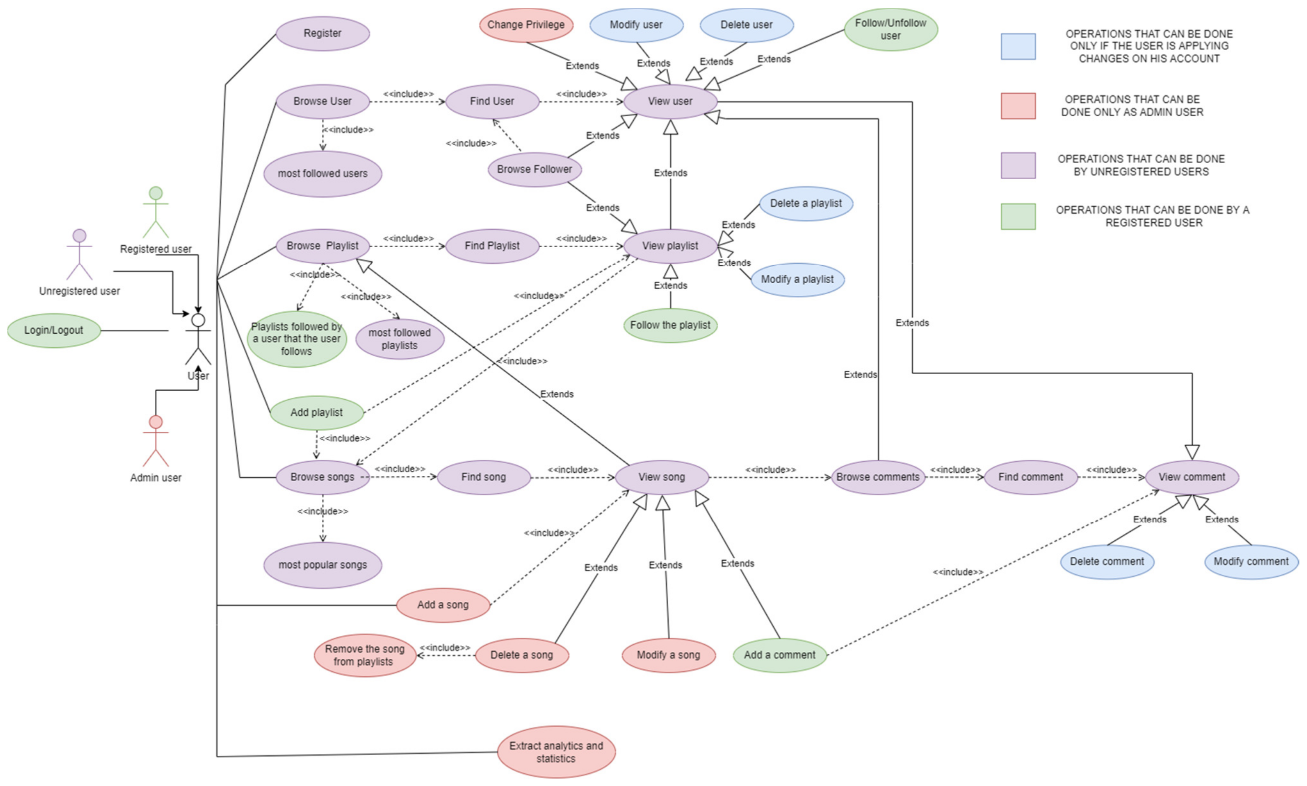


Figure 1: use cases diagram

CLASS DIAGRAM ANALYSIS

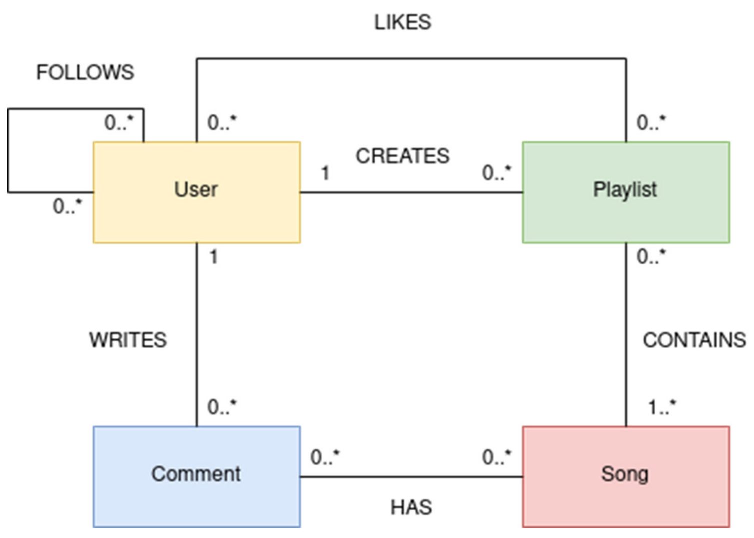


Figure 2: class diagram

- Users: a user is able to use all the basic functions of the application and interact with other users. A user is an Admin user if the attribute isAdmin is true. An admin user can use all the functionalities. Users are characterized by an id and must have unique usernames, in addition, they have a password, a date of birth and a date of creation.
- Playlists: a playlist is a set of songs that is created by a user and shared among the application. It can also be liked by a user. It is characterized by an id, a name and a creation date.
- Songs: a song whose information is offered by the application. It can be added to a playlist, get a comment and a rate. It is identified by an id, its attributes are track and artist.
- Comments: a comment is a review referred to a song, written by a user. It has a body and a vote, it is characterized by an id and a date.

DATA MODEL

MongoDB and Neo4J have been used to store data. MongoDB has been used to save details about songs, playlists and users; relationships and comments are instead stored in Neo4J.

In Neo4J we defined four kinds of nodes:

- User: it represents a user of the application and has the properties id and username.
- Song: it represents a song and has the properties id, track and artist.
- Comment: it represents a comment and has the properties date, vote and body.
- Playlists: it represents a playlist and has the properties id and name.

The relationships we defined are:

- FOLLOWS: a directed relationship that connects two users to represent the fact that a user follows another user.
- LIKES: a relationship that connects together a user and its liked playlists.
- WRITE: a relationship that connects together a user and its comments.
- RELATED: a relationship that connects together a song and its comments.

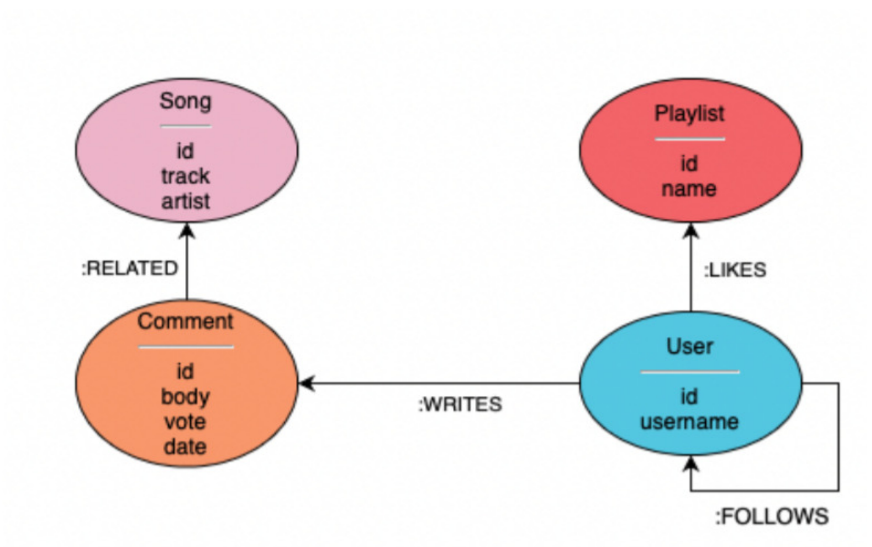


Figure 3: neo4J database organization

In MongoDB three different collections has been defined:

- users collection: to store details about users and some information about the created playlists.
- playlists collection: to store details about playlists and some information regarding contained songs and about the creator.
- songs collection: to store details about songs and some information regarding playlists that contain that song.

As defined later in the documentation, another collection was used to store information about what artists appear frequently together in playlists. The information in this collection is used to suggest songs to users when they create new playlists.

ARCHITECTURAL DESIGN

In terms of application architecture, it has been built according to the Client-Server paradigm.

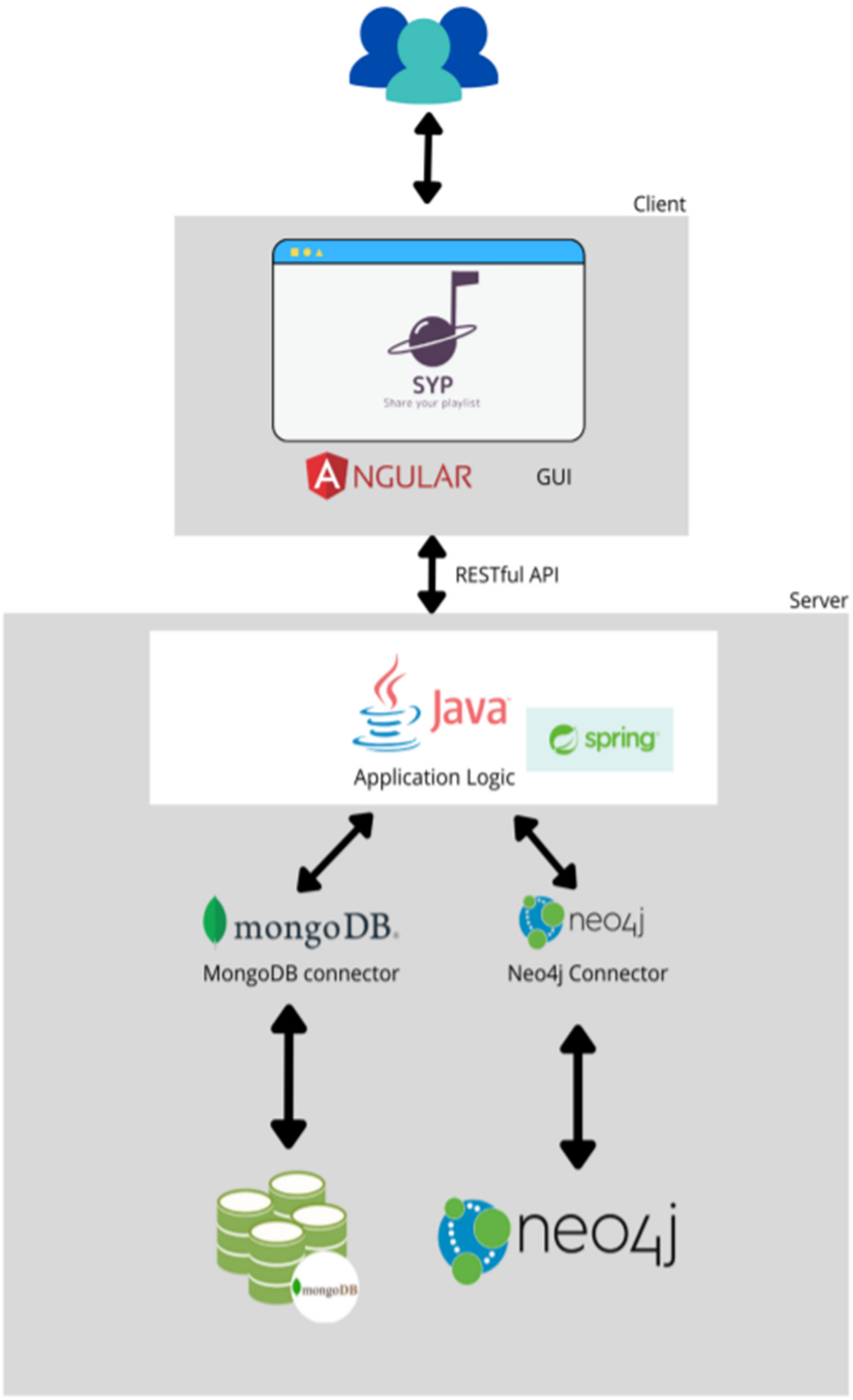


Figure 4: application architecture

CLIENT-SIDE

The client of the application has been developed using Angular framework and development platform, in order to create a single page application. Angular is a component-based framework where every part of the application is a component. Each component consist of:

- An HTML template that declares what renders on the page;
- A Typescript class that defines the behavior of the component;
- A CSS selector that defines the component style.

Angular implements the dependency injection design pattern and it is used for injecting in the components the services used to retrieve data from the server. These data are retrieved using REST API, an application programming interface that permits the exchange of the data from the client to the server over the HTTP protocol in a JSON format.

SERVER-SIDE

The server-side of the application has been developed using Java and Spring framework. It builds a Java Web API that can be exploited by the front-end, communicating thus with the http protocol and exchanging information through JSON documents. The application is divided in different packages, organized according to the suggested structure given in Spring documentation. Weka Java API was used to manage the data mining task of the application.

Maven was used to manage the Java part of the project and all its dependencies. About the version control, git was used along with GitHub. The repository can be found at the link: https://github.com/EhiSuper/syp2.0

DATA MINING

INTRODUCTION

When a user adds a new playlist he has to search for the songs he wants to insert. An improvement of the user experience would be to be able to reliably suggest songs on the basis of the already inserted songs, so that the system will help the user in retrieving easily the songs that he would like to put in that specific playlist.

The idea was to mine patterns of songs or artists that were put together in playlists, so that when some songs or artists appear very often together there is a higher probability that a user that inserts one of these songs in a playlist wants to insert also the other ones.

The system thus should first mine what songs or artists are correlated and then exploit this information to smartly suggest users with songs they may want to insert at that moment.

DATASET ANALYSIS

The dataset for the data mining task is taken from the application database because pattern mining should be done on actual stored data about playlists and songs and not from a static one. In any case, to build the initial database, we chose the spotify playlist dataset from Kaggle whose link is the following: https://www.kaggle.com/andrewmvd/spotify-playlists.

It contains tabular data with four attributes per record: the user_id, the playlist name, the track name and its artist, for a total dimension of 1 Gb. For practical purposes the dataset has been reduced and only 4% of the total dataset has been used to be stored in application databases. The total dimension of the database in MongoDB is about 57 Mb and in Neo4J is about 70 Mb.

Once the database was loaded, an analysis was performed on songs and artists to understand better what were the characteristics of our playlists and what could be the most interesting solution for the application. We discovered that the majority of songs were contained in only a small number of playlists, which means that a song appeared in only one playlist or a few more.

As we can see from the graph below, which represents on a logarithmic scale the percentage of songs and artists that are contained in a specific amount of playlists for different amounts of playlists.

We have higher percentages of songs that are contained in just a small number of playlists. Instead the percentages of artists are better distributed among a higher amount of playlists.

For this reason we decided to mine what artists go together in playlists, and thus find association rules about artists.

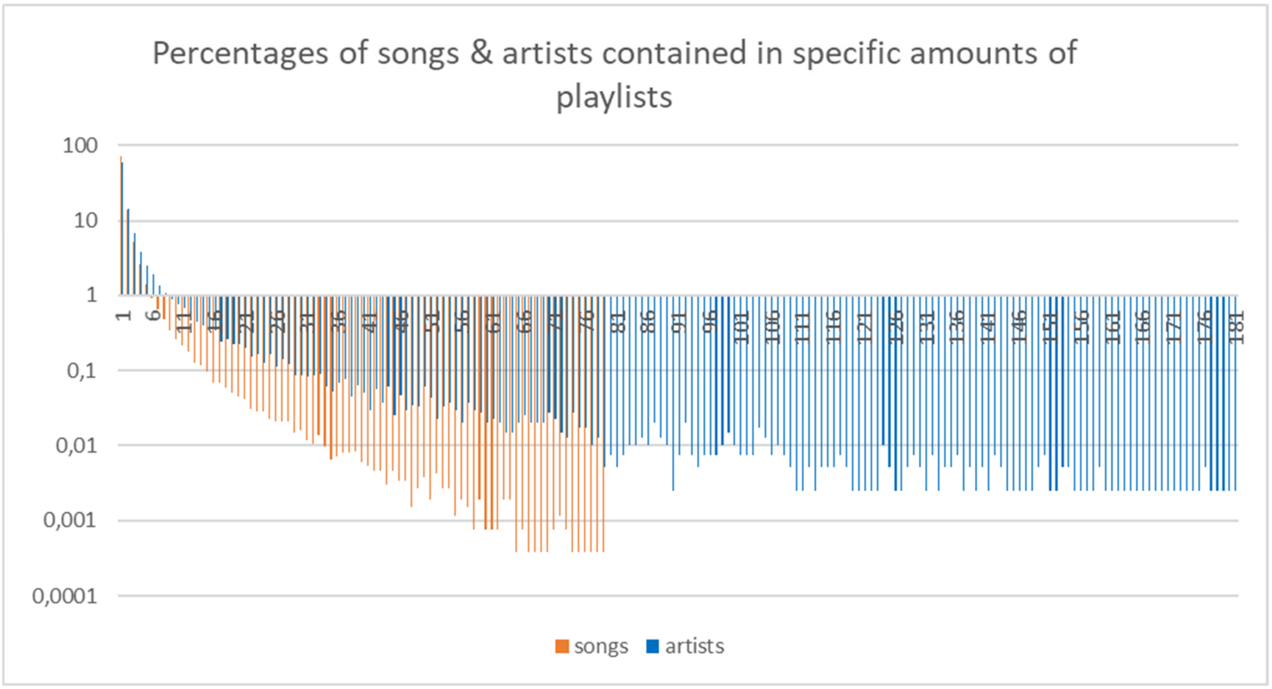
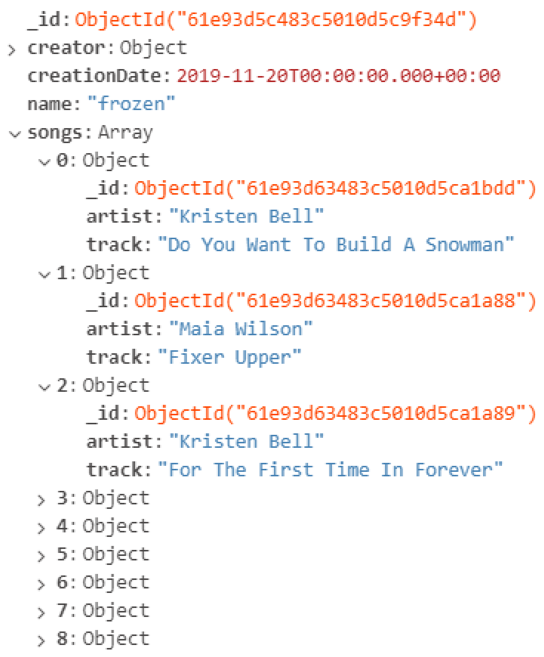


Figure 5: songs & artists distribution

DATA PREPROCESSING

Data about playlists and contained songs were saved in MongoDB. In order to use the application database to perform the pattern mining algorithms we had to transform the representation of our dataset.



```
  _id: ObjectId("61e93d5c483c5010d5c9f34d")
> creator: Object
  creationDate: 2019-11-20T00:00:00.000+00:00
  name: "frozen"
v songs: Array
  v 0: Object
      _id: ObjectId("61e93d63483c5010d5ca1bdd")
      artist: "Kristen Bell"
      track: "Do You Want To Build A Snowman"
  v 1: Object
      _id: ObjectId("61e93d63483c5010d5ca1a88")
      artist: "Maia Wilson"
      track: "Fixer Upper"
  v 2: Object
      _id: ObjectId("61e93d63483c5010d5ca1a89")
      artist: "Kristen Bell"
      track: "For The First Time In Forever"
  > 3: Object
  > 4: Object
  > 5: Object
  > 6: Object
  > 7: Object
  > 8: Object
```

Figure 6: MongoDb playlists collection

In order to use the FPGrowth algorithm offered by the Weka Java API, we needed to transform this dataset in a format with a list of binary attributes representing the presence or absence of each artist. A row of our dataset was a list of true or false values where true indicates that artist is present in the playlist and false indicates that he's not. An arff representation of the final dataset is shown below.

```
@attribute 'Kristen Bell' {t,f}
@attribute 'Maia Wilson' {t,f}
...
@attribute 'Daft punk' {t,f}

@data

t,t,...,f
```

Figure 7: dataset format for FPGrowth

To obtain this format we exploited a Weka filter called Denormalize, that took in input our dataset and created as output the dataset we needed. The initial dataset consisted of a list of playlists where each playlist contained the list of contained songs and relative artists. We extracted artists contained in each playlist and created a normalized dataset where in each row there was the playlist id and a single artist contained in that playlist. This was the format the Denormalize filter needed.

```
1, 'Kristen Bell'
1, 'Maia Wilson'
...
1850, 'Daft punk'
...
```

Figure 8: normalized dataset

PATTERN MINING

To perform pattern mining on our dataset we had basically two possibilities: Apriori algorithm or FPGrowth algorithm. Because of the size of our dataset, in particular the fact that we have almost 40000 attributes, the Apriori algorithm did not provide what we wanted, occupying too much memory and crashing when trying to perform pattern mining. On the other hand, the FPGrowth algorithm was much better and efficient with respect to Apriori.

The idea was not only to find interesting rules but also to have a consistent number of rules so that the system could suggest songs to the user in a higher number of cases. In order to find the best association rules for our application we had to choose among different configurations of the minimum support and the minimum confidence thresholds.

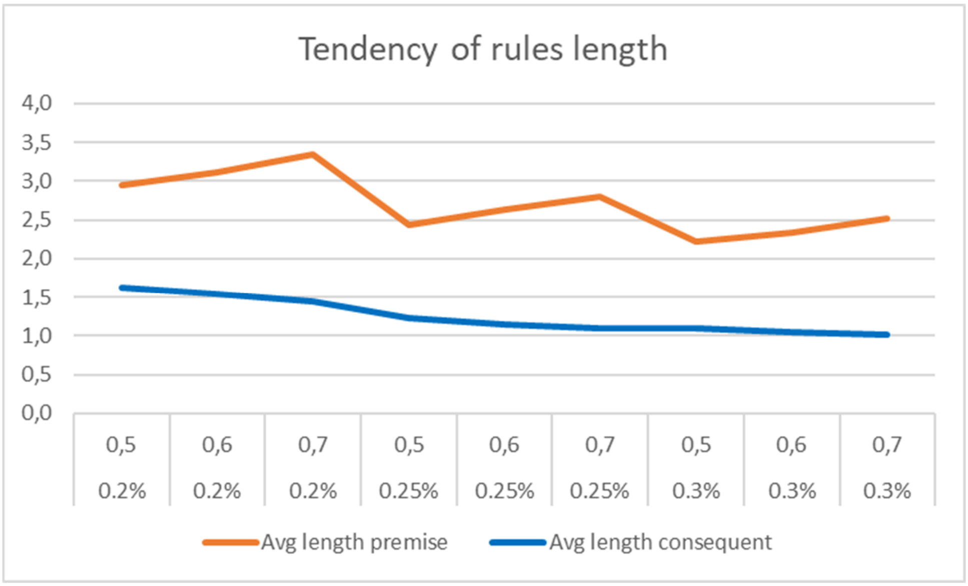
In the figures below are shown the different statistics obtained with different combinations of the two thresholds. In particular we decided to try with a minimum support of 0.2%, 0.25% and 0.3% while for the minimum confidence we selected the values 50%, 60% and 70%. It is important to underline that these values of min support are coherent with our application domain and the nature of our application. It depends on the diversity of the world of music and how different a playlist can be from another.

Min support	Min confidence	Max length premise	Min length premise	Avg length premise	Max length consequent	Min length consequent	Avg length consequent
0.2%	0,5	7	1	2,9458	6	1	1,6288
0.2%	0,6	7	1	3,1183	6	1	1,5348
0.2%	0,7	7	1	3,3369	5	1	1,4398
0.25%	0,5	4	1	2,4412	3	1	1,2306
0.25%	0,6	4	1	2,6278	3	1	1,1460
0.25%	0,7	4	1	2,7944	3	1	1,0996
0.3%	0,5	4	1	2,2177	2	1	1,0994
0.3%	0,6	4	1	2,3299	2	1	1,0399
0.3%	0,7	4	1	2,5254	2	1	1,0141

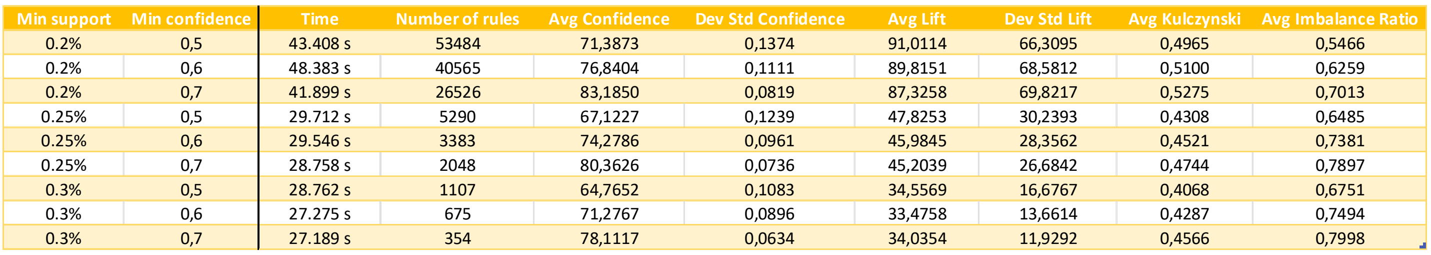
Figure 9: rules length statistics

In the table above we described the length of the rules found for each combination of minimum support and minimum confidence. The most relevant characteristics that resulted are the average length of premise and consequents. As we can see from the figure below, both tend to decrease as we increase the minimum support, but this shouldn't surprise: in general, the probability of having longer predicates decreases when we increase minimum support, and small rules tend to have higher supports.

It is interesting that increasing the minimum confidence, the average length of the premise increases independently from the value of the minimum support. It seems that in our application domain, stronger rules are the ones with an higher number of predicates in the premise, so that they are not pruned when we increase the minimum confidence. This is understable because the confidence of the rule depends on the inverse of the support of the premise, so if you select the rules with higher confidence you are doing an average on rules that have a lower support for the premise, the ones that in general have more artists.



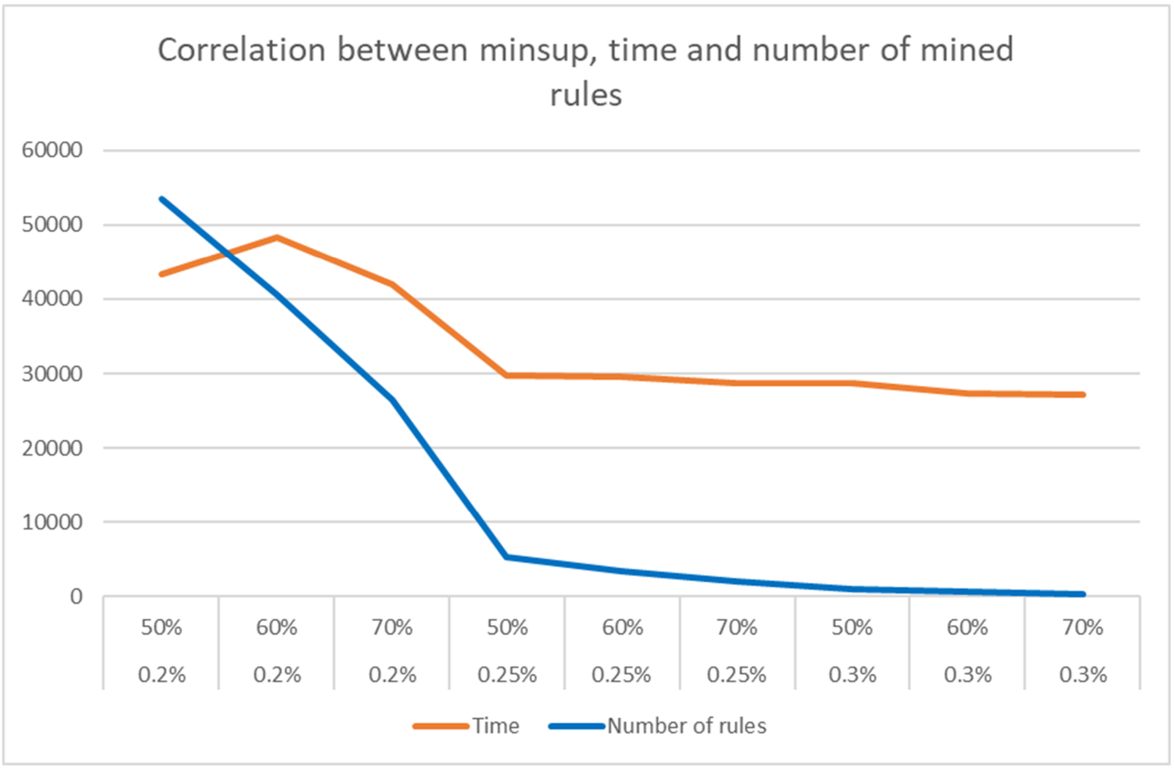
Additional statistics were computed to understand better what kind of rules we had and to have a better decision in what configuration we should choose among the ones we tried.



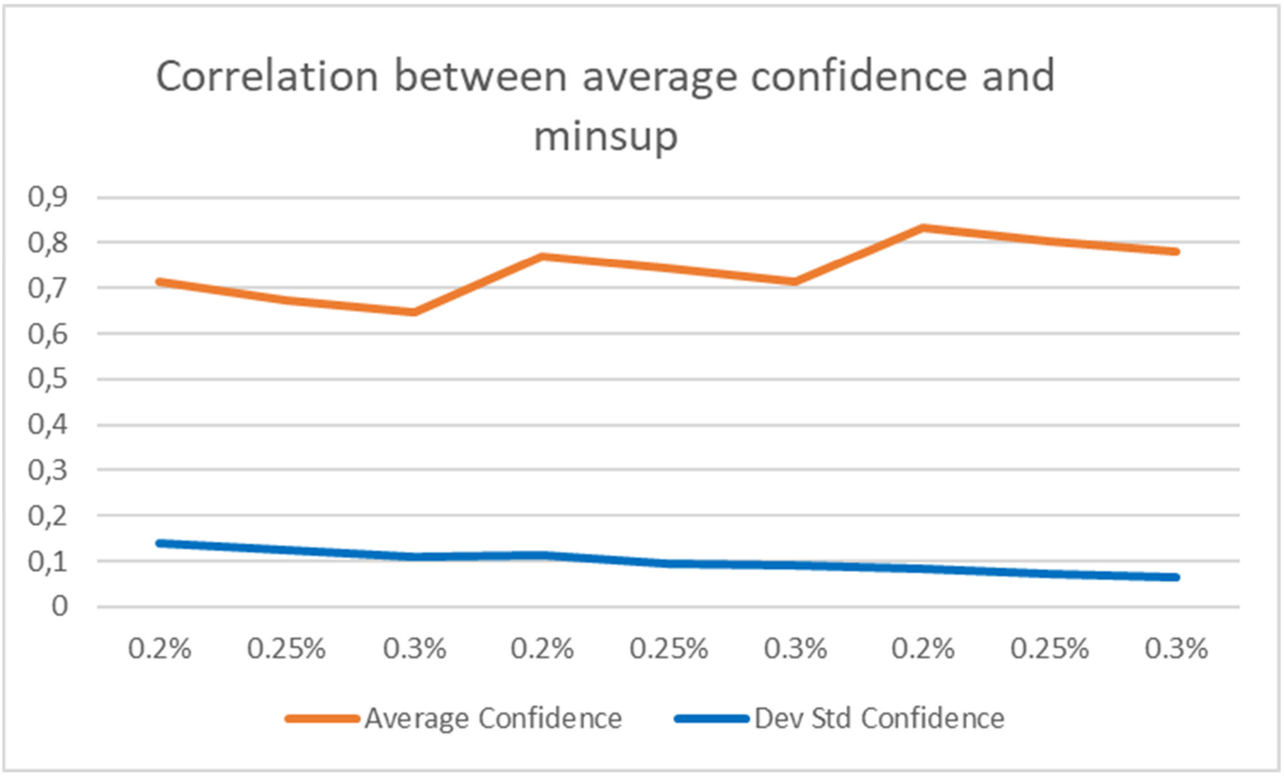
Min support	Min confidence	Time	Number of rules	Avg Confidence	Dev Std Confidence	Avg Lift	Dev Std Lift	Avg Kulczynski	Avg Imbalance Ratio
0.2%	0,5	43.408 s	53484	71,3873	0,1374	91,0114	66,3095	0,4965	0,5466
0.2%	0,6	48.383 s	40565	76,8404	0,1111	89,8151	68,5812	0,5100	0,6259
0.2%	0,7	41.899 s	26526	83,1850	0,0819	87,3258	69,8217	0,5275	0,7013
0.25%	0,5	29.712 s	5290	67,1227	0,1239	47,8253	30,2393	0,4308	0,6485
0.25%	0,6	29.546 s	3383	74,2786	0,0961	45,9845	28,3562	0,4521	0,7381
0.25%	0,7	28.758 s	2048	80,3626	0,0736	45,2039	26,6842	0,4744	0,7897
0.3%	0,5	28.762 s	1107	64,7652	0,1083	34,5569	16,6767	0,4068	0,6751
0.3%	0,6	27.275 s	675	71,2767	0,0896	33,4758	13,6614	0,4287	0,7494
0.3%	0,7	27.189 s	354	78,1117	0,0634	34,0354	11,9292	0,4566	0,7998

Figure 10: additional statistics about rules

As we can see from the table above and the relative graph below, the number of rules depends a lot from the value chosen for the minimum support threshold. As we can expect if we decrease the minimum support value the number of rules found increases a lot. Interesting is how this number varies also with a small difference of the minsup from 0.2% to 0.25%. This suggests that if we decrease the min sup value further we would find a very large, yet less interesting, number of rules that would increase the execution time of the algorithm. It is also interesting to notice that the number of association rules discovered don’t change so much from a min sup value of 0.25% to 0.30% and the fact that also the execution time remains more or less the same.



As we can see from the graph below, if we increase the minimum support , keeping the same minimum confidence, the average confidence of the rules found slightly decreases. It depends on the fact that it is more difficult to find rules with both high confidence and support. Obviously the standard deviation of the average confidence decreases because increasing the minimum confidence the range of the confidence of the rules discovered decreases.



About the average lift found we can say that in general a high value of the lift represents a strong correlation among our rules, but because the lift is not a null invariant measure it suffers from the presence of a lot of playlists that don't contain the rule. So we can conclude that in this case the lift is not a reliable statistic.

More reliable in this situation is the kulczynski measure because it is a null invariant measure. It generally works together with the imbalance ratio (IR) measure. We can see from the average imbalance ratio that we generally have skewed rules. This is because in our association rules the premise is generally composed by 2 or more artists meanwhile the consequent is generally composed by one artist, so the support of the consequent is generally higher than the support of the premise. The value of the kulczynski measure is slightly less than 0.5 because the confidence of the promise given the consequent in our rules is very low compared to the confidence of the consequent given the promise.

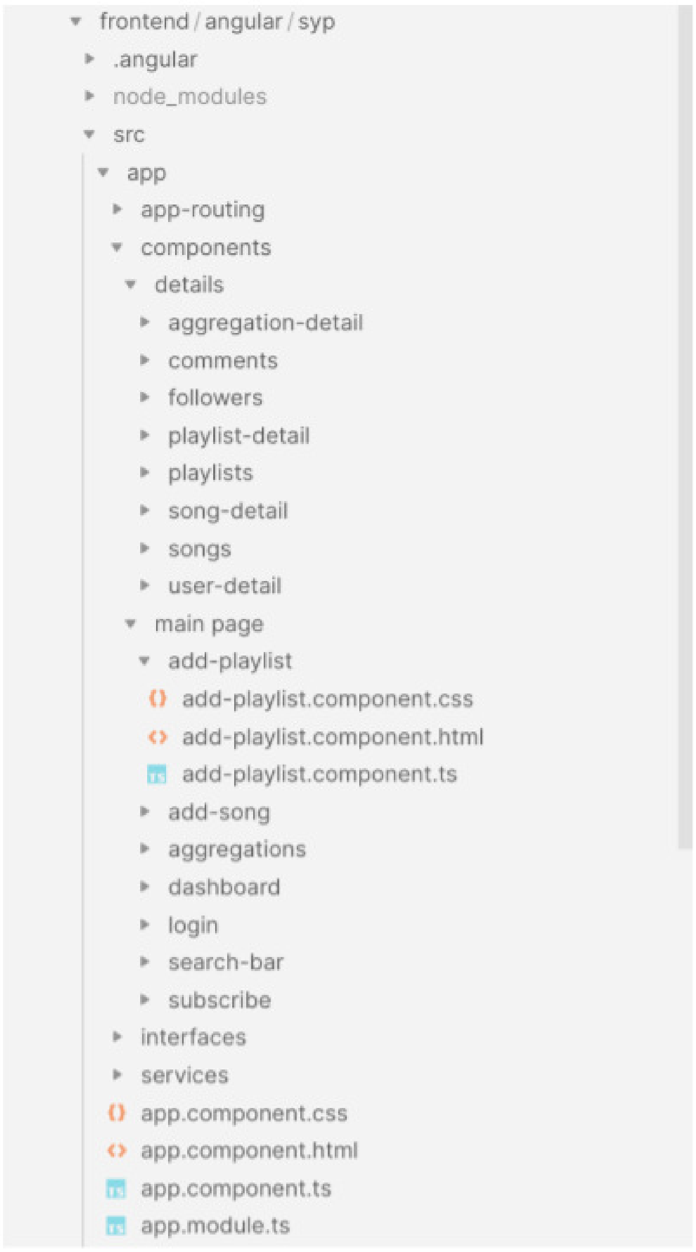
Given these considerations we decided to use in our application a value of minimum support of 0.2% associated with a minimum confidence value of 70%. These values permit us to find interesting rules with a confidence of 70% and in high numbers thanks to the smaller value of the minimum support.

IMPLEMENTATION DETAILS

FRONTEND

The frontend part of the application is in Angular. The structure has been divided in two main parts:

- The main page of the application, containing all the features that the user sees in the dashboard, included the possility of adding a new playlist.
- The details pages of the entities of the application.

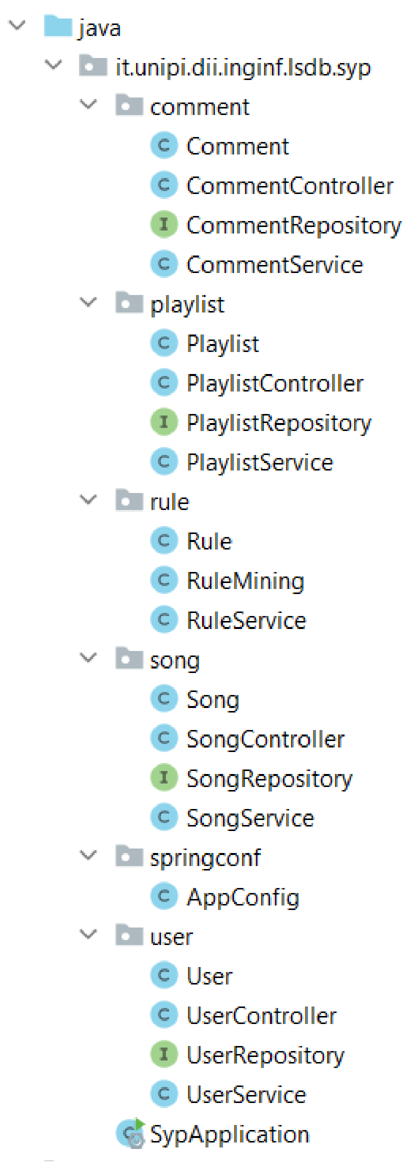


```
▾ frontend / angular / syp
  ▸ .angular
  ▸ node_modules
  ▾ src
    ▾ app
      ▸ app-routing
      ▾ components
        ▾ details
          ▸ aggregation-detail
          ▸ comments
          ▸ followers
          ▸ playlist-detail
          ▸ playlists
          ▸ song-detail
          ▸ songs
          ▸ user-detail
        ▾ main page
          ▾ add-playlist
              add-playlist.component.css
              add-playlist.component.html
              add-playlist.component.ts
          ▸ add-song
          ▸ aggregations
          ▸ dashboard
          ▸ login
          ▸ search-bar
          ▸ subscribe
      ▸ interfaces
      ▸ services
        app.component.css
        app.component.html
        app.component.ts
        app.module.ts
```

Figure 11: structure of Angular project

BACKEND

The backend part was written in Java. The package structure has been divided by features.



```
java
  it.unipi.dii.inginf.lsdb.syp
    comment
      Comment
      CommentController
      CommentRepository
      CommentService
    playlist
      Playlist
      PlaylistController
      PlaylistRepository
      PlaylistService
    rule
      Rule
      RuleMining
      RuleService
    song
      Song
      SongController
      SongRepository
      SongService
    springconf
      AppConfig
    user
      User
      UserController
      UserRepository
      UserService
    SypApplication
```

Figure 12: package structure

The main class, SypBackendApplication, is located in the root package and starts the java application.

The springConf package contains a class used to configure some aspects of Spring.

There is one package for each entity of the application, that contains the entity Java representation, the controller that defines the operations on that entity, and the service which implements the business logic. The connection with MongoDB is handled by the Spring class MongoTemplate, injected in the Service Class, while the connection with Neo4J is handled through the Repository interface injected in the Service Class as well. Each entity class contains all the information needed for that entity. Each entity has been enriched with annotations in such a way that Spring Data MongoDB and Spring Data Neo4J drivers can map instances automatically. These classes are automatically serialized in JSON objects to send them to the front-end. Each attribute could be null (not all the information related to an object is always needed, or always available), but the Spring default serializer has been configured to consider only non-null attributes when sending these objects to the front-end.

There is a package named rules in which we developed the data mining task of the application:

- The Rule class implements the Java representation of our association rules, as we defined them in the database. In details, we decided to use a MongoDB collection “rules” to store the mined rules and use it to retrieve relevant rules at the moment of the creation of a playlist.

- The RuleService class implements the connection with mongoDB and defines all the useful methods to save and retrieve rules.

- The RuleMining class implements the actual data mining task, this class is used to retrieve and trasform the dataset, and implements FPGrowth algorithm. Java Weka API has been used to perform this tasks.

- The RuleController class provides to the administrators the endpoint to start the mining process in a synchronized way.

It is also important to underline how we decided to suggest songs to the user, this was done implementing one additional feature in song package, namely the “suggested songs” feature.

In details, a new controller method specifically for the request of suggesting songs was added to the SongController class. Additional methods were instead added to the SongService class to obtain the list of suggested songs. We first check if an association rule whose premise contains precisely the specific set of artists inserted in the playlist is present, if it is, we use it to retrieve correlated artists. If it is not we search for rules whose premise contains the artists of the selected songs, but can have also other artists. If no rules are found, we search rules that have at least one of the selected artists in the premise. This was done to be able to suggest something to the user even if that precise set of artist didn’t appear in a mined association rule. Once we get the correlated artists, we retrieve their songs ordered by popularity (number of playlists that contain that song) and display the top ten to the user.