UNIVERSITÀ DI PISA

COMPUTER ENGINEERING

Data Mining and Machine Learning

WORKGROUP PROJECT DOCUMENTATION

Design and development of *“SoundHabit”*: an Application embedding Machine Learning Algorithms

Students

Federica Baldi

Daniele Cioffo

Academic Year 2020/2021

CONTENTS

1 DESIGN

1.1 THE APPLICATION

SoundHabit is an application whose main purpose is to help users discover new songs that they will most likely enjoy.

The application can automatically extract music information and classify the songs into their musical genres.

Moreover, the application generates for each user a customized list of recommended songs, based on their tastes and especially on the genres they appear to like best.

The only thing users need to do is to select, from the list of featured songs, the ones they like the most and let the application do all the work.

As a result, users can find out more about their personal tastes and hopefully answer the recurring question: *"What kind of music are you into?"*.

1.2 REQUIREMENTS

1.2.1 Main actors

The application is meant to be used by three different types of actors:

- *Anonymous Users,* who are only allowed to either register or login (if they already have an account).
- *Standard Users,* who are the primary users of the application. They can search for songs (applying filters on demand), like them or unlike them, and see personalized recommendations based on their tastes.
- *Administrators*, who are responsible for adding new songs to the application's database.

1.2.2 Functional requirements

- The application must provide a registration form in order to allow new users to sign up as *Standard Users*[1].
- The application must handle the login process, so that *Anonymous Users* can identify themselves via a *username* and a *password*.
- *Anonymous Users* must be denied the opportunity to access the functionalities offered by the application.

[1] Note that it is not possible to sign up as *Administrators*. In fact, application *Administrators* are already pre-registered.

- *Standard Users* must be given the ability to browse the list of featured songs, optionally using parameters (such as *title, author, genre, previously liked*).
- While viewing the details of a song, *Standard Users* must be given the opportunity to *like* the song or *unlike* it (if they had previously liked it).
- *Standard Users* must be shown a list of suggested songs based on their tastes (for instance, the genres of the song they previously liked).
- *Administrators* must be given the possibility to add songs to the database, specifying their details.
- *Administrators* must be provided with the opportunity to let the application automatically determine the genre of the song to be inserted.

1.2.3 Non-functional requirements

The following list outlines the non-functional requirements of the application.

- The application must embed one or more machine learning algorithms.
- *Usability*: The application must be *user friendly*, that is easy to use, with a simple and intuitive user interface.

1.3 Use Cases

Figure 1 shows the UML use case diagram representing users' interaction with the system.

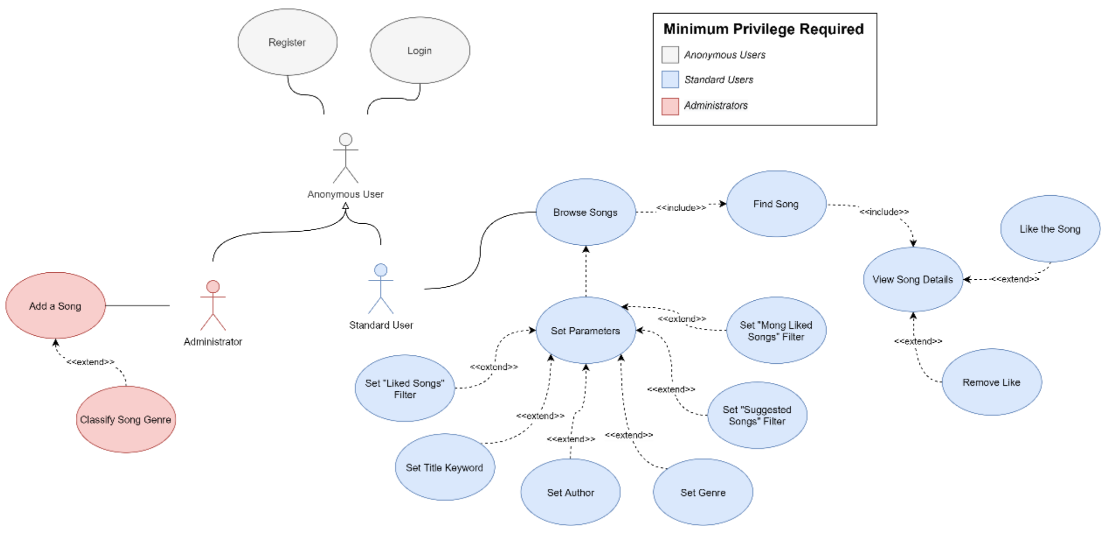
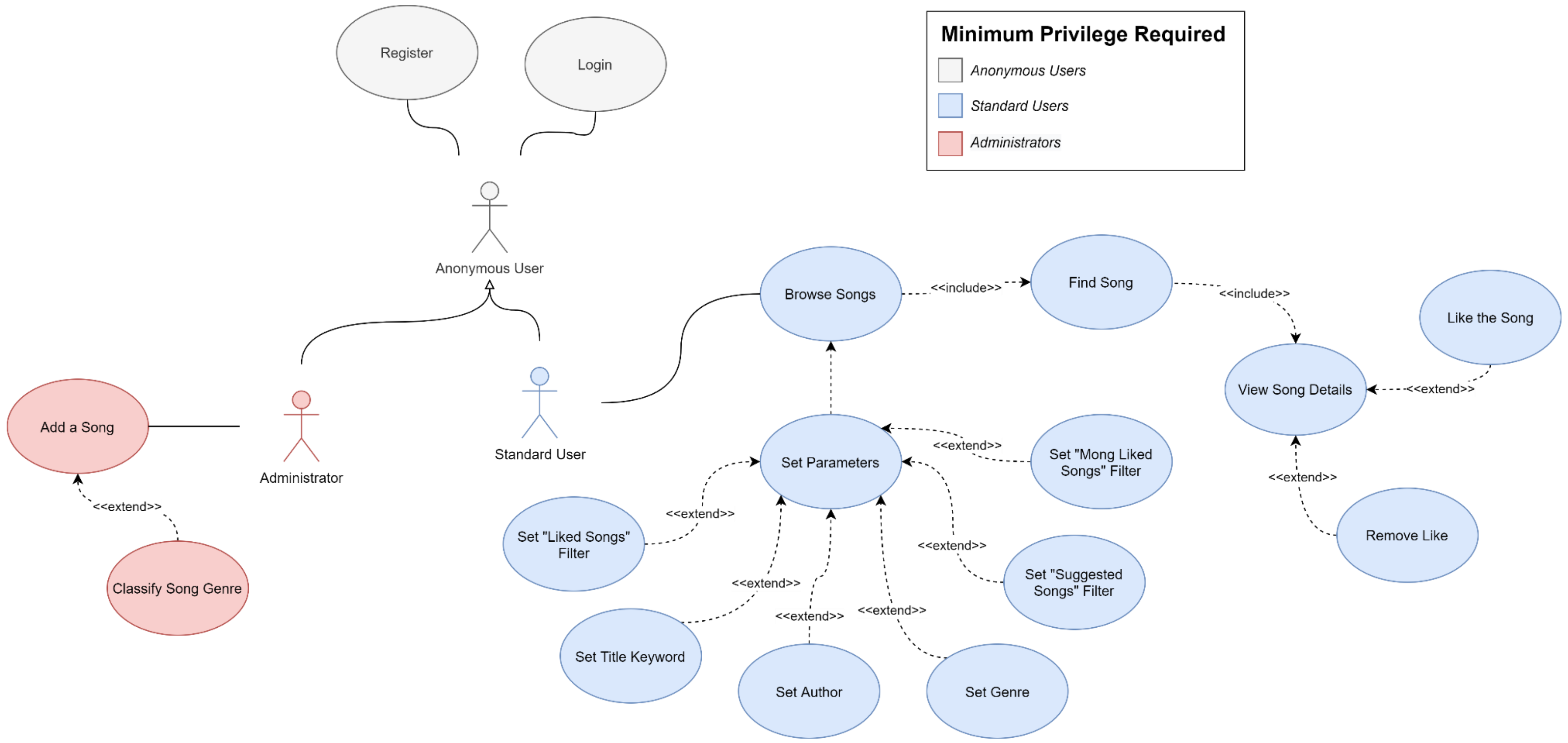


Figure 1. UML diagram of the main *Use Cases*


Register
Login
Anonymous User
Minimum Privilege Required
Anonymous Users
Standard Users
Administrators
Browse Songs
<<include>>
Find Song
<<include>>
Like the Song
View Song Details
<<extend>>
Add a Song
Administrator
Standard User
Set Parameters
<<extend>>
Set "Mong Liked Songs" Filter
<<extend>>
Remove Like
<<extend>>
Classify Song Genre
Set "Liked Songs" Filter
<<extend>>
<<extend>>
Set "Suggested Songs" Filter
<<extend>>
<<extend>>
<<extend>>
Set Title Keyword
Set Author
Set Genre

1.4 ANALYSIS CLASSES

Figure 2 shows the main entities of the application and the relationships between them.

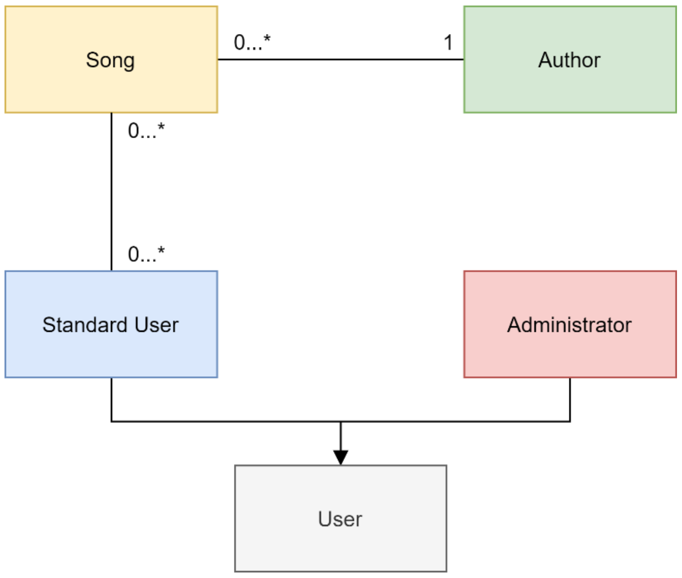


Figure 2. UML diagram of the *Analysis Classes*

Users are characterized by a *username* and a *password. Users* must be either *Standard Users* or *Administrators.*

Each *Standard User* may have liked none or many *Songs*. Similarly, each *Song* may have been liked zero of many times.

Songs are characterized by their *title*, *duration, genre,* an associated *image* and a *link* so you can listen to them. Moreover, each *Song* must have its own *Author*.

On the other hand, *Authors* may have zero or many *Songs* associated with them. Each *Author* is characterized by its *name*.

1.5 DATA MODEL

We decided to use a graph database to store our data. Thanks to it we can analyze in a performing way the relationships among users, song and author.

We have four types of nodes in our graph:

- **User**: It is the node that represents the user entity inside the graph. Its attributes are: *firstName*, *lastName*, *username*, *password*. To better model the entity User, we decided to use an *Administrator* label, which allow us to divide administrators from basic users.
- **Song**: It is the node that represents the song entity inside the graph. Its attributes are *name*, *songLink*, *imageLink, author*. To better model the entity Song, we

decided to use a different label for each genre, so we will have the labels *Blues*, *Classical*, *Jazz*, *Metal*, *Pop*, *Rock*.

There is also one relationship:

- **LIKES**: If the User *u* has liked a Song *s*, then there will be a relation LIKES from *u* to *s.*

We choose Neo4j as DBMS for the Graph database. There is also the necessity to use some constraints on the database:

- The username of the user must be unique and always present:

```
CREATE CONSTRAINT username_constraint ON (u:User)
ASSERT (u.username) IS NODE KEY
```

- The name of the song and the name of the author must be the key to the node Song, so together they must be unique and always present.

```
CREATE CONSTRAINT name_author_song_constraint ON
(s:Song) ASSERT (s. name, s.author) IS NODE KEY
```

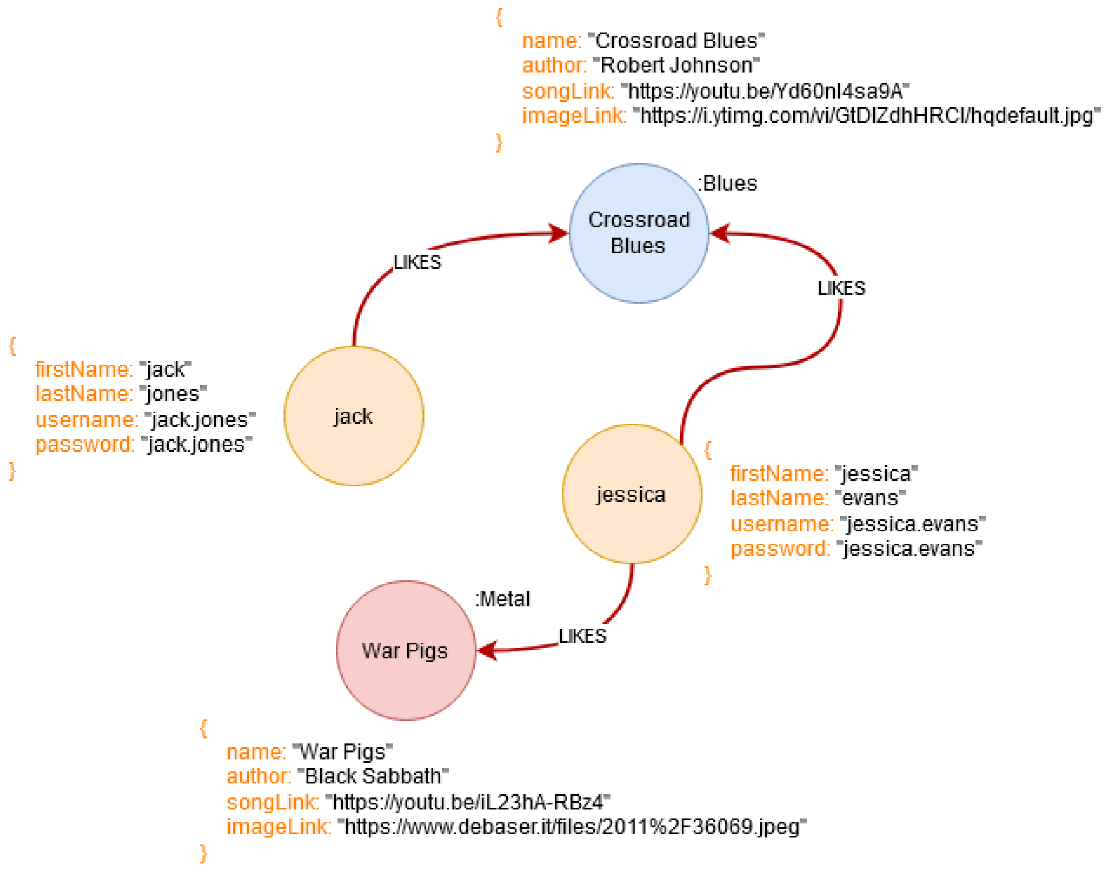


Figure 3. A partial example of GraphDB nodes and relationship

1.6 System Architecture

The architectural pattern used for the design of the overall system is the client-server one, as shown in Figure 4. Clients, for instance the users of the application, will request to query the database or to extract the song features to the server.

1.6.1 Client Side

On the client side, the Presentation Layer and the Logic Layer run. The *Presentation Layer* consists of a Graphical User Interface that users can access directly.

The *Logic Layer* is in charge of processing requests coming from the *Presentation Layer*, computing all the right calculations, and passing them to the server (via a specific communication protocol).

1.6.2 Server Side

The server will consist of two nodes (which can also run on the same machine):

- on one node, a copy of the database is maintained, and the Data Layer runs;
- on the other node there will be a server that will receive files containing music, save them temporarily, extract features from them and send the result back to the clients.

1.6.3 Frameworks

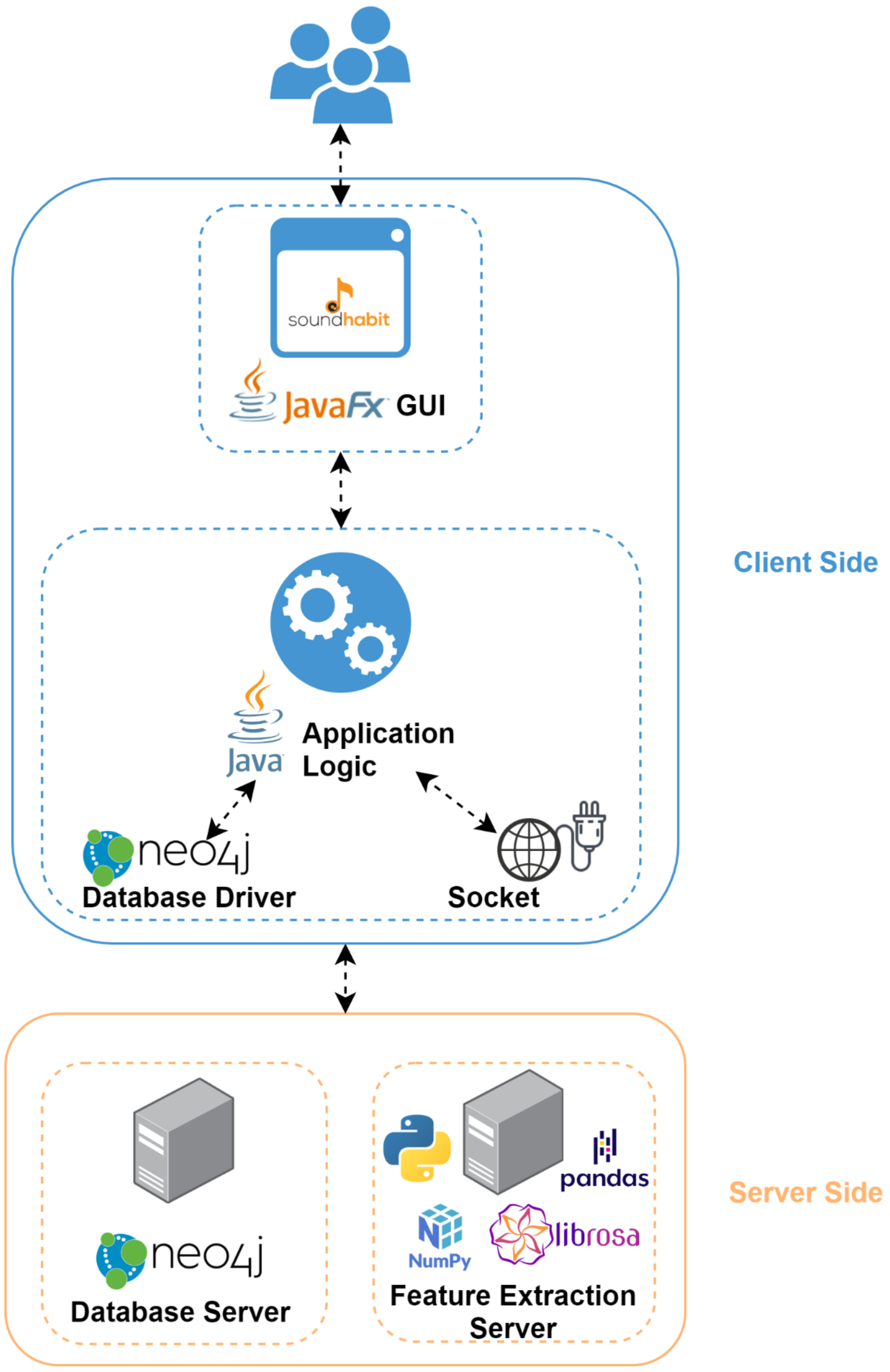
The application code will be written in Java, with the addition of JavaFX for the GUI. Concerning databases, Neo4j will be used.

The server that will extract features from files containing music tracks will be written in Python. In particular, the libraries Librosa, Pandas, and Numpy will be used.

Apache Maven will be used as a tool for building and managing the whole application, while for version control Git (in particular, GitHub) will be used.

All the source code for the application can be found on GitHub at the following link: https://github.com/danielecioffo/SoundHabit.

Figure 4. System architecture



2 Machine Learning

2.1 Introduction

A music genre is a *conventional category that identifies some pieces of music as belonging to a shared tradition or set of conventions*[2].

Classifying music genres has always been a task for humans, but, as the music industry has progressed, it has become increasingly important to be able to automate this process.

Indeed, to classify a music sample or song manually, the person has to listen to the song and select the genre. This is a time-consuming job and the person is required to have extensive knowledge in the music field.

Moreover, this approach is not scalable in real-world applications, with millions of songs in digital libraries and hundreds of thousands of songs releasing every year on digital music services.

In any case, automating the music classification is far from easy. First of all, the universe of musical genres is extremely vast, and new labels are being created all the time.

Second, although some features may uniquely characterize a genre, the boundaries between genres are not so clear, and often a single song can have influences from different genres.

The objective of our application, like that of the leading companies in the sector such as Spotify, Soundcloud and Shazam, is to overcome these difficulties and manage to classify the musical genre of songs as a first step to provide recommendations to users based on their tastes.

2.2 Dataset

It is now important to talk about the dataset used for the classification process.

After several research, we decided to use the GTZAN dataset, often used in various papers related to the classification of musical genres. It contains 10x100 songs, 100 songs for each one of the 10 main genres (blues, classical, country, disco, hip hop, jazz, metal, pop, reggae, rock). Its volume is around 1.23GB.

The link of the dataset is: marsyas.info/downloads/datasets.html

But since in various papers we have noticed that it is almost impossible to achieve good performances with many genres without relying on Deep Learning approaches, we have decided to focus our attention on 6 main genres: *blues, classical, jazz, metal, pop, rock*.

[2] Samson, Jim. "Genre". In Grove Music Online. Oxford Music Online.

2.3 Pre-processing

2.3.1 Feature Extraction

First we had to extract the audio features from the audio files. Indeed, the machine learning model isn't trained directly using the music, but it is trained by extracting features of the music sample. This is probably one of the most important steps, because the performance of our application will depend on how the features we choose to extract can characterize well the segments of audio.

To do this, we used the Librosa library, which allows you to obtain all this information in a simple and intuitive way. This feature extraction step is part of the preprocessing and lays its foundation on the *Short-time fourier transform,* which is implemented in the Librosa library and involves splitting an audio signal into frames and then taking the Fourier Transform of each frame.

The features we have decided to extract are the following:

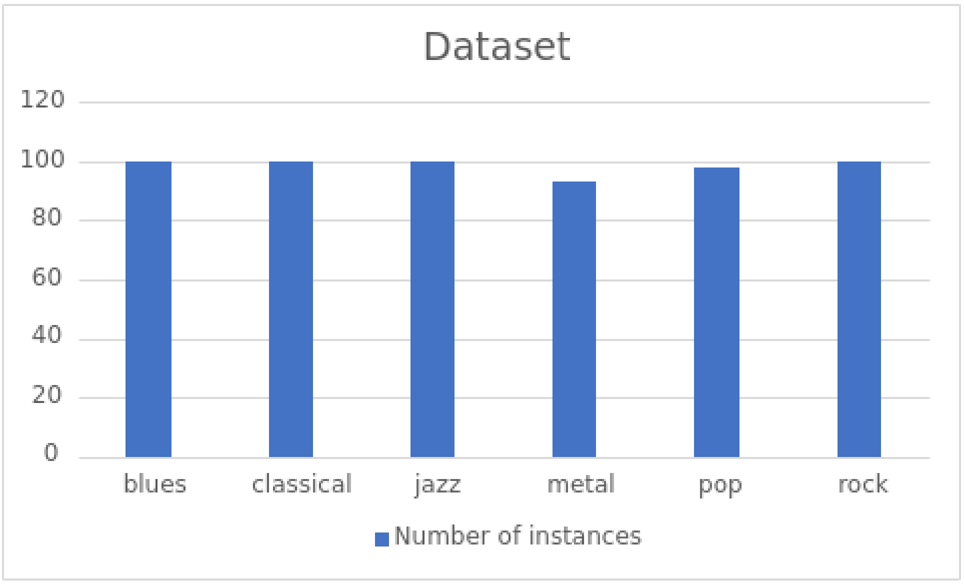
- **Chroma Frequencies**: The entire spectrum is projected onto 12 bins representing the 12 distinct semitones (or chorma) of the musical octave. The Librosa function returns the normalized energy for each chroma bin at each frame.
- **RMS**: Root-Mean-Square value for each frame.
- **Spectral Centroid**: indicates where the "*centre of mass*" for a sound is located and is calculated as the weighted mean of the frequencies present in the sound. For example, the spectral centroid for blues song will lie somewhere near the middle of its spectrum while that for a metal song would be towards its end. Each frame of a magnitude spectrogram is normalized and treated as a distribution over frequency bins, from which the mean (centroid) is extracted per frame.
- **Spectral Bandwidth**: frequency bandwidth for each frame.
- **Spectral Roll-off**: a measure of the shape of the signal. It represents the frequency below which a specified percentage of the total spectral energy, e.g. 85%, lies. The roll-off frequency is defined for each frame as the center frequency for a spectrogram bin such that at least roll_percent (0.85 by default) of the energy of the spectrum in this frame is contained in this bin and the bins below.
- **Zero crossing rate**: the rate at which the signal changes from positive to negative or back in each frame. (It usually has higher values for highly percussive sounds like those in metal and rock.).
- **MFCC (**Mel-frequency cepstral coefficients): a small set of features (usually about 10–20, but they can also be many more) which concisely describe the overall shape of the spectrum of the amplitude of the signal. We decided to calculate 20 MFCC, which are more than enough for our purposes.

We decided to take the mean when we had arrays of values, in order to get a single attribute.

So, at the end of this process, we have 26 audio features for each song. Once all the features were extracted, it was possible to create the actual dataset, which is used in the next steps.

2.3.2 Data Transformation

In addition to extracting features, we had to remove duplicates from our dataset, this is because the GTZAN dataset contained song duplicates. However, these duplicates were not many, so the classes are still balanced.



Finally, some algorithms need further filters, for example in the case of K-NN we needed to normalize the data, in order to give the same weight to all the features when calculating the Euclidean distance.

We chose the *z-score* normalization, in which we subtract the mean and divide by the standard deviation. We did not use the *min-max* normalization because it is more affected to the outliers.

2.4 Models Evaluation and Selection

The problem our application poses is a multiclass classification one; indeed, as previously anticipated, each song can belong to one of the six genres we decided to use.

In order to classify a new song whose genre is not known, we had to build and train a classification model.

Our goal was to obtain the best accuracy and, in particular, we targeted at least an accuracy of 70% (it is the one achieved on average by humans when classifying music genres[3]).

In order to build the best model possible, we decided to test different classifiers and evaluate their results on our dataset.

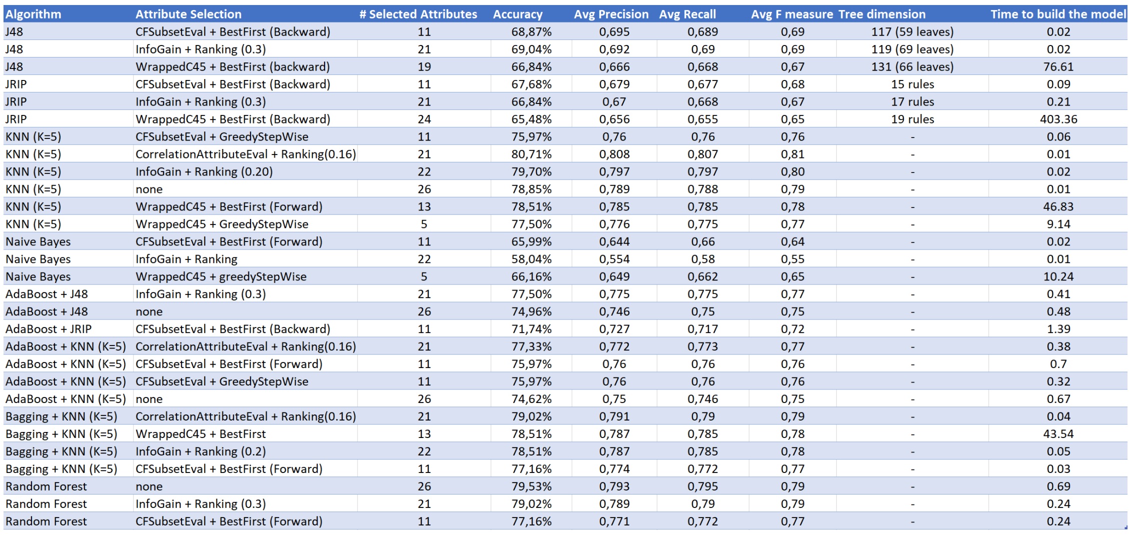
[3] Gjerdingen, Robert O. and Perrott, David (2008) *'Scanning the Dial: The Rapid Recognition of MusicGenres'*, Journal of New Music Research, 37: 2, 93 — 100

Thus, in the first phase of our analysis we tested various classifiers in combination with different attribute selection algorithms and using 10-fold cross-validation.

Among the various classifiers tested we have *J48* (an open source Java implementation of the C4.5 *decision tree algorithm*), *JRIP* (a propositional *rule learner*, RIPPER) *IBk* (*K-nearest neighbors*), and *Naïve Bayes*. We also tested ensemble methods such as *AdaBoost*, *Bagging*, and *Random Forest*.

As for the attribute selection algorithms, we mainly used *InfoGainAttributeEval* (the information gain with respect to the class is measured), *CFSubsetEval* (the individual predictive ability of each feature is considered, along with the degree of redundancy between them), and *CorrelationAttributeEval* (the correlation between the attribute and the class is measured), experimentally adjusting the parameters.

The results are shown in Figure 5, and they are also visually summarized in the box-plot in Figure 6.



Algorithm	Attribute Selection	# Selected Attributes	Accuracy	Avg Precision	Avg Recall	Avg F measure	Tree dimension	Time to build the model
J48	CFSubsetEval + BestFirst (Backward)	11	68,87%	0,695	0,689	0,69	117 (59 leaves)	0.02
J48	InfoGain + Ranking (0.3)	21	69,04%	0,692	0,69	0,69	119 (69 leaves)	0.02
J48	WrappedC45 + BestFirst (backward)	19	66,84%	0,666	0,668	0,67	131 (66 leaves)	76.61
JRIP	CFSubsetEval + BestFirst (Backward)	11	67,68%	0,679	0,677	0,68	15 rules	0.09
JRIP	InfoGain + Ranking (0.3)	21	66,84%	0,67	0,668	0,67	17 rules	0.21
JRIP	WrappedC45 + BestFirst (Backward)	24	65,48%	0,656	0,655	0,65	19 rules	403.36
KNN (K=5)	CFSubsetEval + GreedyStepWise	11	75,97%	0,76	0,76	0,76	-	0.06
KNN (K=5)	CorrelationAttributeEval + Ranking(0.16)	21	80,71%	0,808	0,807	0,81	-	0.01
KNN (K=5)	InfoGain + Ranking (0.20)	22	79,70%	0,797	0,797	0,80	-	0.02
KNN (K=5)	none	26	78,85%	0,789	0,788	0,79	-	0.01
KNN (K=5)	WrappedC45 + BestFirst (Forward)	13	78,51%	0,785	0,785	0,78	-	46.83
KNN (K=5)	WrappedC45 + GreedyStepWise	5	77,50%	0,776	0,775	0,77	-	9.14
Naive Bayes	CFSubsetEval + BestFirst (Forward)	11	65,99%	0,644	0,66	0,64	-	0.02
Naive Bayes	InfoGain + Ranking	22	58,04%	0,554	0,58	0,55	-	0.01
Naive Bayes	WrappedC45 + greedyStepWise	5	66,16%	0,649	0,662	0,65	-	10.24
AdaBoost + J48	InfoGain + Ranking (0.3)	21	77,50%	0,775	0,775	0,77	-	0.41
AdaBoost + J48	none	26	74,96%	0,746	0,75	0,75	-	0.48
AdaBoost + JRIP	CFSubsetEval + BestFirst (Backward)	11	71,74%	0,727	0,717	0,72	-	1.39
AdaBoost + KNN (K=5)	CorrelationAttributeEval + Ranking(0.16)	21	77,33%	0,772	0,773	0,77	-	0.38
AdaBoost + KNN (K=5)	CFSubsetEval + BestFirst (Forward)	11	75,97%	0,76	0,76	0,76	-	0.7
AdaBoost + KNN (K=5)	CFSubsetEval + GreedyStepWise	11	75,97%	0,76	0,76	0,76	-	0.32
AdaBoost + KNN (K=5)	none	26	74,62%	0,75	0,746	0,75	-	0.67
Bagging + KNN (K=5)	CorrelationAttributeEval + Ranking(0.16)	21	79,02%	0,791	0,79	0,79	-	0.04
Bagging + KNN (K=5)	WrappedC45 + BestFirst	13	78,51%	0,787	0,785	0,78	-	43.54
Bagging + KNN (K=5)	InfoGain + Ranking (0.2)	22	78,51%	0,787	0,785	0,78	-	0.05
Bagging + KNN (K=5)	CFSubsetEval + BestFirst (Forward)	11	77,16%	0,774	0,772	0,77	-	0.03
Random Forest	none	26	79,53%	0,793	0,795	0,79	-	0.69
Random Forest	InfoGain + Ranking (0.3)	21	79,02%	0,789	0,79	0,79	-	0.24
Random Forest	CFSubsetEval + BestFirst (Forward)	11	77,16%	0,771	0,772	0,77	-	0.24

Figure 5. Results table of 10-fold cross-validation performed on different classifiers

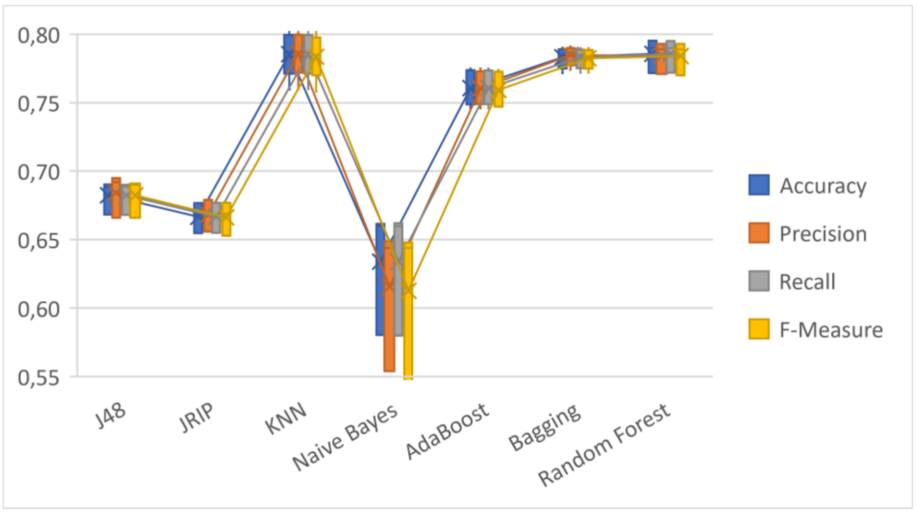


Figure 6. Average values of *accuracy, precision, recall* and *F-measure* for different classifiers

Algorithm	Attribute Selection	# Selected Attributes	Accuracy	Avg Precision	Avg Recall	Avg F measure	Tree dimension	Time to build the model
J48	CFSubsetEval + BestFirst (Backward)	11	68,87%	0,695	0,689	0,69	117 (59 leaves)	0.02
J48	InfoGain + Ranking (0.3)	21	69,04%	0,692	0,69	0,69	119 (69 leaves)	0.02
J48	WrappedC45 + BestFirst (backward)	19	66,84%	0,666	0,668	0,67	131 (66 leaves)	76.61
JRIP	CFSubsetEval + BestFirst (Backward)	11	67,68%	0,679	0,677	0,68	15 rules	0.09
JRIP	InfoGain + Ranking (0.3)	21	66,84%	0,67	0,668	0,67	17 rules	0.21
JRIP	WrappedC45 + BestFirst (Backward)	24	65,48%	0,656	0,655	0,65	19 rules	403.36
KNN (K=5)	CFSubsetEval + GreedyStepWise	11	75,97%	0,76	0,76	0,76	-	0.06
KNN (K=5)	CorrelationAttributeEval + Ranking(0.16)	21	80,71%	0,808	0,807	0,81	-	0.01
KNN (K=5)	InfoGain + Ranking (0.20)	22	79,70%	0,797	0,797	0,80	-	0.02
KNN (K=5)	none	26	78,85%	0,789	0,788	0,79	-	0.01
KNN (K=5)	WrappedC45 + BestFirst (Forward)	13	78,51%	0,785	0,785	0,78	-	46.83
KNN (K=5)	WrappedC45 + GreedyStepWise	5	77,50%	0,776	0,775	0,77	-	9.14
Naive Bayes	CFSubsetEval + BestFirst (Forward)	11	65,99%	0,644	0,66	0,64	-	0.02
Naive Bayes	InfoGain + Ranking	22	58,04%	0,554	0,58	0,55	-	0.01
Naive Bayes	WrappedC45 + greedyStepWise	5	66,16%	0,649	0,662	0,65	-	10.24
AdaBoost + J48	InfoGain + Ranking (0.3)	21	77,50%	0,775	0,775	0,77	-	0.41
AdaBoost + J48	none	26	74,96%	0,746	0,75	0,75	-	0.48
AdaBoost + JRIP	CFSubsetEval + BestFirst (Backward)	11	71,74%	0,727	0,717	0,72	-	1.39
AdaBoost + KNN (K=5)	CorrelationAttributeEval + Ranking(0.16)	21	77,33%	0,772	0,773	0,77	-	0.38
AdaBoost + KNN (K=5)	CFSubsetEval + BestFirst (Forward)	11	75,97%	0,76	0,76	0,76	-	0.7
AdaBoost + KNN (K=5)	CFSubsetEval + GreedyStepWise	11	75,97%	0,76	0,76	0,76	-	0.32
AdaBoost + KNN (K=5)	none	26	74,62%	0,75	0,746	0,75	-	0.67
Bagging + KNN (K=5)	CorrelationAttributeEval + Ranking(0.16)	21	79,02%	0,791	0,79	0,79	-	0.04
Bagging + KNN (K=5)	WrappedC45 + BestFirst	13	78,51%	0,787	0,785	0,78	-	43.54
Bagging + KNN (K=5)	InfoGain + Ranking (0.2)	22	78,51%	0,787	0,785	0,78	-	0.05
Bagging + KNN (K=5)	CFSubsetEval + BestFirst (Forward)	11	77,16%	0,774	0,772	0,77	-	0.03
Random Forest	none	26	79,53%	0,793	0,795	0,79	-	0.69
Random Forest	InfoGain + Ranking (0.3)	21	79,02%	0,789	0,79	0,79	-	0.24
Random Forest	CFSubsetEval + BestFirst (Forward)	11	77,16%	0,771	0,772	0,77	-	0.24

As can be seen from the results not all classifiers are able to achieve the same level of accuracy, even when trying various attribute selection algorithms.

In addition, attribute selection algorithms fail to reduce the dimensionality of the dataset by much, and often when they do, they also reduce the accuracy of the model.

Our explanation for this is that, since the task of classifying the music genre is complex, many of the features we extracted are important to characterize the different samples and assign them to their respective class.

At this early stage, the best models appeared to be *K-nearest neighbors* and the *ensemble methods*.

In order to determine if one model was better than the others with a statistical significance, we selected the models with highest accuracy and employed the *t*-test with pairwise comparison for each 10-fold cross-validation round.

The results of the test are shown in Figure 7, *IBk* was used as test base.

		Classifiers			
		CorrelationAttributeEval + Ranking IBk (k = 5)	RandomForest	InfoGainAttributeEval + Ranker AdaBoostM1 + J48	CfsSubsetEval + BestFirst Bagging + IBk (k = 5)
Comparison field	Accuracy	79.74	78.58	76.01 *	76.58
	F-measure	0.80	0.74 *	0.69 *	0.75
	Time for testing	0.00	0.00 *	0.00 *	0.02 v

Figure 7. t-test for best performing classifiers

The statistical test confirmed that *K-nearest neighbors* isn't better than the others as much in *accuracy*, but rather in the *F-measure* (our classes are not perfectly balanced). We therefore decided to adopt that as our classifier.

The statistical test also showed that the time required for testing is significantly less for algorithms such as Random Forest and J48. This result is not surprising since *K-nearest neighbors* is a lazy learner, but since the dataset used for training is not very large we decided it was not a problem.

The next thing we wanted to test was whether there was a significant difference in using different threshold values for the attribute selection algorithm.

The test assessed that there was no statistical difference in accuracy when using a threshold of 0.16 or 0.20, as shown in Figure 8.

Since using a threshold of 0.20 led to fewer selected attributes, we decided to adopt it. This way, the classification process will be faster.

		Threshold			
		0.16	0.2	0.25	0.3
Comparison field	Accuracy	79.74	78.21	72.77 *	51.72 *
	Attributes Selected	21.39	19.06 *	13.30 *	1.99 *

Figure 8. t-test comparing different thresholds

2.5 Conclusions

After the various analyses just described, we can say that we are satisfied with the results obtained by the *K-nearest neighbors algorithm* on the dataset.

In fact, we were able to exceed "human accuracy", although to do so we had to reduce the number of genres to six. However, we think that the six genres chosen are able to represent those that have always been the main musical universes.

Obviously, we think that there could still be room for improvement using more advanced technologies (e.g., Convolutional Neural Network), but we still think that most of the problems we had in getting high accuracy were due to the dataset.

In our opinion, in fact, the dataset is too small to be able in the training process to cover the various nuances contained in the genres and the risk is unfortunately that of overfitting.

In any case, using the application we developed, we were able to test the algorithm with new songs (not used in training or testing) obtaining good results.

What we noticed is that the classifier performs better with some specific genres and worse with others. For instance, metal is often misclassified into rock, and blues and jazz are sometimes interchanged.

However, these errors are partly understandable, since metal has its roots in rock and, similarly, jazz was born from the union of blues with other rhythms.

This again highlights how the task of classifying the musical genre of a song is far from easy and that automating this process requires advanced machine learning algorithms.

3 IMPLEMENTATION

3.1 MAIN MODULES

The implementation code is divided into two main modules: *FeatureExtractor* and *SoundHabit*.

- **FeatureExtractor**: is the module that contains the Python functions needed to extract audio features from songs. In fact, in the *Extractor.py* file the *extract_feature* function has been defined to extract these features from an audio signal. The server implementation (single process) is defined in the *FeatureExtractorServer.py* file, which waits for requests, receives the file to be processed and returns the extracted features to the client. In the *main.py* file there is also the function used to generate the dataset from the original one containing the audio files.
- **SoundHabit**: is the actual application, whose implementation will be analyzed in more details in the next section. The application was developed following the *MVC* (Model, View, Controller) pattern. The View displays the data contained in the model and was mainly developed using .fxml files, which allow you to write all the graphic components and their properties in separate files. The Model provides methods for accessing data useful for the application. The Controller receives the commands from the user and implements them by changing the status of the other two components. This division allowed us to completely separate the three components, for a more readable and maintainable code.

3.2 MAIN PACKAGES AND CLASSES

In this section the main packages of SoundHabit module and their respective classes will be presented.

3.2.1 it.unipi.dii.inginf.dmml.soundhabit.app

This package contains the main class, that starts the application.

Classes:

- **SoundHabit**: this class extends Application and implements the start method.

3.2.2 it.unipi.dii.inginf.dmml.soundhabit.classification

This package contains the classes needed for the classification process. These classes take advantage of the Weka library to build the music genre classifier.

Classes:

- **Classifier**: This class implements a K-NN classifier, performing all necessary pre-processing operations on the dataset. Through the *classify* function it is possible to classify a new instance. We used the singleton design pattern, to build the classifier only once.

- **SongFeature**: This class stores all the audio features inherent to a song. It is a Java Bean needed to store the information needed by the classifier.

- **FeatureExtractor**: this class acts as a client to the FeatureExtractorServer. So it takes care of sending a feature extraction request and then sends the file to the server. In order to send even large files, a packet splitting mechanism was used, first transmitting the size of the file to the server, in order to synchronize on the number of packets that will be transmitted.

3.2.3 it.unipi.dii.inginf.dmml.soundhabit.config

This package is used to handle the configuration parameters, stored in *config.xml*. The schema for the validation is in *config.xsd*. The validation is very important to be sure of the correctness of the *config.xml* file.

Classes:

- **ConfigurationParameters**: this class stores all the configuration parameters needed by the application. For example the IP for the Neo4j database. These values do not need to be changed, so only get methods are provided. We used the singleton design pattern for this class, to read the file only once.

3.2.4 it.unipi.dii.inginf.dmml.soundhabit.controller

This package contains the classes required for the controller part of the MVC pattern. For each different page to be shown to the user, a special controller has been implemented, which manages the events resulting from the actions taken by the user and updates the model and the view.

Classes:

- **AdminPageController**: this class manages the administration page, in which the administrator can insert new songs. From this page it is also possible to open the classifier page.

- **ClassifyPageController**: this class handles the page in which the administrator can do a classification request. The results will be shown with a chart, that summarizes the affinity for each class.

- **DiscoveryPageController**: this class manages the discovery section of the application. So it is in charge of showing the results of search made by the user.

- **SongController**: this class manages the single song and all the operations that can be done on it.

- **WelcomePageController**: this class manages the login/register page of the application.

3.2.5 package it.unipi.dii.inginf.dmml.soundhabit.model

This package contains the classes required for the model. These classes are the Java bean for our application.

Classes:

- **User**: This class stores all the information about a user, like the username, the password, and so on.
- **Song**: This class stores all the information related to one song, like the name, the author, a link through which you can listen to the song, and so on.
- **Genre**: This class is an Enumerated Type, a class that extends *java.lang.Enum*. It stores the type of music genre that are used inside the application.
- **Session**: This class is used to maintain the information of the session, like the logged user. We used the singleton design pattern for this class.

3.2.6 it.unipi.dii.inginf.dmml.soundhabit.persistence

This package deals with managing the persistency of data, in fact it contains the class used to interface with the database.

Classes:

- **Neo4jDriver**: this class is responsible for implementing all the queries that have to be run on Neo4j. We used the Singleton design pattern, because a single instance of this driver must be shared by all application classes.

3.2.7 it.unipi.dii.inginf.dmml.soundhabit.utils

This package contains a class used to store all the utility functions that we use in the application.

Classes:

- **Utils**: this class is used for containing some utility functions used inside the application (to avoid code replication).