UNIVERSITÀ DI PISA

# Data Mining and Machine Learning Project VoiceIDNotes

Federico Minniti

Matteo Del Seppia

January 2022

# Contents

# 1 Design

## 1.1 The application

VoiceIDNotes is a simple note-taking application focused on keeping users' notes private. The users can access their notes only after performing the login, either letting the app use a model to identify their voice or using their username and password.
The key idea is that everyone has their own particular and personal voice. VoiceIDNotes allows users to have an easy login phase because once recognized their voices a simple 4-digit pin will be required
Once logged in, users can create a new note, modify or delete already existing notes or research among their notes through some research parameters such as the title or the date of creation.
Lastly, users can modify their pin or access password through a personal page

## 1.2 Requirements

### 1.2.1 Main actors

VoiceIDNotes is meant to be used by two different types of actors:

- *Anonymous Users*, who are only allowed to register or login (if they already have an account)
- *Standard Users*, who can use the application to search, write and edit their private notes

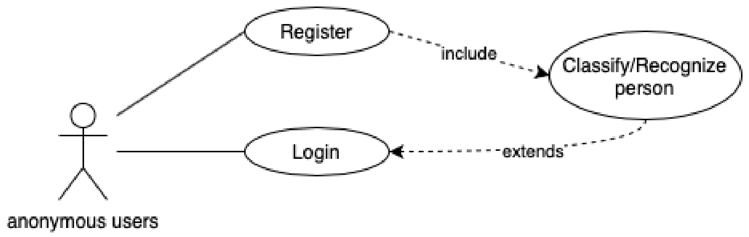
### 1.2.2 Functional requirements

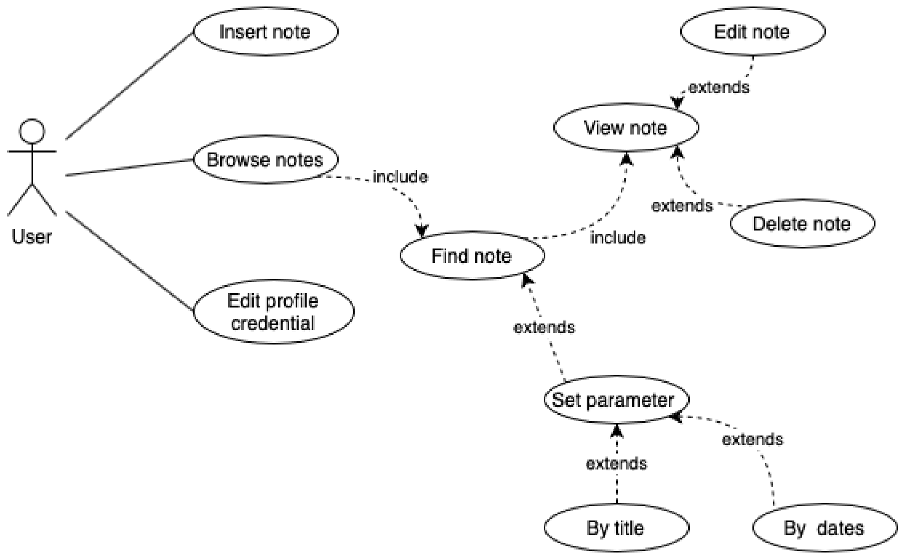
- VoiceIDNotes must provide a way for *Anonymous Users* to register and become *Standard Users*
- VoiceIDNotes must deny to the *Anonymous Users* the access to the app and to all functionalities designed for the *Standard Users*
- VoiceIDNotes must provide a way to perform the login with credentials
- VoiceIDNotes must offer a way to perform an easy login through a voice identification process and a PIN
- *Standard Users* must be given the opportunity to browse their notes, optionally using filters (*title*, *creation date*)
- *Standard Users* must be given the opportunity to save/modify/delete notes
- *Standard Users* can modify their pin or access password

### 1.2.3 Non-functional requirements

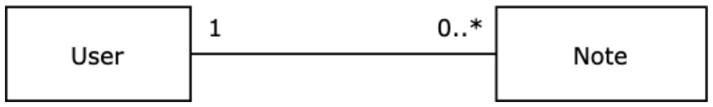
- VoiceIDNotes must embed at least one machine learning algorithm
- VoiceIDNotes should be intuitive and easy to use

## 1.3 Use cases





## 1.4 Analysis classes



- Each user can have zero or more privately saved notes
- Each note is written by a single user

- Users are characterized by a username, a password and a PIN
- Notes are characterized by a title, a text and a creation timestamp

## 1.5 Data Model

Given the simple storage requirements of our application, we decided to use a key-value database to store the users' account information and private notes
In order to keep information about the user and to have a fast delete/update/insert operations, we have structured the following key:

*user:*$\langle$*username*$\rangle$*:*$\langle$*attribute*$\rangle$

The key for the notes has been designed to keep track of the properties of a note:

*note:* $\langle$*username*$\rangle$*:* $\langle formatted\_timestamp \rangle$ *:* $\langle$*attribute*$\rangle$

*formatted_timestamp* is the creation timestamp of the note formatted into a string with the pattern *yyMMddHHmmss*, where

- *yy* are the the last two digits of the year
- *MM* is the numeric month
- *dd* is the number of the day in the month
- *HH* is the hour
- *mm* are the minutes
- *ss* are the seconds

## 1.6 System Architecture

The architectural design of the system is client-server. Clients will request to extract the features of their voice to the server.

### 1.6.1 Client

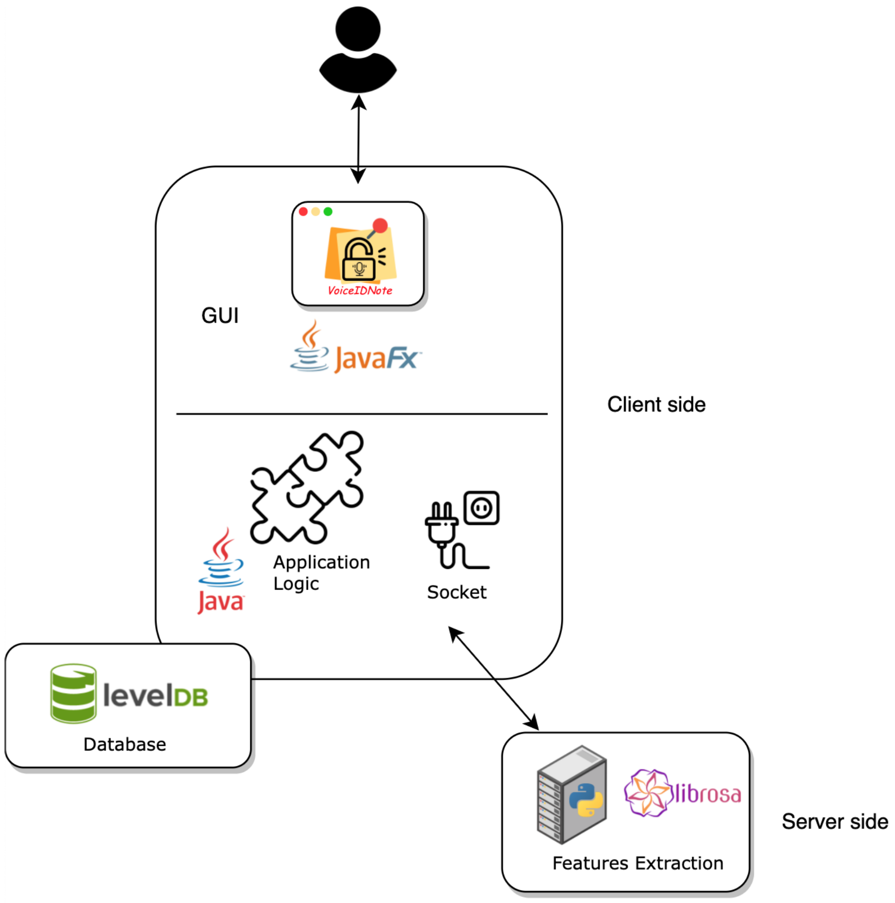
On the client side, the Presentation Layer (GUI) and the Logic Layer are running. The Logic Layer has to process requests coming from the Presentation Layer and forwarding them to the server, with a communication protocol, or to the local key-value database.

### 1.6.2 Server

The server side consists of a single node, who can also run on the same machine of the client, that will receive and save temporarily audio files to extract and send back to the clients the voice features.

### 1.6.3 Frameworks

VoiceIDNotes code will be written in Java, with JavaFX for the GUI.
LevelDB will be used as database.
The server that will extract features from files containing voice audio will be written in Python with a particular library for feature extraction of audio file: Librosa. (we also use Pandas and Numpy).
Maven will be used as a tool for building and managing the whole application.
Regarding the version control Git (in particular, GitHub) will be used.
The source code for the application can be found on GitHub at:
`https://github.com/federicominniti/VoiceIDNotes`.
It is a repository divided in two parts: one for the feature extraction server and the other for the client application.



# 2 Machine Learning

## 2.1 Introduction

The machine learning problem that we've tried to solve for the development of this application is known in literature as text-independent voice identification. The goal is building a classifier that recognizes the voice of a user based on a small audio sample of their voice, independently of whatever the user is saying. The approach we have chosen to use is obviously only feasible for applications with a very limited number of users like VoiceIDNotes.
The objective of VoiceIDNotes is, in fact, to implement a simple system of text-independent voice recognition to allow the users to perform a fast login to the application.

## 2.2 Dataset

The dataset we have used is part of the LibriSpeech dataset, used in various papers to solve the problem of both text-dependent and text-independent voice identification and also speech recognition. LibriSpeech is a corpus of approximately 1000 hours of 16kHz read English speech, derived from read audiobooks from the LibriVox project. The recordings of the audiobooks are split audio files of approximately the same length in seconds (6-10s). After reading some papers about the problem of text-independent voice identification, we have decided to take a subset of the LibriSpeech dataset consisting of 4000 audio samples by 40 different speakers (100 audios per speaker, 20 men and 20 women). Link: `https://www.openslr.org/12/`

## 2.3 Pre-processing

### 2.3.1 Feature extraction

Sounds are composed of different frequencies, amplitude, wavelength, and other related features. Our goal is to extract these characteristics of sounds to build a machine learning classifier. In order to determine the sound features relevant to speech processing for speaker recognition, we first need to understand the different type of audio features. Audio features can be broadly classified into three types:

- **Rhythmic features**, which are mainly related to musical notes and are widely used in applications of music information retrieval
- **Temporal features**, describing an audio signal over a sampled period of time and used to understand the continuity of an audio signal (for example, detecting sudden changes)
- **Spectral features**, which are considered to be the most effective in speech and voice recognition because they describe audio signals very similarly to how the human ear perceives sounds

Studies suggest that MFCCs represent the closest relation to the human earing model. MFCCs are a small set of features which concisely describe the overall shape of the spectrum of the amplitude of the signal. The number of MFCCs to use depends on the sampling rate of the audios:

$$n = \frac{sampling\_rate + \frac{sampling\_rate}{2}}{2}.$$

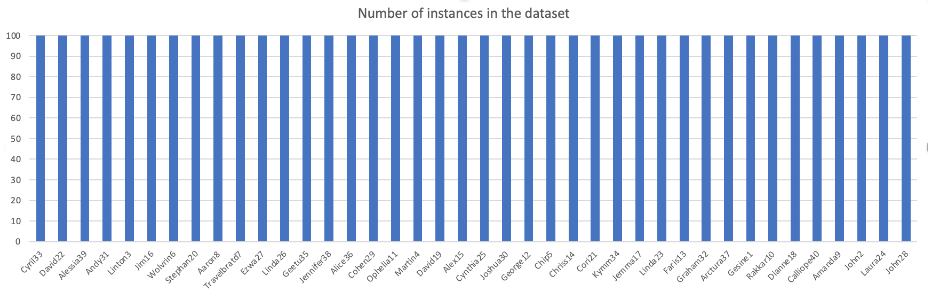
With a 16KHz sampling rate we have to use 13 MFCCs (12 + 1 for the energy). Along with the 13 MFCCs, we decided to extract also their variations, known as Delta MFCCs. Similarly, we can extract also the variation of the Delta coefficients, which are called MFCC Delta-Delta coefficients, sometimes also referred to as accelerations.
For the MFCCs, Deltas and Delta-Deltas we've decided to take the mean of the arrays of values, in order to get a single attribute for each the 39 features (13 MFCCs + 13 Deltas + 13 Delta-Deltas). At the end of the feature extraction step, each audio is transformed into a tuple containing 39 audio features and a label.

### 2.3.2 Data transformation

We've used the z-score normalization in order to give the same weight to all the features.
The z-score is the signed distance each sample value is from the mean in standard deviations:

$$z = \frac{x-\mu}{\sigma}$$

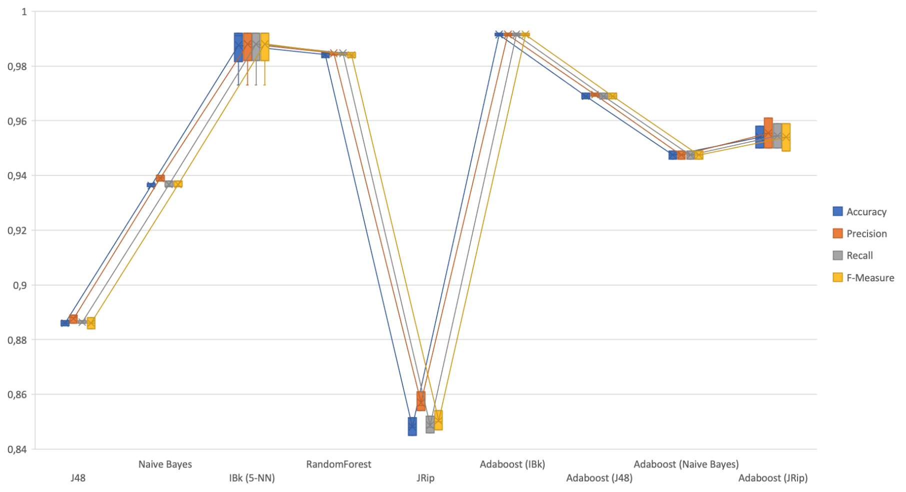
where $x, \mu, \sigma$ are respectively a sample value, the mean and the standard deviation of the distribution of values of a certain attribute.
We haven't chosen min-max discretization because there is a problem in handling outliers due to the fact that all the other points are compressed in a small area.
The classes are already balanced, with 100 samples for each label.



## 2.4 Models evaluation and selection

In order to build the best model possible, we've tested different classifiers and evaluated their results on our dataset. Our goal was to find the classifier with both the highest accuracy and the lowest time required to build the model. 10-fold cross-validation has been used to test the classifiers.

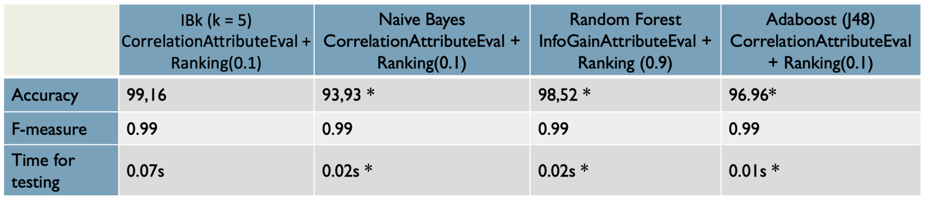
| Algorithm | Attribute Selection | # Selected Attributes | Accuracy | Avg Precision | Avg Recall | Avg F measure | Tree dimension | Time to build model |
|---|---|---|---|---|---|---|---|---|
| J48 | null | 1-39 (tot. 39) | 88,7 | 0,889 | 0,887 | 0,888 | 397 | 0.22s |
| J48 | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot. 14) | 88,5 | 0,888 | 0,886 | 0,886 | 407 | 0.27s |
| J48 | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot. 13) | 88,6 | 0,886 | 0,886 | 0,884 | 409 | 0.12s |
| Naive Bayes | InfoGainAttributeEval + Ranking (0.9) | 1-13 (tot. 13) | 93,6 | 0,938 | 0,936 | 0,936 |  | 0.11s |
| Naive Bayes | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot. 13) | 93,7 | 0,94 | 0,938 | 0,938 |  | 0.03s |
| Naive Bayes | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot. 14) | 93,6 | 0,939 | 0,936 | 0,936 |  | 0.16s |
| IBk (5-NN) | null | 1-39 (tot. 39) | 97,3 | 0,973 | 0,973 | 0,973 |  | 0.01s |
| IBk (5-NN) | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot. 14) | 99,2 | 0,992 | 0,992 | 0,992 |  | 0.14s |
| IBk (5-NN) | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot. 13) | 99,1 | 0,992 | 0,992 | 0,992 |  | 0.03s |
| IBk (5-NN) | InfoGainAttributeEval + Ranking (0.9) | 1-13 (tot. 13) | 99 | 0,991 | 0,991 | 0,991 |  | 0.1s |
| IBk (5-NN) | CFSubsetEval + GreedyStepWise | 1-13, 27 (tot. 14) | 99,2 | 0,992 | 0,992 | 0,992 |  | 0.14s |
| RandomForest | null | 1-39 (tot. 39) | 98,5 | 0,985 | 0,985 | 0,985 |  | 3.10s |
| RandomForest | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot. 14) | 98,3 | 0,984 | 0,984 | 0,983 |  | 2.21s |
| RandomForest | InfoGainAttributeEval + Ranking (0.9) | 1-13 (tot .13) | 98,45 | 0,985 | 0,985 | 0,984 |  | 2.09s |
| JRip | CFSubsetEval + GreedyStepWise | 1-13, 27 (tot. 14) | 84,8 | 0,854 | 0,848 | 0,85 |  | 1.45s |
| JRip | InfoGainAttributeEval + Ranking (0.9) | 1-13 (tot. 13) | 85,15 | 0,861 | 0,852 | 0,854 |  | 1.56s |
| JRip | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot. 13) | 84,5 | 0,856 | 0,846 | 0,847 |  | 1.36s |
| Adaboost (IBk) | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot. 13) | 99,1 | 0,991 | 0,991 | 0,991 |  | 9.41s |
| Adaboost (IBk) | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot.14) | 99,2 | 0,992 | 0,992 | 0,992 |  | 9.81s |
| Adaboost (J48) | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot.13) | 97 | 0,97 | 0,97 | 0,97 |  | 1.91s |
| Adaboost (J48) | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot.14) | 96,8 | 0,969 | 0,968 | 0,968 |  | 2.13s |
| Adaboost (Naive Bayes) | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot. 13) | 94,6 | 0,946 | 0,946 | 0,946 |  | 2.19s |
| Adaboost (Naive Bayes) | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot. 14) | 94,9 | 0,949 | 0,949 | 0,949 |  | 2.95s |
| Adaboost (JRip) | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot.14) | 95,8 | 0,961 | 0,959 | 0,959 |  | 11.64s |
| Adaboost (JRip) | InfoGainAttributeEval + Ranking (0.9) | 1-13 (tot.13) | 95 | 0,95 | 0,95 | 0,949 |  | 11.5s |



As we can see in the graph the three best algorithms for our dataset are: IBk, RandomForest and Adaboost(IBk). On the other hand, the worst algorithm proved to be JRip. Obviously, we must also consider the time to build the model, since Adaboost(IBk) took more or less 10 seconds, while RandomForest within two and three seconds. These times are very high for our use compared to the 0.1 seconds of IBk.

| Algorithm | Attribute Selection | # Selected Attributes | Accuracy | Avg Precision | Avg Recall | Avg F measure | Tree dimension | Time to build model |
|---|---|---|---|---|---|---|---|---|
| J48 | null | 1-39 (tot. 39) | 88,7 | 0,889 | 0,887 | 0,888 | 397 | 0.22s |
| J48 | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot. 14) | 88,5 | 0,888 | 0,886 | 0,886 | 407 | 0.27s |
| J48 | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot. 13) | 88,6 | 0,886 | 0,886 | 0,884 | 409 | 0.12s |
| Naive Bayes | InfoGainAttributeEval + Ranking (0.9) | 1-13 (tot. 13) | 93,6 | 0,938 | 0,936 | 0,936 | | 0.11s |
| Naive Bayes | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot. 13) | 93,7 | 0,94 | 0,938 | 0,938 | | 0.03s |
| Naive Bayes | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot. 14) | 93,6 | 0,939 | 0,936 | 0,936 | | 0.16s |
| IBk (5-NN) | null | 1-39 (tot. 39) | 97,3 | 0,973 | 0,973 | 0,973 | | 0.01s |
| IBk (5-NN) | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot. 14) | 99,2 | 0,992 | 0,992 | 0,992 | | 0.14s |
| IBk (5-NN) | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot. 13) | 99,1 | 0,992 | 0,992 | 0,992 | | 0.03s |
| IBk (5-NN) | InfoGainAttributeEval + Ranking (0.9) | 1-13 (tot. 13) | 99 | 0,991 | 0,991 | 0,991 | | 0.1s |
| IBk (5-NN) | CFSubsetEval + GreedyStepWise | 1-13, 27 (tot. 14) | 99,2 | 0,992 | 0,992 | 0,992 | | 0.14s |
| RandomForest | null | 1-39 (tot. 39) | 98,5 | 0,985 | 0,985 | 0,985 | | 3.10s |
| RandomForest | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot. 14) | 98,3 | 0,984 | 0,984 | 0,983 | | 2.21s |
| RandomForest | InfoGainAttributeEval + Ranking (0.9) | 1-13 (tot .13) | 98,45 | 0,985 | 0,985 | 0,984 | | 2.09s |
| JRip | CFSubsetEval + GreedyStepWise | 1-13, 27 (tot. 14) | 84,8 | 0,854 | 0,848 | 0,85 | | 1.45s |
| JRip | InfoGainAttributeEval + Ranking (0.9) | 1-13 (tot. 13) | 85,15 | 0,861 | 0,852 | 0,854 | | 1.56s |
| JRip | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot. 13) | 84,5 | 0,856 | 0,846 | 0,847 | | 1.36s |
| Adaboost (IBk) | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot. 13) | 99,1 | 0,991 | 0,991 | 0,991 | | 9.41s |
| Adaboost (IBk) | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot.14) | 99,2 | 0,992 | 0,992 | 0,992 | | 9.81s |
| Adaboost (J48) | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot.13) | 97 | 0,97 | 0,97 | 0,97 | | 1.91s |
| Adaboost (J48) | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot.14) | 96,8 | 0,969 | 0,968 | 0,968 | | 2.13s |
| Adaboost (Naive Bayes) | CorrelationAttributeEval + Ranking(0.1) | 1-13 (tot. 13) | 94,6 | 0,946 | 0,946 | 0,946 | | 2.19s |
| Adaboost (Naive Bayes) | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot. 14) | 94,9 | 0,949 | 0,949 | 0,949 | | 2.95s |
| Adaboost (JRip) | CFSubsetEval + BestFirst (Backward) | 1-13, 27 (tot.14) | 95,8 | 0,961 | 0,959 | 0,959 | | 11.64s |
| Adaboost (JRip) | InfoGainAttributeEval + Ranking (0.9) | 1-13 (tot.13) | 95 | 0,95 | 0,95 | 0,949 | | 11.5s |

For the determination of the best model suitable for our application, we employed a t-test with pairwise comparison for each 10-fold cross validation round on the classifiers who proved to be more accurate and fast.



| | IBk (k = 5) CorrelationAttributeEval + Ranking(0.1) | Naive Bayes CorrelationAttributeEval + Ranking(0.1) | Random Forest InfoGainAttributeEval + Ranking (0.9) | Adaboost (J48) CorrelationAttributeEval + Ranking(0.1) |
|---|---|---|---|---|
| Accuracy | 99,16 | 93,93 * | 98,52 * | 96.96* |
| F-measure | 0.99 | 0.99 | 0.99 | 0.99 |
| Time for testing | 0.07s | 0.02s * | 0.02s * | 0.01s * |

The t-test confirmed that K-NN with K=5 is better than the others in terms of accuracy, but it's a little worse in terms of testing time, which is really not surprising. Anyway, we think that such a testing time is still good, so we've decided to implement the application with K-NN.

## 2.5 Conclusions

### 2.5.1 General considerations

As we can see from results we obtained, after applying attribute selection the accuracy remains more o less equal, except for IBk that is slightly increased. We can see some differences in times to build models that decreased thanks to the maintenance of only MFCC coefficients and in some cases of the 27th delta-delta coefficient.
After the various analyses just described, we can say that we are satisfied with the results obtained by K-NN with K=5. In fact, we were able to achieve a very high accuracy combined with a very low time required to build the model. We also think that these high results in terms of accuracy of the models are due to the optimal choice of the dataset, which contained fairly clean audios of people talking without noise caused by external sounds like cars passing in the background, city sounds and other things.

### 2.5.2 Adjustments needed for the implementation

Obviously we couldn't ask the users to register 100 audios of 10 seconds at the moment of signing up, because this would have taken at least 17 minutes.
We've tried to solve this problem by letting new users to register only 10 audios of their voice and then apply SMOTE to their class to create 90 more synthetic samples to be used for the classification. We expect that the accuracy in recognizing new users will be low on average and not comparable with the others estimated in the analysis; so we decided to update the dataset by replacing old samples with new samples of the users' voice recorded during successful or failed attempts to login with the voice recognition system. This will increase the accuracy of identifying new users over time, hopefully reaching at one point the level of accuracy measured in the model evaluation step. Doing so

we also guarantee that changes of the users' voice over time will be taken in consideration by the classification system.

# 3 Implementation

## 3.1 Main Modules

- **VoiceIDNotesFeatureExtractor**: contains the Python scripts needed to extract audio features from songs and the implementation of the server. The server implementation is single process and is defined in the Server.py file. In the main.py file there is also the function used to generate the dataset from the original audio files.
- **VoiceIDNotesApp**: is the actual application, developed following the MVC (Model, View, Controller) pattern. That guarantees to have a good separation and maintainability of the code.

## 3.2 Main packages and classes

### 3.2.1 it.unipi.dii.inginf.dmml.voiceidnotesapp.app

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

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

### 3.2.2 it.unipi.dii.inginf.dmml.voiceidnotesapp.classification

This package contains the classes needed to handle the classification process. The Classifier class takes advantage of the Weka library. The package also contains classes to structure voice features and to perform the communication with the feature extraction server.
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.
- **VoiceFeature**: this class stores all the audio features of a certain voice sample.
- **FeatureExtractor**: this class acts as a client to the Python server. In order to send even large files, a packet is split in more than one part. Sequence of packets transmitted to the server: command, size of the file, chunks of the file

### 3.2.3 it.unipi.dii.inginf.dmml.voiceidnotesapp.config

This package is used to handle the configuration parameters, extracted in config.xml and validated with the config.xsd schema.
Classes:

- **Config**: this class stores all the configuration parameters of the application (feature extraction server IP and Port, dataset location). Users can edit parameters that are stored in config.xml file. Validation process of config.xml file with config.xsd file and the retrieving parameters process, are performed in this class.

### 3.2.4 it.unipi.dii.inginf.dmml.voiceidnotesapp.controller

This package contains controllers to handle the interaction of the user with the GUI that is created with .fxml files stored in ./src/main/resources/fxml
Classes:

- **LoginPageController**: this class manages the login page of the application where users can record their voice to be identified
- **RegisterPageController**: this class manages the registration page of the application, where users must record 10 audio samples and insert their information for the sign up.
- **MyNotesController**: this class manages the note-taking part of the application
- **ProfileController**: this class manages the editing of the logged user credentials

### 3.2.5 it.unipi.dii.inginf.dmml.voiceidnotesapp.model

This package contains the classes required to structure and handle the back-end model.
Classes:

- **User**: this class maintains the user's information
- **Session**: this class is used to maintain the logged user session
- **Note**: this class is used to maintain information about a single note

### 3.2.6 it.unipi.dii.inginf.dmml.voiceidnotesapp.persistence

This package deals with the storage of the data querying LevelDB.
Classes:

- **LevelDBDriver**: this class implements all the queries that have to be run on the local LevelDB instance, to maintain information about notes and users.

Possible future implementation: create a remote database server to make available all stored data about registered users from any client

### 3.2.7 it.unipi.dii.inginf.dmml.voiceidnotesapp.utils

This package contains utility classes for the application, with functions and constants used in more classes of the project to perform general tasks. Classes:

- **VoiceRecorder**: this class manages the recording of audio samples during the login and register phase.
- **CSVManager**: this class performs operations on the CSV dataset of registered users
- **Utils**: this class is useful for have always available generally utility functions