University of Pisa

SCUOLA DI INGEGNERIA

Corso di Laurea in Artificial Intelligence and Data Engineering

Data mining project documentation

Candidati

Giacomo Mantovani

Stefano Poleggi

Relatori

Prof. Francesco Marcelloni

Prof. Pietro Ducange

Anno Accademico 2019–2020

Contents

1 Introduction

The **TweetQuake** application offers a real-time serious earthquake detection service in Italy. When the application opens up it will start fetching tweets and analyzing them to recognize ones related to an earthquake. The application will show the number of earthquake tweets recognized so far. When some earthquake tweets are recognized the application show a warning message, if there is a trend of earthquake tweets then the application will show an emergency message. If the button at the end of the page is clicked then a new window containing the last recognized earthquakes opens up.

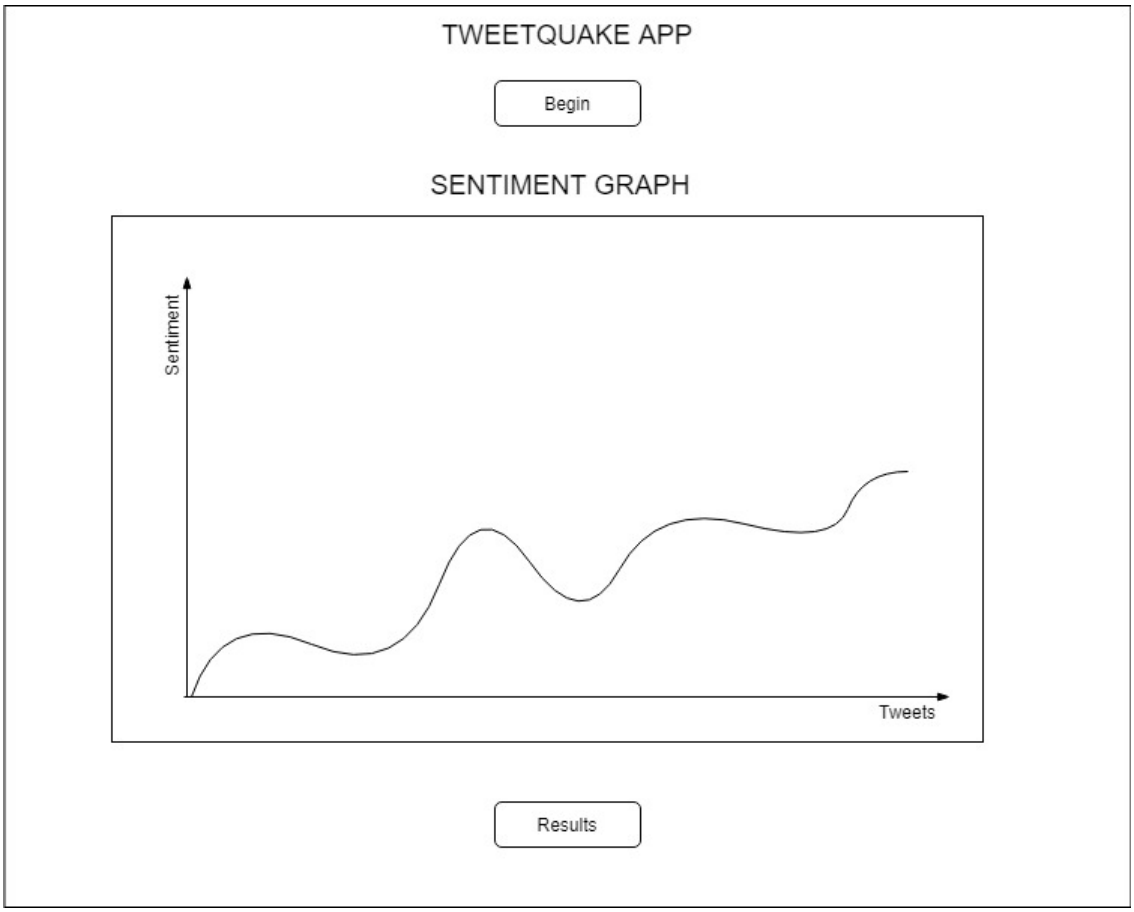

Figure 1: Home Page Mockup

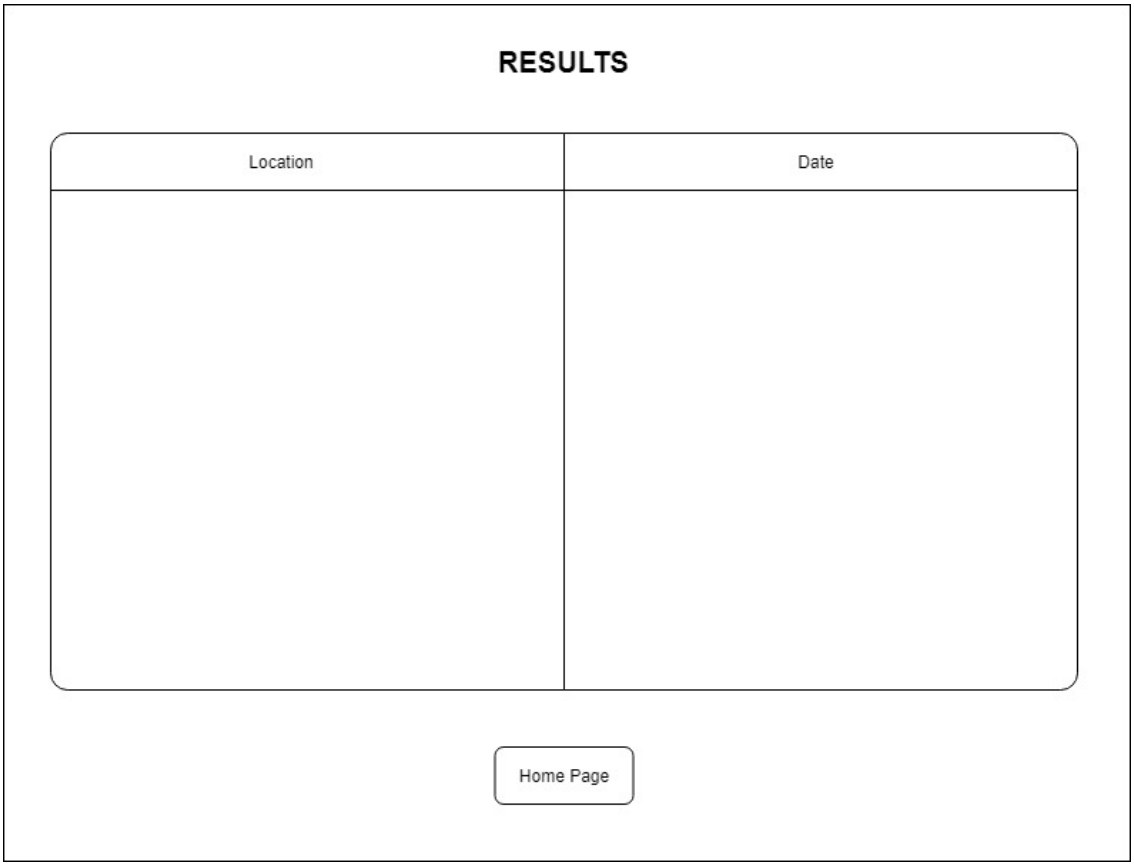

Figure 2: Results Page Mockup

2 Analysis and workflow

2.1 Requirements

2.1.1 Functional requirement

The system has to allow the user to carry out basic functions such as:

- To start the real-time fetching.
- To stop the real-time fetching.
- To view the most recent recognized earthquakes from the application.

The system has to iteratively perform the following operations:

- Real-time fetching of tweets.
- Perform a classification of the tweets and obtain the label (earthquake or non-earthquake).
- When some earthquake tweet is regocnized show a warning message.
- When an earthquake tweet trend is recognized show an emergency message and add the relative earthquake information into the database.

2.1.2 Non-functional requirements

- Usability, ease of use and intuitiveness of the application by the user.
- The system should provide a high level of accuracy.

2.2 Use Cases

Actors

- User: this actor represents a user of the application
- System: this actor represent the system
- Twitter: this actor represent the Twitter service

2.2.1 Use Cases Description

- Twitt Search : This use case can be performed by the user to start the real-time tweets fetching.
- Retrive Tweet: This use case represents the action of getting a tweet from twitter.
- Twitter Connection: This use case represents the connection with the Twitter service.
- HTTP Request for Connection: This use case represents the Request for the connection to the Twitter Service.
- HTTP Response for Connection: This use case represents the Response for the connection from the Twitter Service.
- HTTP Request for Tweets: This use case represents the Request for tweets to the Twitter service.
- HTTP Response for tweets: This use case represents the Response for tweets from the Twitter service.
- Perform Text Analysis: This use case represents the process of Text analysis performed by the system.
- Apply Classification Algorithm: This use case represents the classification of the tweets performed with the previously generated classifier by the "Generate Classifier" use case.
- Show Warning Message: This use case displays an warning message if a few earthquake tweet are recognized.
- Show Emergency Message: This use case displays an emergency message if a some earthquake tweet are recognized.
- Add Earthquake To Database: This use case represent the insert of the recognized earthquake into the database.
- Display Results: This use case shows the results of the classification.
- Browse earthquakes: The system browses the earthquakes on the database.
- Find earthquake: The system selects an earthquake on the database.
- Display Earthquake: This use case shows the earthquake on the application.
- Generate Classifier: This use case is performed automatically by the system and it generates a classifier using a training set.
- Browse Tweets: The system browses the tweets on the trainig dataset.
- Train the classifier: Perform an algorithm to generate the classifier.

- Stop Real-Time fetching: This use case allows the user to stop the real-time tweets fetching.

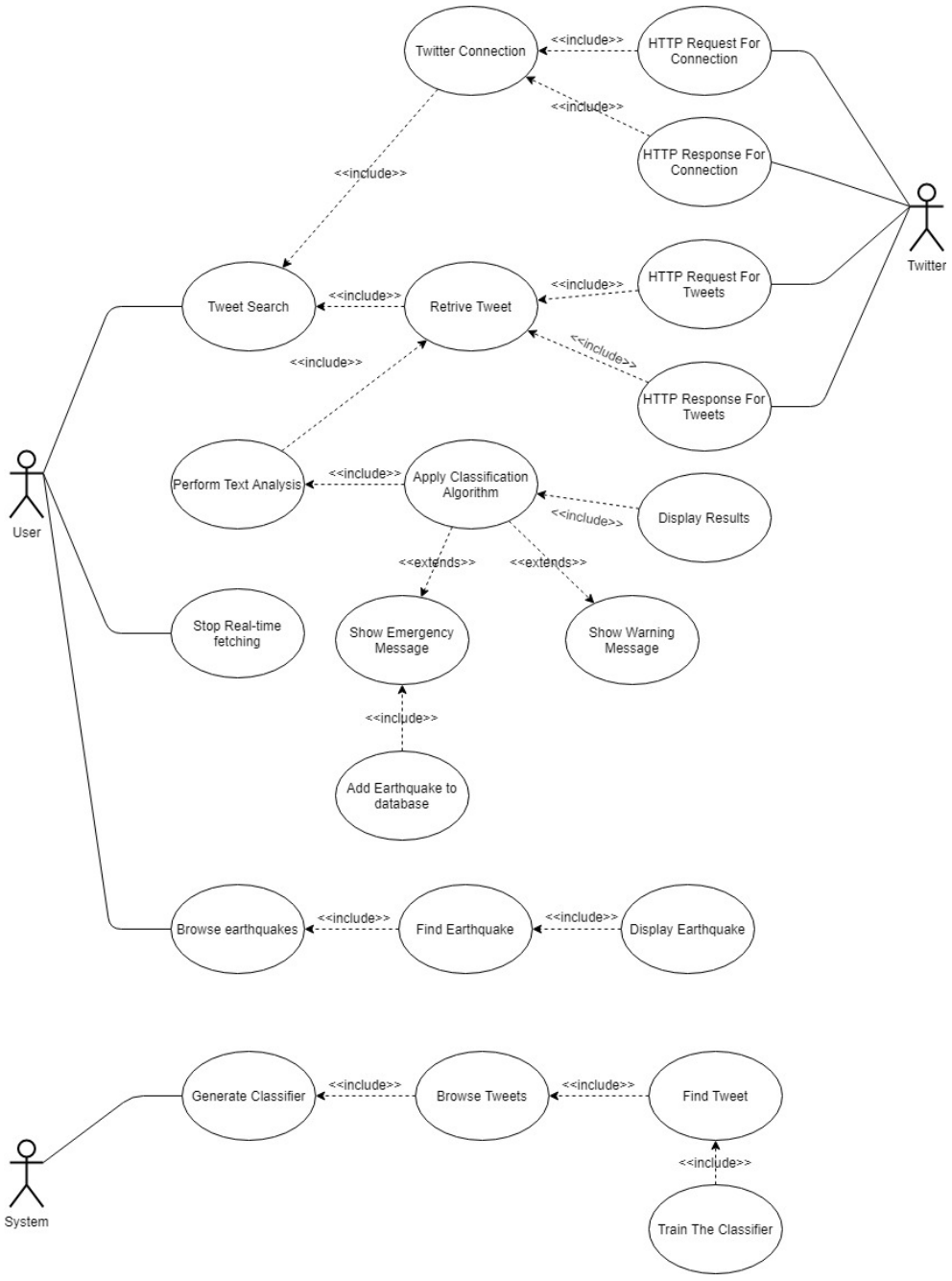

Figure 3: Use cases diagram

2.3 System Flow Diagram

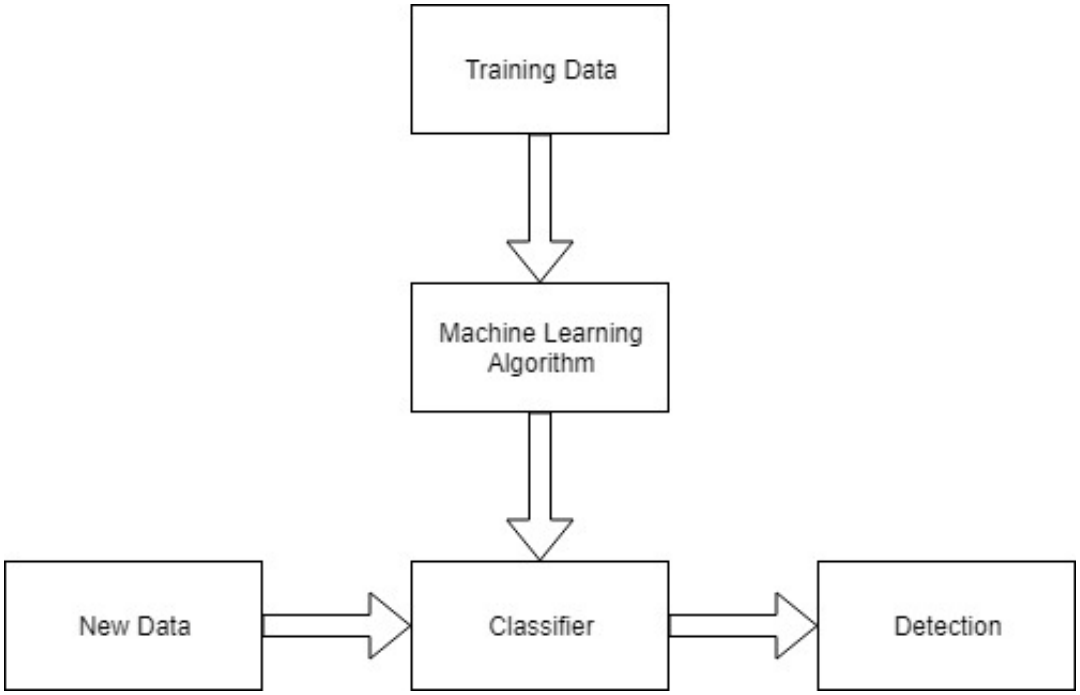

Figure 4: Flow Diagram

2.4 Text analysis process

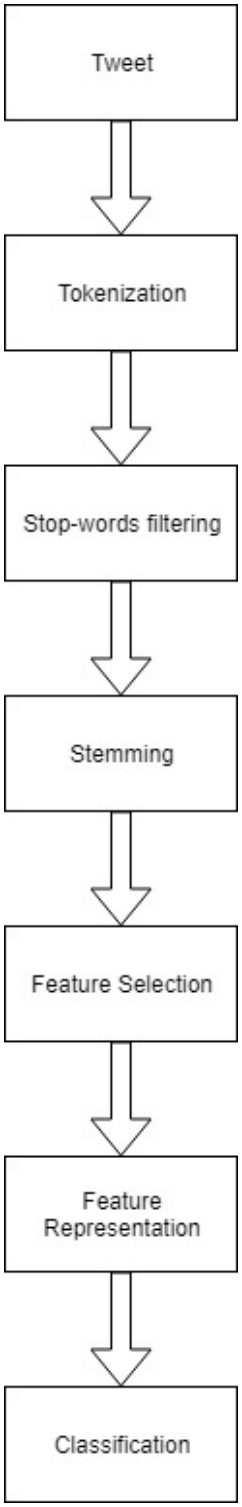

Figure 5: Text Analysis Process

2.5 Data analysis

2.5.1 Building the classifier

In order to build our classifier raw tweets have been scraped using a twitter scraping tool called Twint written in Python. Given location, tags to search for and a timestamp (or a date interval), returns the informations about the tweets matching those parameters. For more information visit: Twint Github Page. To build the training set two different queries have been used:

- The first query selects the tweets in Italy matching the keyword "terremoto".
- The second query selects the tweets in Italy without tags.

After that the labels (earthquake, non-earthquake) have been added manually to obtain a good training (and test) set to generate a classifier. We ended up with a balanced dataset of about 300 tweets per class, 600 labeled tweets.
The training data will be stored in an arff file, after cleaning the text of the tweet. The second query is necessary to obtain better results because without the tweets obtained from it the classifier would suffer from overfitting.
Once the training data is ready a classification algorithm is applied in order to build the classifier.

2.5.2 2016-2017 earthquakes database

Another database was created by collecting (in the same way) tweets between the dates 2017-06-30 and 2016-06-01. With that database we managed to create an histogram to check if there are spikes of tweets corresponding to catastrofic earthquakes in order to perform an analysis. The total number of analyised tweets is 304307.

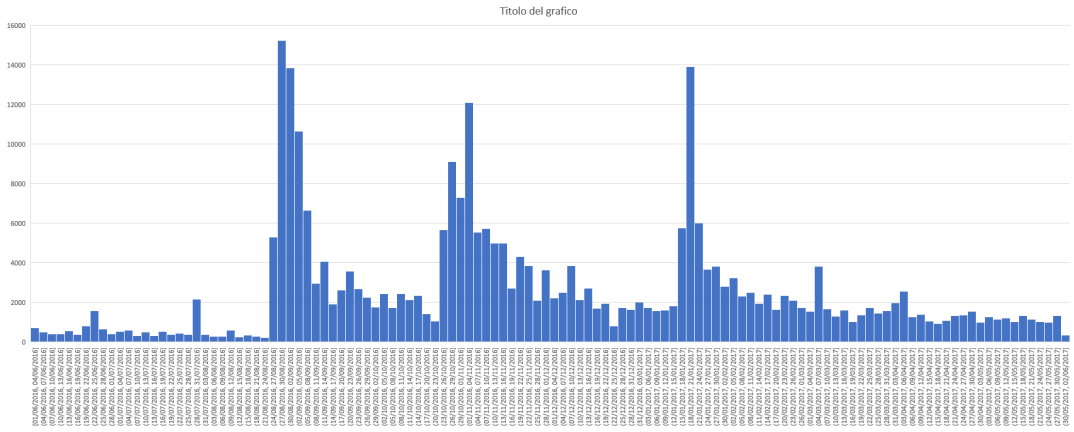

Figure 6: 2016-2017 tweets

As we can see there are 3 major spikes relative to the 24-08-2016, 30-10-2016 and 18-01-2017 huge earthquakes. We created the same histogram after performing a classification on the tweets using the classifier obtained from the training set.

2.5.3 Application parameters

Two parameters are used in this application:

- Number of tweets classified as "earthquake" to detect before sending a warning message (Warning threshold).
- Number of tweets classified as "earthquake" to detect before sending an emergency message (Emergency threshold).

We tested how many earthquake tweets are detected in a 10 minutes time span of a real serious earthquake, a non serius earthquake and a random time span without any earthquake. After that, the threshold values have been setted based on the values obtained from the tests.

2.6 Model analysis

2.6.1 Choosing the classifier

The choice of the classifier has been made by evaluating some quality measures of different classifiers applying a 10-fold cross-validation to find the best one for our application. We performed a cost sensitive classification schema, because we wanted to give an higher misclassification error cost for the "earthquake" class rather than "non earthquake" class.

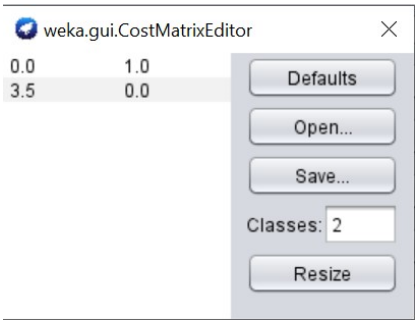

Figure 7: Cost matrix.

The results obtained from the different classifiers is show in the figure below.

Classifier	Accuracy (%)
SMO	90.62%
J48	88.16%
1NN	89.97%
2NN	88.16%
3NN	85.70%
NB	84.05%

Figure 8: confusion matrix

We ended up choosing the SMO classifier since it has the higher accuracy and the number of false negative is fairly low, which is good in our case since we don't want to classify non earthquake tweets as earthquake, but it is acceptable to have a slightly higher number of false positive (this is the reason for applying the cost sensitive classifier to our model). The confusion matrix obtained from the classification is the following:

```
=== Confusion Matrix ===

   a   b   <-- classified as
 253  51 |   a = earthquake
   6 290 |   b = non-earthquake
```

Figure 9: classifiers accuracy

3 Design

3.1 Software architecture

The application is designed over 3 different layers, see figure 10:

- Front-end
- Middleware
- Back-end

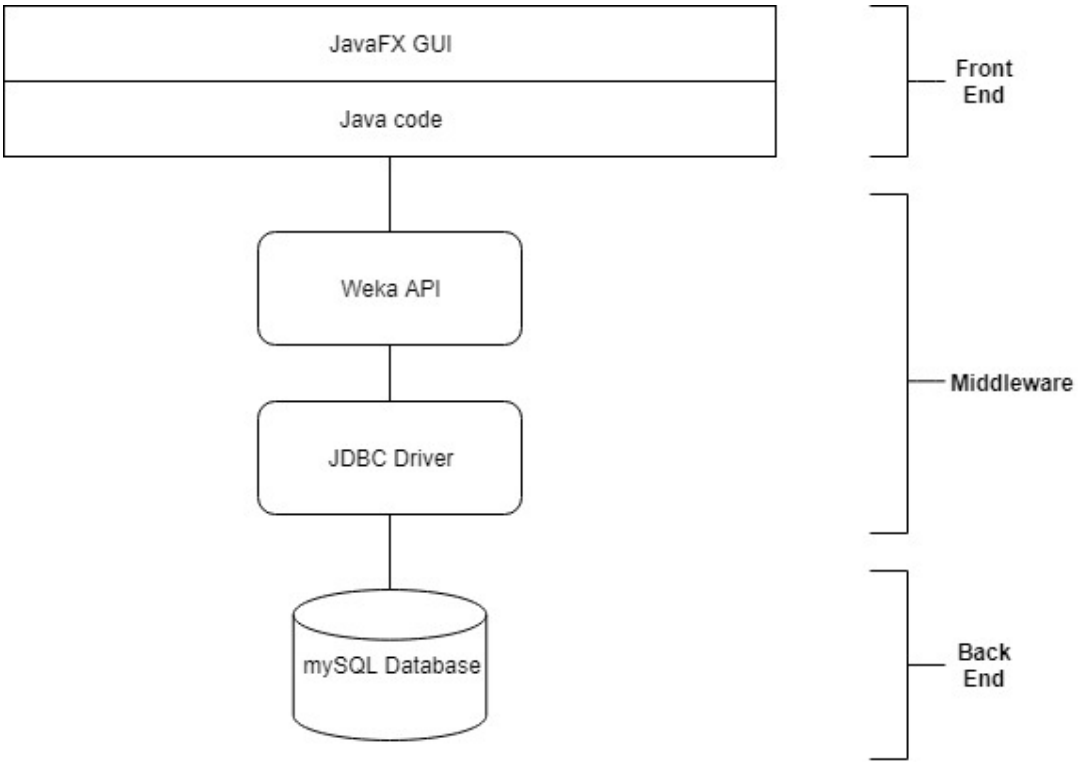

Figure 10: Software architecture diagram

4 Implementation

4.1 Used technologies

The application is developed in java programming language, version 11.0.4, and in JavaFX system to create the GUI, version 11, so it should run on each platform in which JVM is installed, but the application is tested and guardantee on Ubuntu 16 and Window OS. Moreover Maven is used to build and mantain the project, version 3.8.0.
The java driver for mongo manage the comunication between client application layer and mongo backend layer, version 3.11.2.
For the background layer it is used Apache as web-server, version 2.4 (including MySql).
So this application is tested using these technologies, considering these particular versions: for other versions the correct execution isn't guaranteed.

4.2 Building the classifier

The following java-code shows the training part of the process to obtain our classifier using the SVM classification method.

```
LovinsStemmer stemmer = new LovinsStemmer();

WordsFromFile stopwordHandler = new WordsFromFile();
File stopwordsFile = new File("./src/main/resources/stopwords_it.txt");
stopwordHandler.setStopwords(stopwordsFile);

StringToWordVector stringToWordVector = new StringToWordVector(1000);
stringToWordVector.setOutputWordCounts(true);
stringToWordVector.setStemmer(stemmer);
stringToWordVector.setStopwordsHandler(stopwordHandler);

InfoGainAttributeEval igAttributeEval = new InfoGainAttributeEval();
Ranker ranker = new Ranker();
ranker.setOptions(new String[] { "-T", "0.0" });

AttributeSelection attSelect = new AttributeSelection();
attSelect.setEvaluator(igAttributeEval);
attSelect.setSearch(ranker);

MultiFilter multiFilter = new MultiFilter();
Filter[] twoFilters = new Filter[2];
twoFilters[0] = stringToWordVector;
twoFilters[1] = attSelect;
multiFilter.setFilters(twoFilters);

SMO smo = new SMO();
CostSensitiveClassifier costSensitiveClassifier = new CostSensitiveClassifier();
Reader reader = new BufferedReader(new FileReader("./costMatrix.cost"));
CostMatrix costMatrix = new CostMatrix(reader);
costSensitiveClassifier.setClassifier(smo);
costSensitiveClassifier.setCostMatrix(costMatrix);

FilteredClassifier fc = new FilteredClassifier();
```

```
fc.setFilter(multiFilter);
fc.setClassifier(costSensitiveClassifier);
fc.buildClassifier(data);
```

4.3 Create

Adding an earthquake to the database.

```
public void addDetectedEarthquake(Timestamp timestamp, double lat, double lon) {
    try {
        String position = lat + " " + lon;
        if (lat == 0 && lon == 0) {
            position = "NOT FOUND";
        }
        insertEarthquakeStatement.setTimestamp(1, timestamp);
        insertEarthquakeStatement.setString(2, position);
        int i = insertEarthquakeStatement.executeUpdate();
        System.out.println("Added " + i + " row");
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

4.4 Read

This functionality returns a list of tweets.

```
public List<Earthquake> getEarthquakesList(){
    List<Earthquake> list = new ArrayList<Earthquake>();
    Earthquake earthquake = null;
    try {
        result = getLastDetectedStatement.executeQuery();
        while (result.next()) {
            Timestamp time = result.getTimestamp("Timestamp");
            String location = result.getString("Position");
            String timeString = time.toString();
            timeString = timeString.substring(0, timeString.length()-2);
            earthquake = new Earthquake(timeString, location);
            list.add(earthquake);
            if(list.size() >= 10)
                break;
        }
        return list;
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return list;
}
```

4.5 Classification of tweets

This functionality performs the classification of tweets and count the number of earthquake tweets recognized.

```
int count = 0;
for (int i = 0; i < data.numInstances(); i++) {
    String predictClass = trainingSet.classAttribute()
    .value((int) fc.classifyInstance(data.instance(i)));
    if (predictClass.equals("earthquake")) {
        count++;
    }
}
```