Vienna University of Technology

Project report on topic

Little Sister Confuses Big Brother

Author:
Srdan Bisanovic

Supervisor:
Dr. Horst Eidenberger

September 12, 2017

Abstract

Little Sister Confuses Big Brother is an application that retrieves all available posts from a Twitter account that the user specifies. These tweets are then classified in two groups with the help of artificial intelligence: those on the *"USA-Russia Relations"* topic, and others. The ones that are on this topic, are then inverted in such way that the user gets the opposite version of the original tweet — and therefore incorrect information. This way, the application manages to confuse the people who use it.

The project consists of two challenges: classification and inversion. The first one requires various data retrieval techniques, text and natural language processing. Our highest classification accuracy in this part was 91.51%. The second one is based on English grammar rules and vocabulary. The results in this part are a bit hard to measure since there is no strict guidelines on how the desired outcome should look like. We can say that the application will successfully invert more than 90% of the tweets. By "successful", we mean that the tweet will have the opposite form of its initial information, without breaking the grammar rules and losing the sense of the sentence.

This report contains instructions on how to run and use the application, a detailed description of the methods and ideas used behind the two subparts, the architectural overview of the whole project and the hours breakdown of the time spent on this project.

1 User guide

1.1 Setting the environment

The whole project has been done in a *Maven* environment, which means that *Java 8* and the *Maven framework* have to be installed and set before the first run of the application. Instructions on how to install and set *Java 8* can be found here: [2]. *Maven* setup can be found on their official web page: [1].

1.2 Running the project

Once everything is installed and set, we can open the console, navigate to the *twitter* folder, and set up the project from there with the following command.

```
$ mvn package
```

Once the build process is finished, we can run the application. There are two available modes. The first one runs the main program, that fetches all the tweets from the specified account, classifies and inverts them. This mode can be run with the command:

```
$ mvn exec:java -Dexec.mainClass="App"
```

The second mode performs the whole training process of the machine learning component and then runs already downloaded tests on the newly trained classifier. It will print out three results in the console: general accuracy on all test tweets, accuracy on the positive tweets (the ones that are on the *"USA-Russia Relations"* topic) and accuracy on the negative tweets (all other tweets). This mode can be started with the command:

```
$ mvn exec:java -Dexec.mainClass="Analyser"
```

1.3 Use case diagram

Figure 1: Use case diagram for the main application.

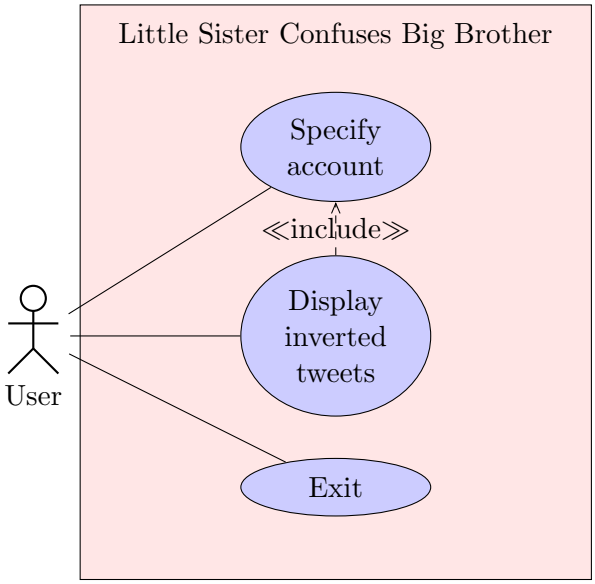

Figure 1 displays the use case diagram of the main application. This diagram is quite simple since there are only three actions performable by the user: *specify the desired account*, *display inverted tweets* and *exit the program*. In order to display inverted tweets, a valid account of interest must be given first. By valid, we mean that the field is not empty and that it's a name of a person or organization. Random arrays of characters won't return any results. The *exit* command can be executed any time, regardless of the previous action.

2 Classification

As we already said, the task od this part is to fetch the tweets from the desired account and picks the ones about *"USA-Russia Relations"*. This means that we have two possible categories for all tweets: accept and reject. We say that the tweets on this topic are positive and all others are negative. We will use this notation throughout the report. In order to perform the categorization, we had to do these two steps: collect the data and build the classification system.

2.1 Collecting the data

The first task was to collect the data that will be used for training and testing. This means that we need to have two different sets of posts for these two purposes. Out training set had 2290 positive posts and the testing set had 8092 posts. In the testing set, 3320 posts were positive and 4772 were negative. The special case of the negative posts were the ones that mention USA and Russia, but in some other context (i.e. sports). We had 1501 of these for testing. In order to collect all this data, we fetched all the tweets from 79 politics and 7 sports accounts. The majority of these accounts were from USA or England because of language. After that, we ran a search for all the tweets that contained the word "Russia" or any other word related to this country

("Putin", "Moscow" etc.). The next step was to manually go over the chosen posts and pick those that truly talk about our subject. The rest is returned to the negative posts. Having done this, we were done with data collecting. All together, we fetched 262366 tweets but, of course, not all of them are used.

2.2 Training and testing

Since the Twitter has absolutely no regulations on how the posts should look like, it is a challenging tasks to format all the posts in some way suitable for training and testing. Tweets can contain emoticons, hashtags, links and abbreviations. Besides that, because the tweet length is limited to 140 characters, there is a lot of grammatically incorrect sentences. To use something that chaotic, we had to develop a series of filters that the tweets will go through. Every filter edits or erases the part of the tweet that is't responsible for. These filters are ordered one after another in something we call *Pipeline*.

HTML corrections: When we access Twitter through its API, it will send us all the data in JSON format that includes HTML-encoded entities (i.e. &lt instead of '<'). Since we don't need any encoded information, we have to un-escape these HTML entities.

Expanding abbreviations: Shortened words cannot be recognized by the text processors, so we made a collection of the most used abbreviations and their full forms. This allows us to deal with complete words only.

Lower case: Since many people don't care about correct lower-upper case usage, it is much easier to put everything in lower case and then process the text.

Managing special characters: Different special characters are treated differently since they are used for various purposes. Hyphens ('-'), are very often used to connect words that are usually separated. Because of this, we will delete the hyphens and replace them with spaces. We will also delete '#' from hashtags and '@' from username references, since the rest of these strings can give us words that we can use for classification. Punctuation and other special characters will simply be removed since they don't usually add any necessary meaning to the sentence.

Stemming: First of all, every verb is converted to its infinitive for since it is much easier to process only one tense of the verb than all possible tenses. After this, all the words are reduced to their "stems" in order to group different forms of the same word into a single feature. This stem isn't a grammatical root of the word but it satisfies the property that related words have the same form.

URL removal: The first idea was to connect to all the URL-s in tweets and to replace them with the title of the page they lead to. However, this proved to be very unefficient since it takes much more time to look for every single of these links. Besides that, most of these pages have a title that is the same as the content of the tweet, which results in having the same thing written twice. Other links are just dead, or the servers holding them are too slow. This way, the application can get stuck while trying to connect to a server that doesn't send any response at all, since it doesn't know if the server is down or it just takes a lot of time to transfer the messages. For

these reasons, we decided to completely remove all URLs from the tweets.

Common words: This is the most important part od the pipeline. This filter directly makes the difference in accuracy of the machine learning component. In order to make it easier for the classifier to differentiate between the positive and negative tweets, we should exclude the words that occur in both types of tweets for both training and testing. We first tried to do it by downloading the lists of most common English words from Internet but the results weren't very encouraging for this particular project. For instance, only 11 English words ("the", "to", "be", "in", "of", "rt", "a", "on", "and", "for" and "have") appear more times than the word "Trump" in all the tweets that we had, and this word is certainly not in the list of the most common English words. For this reason, we found the ratio between the number of occurrences in the 5610 positive tweets and all 262366 tweets, for every single word. Having this information, we were able to determine what words are truly crucial for our purpose. The only way to do this was to try different ratio thresholds, meaning that the words that have the ratio above the chosen threshold would be left in the post. The words with the ratio under the threshold will be considered as too common and eliminated from the sentence. The table below represents the machine learning accuracy depending on the ration threshold.

Threshold	All	Positive	Negative
0.0	57.88	90.12	35.46
0.1	77.72	92.89	67.16
0.2	77.39	90.09	68.55
0.3	77.90	91.39	68.52
0.4	78.51	92.86	68.52
0.5	79.28	94.73	68.52
0.6	79.50	94.91	68.78
0.7	81.75	93.98	73.24
0.8	62.59	09.31	99.66
0.9	62.54	09.16	99.69

These results are not very satisfying, so we had to find a way to increase the accuracy. What we haven't considered in the previous case were the posts that mention USA and Russia in some other context, sports for instance. We call these posts *false positives*. The solution for this problem was to leave the words that don't appear at all in positive posts. This way, words like "football" or "basketball" will be left in the posts, which makes it much easier for the machine learning to recognize the false positives. The accuracy table with this method included is given below.

Threshold	All	Positive	Negative
0.0	81.09	90.12	74.81
0.1	88.93	92.14	86.69
0.2	88.84	88.77	88.89
0.3	89.47	90.00	89.10
0.4	90.60	91.63	89.88
0.5	91.49	93.49	90.09
0.6	91.51	93.73	89.96
0.7	92.03	92.65	91.60
0.8	62.49	08.98	99.73
0.9	62.44	08.86	99.73

As we see, the best result all together is when the threshold is 0.7 but we chose the threshold to be 0.6, since that's the result where we get the most positive posts, which are of our biggest interest. There is a better result for positive results in the first table, but the overall accuracy there is too low (79.50%).

Having said that, our best machine learning accuracy is 91.51% overall, 93.73% for positive and 89.96% for negative posts.

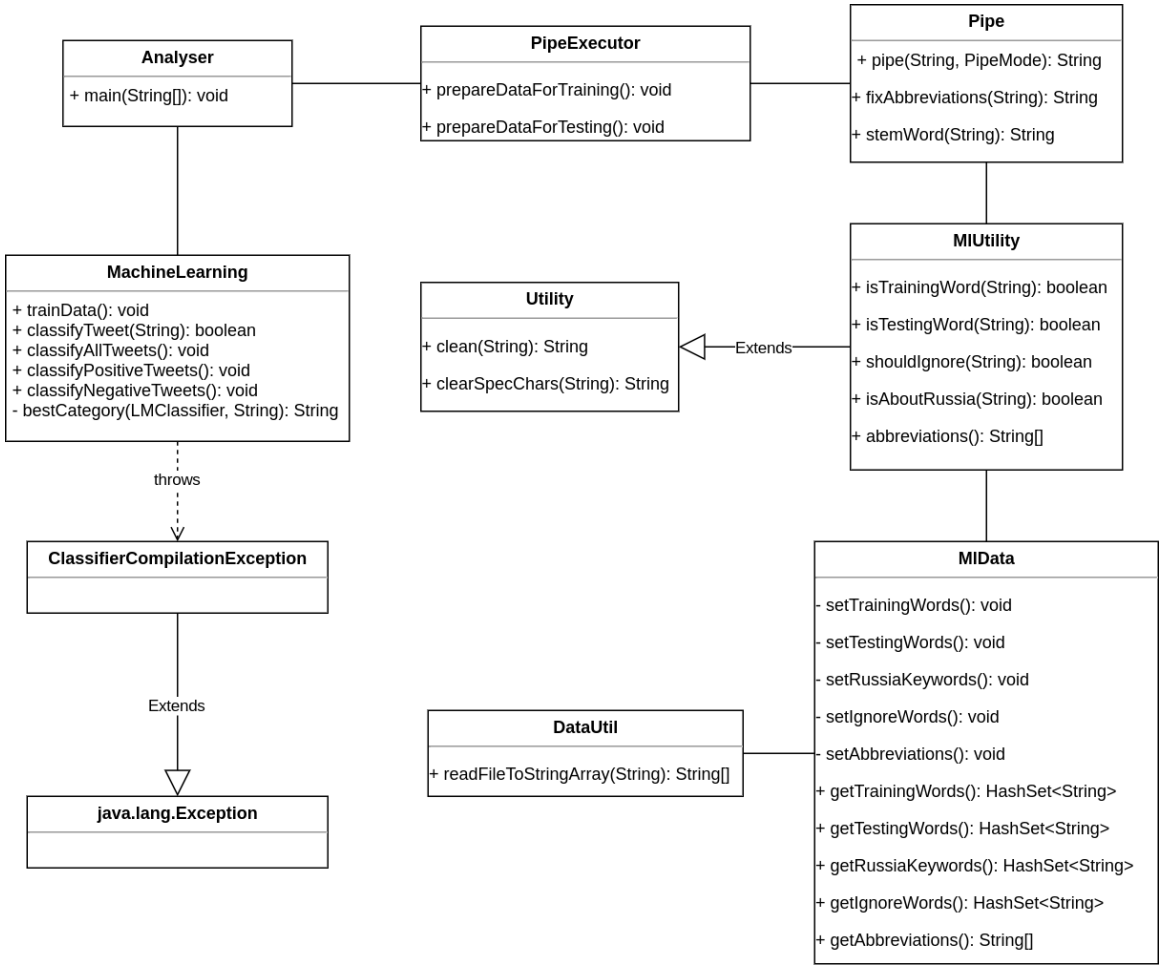

Figure 2: Class diagram for the classification part.

3 Inversion

This is the second part of the application. Its goal is to invert the meaning of the chosen posts by putting the verbs of the sentence in their opposite forms. For instance, the sentence "We will have a meeting." will be turned to "We won't have a meaning." There are three conditions that the resulting sentence should fulfill in order to be considered correct: the meaning should be opposite of the starting sentence, it should stay within boundaries of English grammar rules and the new sentence should have sense. The third requirement was the most challenging because not all sentences have the meaning when we convert them. For instance, "When will you go there?" will become "When won't you go there?", which isn't a very logical thing to ask. This task increases its complexity when sentences become longer and more complex. That's why we had to develop a unique system of rules that determines what sentences will be inverted. Besides that, it also decides what verbs will be converted within the sentence, since we cannot just simply turn everything to its opposite version. In order to explain the whole inversion process, we have to define some new terms first.

Word collections are the files that store different types of words that we needed for this purpose. The most important collection is the verb collection, since it is used for recognizing the verbs in the sentence. Besides that, we also use noun, adjective and name collections.

Tense is a class that we use to store a whole verb form. By verb form, we mean a whole construction of the verb. For instance, the verb phrase "have been doing" is in Present Perfect Continuous. Our class *Tense* can store up to four levels. Each of the levels contains ove part of the verb construction. If we take a look at our previous example, "have" will be stored on level 1, "been" on level 2 and "doing" on level 3. When we iterate through the words in the sentence, each time we come across the verb, we will try to put it to the next level of the *Tense*.The class also contains the knowledge about what kind of a verb can be accepted next. For instance, if have the verb "have" on level 1, we cannot add "is" to level 2, since there is no tense with such construction. The first we fail to add a verb to the next level, we know that we have a complete verb form.
This class also contains some additional information about the sentence we're currently operating on. It keeps the sentence position of the verbs stored in the levels (the word "go" in the sentence "I have to go." has the position 3, since the counting starts from 0). It also contains a boolean variable that knows if a sentence starts with a question word ("what", "who" etc.) because we need that to decide how are we going to invert our verb tense. It also knows if we turn the verb from its positive to its negative or vice versa.

Post formating: Before we can start the post inversion, we have to make sure that all punctuation and other special characters are used correctly with the right spacing around them, since this has a huge impact on splitting the posts in their subparts. We also replace all short modal forms like "I'm" with their longer forms like "I am". Once we have done this, we are ready to start iterating through the sentences.

New sub-sentence is one of the most important concepts in this inversion system. Most of the sentences, that we have processed, have more than one verb construction and are very often built from several sub-sentences, connected with "and", "or" and

similar words. That's why we introduced the *NewSubSentenceHelper* class. This class contains all the rules that signal the start od a new sub-sentence within our main sentence. When we iterate through the words, each word will be tested on the new sub-sentence conditions. These conditions are:

- Is it a conjunction word?
- Does it end with a comma?
- Is it one of the interrogatives ("what", "who", "how")?
- Is it a quotation start?

If we conclude that the currently processing word is a start of a new sub-sentence, we will invert that what we have stored in the *Tense* class, delete everything from it and start building a new verb form. We go by the rule that only one verb can be inverted within a single sub-sentence. That way, we can be sure that the resulting sentence will be some meaningful thought.

Questions: are treated differently than other sentences, because it's very easy to lose the meaning of the sentence if we're dealing with a question. However, if we have a question in negative form like "Aren't you there?", we can always simply turn it to its positive form like "Are you there?". For the questions in positive forms, we created three subgroups, depending on what interrogative is on their start: skip, push and negate questions.

- *Skip questions* are the questions that simply cannot be inverted in any way so that the result is something meaningful. In this group belong the questions that start with the following interrogatives: "when", "where", "whither", "whence", "whether" and "whatsoever". In addition, future questions also belong here — the ones that have the modal verb "will". These questions are always skipped from the negation since we won't get anything meaningful at the end.
- *Push questions* are the ones that start with: "what", "which", "whose", "who" and "whom". Here it also doesn't make much sense to negate the modal verb as in normal sentences, but it makes much sense to negate the main verb. For instance, for the question "Which one do you like?", it doesn't make sense to convert it to "Which one don't you like?" but it makes sense to put it to "Which one do you not like?".
- *Negate questions* are negated just like all other sentences. The questions that start with "why" and "how" are the part of this group.

False positives is another important thing to be careful about while inverting the verbs. It is a very often case that a word is recognized as a verb, since it's written the same way as some verb that really exists, but it's actually some completely other word class (i.e. noun or adjective). In order to avoid these confusions, we had to find to rules that will recognize these false positives. The rules are:

- Are there singular/plural irregularities? (i.e. "Russia deal". From here, we know that the word "deal" is a noun, because if it was a verb, we would have "Russia deals")
- Is there an article before ("the", "a", "an")?

- Is there a conjunction word before although there is no verb before?
- Is there a preposition before or after (words like "in", "from")?
- Is that the start of the sentence (i.e. "Sanctions were discussed". We know that "Sanctions" is a noun here because it's on the first place in the sentence. We exclude real questions from this check.)?
- Is there an adjective before?
- Is there a possessive pronoun before?

Inversion: If the word is a real verb, we first try to add it to the next level of the *Tense*. If this can't be achieved, we have to invert the tense. Other triggers for the inversion process are: new sub-sentence and end of the sentence. Once we enter the inversion process, we first decide if deal with a question. If that's the case we invert it like we explained above. If that's not the case we look for some general cases. For instance, in the sentence "Everybody will understand.", it is grammatically correct to turn "Everybody" to "Nobody" and leave the verb as it is. If there are no such general cases, we proceed to the real inversion. There, we first check if there is a modal verb that can be turned. Otherwise we will fain the main verb, recognize its tense and put it in the correct opposite version.

Clearance: After inversion, we have something we call clearance. For example, if we turn the sentence "There was some evidence." to "There wasn't some evidence.", the word "some" is sufficient. If we replace it with "There wasn't any evidence.", the sentence will gain its sense again. The clearance part performs these non-verb replacements.

After all of these steps are done, the new version of the post will be displayed to the user. Class diagram for the inversion part is displayed on Figure 3.

4 Architecture

We have explained the most complex parts of the application so far. The only part that we haven't explained yet are the data access layer and UI.

4.1 Data access layer

Since we have no database, all our required data is stored in files. We process a very big amount of text and if we had to iterate over all our stored words for each word we process, the iteration costs would be enormous. We chose files because we want to have all our data immediately available. That way, we will read all the files on the program start and store everything in HashSet or HashMap data structures. The files are not used or altered at any later point of program execution. In order to make the architecture more understandable, we separated this layer in *data* and *dao* sub-packages. All hash data structures are stored in the *data* package. Access to this data and basic operations on it (i.e. query if a word is a verb) are implemented in the *dao* package. Every class from the *dao* package has its corresponding *data* class. The only exception is the *VerbUtility* class because it's responsible for two classes in the *data* package: *Modals* and *Verbs*.

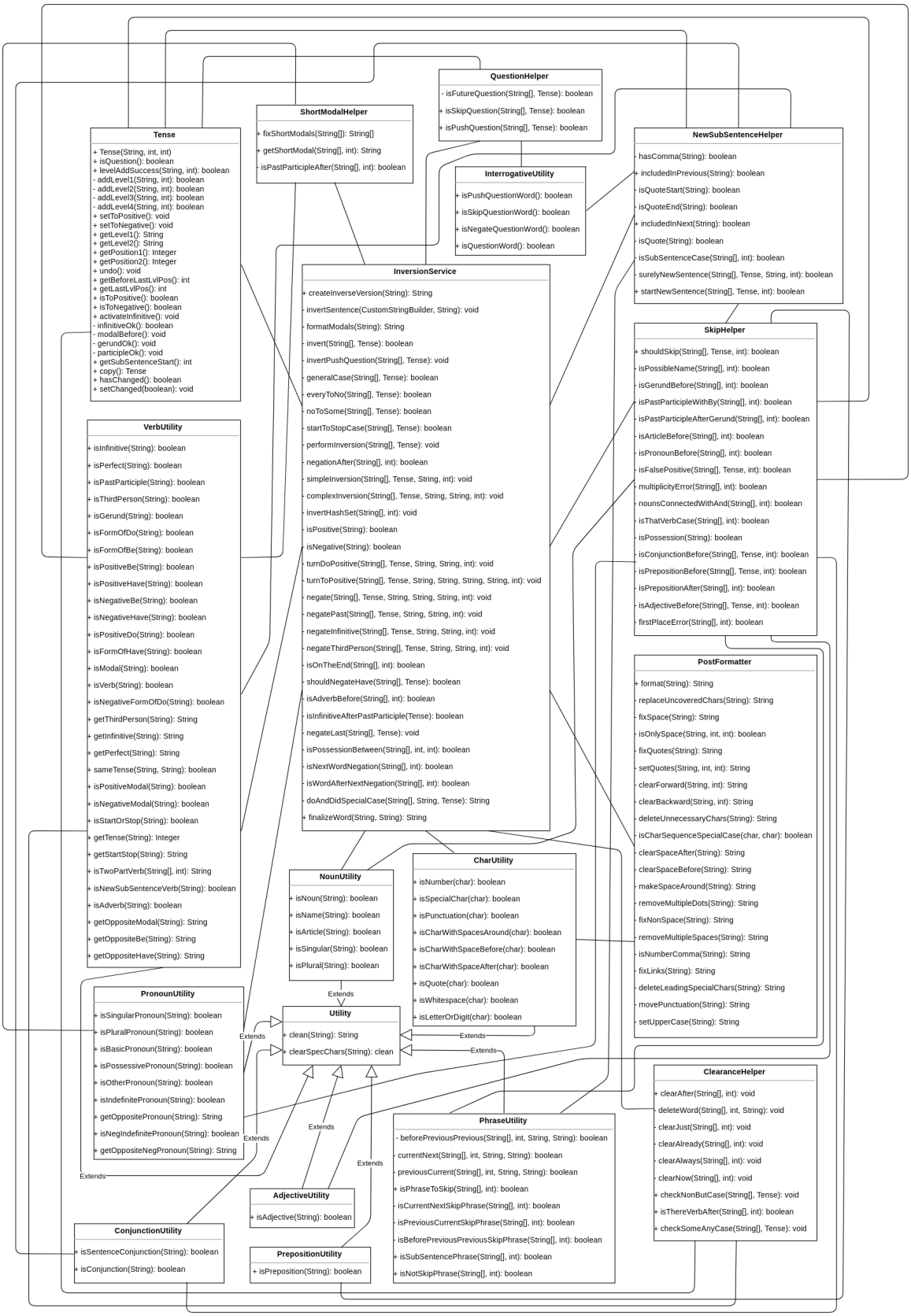

Figure 3: Class diagram for the inversion part.

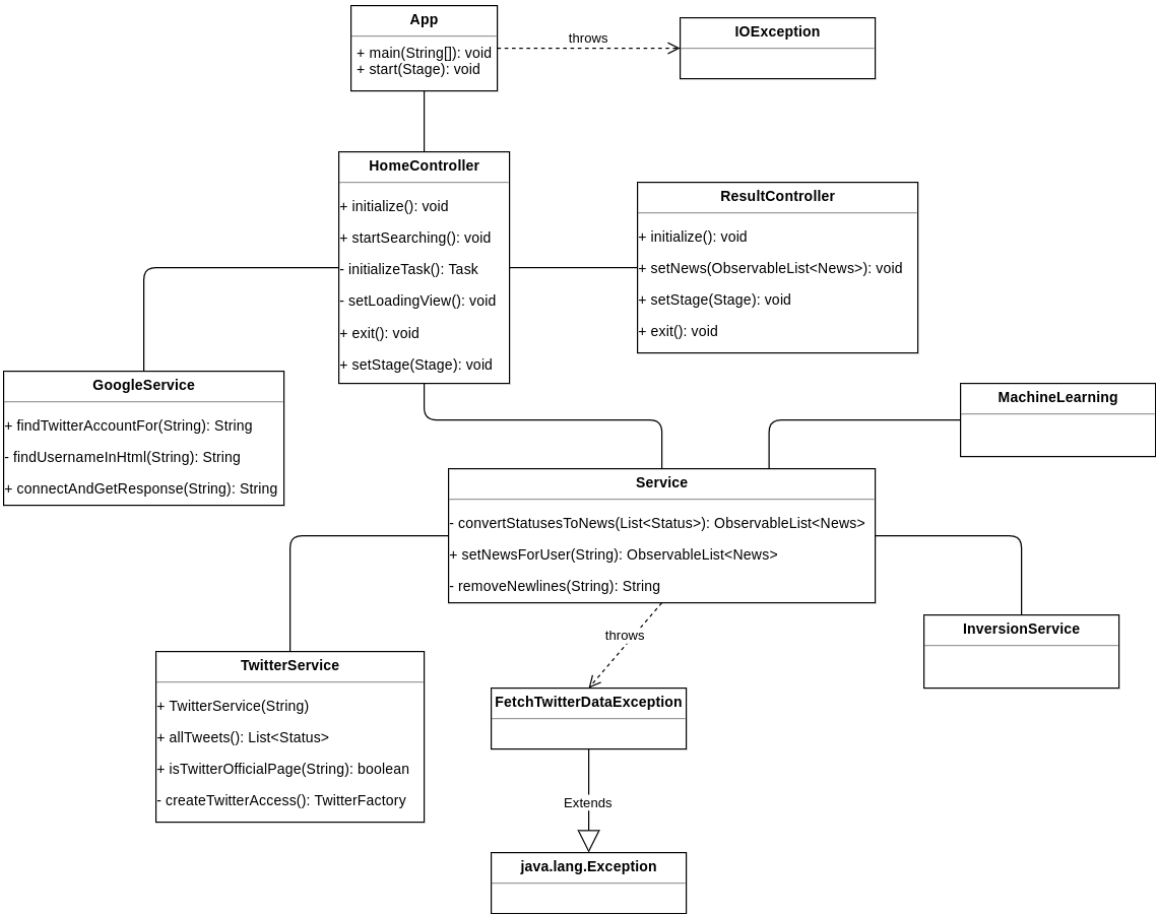

Figure 4: Class diagram for the inversion part.

4.2 User interface

There are only two classes responsible for the UI: *HomeController* and *ResultController*. The first one is our starting screen. It asks the user for the desired Twitter profile. It's also responsible for the error messages if the entry is invalid. While we wait for the application to load and convert all the statuses, the loading screen will be shown. Once the loading is done, the *ResultController* is started. It is only responsible for the table that displays all retrieved tweets. The table contains ordering, publish date, original and inverted tweet.

4.3 All together

Once a correct name is entered, it will use the *GoogleService* class to find the profile of that person or organization. After that, it will send the exact username of the Twitter account to the main *Service* class. This class connects all the dots together. It will first use the Twitter API to fetch the tweets from the specified account, than it will use the machine learning component to classify them and than call the *InversionService* to make the opposite versions of the posts. The class diagram for the whole architecture is given on the Figure 4.

5 Hour breakdown

Task	Description	Time
Literature research	Searching the web for the literature that could be used	4h
Reading the literature	Mostly books on hot wo use Lingpipe framework and how to build classifiers	14h
Planning the architecture	Planning the way different components interact with each other	4h
Building the basic architecture	Creating the most important classes (controller and service classes) and putting them together	5h
UI design	Planning how the UI is going to look	1h
UI programming	Detailed implementation of *HomeController* and *ResultController classes*	11h
Twitter data gathering	Code for Twitter data fetching	8h
Manually classifying tweets	Deciding what tweets are on the specified topic	26h
Implementing machine learning component	Machine learning code	7h
Creation of pipe	Reading about the pipe filters and implementation	15h
GoogleService	Code for accessing the Google search engine	2h
Collecting English words	Search for English word collections, code for automated verb conjugation, manually entering the words	13h
PostFormatter	Code for text foramtter.	6h
SkipHelper	Planning and implementation for recognizing the false positives, fixing bugs	10h
NewSubSentenceHelper	Planning and implementation for new sub-sentences, fixing bugs	18h
ShortModalHelper	Code for turning the short modal verbs to their long forms	1h
ClearanceHelper	Code for inverting the non-verb words	3h
Question handling	Planning how to handle questions, implementation	7h
Simple inversion	Planning and implementation	3h
Complex inversion	Planning and implementation	11h
Special cases handling	Searching for special inversion cases and implementing them	23h
Data access layer	Planning and building the data layer	9h
Refactoring	Making the code more readable, JavaDoc	14h
Documentation	Writing the report	13h
Maven packaging	Setting that is project is maven-executable	1h
All together		229h

References

[1] Apache. Installing Apache Maven. `https://maven.apache.org/install.html`, 2017.

[2] Oracle. JDK 8 and JRE 8 Installation. `https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html`, 2016.