Chatbot trained on movie dialogue

Alexander Roghult
roghult@kth.se

Degree in Computer Science, First Cycle, DD143X
Supervisor: Gabriel Skantze
 Examiner: Örjan Ekeberg
Abstract
A chatbot is a computer program that engages in written or spoken conversation with a human user. This project aims to investigate the possibility of training a chatbot in using movie dialogue in generating the response. Movie dialogue can be found in both movie scripts as well as subtitles, though using subtitles is much easier as they follow a special formatting.

Using one subtitle as a response to each word found in the preceding subtitle, the implemented chatbot links together subtitles. The responses are stored in a frequency distribution table that maps each word to all responses found.

Though the responses generated by the chatbot were not desirable, the response set most often contained responses which would be more fitting. The result drawn from this is that, with further work and improvement, the chatbot could perform acceptably.
Table of Contents

1 Introduction .................................................................................................................. 1

2 Goal .............................................................................................................................. 1

3 Background ................................................................................................................. 1
  3.1 ELIZA ...................................................................................................................... 1
  3.2 A.L.I.C.E. ............................................................................................................... 2
    3.2.1 AIML .............................................................................................................. 2
  3.3 Corpus based approach ......................................................................................... 3

4 Methodology .............................................................................................................. 3
  4.1 Movie scripts .......................................................................................................... 3
  4.2 Movie subtitles ....................................................................................................... 4
    4.2.1 SubRip ............................................................................................................. 4
  4.3 Implementation steps ............................................................................................ 4
    4.3.1 Downloading subtitles .................................................................................... 5
    4.3.2 Subtitle analysis ............................................................................................. 5
  4.4 Error handling ....................................................................................................... 6

5 Results ....................................................................................................................... 7

6 Discussion .................................................................................................................. 9
  6.1 Improvements ......................................................................................................... 9
    6.1.1 Choosing distributors ....................................................................................... 9
    6.1.2 Natural language processing ........................................................................... 9
    6.1.3 More advanced weighting .............................................................................. 9
    6.1.4 Incorporating predetermined responses ......................................................... 10
    6.1.5 Database ......................................................................................................... 10

7 Conclusion ................................................................................................................ 10

8 References ............................................................................................................... 11

Appendix A Chatbot.java ............................................................................................. i

Appendix B SubtitlesWorldParser.java ....................................................................... iii

Appendix C SubRipFileAnalyzer.java .......................................................................... vi

Appendix D NLP.java .................................................................................................. x
1 Introduction
The Turing test is a test of a computer’s ability to display intelligent behavior equivalent to that of a human. Alan Turing introduced the test in his 1950 paper “Computing Machinery and Intelligence”. Even though the test has existed for over 60 years, no program has been able to pass it; though some have been close. A chatbot is a computer program that engages in written or spoken conversation with a human user. For the developer, the goal is to make the chatbot as intelligent as possible and to pass the Turing test. In 1966 Joseph Weizenbaum created a chatbot called ELIZA. It mimicked the responses of a psychotherapist and came very close to passing the Turing test (1).

Today’s chatbots are widely used as support agents for companies providing offline help, personalized service or information acquisition but have also been used for malicious activity. They are often seen flooding chat rooms with spam and advertisements but are also used to fool people into revealing personal information, such as bank account numbers, pin codes or passwords. There is even a published report of a chatbot posing as a person on a dating site (2).

Most chatbots are trained with a large set of text corpora, a large and structured set of texts, or they use predefined responses to certain inputs in order to reply to the human user. There exist several dialogue corpora, such as the Dialogue Diversity Corpus, British National Corpus etc. as well as free AIML sets that can be used in training a chatbot (3) (4).

2 Goal
The goal of this report is to investigate if it is possible to train a chatbot using dialogue from movies. Movie dialogue can be found in both movie scripts and movie subtitles. These are available in large quantities on websites such as SubtitlesWorld, Subscene, Moviesubtitles.org, Internet Movie Script Database, etc. (5) (6) (7) (8). As these websites are constantly being updated with new movies, they could provide a stable source of data for training a chatbot.

3 Background
This section will provide an introduction to the history of chatbots. The method used in two of the most well known chatbots, ELIZA and A.L.I.C.E. will be presented and discussed. Both ELIZA and A.L.I.C.E. use a set of defined rules to generate their response, a different method will also be presented that uses dialogue corpora.

3.1 ELIZA
Joseph Weizenbaum created ELIZA in 1966. It was one of the first attempts at a computer program that gave the user the illusion of talking to another human, while in fact all responses were generated by the program. ELIZA takes a
statement from the user as input, analyzes it and generates a response. The user is then expected to provide another statement.

ELIZA uses so-called scripts to decide on its response and can therefore easily be improved or taught another language simply by adding to it, or changing the script (1). The script is built up of keywords and their associated rules. The statement provided by the user is examined for keywords. When such a keyword is found the response from ELIZA is derived according to the rules belonging to the keyword.

3.2 A.L.I.C.E.
A.L.I.C.E., short for Artificial Linguistic Internet Computer Entity, is a chatbot created by Richard Wallace in 1995. A.L.I.C.E. uses A.I.M.L., Artificial Intelligence Mark-up Language, to store the conversational rules between itself and the user (9). In this way it is similar to ELIZA, using “scripts” in deciding its response to the user. A.I.M.L on the other hand is a more advanced language, allowing more complex rules.

3.2.1 A.I.M.L.
A.I.M.L is an XML-compliant language used for storing rules of interaction for the chatbot. It is built up of several units that in different ways determine the response of the chatbot. The most important units are <category>, <pattern> and <template> (10). The category tag is a basic unit of knowledge in A.I.M.L and contains the rules for matching a certain input with an output. Within the category reside the pattern and template tags. The pattern contains the text that is to be matched with the users input while the template contains the output from the chatbot. The pattern is always written in uppercase and can contain the wildcard symbols “*” and “_”, which will be explained in detail below.

There are three general types of categories in A.I.M.L: atomic, default and recursive categories (11). Atomic categories are categories that do not contain any wildcards. An example of an atomic category can be seen below. The template “Hello.” will be returned to the user if “HI ALICE” is provided as input.

```
<category>
    <pattern>HI ALICE</pattern>
    <template>Hello.</template>
</category>
```

Default categories are categories containing wildcards. The ultimate default category is the one with <pattern>*/pattern>, matching any input. Wildcards are more often used embedded in sentences. The following example is taken from A.I.M.L Primer (11).

```
<category>
    <pattern>I NEED HELP */pattern>
    <template>
        Can you ask for help in the form of a question?
    </template>
</category>
```
Wildcards are used to match more general patterns with a user's input. The category above will respond to any input that starts with the phrase “I NEED HELP”, regardless of the ending.

Recursive categories map patterns to other categories, making it possible to simplify the language, to identify synonymous patterns, divide the input into two different parts, etc. (12). Recursive categories use <srai> and <sr/> tags; the <srai> tag is used to direct the input to another category while the <sr/> tag is used when partitioning an input. The example below is taken from AIML Primer (11).

```
<category>
  <pattern>HELLO *</pattern>
  <template><srai>HELLO</srai> <sr/></template>
</category>

<category>
  <pattern>HELLO</pattern>
  <template>Hi there.</template>
</category>
```

This category uses the <srai> tag to match “HELLO” to another category, in this case the second. It then appends the best response found for the input found in the * wildcard.

### 3.3 Corpus based approach

The two chatbots mentioned earlier use a set of defined rules in determining the response of the chatbot. Another approach is to use a set of text corpora to train the chatbot. The chatbot analyses the corpus with a natural language processing (NLP) system (13). The NLP system can be used for example tokenizing conversations, removing stop words and punctuation; thereby filtering out the key words in determining the context of the phrase (14). The benefits of this method are that the developer does not need to write the chatbot’s responses manually. By clever design and a good NLP system the chatbot can provide the user with intelligent responses. To increase the vocabulary the chatbot only needs to be provided more text corpora.

### 4 Methodology

The chatbot was written in Java as this is the programming language that the author is most practiced in. To train the chatbot it was decided that the corpus-based approach was more reasonable than using AIML. AIML is a powerful language to use for chatbots, though developing a process to transform dialogue extracted from scripts or subtitles into AIML categories seemed a very difficult task.

#### 4.1 Movie scripts

The initial plan was to use movie scripts to train the chatbot. Movie scripts contain all the dialogue between characters and can be accessed on several websites. One of these websites is the Internet Movie Script Database (8).
Upon analyzing different scripts, it became apparent that there was no common way of formatting the scripts. Some scripts were available on the website in HTML, while others needed to be downloaded as PDF files. The way in which the scripts were written also varied greatly; most scripts provided a clear distinction between the different characters’ dialogue, marking the speaker in HTML <b>-tags followed by the line, though some scripts also contain this formatting when describing the environment in the scene. While distinguishing what is dialogue between two characters and what is text describing the scene is easy for a human, this is difficult for a computer program. The program would need to be able to figure out in which format each script was written in order to find the text containing the dialogue. This did not seem a plausible approach for this project so a new approach was needed.

4.2 Movie subtitles
As the movie scripts proved to be a difficult source of training data, movie subtitles were examined as a possible source of dialogue. Movie subtitles only contain the dialogue of a movie while leaving out, in this case, unnecessary text such as scene descriptions. Subtitles also follow formatting rules, which make them much easier to parse. One of the most common subtitle formats is SubRip.

4.2.1 SubRip
SubRip is a software program used to extract subtitles and their timings from video files and DVDs (15). The subtitle is saved in a text file with the extension .srt. The SubRip subtitle format is one of the most common available for download and has a very simple formatting. An example from the movie Pulp Fiction can be seen below:

```
303 00:19:58.527 --> 00:20:02.203
And you will know
My name is the Lord...
```

The first line dictates that the subtitle is the 303rd subtitle to be shown. The second line specifies the time at which the subtitle is to be shown. The start and end time follow the format hours:minutes:seconds.milliseconds and are separated by “-->”. The two following lines contain the subtitle to be shown followed by a blank line, separating this subtitle from the next. The subtitle line can also be on a single line.

4.3 Implementation steps
There are several websites that provide subtitles to movies and TV series; SubtitlesWorld, Subscene, Moviesubtitles.org, etc. (5) (6) (7). None of these websites provide a bundle or package containing all subtitles, therefore requiring each subtitle to be downloaded individually. Doing this manually would be very cumbersome and not allow for new scripts to be found by the chatbot. An application was needed to crawl through the website, downloading each subtitle available. The downloading procedure is time consuming and
therefore stores the subtitles in a local directory. The chatbot can thereafter access them for analysis.

### 4.3.1 Downloading subtitles

The application with the task of downloading subtitles was written in Java. To connect to and parse the website, a third party Java library called jsoup was used. Jsoup provides an API for extracting and manipulating elements from a website. SubtitlesWorld had a good structure in listing subtitles, which would make it easy for the program crawl through. Therefore it was decided that SubtitlesWorld would be used to download subtitles.

The application would start by connecting to [http://movie subtitlesworld.com/lists-a.html](http://movie.subtitlesworld.com/lists-a.html), a page with links to all subtitles where the movie title begins with the letter A. On this page there exist links to other pages containing the rest of the movie subtitles, categorized by the first letter in the title. These are easily extracted with jsoup. Each subtitle link directs to a webpage with download links for the movie’s subtitle in different languages. The application finds the English subtitle and downloads it. The subtitles are stored on the local hard drive to then be used by the chatbot.

### 4.3.2 Subtitle analysis

Before interaction is possible with the chatbot, it needs to analyze the downloaded subtitles. The chatbot analyzes the downloaded subtitles in order to generate a response with the interacting user. As the SubRip subtitle format does not include any standardized way of indicating when a new person is speaking, the chatbot presumes that each new subtitle is a response to the previous. Seen below is an outtake from the movie 500 Days of Summer.

```
173
00:11:17,010 --> 00:11:19,428
So, how's it goin'? 

174
00:11:19,554 --> 00:11:21,472
Pretty good. 
```

In this case the second subtitle, “Pretty good”, will be noted as a response to the first subtitle, “So, how's it goin'?”. Henceforth the first subtitle in the example above will be referenced to as subtitle1 and the second subtitle will be referenced to as subtitle2. It is not likely that the interacting user will provide an input that is exactly the same as a line in a movie. Therefore the chatbot needs to register the responses on a per word basis. As proposed by Bonnie Chantorotwong, a conditional frequency distribution is created, mapping each word to the response (14). To break the sentence into words the Stanford Tokenizer is used, provided by the Stanford Natural Language Processing Group (17). These words will hereafter be referenced to as tokens. Each token in subtitle1 will be added to the frequency distribution with subtitle2 as a response. The frequency distribution is stored in the local memory in a HashMap data structure, storing each token as a key mapping to a HashSet data structure containing all the results as well as their weight.
The timing of the subtitle is also taken into consideration. A subtitle is only added to the frequency distribution as a response if its starting time is within two seconds of the preceding subtitles stopping time. Subtitles not within this timeframe are regarded as not being part of the same conversation.

Most tokens will occur in several subtitle lines and therefore need to point to several different responses. Choosing the right one is crucial to give a correct response to the users input. Each response will therefore be given a value denoting the probability of that token causing the response. The simplest way of doing so is presuming that each token is equally responsible for causing the response. Each token is therefore given the weight of $\frac{1}{N}$, where $N$ is the number of tokens of the input. In the case given above, each token in subtitle1 will have a weight of $\frac{1}{4}$ to return subtitle1 as a response. Let us assume that there also exists a subtitle line, “How are you?”, that has subtitle2 as a response. The weight for each token in the new subtitle, to provide subtitle2 as a response, would be $\frac{1}{3}$. This distribution is shown in Figure 4-1 Frequency distribution.

![Figure 4-1 Frequency distribution](image)

If a subtitle line is written on two lines, the lines sometimes begin with “-”. This indicates that the two lines are spoken by two different people. Though this is not a standard for the SubRip subtitle format it is so commonly used that it is worth taking into consideration upon subtitle analysis. If such a subtitle line is found, it is divided into two separate subtitles.

### 4.4 Error handling

To the SubRip subtitle format there have been added several unofficial formatting options that complicate the subtitle parsing. To make it easier for the viewer, some use notations to indicate if the spoken language is changed, if a character takes over the conversation, etc. These notations are not included in
the SubRip subtitle format and are therefore arbitrary additions that are not followed by all.

Events in the movie that are relevant to the subtitle are sometimes included in the subtitle file. To indicate that these lines are not spoken by any character they are often surrounded by square brackets, “[“ and “]”. This could for example be if the language changes, somebody coughs, etc. The most common use for these is to describe sounds to hearing impaired viewers. The chatbot needs to disregard these lines, since they must not be regarded as a possible response to an input.

Different parties, some more reliable than others, release the subtitles downloaded from SubtitlesWorld. The chatbot does not take into consideration which distributor has made the subtitle, but only takes the first one available. Upon running the subtitle analysis the chatbot will sometimes run into errors in the file. Some of the errors the chatbot came upon were unrecognizable characters as well as formatting violating the SubRip format. If the chatbot discovers any of these errors, the analysis for that subtitle is aborted.

5 Results
The reached result is a chatbot that utilizes movie subtitles to generate responses to the user. The initial choice of using movie scripts was discarded due to the fact that subtitles follow a specific formatting while there is no predefined way of writing scripts. Parsing subtitles into data for the chatbot is therefore much easier.

The chatbots responses are most often not satisfactory. The responses often halt the conversation and are sometimes not even understandable in the context. The beginning of a simple conversation can be seen in Figure 5-1 Example of conversation. The green texts are that of the user while the black texts are the responses from the chatbot.

Chatbot started:
Hello
you came back in the middle of a restricted area
How are you today?
are you alright?
I am good thanks how are you?
okay.
What are you going to do today?
you...
What are your plans for today?
stay here.
The weather is nice today
- yeah.

Figure 5-1 Example of conversation
As seen in Figure 5-1 Example of conversation, running the chatbot and providing the input “Hello” will, with the current downloaded subtitles, result in the response “You came back in the middle of a restricted area”. This may not be the most desirable response from the chatbot, especially as it most likely is the first response to the user. The weight of this response is set to 1, indicating that upon analysis, the token “hello” was a sole contributor to the response “You came back in the middle of a restricted area”. Searching through the subtitles also shows that this is the case. Below is an outtake from the subtitle file “1201.DVDRip.Koi.en.srt”.

247  
00:12:52,051 --> 00:12:53,352  
HELLO.

248  
00:12:53,419 --> 00:12:56,088  
YOU CAME BACK IN THE  
MIDDLE OF A RESTRICTED AREA

The frequency distribution also contains several other responses with 1 as a weight, some of these providing the much more desirable; “Hello!”, “It’s a pleasure to be here!” and “What’s up?” are a few.

Continuing with the input “What are you going to do today?” the chatbot responds with “You...” with a weight of 1.74, again not a response adding meaning to the conversation. Among the list of responses “Nothing” can be found, with a weighting just below at 1.72. Though this response more or less halts the conversation, it is still on topic.

Providing the chatbot with 350 subtitle files to analyze derived the results of the project. Exceeding this amount greatly increased the time of analysis and lead to the chatbot crashing due to exceeded memory capacity. The reason for this is that the frequency distribution HashMap grows extremely large. At 350 subtitle files the frequency distribution holds around 45000 tokens mapping around 1.4 million responses. The responses are of course not unique, as they are added to the frequency distribution for each token pointing to it. In Table 5-1 Analysis statistics, the running time, amount of tokens and the total amount of responses found are listed. When reaching 450 subtitle files, the chatbot crashes due to exceeding the memory capacity.

<table>
<thead>
<tr>
<th>Subtitle files</th>
<th>Analysis time (ms)</th>
<th>Tokens</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>2719</td>
<td>29069</td>
<td>642475</td>
</tr>
<tr>
<td>200</td>
<td>3444</td>
<td>33899</td>
<td>847489</td>
</tr>
<tr>
<td>250</td>
<td>4432</td>
<td>38032</td>
<td>1042274</td>
</tr>
<tr>
<td>300</td>
<td>5015</td>
<td>42020</td>
<td>1223482</td>
</tr>
<tr>
<td>350</td>
<td>5873</td>
<td>45610</td>
<td>1433292</td>
</tr>
<tr>
<td>400</td>
<td>83502</td>
<td>49681</td>
<td>1634926</td>
</tr>
<tr>
<td>450</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

*Table 5-1 Analysis statistics*
6 Discussion
Though the project was not a success in creating a chatbot possible of providing a user with reasonable responses, it was a success in determining weather a chatbot can be trained using movie dialogues. With just a basic implementation, the chatbot shows promise in producing responses relevant to the users input. It is hard to tell whether a more advanced implementation would be able to pass the Turing test, though it is my belief that the chatbot could be made advanced enough to keep the conversation on topic.

6.1 Improvements
There remain several improvements that can be applied to the chatbot. However, as the goal of the project was only to examine the possibility of using movie dialogue and not to implement a fully functioning chatbot, these improvements were not prioritized.

6.1.1 Choosing distributors
There are several different distributors releasing subtitles to movies and TV-series, some more trustworthy than others. By finding the more reliable distributors that are consistent in their formatting, the quality of the chatbots data would greatly increase. For example, not all distributors use “.” to indicate that a different character is speaking. The chatbot has then no way of knowing that the subtitle should actually be split. A response using this subtitle will most likely not be adequate.

6.1.2 Natural language processing
Un unexpected obstacle was analyzing natural language. With a deeper knowledge in computational linguistics the subtitle analysis could be greatly improved. A more advanced NLP system could for example be used; one that discovers spelling errors, transforms contractions to their original words and groups together words. Contractions are shortened versions of a word, syllable or word group; as “don’t” is a contraction of the words “do not”. This way a more uniform dataset is provided for the chatbot. This example also makes the importance of grouping together words clear. Some words should not be analyzed by themselves but should instead be grouped together with other words. Grouping together “do not” and using it as a single token can preserve the meaning the words add to the sentence, helping the chatbot in choosing a better response.

6.1.3 More advanced weighting
The chatbot uses a very simple method of weighting the responses. Simply giving each token the same weight is a naïve approach, which does not correlate with reality. Each word in a sentence does not provide equal meaning to it. It is more likely that words occurring less often are more responsible for the response. By taking this into consideration a more probable weighting could be given each token.
6.1.4 Incorporating predetermined responses
To help the chatbot with the most common inputs, a set of predetermined responses could be created. These responses could be used in combination with the data derived from the movie subtitles. The chatbot could, for example, also be compliant in handling AIML. This way already existing AIML sets could be downloaded and incorporated with the response data provided by subtitles. Examples of input which would contribute from this are “Who are you?”, “What day is it today?” and “Who created you?”. These are all questions which can not be answered by movie subtitles as they are questions regarding the chatbot or information about the present time.

6.1.5 Database
The implemented chatbot stores all of its responses in the primary memory. This means that it needs to read through and analyze the subtitles each time it is restarted. It is also prevented from using a large amount of subtitle files since this exceeds the memory. By storing the responses in a database a much larger dataset could be used and would remove the necessity of analyzing subtitles on startup. If the chatbot was implemented on a webserver, a scheduled job could search for new subtitle files, adding them to the dataset as they are made available. The dataset would then be continuously updated for the chatbot.

7 Conclusion
In conclusion this project has shown that it is possible to create a chatbot using movie dialogue. Though the chatbots responses were not always reasonable, there existed responses in its dataset that would be satisfactory. By applying improvements it could be possible for the chatbot to use the most preferable response as well as increasing its source of data. These improvements were not implemented due to the time constraints as well as the goal of the project. In order to perfect a chatbot based on movie dialogue a greater knowledge in computational linguistics is needed. By grouping together certain words into single tokens the chatbots dataset will be more precise in its response evaluation.
8 References


import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;

public class Chatbot {

    private static final int NUMBER_OF_FILES = 350;
    private HashMap<String, HashSet<WordResponse>> frequencyDistribution;

    public Chatbot() {
        frequencyDistribution = new HashMap<String, HashSet<WordResponse>>() {
    }

    public void initialize() {
        // read through all downloaded subtitles.
        analyzeSubtitles();
    }

    /*
     * Goes through the downloaded subtitles and adds them to the frequency
     * distribution
     */
    private void analyzeSubtitles() {
        SubRipFileAnalyzer analyzer = new SubRipFileAnalyzer();
        File dataDirectory = new File(Constants.SUBTITLES_DIR);
        File[] subtitleFiles = dataDirectory.listFiles();

        for (int i = 0; i < NUMBER_OF_FILES; i++) {
            File subtitleFile = subtitleFiles[i];
            if (subtitleFile.getName().endsWith(".srt")) {
                analyzer.analyze(subtitleFile, frequencyDistribution);
            }
        }
        System.out.println("Done analyzing.");
    }

    /*
     * Start talking to the chatbot.
     *
     */
    public void start() throws IOException {
        System.out.println("Chatbot started:");
        while (true) {
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            String input = br.readLine();
            String response = getResponse(input);
            System.out.println(response);
        }
    }
}
private String getResponse(String input) {
  String response = generateResponse(input);
  return response != null ? response :
            "I am sorry, I don't have a response to that."
;
}

/**
* Uses the frequency distribution to find an appropriate response. Each
* response found for each token in the input string is added to a
* HashMap.
* The response that contains the highest value, i.e. the most hits, is
* returned.
* /
private String generateResponse(String input) {
  String[] inputTokens = NLP.tokenize(input);
  HashMap<String, Double> responseDistribution = new HashMap<String,
               Double>();

  // find all responses for each token
  for (String inputToken : inputTokens) {
    HashSet<WordResponse> tokenResponses = frequencyDistribution
                             .get(inputToken.toLowerCase());
    if (tokenResponses != null) {
      for (WordResponse response : tokenResponses) {
        Double responseValue = responseDistribution.get(response
                              .getResponse());
        Double newValue = response.getValue();
        if (responseValue != null) {
          newValue += responseValue;
        }
        responseDistribution.put(response.getResponse(), newValue);
      }
    }
  }

  String bestResponse = null;
  Double bestResponseValue = (double) -1;

  // go through each entry in the HashMap and find the response with the
  // highest value
  for (Map.Entry<String, Double> entry : responseDistribution.entrySet()) {
    if (entry.getValue() >= bestResponseValue) {
      bestResponseValue = entry.getValue();
      bestResponse = entry.getKey();
    }
  }

  return bestResponse;
}
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;

public class SubtitlesWorldParser {

    private int numberOfDownloads = 0;
    private HashSet<String> alreadyFinishedUrls;

    public void downloadAllSubtitles() throws IOException {
        System.out.println("Downloading subtitles...");
        ArrayList<String> alphabeticalUrls = getAlphabeticalUrls();
        alreadyFinishedUrls = getAlreadyFinishedUrls();

        File alreadyFinishedUrlsFile = new File(Constants.DATA_DIR + File.separator + Constants.FINISHED_URL_FILE_NAME);
        FileWriter fileWriter = new FileWriter(alreadyFinishedUrlsFile, true);
        BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

        for (String url : alphabeticalUrls) {
            if (!alreadyFinishedUrls.contains(url)) {
                downloadAllSubtitles(url, bufferedWriter);
                bufferedWriter.write(url);
                bufferedWriter.newLine();
                bufferedWriter.flush();
            } else {
                System.out.println("Already completed "+ url);
            }
        }

        try {
            bufferedWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Returns a list of the URLs already visited.
     *
     */
    private HashSet<String> getAlreadyFinishedUrls() {
        HashSet<String> finishedUrls = new HashSet<String>();
        BufferedReader br = null;
        File finishedUrlsFile = new File(Constants.DATA_DIR + File.separator
                + Constants.FINISHED_URL_FILE_NAME);

        try {
            br = new BufferedReader(new FileReader(finishedUrlsFile));
            String line;
            while ((line = br.readLine()) != null) {
                finishedUrls.add(line);
            }\n        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        return finishedUrls;
    }
}
if (finishedUrlsFile.exists()) {
    try {
        String currentLine;
        br = new BufferedReader(new FileReader(finishedUrlsFile));
        while ((currentLine = br.readLine()) != null) {
            finishedUrls.add(currentLine);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (br != null) {
            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    } else {
        try {
            finishedUrlsFile.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return finishedUrls;
}

private void downloadAllSubtitles(String url, BufferedWriter bufferedWriter)
    throws IOException {
    System.out.println("Downloading all subtitles from " + url);
    Document doc = Jsoup.connect(url).get();
    Elements movieLinks = doc.getElementById("listing_container").select("a[href]");

    for (Element movieLink : movieLinks) {
        try {
            String subtitleUrl = movieLink.attr("href");
            if (!alreadyFinishedUrls.contains(subtitleUrl)) {
                try {
                    downloadSubtitle(subtitleUrl);
                    bufferedWriter.write(subtitleUrl);
                    bufferedWriter.newLine();
                    bufferedWriter.flush();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            } else {
                System.out.println("Already downloaded ", subtitleUrl + ", skipping.");
            }
        } catch (NoSubtitleFoundException e) {
            e.printStackTrace();
        }
    }
private void downloadSubtitle(String movieUrl) throws IOException, NoSubtitleFoundException {
    Document doc = Jsoup.connect(movieUrl).get();
    Element container = doc.getElementById("listing_container");
    if (container != null) {
        Elements subtitleLinks = container.select("a[href]");
        if (subtitleLinks.size() < 1) {
            throw new NoSubtitleFoundException("No subtitle found on " + movieUrl);
        }
        for (Element subtitleLink : subtitleLinks) {
            if (subtitleLink.attr("name").equals("English")) {
                UrlDownload.downloadFileFromUrl(subtitleLink.attr("href"),
                        Constants.SUBTITLES_DIR);
                numberOfDownloads++;
                break;
            }
        }
    } else {
        System.err.println("listing_container not found at " + movieUrl);
    }
}

public ArrayList<String> getAlphabeticalUrls() throws IOException {
    ArrayList<String> urls = new ArrayList<String>();
    Document doc = Jsoup.connect(Constants.LIST_URL).get();
    Elements links = doc.getElementById("listing").children();
    for (Element link : links) {
        // get the value from href attribute
        String url = link.attr("href");
        urls.add(url);
    }
    return urls;
}

public int getNumberOfDownloads() {
    return numberOfDownloads;
}
Appendix C    SubRipFileAnalyzer.java

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.HashSet;

public class SubRipFileAnalyzer {

    private static final long MAX_TIME_DIFF = 2000;

    private Subtitle currentSubtitle = null;
    private Subtitle nextSubtitle = null;

    /**
     * Reads the given subtitle file and adds to the frequency distribution.
     * Each subtitle is added as a response to each word of the previous subtitle.
     * The response is weighted as a factor of the subtitles length.
     * /
     * public HashMap<String, HashSet<WordResponse>> analyze(File subtitleFile,
     * HashMap<String, HashSet<WordResponse>> frequencyDistribution) {
     * BufferedReader br = null;
     * try {
     *     br = new BufferedReader(new FileReader(subtitleFile));
     * } catch (FileNotFoundException e) {
     *     System.err.println(subtitleFile.getName() + " not found.");
     *     e.printStackTrace();
     *     return frequencyDistribution;
     * }
     * boolean firstRun = true;
     * currentSubtitle = null;
     * nextSubtitle = null;
     * String numberString;
     * int line = 0;
     * try {
     *     while ((numberString = br.readLine()) != null) {
     *         line++;
     *         numberString = numberString.trim();
     *         if (numberString.equals("")) {
     *             // error in file, return
     *             return frequencyDistribution;
     *         }
     *         String subtitleString = "";
     *         String s;
     *         boolean newSubtitleFound = false;
     *         String nextSubtitleText = "";
     *         while (!((s = br.readLine()) == null || s.trim().equals(""))) {
     *             subtitleString += s.trim() + " ";
     *             if (s.equals("")) {
     *                 newSubtitleFound = true;
     *                 System.err.println("Blank line.");
     *                 continue;
     *             }
     *             String timeString = br.readLine();
     *             if (timeString == null) {
     *                 throw new Exception();
     *             }
     *             String subtitleString2 = "";
     *             String s2;
     *             level2SubtitleFound = false;
     *             String nextSubtitleText2 = "";
     *             while (!((s2 = br.readLine()) == null || s2.trim().equals(""))) {
     *                 subtitleString2 += s2.trim() + " ";
     *                 if (s2.equals("")) {
     *                     level2SubtitleFound = true;
     *                     System.err.println("Blank line.");
     *                     continue;
     *                 }
     *                 String timeString2 = br.readLine();
     *                 if (timeString2 == null) {
     *                     throw new Exception();
     *                 }
     *             }
     *             String subtitleString3 = "";
     *             String s3;
     *             level3SubtitleFound = false;
     *             String nextSubtitleText3 = "";
     *             while (!((s3 = br.readLine()) == null || s3.trim().equals(""))) {
     *                 subtitleString3 += s3.trim() + " ";
     *                 if (s3.equals("")) {
     *                     level3SubtitleFound = true;
     *                     System.err.println("Blank line.");
     *                     continue;
     *                 }
     *                 String timeString3 = br.readLine();
     *                 if (timeString3 == null) {
     *                     throw new Exception();
     *                 }
     *             }
     *             if (newSubtitleFound) {
     *                 currentSubtitle = createSubtitle(subtitleString);
     *                 nextSubtitle = createSubtitle(subtitleString2);
     *                 nextSubtitle = createSubtitle(subtitleString3);
     *             }
     *             newSubtitleFound = false;
     *             level2SubtitleFound = false;
     *             level3SubtitleFound = false;
     *             if (currentSubtitle != null) {
     *                 currentSubtitle.addResponse(new WordResponse(subtitleString,
     *                     level2SubtitleFound ? nextSubtitle : null,
     *                     level3SubtitleFound ? nextSubtitle : null));
     *                 currentSubtitle = null;
     *             }
     *             if (nextSubtitle != null) {
     *                 nextSubtitle.addResponse(new WordResponse(nextSubtitleText,
     *                     currentSubtitle, null));
     *                 nextSubtitle = null;
     *             }
     *         }
     *     }
     * }
     * return frequencyDistribution;
     * }
     *
     * private Subtitle createSubtitle(String subtitleString) {
     *     Subtitle subtitle = new Subtitle();
     *     String[] words = subtitleString.split(" ");
     *     for (String word : words) {
     *         System.err.println("Creating response for "+ word);
     *         subtitle.addResponse(new WordResponse("" + word));
     *     }
     *     return subtitle;
     * }
     *
     * private void printSubtitle(Subtitle subtitle) {
     *     System.err.println("Subtitle: "+ subtitle.toString());
     * }
     *
     * private void printWordResponse(WordResponse response) {
     *     System.err.println("Response: "+ response.toString());
     * }
     *
     * public static void main(String[] args) {
     *     File subtitleFile = new File("subtitle_file.txt");
     *     HashMap<String, HashSet<WordResponse>> frequencyDistribution = new HashMap<>();
     *     HashMap<String, HashSet<WordResponse>> frequencyDistribution2 = new HashMap<>();
     *     HashMap<String, HashSet<WordResponse>> frequencyDistribution3 = new HashMap<>();
     *     HashMap<String, HashSet<WordResponse>> frequencyDistribution4 = new HashMap<>();
     *     frequencyDistribution.put("Subtitle 1", new HashSet<WordResponse>());
     *     frequencyDistribution2.put("Subtitle 2", new HashSet<WordResponse>());
     *     frequencyDistribution3.put("Subtitle 3", new HashSet<WordResponse>());
     *     frequencyDistribution4.put("Subtitle 4", new HashSet<WordResponse>());
     *     try {
     *         SubRipFileAnalyzer analyzer = new SubRipFileAnalyzer();
     *         analyzer.analyze(subtitleFile, frequencyDistribution);
     *         analyzer.analyze(subtitleFile, frequencyDistribution2);
     *         analyzer.analyze(subtitleFile, frequencyDistribution3);
     *         analyzer.analyze(subtitleFile, frequencyDistribution4);
     *     } catch (IOException e) {
     *         System.err.println("Error: "+ e.getMessage());
     *     }
     * }
     */
}

public class Subtitle {

    private final String title;
    private final long startTime;
if (s.trim().startsWith("-")) && subtitleString.length() > 0) {
    newSubtitleFound = true;
    nextSubtitleText = s;
} else {
    subtitleString += s + " ";
}
}
long startTime = parse(timeString.split("-->")[0]);
long stopTime = parse(timeString.split("-->")[1]);
int number;
number = Integer.parseInt(numberString);
nextSubtitle = new Subtitle(number, startTime, stopTime, subtitleString);
if (firstRun) {
    firstRun = false;
    currentSubtitle = nextSubtitle;
} else {
    addToFrequencyDistribution(frequencyDistribution);
    if (newSubtitleFound == true) {
        nextSubtitle = new Subtitle(number, startTime, stopTime, nextSubtitleText);
        addToFrequencyDistribution(frequencyDistribution);
    }
}
} catch (IOException e) {
    System.err.println("An error occurred in " + subtitleFile.getName() + " on line " + line + " (IO)");
    // e.printStackTrace();
} catch (NumberFormatException e) {
    System.err.println("An error occurred in " + subtitleFile.getName() + " on line " + line + " (NumberFormat)");
    // e.printStackTrace();
} catch (IndexOutOfBoundsException e) {
    System.err.println("An error occurred in " + subtitleFile.getName() + " on line " + line + " (IndexOutOfBoundsException)");
    // e.printStackTrace();
} catch (Exception e) {
    System.err.println("An error occurred in " + subtitleFile.getName() + " on line " + line + " (?)");
    // e.printStackTrace();
}
} finally {
    try {
        br.close();
    } catch (IOException e) {
        System.err.println("Unable to close reader.");
        e.printStackTrace();
    }
}

return frequencyDistribution;

/**
 * Checks that the two subtitles are within the largest allowed time period.
 * */
private boolean isTimeOk() {
    long nextStartTime = nextSubtitle.startTime();
    long currEndTime = currentSubtitle.stopTime();
    long diff = nextStartTime - currEndTime;
    return diff < MAX_TIME_DIFF;
}

/**
 * Checks if the given subtitle is OK to use.
 * *
 * private boolean subtitlesAreOk() {
 *     String nextSub = nextSubtitle.getSubtitle();
 *     if (currentSubtitle == null) {
 *         return false;
 *     } else if (nextSub == null) {
 *         return false;
 *     } else if (nextSub.trim().equals("")) {
 *         return false;
 *     } else if (nextSub.trim().startsWith("[") &&
 *         nextSub.trim().endsWith("]")) {
 *         return false;
 *     }
 *     return true;
 * }

/**
 * Parse the time in the subtitle file.
 * *
 * private static long parse(String in) {
 *     String[] split = in.split(":");
 *     long hours = Long.parseLong(split[0].trim());
 *     long minutes = Long.parseLong(split[1].trim());
 *     long seconds = Long.parseLong(split[2].split(",")[0].trim());
 *     long millies = Long.parseLong(split[2].split(",")[1].trim());
 *     return hours * 60 * 60 * 1000 + minutes * 60 * 1000 + seconds * 1000
 *         + millies;
 * }

/**
 * Links the subtitles in the frequency distribution.
 * *
 * private void addToFrequencyDistribution(
 *     HashMap<String, HashSet<WordResponse>> frequencyDistribution) {
 *     if (isTimeOk()) {
 *         if (subtitlesAreOk()) {
 *             addToFrequencyDistribution(frequencyDistribution,
 *                 currentSubtitle.getSubtitle(), nextSubtitle.getSubtitle());
 *             currentSubtitle = nextSubtitle;
 *             nextSubtitle = null;
 *         }
 *     } else {
 *         currentSubtitle = nextSubtitle;
 *     }
 * }
"
```java
private void addToFrequencyDistribution(
    HashMap<String, HashSet<WordResponse>> frequencyDistribution,
    String currSubtitleText, String nextSubtitleText) {
    String[] tokens = NLP.tokenize(currSubtitleText);
    double size = tokens.length;
    WordResponse wordResponse = new WordResponse(nextSubtitleText, 1.0 / size);
    for (String token : tokens) {
        HashSet<WordResponse> responses = frequencyDistribution.get(token);
        if (responses == null) {
            responses = new HashSet<WordResponse>();
        }
        responses.add(wordResponse);
        frequencyDistribution.put(token, responses);
    }
}
```
import java.io.Reader;
import java.io.StringReader;
import java.util.List;
import edu.stanford.nlp.ling.CoreLabel;
import edu.stanford.nlp.process.CoreLabelTokenFactory;
import edu.stanford.nlp.process.LexedTokenFactory;
import edu.stanford.nlp.process.PTBTokenizer;

public class NLP {

    public static String[] tokenize(String input) {
        input = input.replaceAll("[^a-zA-Z ]", "").toLowerCase();
        Reader reader = new StringReader(input);
        LexedTokenFactory<CoreLabel> tokenFactory = new CoreLabelTokenFactory();
        PTBTokenizer<CoreLabel> ptbt = new PTBTokenizer<CoreLabel>(reader,
            tokenFactory, "");
        List<CoreLabel> tokens = ptbt.tokenize();
        int size = tokens.size();
        String[] tokenStrings = new String[size];
        for (int i = 0; i < size; i++) {
            tokenStrings[i] = tokens.get(i).toString();
        }
        return tokenStrings;
    }
}

Appendix D  NLP.java

import java.io.Reader;
import java.io.StringReader;
import java.util.List;
import edu.stanford.nlp.ling.CoreLabel;
import edu.stanford.nlp.process.CoreLabelTokenFactory;
import edu.stanford.nlp.process.LexedTokenFactory;
import edu.stanford.nlp.process.PTBTokenizer;

public class NLP {

    public static String[] tokenize(String input) {
        input = input.replaceAll("[^a-zA-Z ]", "").toLowerCase();
        Reader reader = new StringReader(input);
        LexedTokenFactory<CoreLabel> tokenFactory = new CoreLabelTokenFactory();
        PTBTokenizer<CoreLabel> ptbt = new PTBTokenizer<CoreLabel>(reader,
            tokenFactory, "");
        List<CoreLabel> tokens = ptbt.tokenize();
        int size = tokens.size();
        String[] tokenStrings = new String[size];
        for (int i = 0; i < size; i++) {
            tokenStrings[i] = tokens.get(i).toString();
        }
        return tokenStrings;
    }
}
