VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
INFORMATION TECHNOLOGIES STUDY PROGRAM

Problem-based project

Lexicon-Based Sentiment Analysis Using Plutchik's Model

Done by:
Domas Janiūnas
Anupras Krištapavičius
Arsenij Nikulin

Supervisor:
dr. Agnė Brilingaitė

Vilnius
2024

Contents

Abstract

Sentiment analysis, sometime referred to as opinion mining, is a branch of natural language processing and is responsible for extracting the emotions of a given text. It is usually used to research customer opinions or analyse product review, although it can be used for many different purposes like social media monitoring. There already exists many solutions for this problem, but most of them only offer results where sentiment is either positive, negative or neutral. The goal of this project is to create a lexicon and rule based sentiment analysis application which classifies the text using eight different emotions. The eight emotions were proposed by psychologist R. Plutchik [12] and are joy, sadness, trust, disgust, fear, anger, surprise and anticipation. This model was chosen because it has better granularity for the results and also provides polarity between the emotions (e.g. joy is opposite of sadness) which simplifies the analysis process. For analysis, a custom system based on the ideas of VADER analysis model was created. VADER was chosen, because it already was able to take negation and amplification into account which makes the results more accurate. The proposed design was implemented as a command-line application written in C++ with a local custom database in JSON format.

Santrauka

Žodynu Pagrįsta Jausmų Analizė Naudojant Plutchiko Modelį

Emocinė analizė yra natūralios kalbos apdorojimo šaka, kuri išskiria emocijas iš suteikto teksto. Dažniausiai ji yra naudojama ištirti klientų nuomonę ar išanalizuoti produkto apžvalgą, tačiau gali būti naudojama ir kitoms paskirtims kaip socialinių tinklų priežiūrai. Šiuo metu jau egzistuoja daug šios problemų sprendimų, bet didžioji dalis jų emocijas išskirsto į pozityvias, negatyvias arba neutralias. Šio projekto tikslas yra sukurti leksikonu ir taisyklėmis pagrįstą analizės programėlę, kuris suklasifikuos tekstą į aštuonias skirtingas emocijas. Šios aštuonios emocijos yra pagrįstos psichologo R. Plutchiko [12] tyrimu, kuriame jis išskyrė *(džiaugsmą, pasitikėjimą, baimę, nuostabą, liūdesį, pasibjaurėjimą, pyktį, laukimą)*. Šis modelis buvo pasirinktas, nes jis yra pakankamai detalus bei turi priešingas emocijas (pvz. džiaugsmas yra priešingas liūdesiui), o tai supaprastina analizės procesą. Analizei atlikti yra pasitelkiama modifikuota sistema, kurios sukūrimas pagrįsta VADER analizės modeliu. VADER analizės modelis buvo pasirinktas, nes jis jau gali susidoroti su neigimu bei emocijas sustiprinančias žodžiais, o tai suteikia rezultatams tikslumo. Pasiūlyta sistema buvo įgyvendinta kaip komandinės eilutės programėlė parašyta C++ kalba su lokalia duomenų baze JSON formatu.

Introduction

In the current times, the amount of data being uploaded to the internet is growing exponentially and the insurmountable amount of data is becoming harder and harder to analyse by humans. Therefore, this conundrum requires help from computers and automated systems. One of the spheres, where this help could be applied, is sentiment analysis. Sentiment analysis is a process of understanding and classifying the emotions expressed in a text. The emotional classification of text could be used for many different purposes: to quickly analyse and summarise product reviews and customer opinions; to monitor forums and social media for toxic behaviour; to analyse and accordingly adjust public speeches or presentations and etc.

There are many already existing solutions for solving this problem but in this paper we aim to explain and showcase a new lexicon-based and rule-based solution which uses eight different emotions for classifying the text. The analysis will be done using the ideas of VADER [7] analysis model adapted to eight basic emotions proposed by psychologist Robert Plutchik [12] which are joy, sadness, trust, disgust, fear, anger, surprise and anticipation. Although using a lexicon-based approach will yield less accurate results than using a machine-learning method, the aim of this solution is to create a guide for humans and not a very accurate and definitive decision maker. The final created product is a command line application which takes in text in form of a file or web page link and returns easy-to-understand and interpret results.

This paper outlines the thought process, design and implementation of our sentiment analysis project. The thought process section describes and explains the choices made, more specifically, why we use the emotions proposed by R. Plutchik and why we adapt the VADER model. In the design segment the general ideas and structure of the application is described, while implementation section presents our specific realisation of the proposed design.

1 Related Works

Sentiment analysis [9] is a quite popular topic, with many different sections and models. There are two main approaches to sentiment analysis - lexicon-based and machine-learning. *Lexicon-based* approach uses dictionaries containing words and assigned emotional values to detect keywords in a text and calculate score by summing it up [14]. Some models like *Bag-of-Words (BOW)* simply looks for keywords [13], while some advanced models like *Valance Aware Dictionary for Emotional Reasoning (VADER)*[7] takes word order, negation, intensification and punctuation into account. *Machine-learning* approaches use different learning techniques and training data to analyse text [8]. Usually ML approach is much more accurate than *Lexicon-based*, since it is able to detect more nuanced details, but *Lexicon-based* approach is less complex and faster, requiring less resources with increased adaptability.

To do any sentiment analysis emotions need to be decided on. Often sentiment analysis services only offer positive and negative emotions, but it only gives a very general quick overview and these solutions lack any depth. Therefore, more complex emotional classification needs to be chosen and in this projects case Plutchik's wheel of emotions [12] is used.

Additionally sentiment analysis needs data to analyse. While it could be a simple input of raw text users may be interested in having ability to analyse what is said in a forum post or an article without needing to copy all the text manually. To deal with this a technique of web-scraping [4] needs to be used. Also analysis requires the databases to be filled and lexicon-based approach requires sentiment lexicons [**?**], which are dictionaries filled with keywords and their emotional scores.

Furthermore, natural language processing [3] needs the text to be seperated, or in other words tokenised [10].

2 Sentiment Analysis

Sentiment analysis is a branch of natural language processing interested in analysing the emotions and classifying the text. The analysis could only cover whether text is positive or negative, or it could go much deeper and analyse how strong are specific emotions in the text.

Sentiment analysis includes many distinct tasks like detecting emotions, opinions both obvious and subtle. Due to broadness of the task there many different approaches to achieve sentiment analysis that have their specific use-cases and have to be chosen accordingly. Furthermore, all methods can be categorised into statistical or machine-learning approaches.

2.1 Classification of Emotions

To start doing any type of sentiment analysis at first it is important to decide what kind of emotions are going to be analysed. The more emotions are taken into account, the more granular the results become, but at the same time complexity of sentiment analysis system increases. As human beings are able to express and feel many different emotions. Some are complex and may even be exclusive to a specific part of the human population. Thus, the basic emotions for the analysis should be carefully chosen.

The most popular and simple way is to just separate everything to positive, negative or neutral emotions. This approach has been done many times and lacks depth. Furthermore, another popular approach is done using the basic emotion classification presented by Jack E. et al.[6]. This idea includes four basic emotions: *happiness, sadness, fear/surprise* (fast approaching danger), *anger/disgust* (static danger). Alternatively one may choose psychologists Ekmans P. [11] conclusions that people show six basic emotions: *sadness, happiness, fear, anger, surprise, disgust*. The last two models have more depth, but they lack clear opposite emotions, which could cause problems in the implementation and increase complexity of the main analysis algorithm.

Finally, we singled out Plutchik's [12] Wheel of Emotions (see Figure 1), which defines eight basic emotions: *joy, trust, fear, surprise, sadness, disgust, anger, anticipation*. This approach has several benefits including clear opposite emotions and higher depth. Existence opposite emotion combinations (*joy-sadness, trust-disgust, fear-anger, surprise-anticipation)* allow to create an algorithm which would be capable of dealing with negations. Additionally this model allows for higher depth by having eight basic emotions and derived emotions. For example, *joy* and *trust* create *love* while *fear* and *disgust* create *shame*.

2.2 Lexicon-Based Approach

The lexicon or keyword-based approach is based on word and phrase recognition using sentiment dictionaries. A sentiment dictionary is a collection of keywords with assigned emotional values. Usually the assigned emotional values only include how positive or negative the word is, but sometimes they also contain specific emotions like joy, anger and sadness. To calculate the final result, values of all found keywords are summed up, then positive and negative (or other emotions) sums are compared to make final conclusions.

The main advantage of lexicon models is their low complexity, which results in faster analysis and lower resource usage. It also allows for quicker implementation of such models and easier manipulation of data sets for adapting to different types of text or even other languages. Finally, it

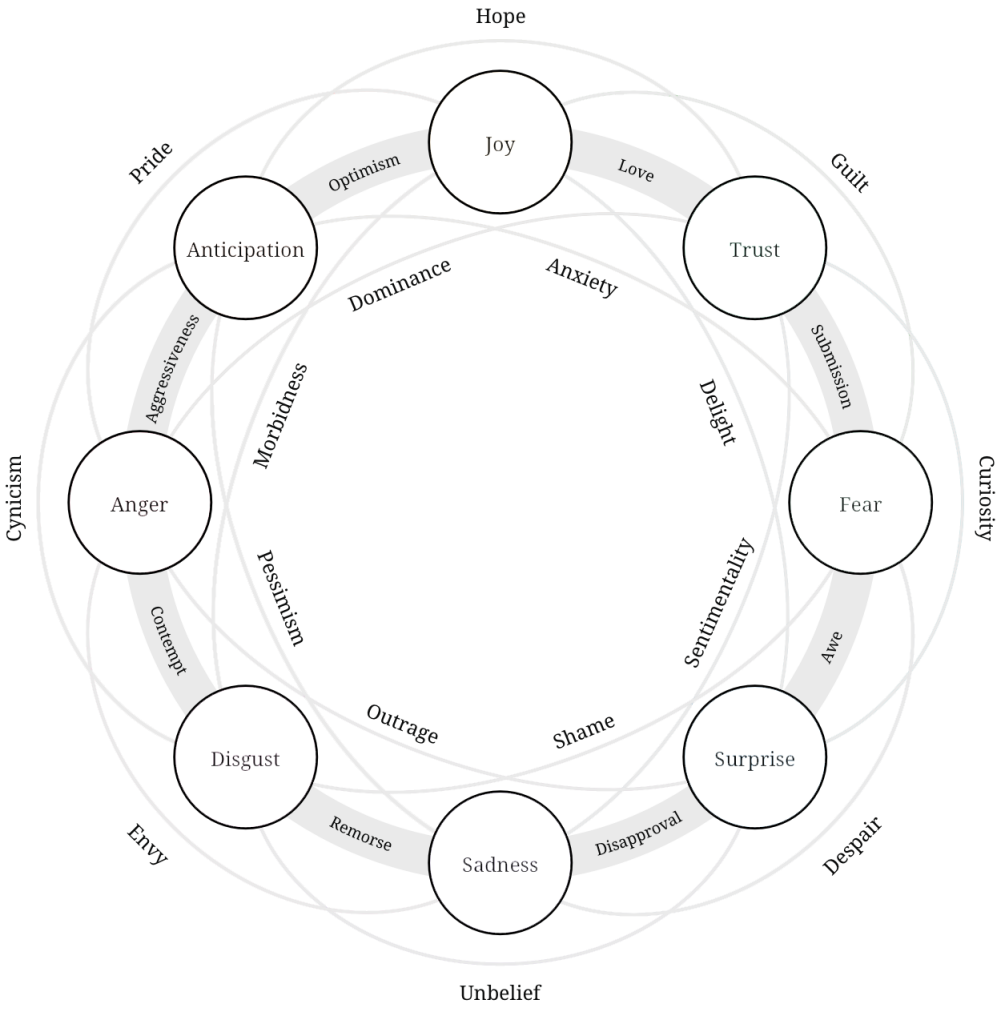

Figure 1. Interpretation of Plutchik's Wheel of Emotions Showcasing Different Dyads

is easier to understand how the analysis results were achieved and what problems or inaccuracies it might have.

They biggest disadvantage of keyword-based approaches is their inability to deal with nuances of human language [2]. This includes literary techniques like sarcasm, irony, complex sentence structures, it is also unable to take context into account and has difficulties with words that have several different meanings. For example, dictionary-based models would always count the word "death" although the death could be referencing the death of some trend and not a person. These problems mean that these models are much less accurate than machine-learning counterparts, although still usable in appropriate situations, when highest possible accuracy is not required.

2.3 Machine-Learning Approach

Machine-learning algorithms can be quite successfully used for sentiment analysis. There are many existing models and libraries, especially for python, for easier implementation. To help train the ML models same dictionaries that are used for lexicon-based approach can be used to combine both ML and lexicon-based approach into one for the best results.

The main advantage machine-learning models have over statistical approach is its accuracy in understanding natural language [8]. ML models achieve this by being able to understand the

context of a word and changing emotional values based on it. Moreover, ML can detect and more correctly interpret sarcasm or jokes, read "between the lines". This approach requires large amounts of training data to begin analysing the text, but the more training data is given, the more accurate the algorithm becomes. Finally, there are many ready-to-use models that just need to be implemented into the application and trained, although some of them even come pre-trained.

The accuracy of ML approach comes with its own price. These models require much more resources like computational power and training data. Large amounts of training data has to be prepared by humans and fed to the algorithm for it to be trained. In addition, a specialist has to constantly supervise the learning process and not let the ML stray away. Finally, the decision making of ML algorithms are not very clear and it is harder to analyse how the results were calculated.

2.4 Analysis Models

For this project we chose to use a lexicon-based approach, since we want to do interpretation of emotions we have chosen and keyword-based approach is more fit to be adapted to our needs. Furthermore, our product does not require very high accuracy and works more like a guide, meaning the added complexity of ML is not worth the extra accuracy. Lexicon-based approach has many different models meant for different purposes and accuracy levels, but two quite popular and fit for our purposes are BoW and VADER.

BoW (bag-of-words) - a simple model used for very shallow and general interpretation of the text. BoW model looks at the text as if it was just a collection words or a "bag of words", where their location is and nearby words do not matter [5]. To calculate the final results the emotional values of keywords multiplied by occurrences are summed up. While BoW is a good model for simple and quick sentiment analysis of texts positivity/negativity, but deeper analysis requires more advanced models.

VADER (Valence Aware Dictionary for Sentiment Reasoning) is a lexicon and rule-based model. It has similarities to BoW, since it searches for known keywords in a given text and calculates score based on keywords sentiment values. However, unlike BoW, VADER tries to take punctuation, negations and amplifiers (words like "very") into account, making the model much more accurate than BoW [7]. Each keyword in VADERs dictionary contains a positivity value from -1 to 1 and negation can invert the score, amplification can increase/decrease the score. Final score is calculated by summing up modified sentiment values of found keywords. Although this approach only incorporates positive and negative sentiment, we want to adapt the ideas of this model to our needs and using Plutchik's wheel of emotions and represent.

2.5 Existing Solutions and Our Take

There are many already existing solutions for the problem of sentiment analysis. For example, MonkeyLearn, one of the first results in google search for sentiment analysis, provides robust sentiment analysis tools for meant for brand-building. It incorporates sentiment analysis based on positive/negative values, allows to sort results based on topics and categories. Moreover, it provides a user-friendly UI very clearly showcasing results and change over time.

Some other tools have sentiment analysis incorporated as only a feature in its large catalog. For example, tech giant Amazon has it incorporated in analytics packet of AWS. Based on machine-learning and capable of doing many more tasks that just sentiment analysis. Furthermore, many

free libraries already exist for easier ML model implementation in your own code. Python has a large collection of libraries called Natural Language Toolkit (NLTK) which has large amounts of different models from ML to statistical, including VADER and BoW.

This presents a large issue for our project of how do we stand out, what do we do, that is special. During our research we noticed that most, if not all, of these solutions classify text as either positive, negative or neutral, but our aim is to go deeper. Our goal is to extract much more specific emotions from the text and find out why the text is positive or negative, is it happy or sad, angry or fearful and etc. The Plutchiks dyads allow for even deeper analysis, since it allows to make combinations of emotions, for example if calculations return, that the text is mostly happy and fearful, then it is possible the actual emotion is guilt. This makes our project stand out and offer a different type of analysis.

3 Design

3.1 Functional Requirements

The final product is a command-line application which meets the following functional requirements:

- The application must be UNIX-OS compatible and portable.
- Input to the application can be given by passing input as a parameter or once the launched application prompts the user for input.
- Should be able to take in three different kinds of input: plain text, text file or URL.
- If URL is supplied, should download websites HTML and scrape it for the text user wants analysed, omitting unnecessary data like advertisements or interface.
- The text should be analysed using either BoW or VADER model according to users choice. Analysis should give numerical scores using aggregated values. Emotion distribution and emotional flow through the text should also be calculated.
- The application should be able to analyse a sentiment around a specific user chosen topic inside of the given text instead of analysing the whole text.
- The application should write logs for general application flow showcasing input, flags, errors and other information which allows users and developers to detect what when wrong.
- The application should write logs for analysis flow showcasing what parts of the text are being analysed, intermediate scores, what keywords, prefixes and other value modifying elements were detected.
- The user should be able to choose output formatting between nicely formatted text with emotion names, total scores and distribution percentages or comma-separated values showing the total scores.
- The user should be able to choose output destination between terminal, text file or PDF file.
- In case of PDF output, graphs showcasing score distribution and emotional flow should be generated.
- For different input, output and analysis settings, command line arguments should be used.
- The database should be a local file which can be easily updated, imported/exported or adjusted by the user or developer for different situations.

3.2 Architecture

The system is a single local application and does not require connection to any outside servers. All the features are handled inside the application by four main modules: input, analysis, database and output. The input module is responsible for handling all user input. Three accepted types of input are: plain text, text file and an URL. In case of the URL input, the system must must scrape the website for the text the user wants analysed. Another responsibility of the input module is to tokenise the text after the input was handled. The text is separated into paragraphs, sentences and individual words and then passed to analysis module (see Figure 2).

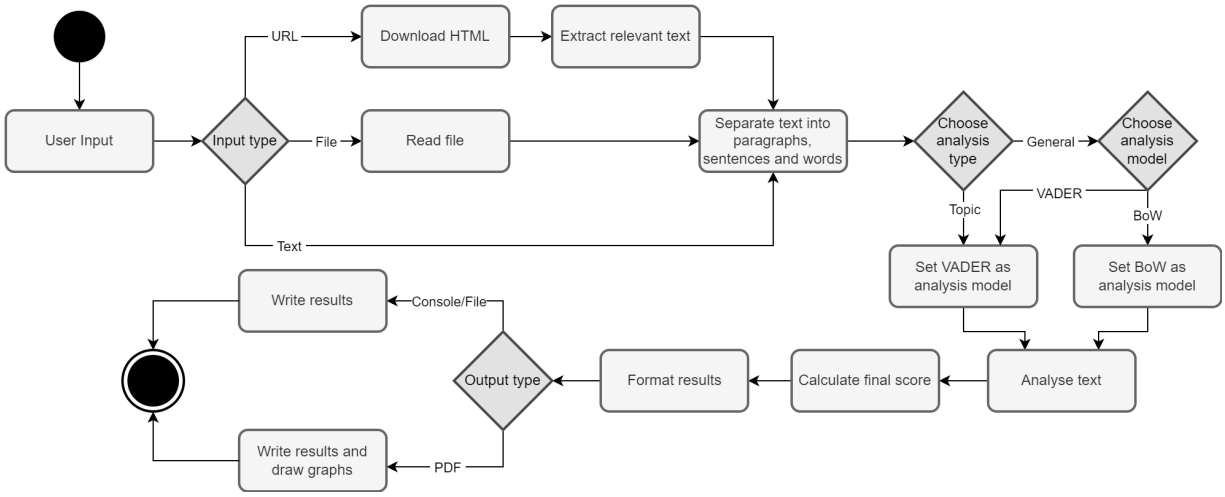

Figure 2. Flow of The Application

Analysis module works together with the database. The analyser is responsible for going through the tokenised text, calculating and keeping scores. To detect keywords during the analysis it makes calls to the database and request to find specific keywords, affixes and other information used to calculate the score. The database is responsible for reading and keeping the data used as a sentiment dictionary. Moreover, it takes care of searching for keywords and other data. At the end of the analysis the calculated scores are passed on to output module.

The final step of the execution is formatting and delivering the output. This is handled by the output module. It is responsible for interpreting the results, calculating distribution and formatting them to be understood by humans. The output can be delivered by three main ways: terminal, text file or PDF document.

3.3 Database

Sentiment analysis using a lexicon-based approach requires a sentiment dictionary, which provides keywords and their emotional values. While there are many dictionaries, like SentiWordNet [1], which are very popular and extensive, but all of them are meant for analysis of whether the text is positive or negative. This means the project requires a custom database to be created for it to work with eight emotions. This database would work as a sentiment dictionary containing keywords and their assigned emotional scores. Additionally, the VADER analysis can take negation and amplification into account, therefore the database must also contain affixes, negations, amplifying and dampening words.

3.3.1 Storing Keywords and Emotional Scores

The most important part of the database is the keywords. One option for storing the keywords is to store every single possible form of a word and have a single score assigned to all of them. This method allows for a fast search using a hash table or similar technology, but storing every single form causes some issues. There would be too much redundancy, making the database very large and inefficient. Moreover, adding new keywords to the database would cause issues, since you would need to make sure to add every possible form.

To address these issues, we chose to store only the root of each word and use separately stored prefixes to modify the score. Although, this approach requires to search for keywords using iterative search, it also significantly simplifies the insertion and storage of keywords. To avoid incorrect keywords being detected, the database also stores all English prefixes which are used in cases when the root is found in the middle of the word, where it is checked if the part before the root is a known prefix. This helps avoid such cases as a keyword root *'ill'* being detected in the word *'will'*.

Every keyword (root) is assigned an emotional score which is made up of eight real numbers from 0 to 1, where each number represents those words' correlation with each of the eight emotions (see Appendix A). Since each emotion has its opposite emotion, this could be done using four values from -1 to 1. This approach would save space but would also cause problems with the positive and negative emotions cancelling each other out. For example, if the text contains many happy words, but also many sad words, the result would be that there are very few emotions in the text. Therefore, each emotion's score is stored separately.

3.3.2 Storing Affixes, Negations and Amplifiers

For storing affixes there are two main sections: general prefixes and score-modifying affixes. General prefixes is a list that includes every English prefix and is used to detect the root of the word correctly as mentioned before. Score-modifying affixes is a list of all prefixes and suffixes that can somehow affect the keywords emotional score. Some affixes (e.g. *'un-'*) can have a negating effect, inverting the emotional scores (e.g. joy's score becomes sadness' score). Other affixes (e.g. *'-ier'*) can have a dampening or amplifying effect, where each emotion's score for that keyword is increased or decreased by a multiplier. Every score modifying affix has a multiplier assigned, where -1 means the affix is negating and values between 0 and 2 are assigned to dampening/amplifying affixes (see Appendix A).

Negating, amplifying, and dampening words work similarly to affixes. Negating words (e.g. *'not'*) are assigned the multiplier of -1, whereas dampening or amplifying words (e.g. *'very'*) are assigned a multiplier between 0 and 2. Because these words are short and usually only one form, they are stored in full, instead of just the root. This allows for fast detection since hash tables can be used for searching for it in the database.

For the purposes of faster analysis, common words (e.g. articles *'a', 'the'*) are stored in the database. The common words list is much shorter than keyword list and stores full words, thus hash table can be used, so it is faster to check if the word should be skipped without searching for it in the keyword list. Additionally, for more accurate analysis, connecting words (e.g. *'and'*) are also stored. They allow to detect negation or amplification much further away from the keyword.

3.4 Analysis

3.4.1 VADER Analysis Model

The original VADER model is meant for analysis using positive/negative values, meaning the already existing implementations and libraries cannot be used, so for the purposes of this project we had to create our own implementation adapting the ideas of the VADER model for our purpose.

```
   Input: Array of words (sentence)
   Output: Array of emotional scores
 1 function calculateScore(words)
 2     declare real array totalScore[8] = [0, 0, 0, 0, 0, 0, 0, 0]
 3     for i = 0 to wordCount − 1 do
 4         foundKey ← findKeyword(words[i])
 5         if foundKey is null then
 6             continue
 7         keyScore ← foundKey.score
 8         keyScore ← keyScore × findA   x(words[i])
 9         keyScore ← keyScore × findMultiplier(words, i)
10         totalScore ← totalScore + keyScore
11     if sentence ends with "!" then
12         totalScore ← totalScore × 1.5
13     return totalScore
14
15 function findMultiplier(words, index)
16     multiplier ← 1
17     n ← 2
18     for j = 0 to n do
19         if index − j < 0 then
20             break
21         if isConnectingWord(words[index − j]) then
22             n ← n + 1
23             continue
24         amplifier ← findAmplifier(words[index − j])
25         if amplifier is not null then
26             multiplier ← multiplier × amplifier.multiplier
27         else
28             multiplier ← multiplier × findNegation(words[index − j])
29     return multiplier
```

Algorithm 1: VADER Analyser Algorithm

Algorithm 1 consists of two primary functions: *calculateScore* and *findMultiplier*, designed to analyse an array of words and return an array of emotional scores. The details of each function are explained below.

The *calculateScore* function accepts an array of words as input and aims to analyse the emotional content of the sentence. It initializes a real number array, *totalScore*, with eight zeros,

representing scores for each of the eight basic emotions. The algorithm iterates through each word in the given sentence, checking if the word is defined as a keyword in the emotional database. If a keyword is found, it proceeds to calculate a *keyScore* by considering affixes get from the *findAffix* function and multiplies it by the corresponding multiplier get from the *findMultiplier* function. The resulting *keyScore* is then added to the *totalScore*. If no keyword is found, the algorithm continues to the next word. Before returning the *totalScore*, the algorithm checks if the sentence ends with an exclamation mark. If it does, the *totalScore* is multiplied by a constant *exclMultiplier*. The final emotional scores are then returned.

The *findMultiplier* function takes an array of words and the index of a keyword found in the sentence. It checks neighboring words to determine if they influence the emotional score. The algorithm initializes two local variables: *multiplier*, with a default value of 1 and *n* with a default value of 2. A for loop iterates through neighboring words, ensuring the index is within bounds. For each word, it checks if it is defined as a connecting word by searching match in *isConnectingWord* function. If it true, then the search will be extended by increasing *n* variable by 1 and will go to new loop iteration. If connecting word not found, then searches amplifying word in the emotional database using *findAmplifier* function. If found, the *multiplier* is changed by the amplifier's multiplier. If the word is not an amplifier, the algorithm, by calling *findNegation* function, checks if it is a negating word and adjusts the multiplier. The final *multiplier* is returned to be used in the calculation of the emotional scores.

Functions *findKeyword, findAffix* and *findMultiplier* search for the given parameter in the database returning the found object or null for *findKeywords* functions or 1 for other functions if the parameter does not exist in the database.

3.4.2 Real World Example

In this segment the analysis process will be showcased by analysing the given text:*"The weather was gloomy, she was very unhappy."* step by step. Before the analysis begins, the text is separated into an array of words:[*"The", "weather", "was", "gloomy", "she", "was", "very", "unhappy", "."*]. Also at the beginning of *calculateScore* an array $totalScore$ is initialised.

```
The | weather | is | gloomy | she | is | very | unhappy |.
totalScore = [0, 0, 0, 0, 0, 0, 0, 0]
```

The *for* loop is started with $i = 0$ and that word is searched for in the database with *findKeyword*. The loop keeps repeating until a keyword is found. Therefore, the code iterates until $i = 3$ and the word is *gloomy*. It is defined in the database, so its score gets assigned to *keyScore*. Also it is checked if it has an affix, that is defined in the database, but in this case it does not, so *keyScore* is not impacted.

```
The | weather | is | gloomy | she | is | very | unhappy |.
totalScore = [0, 0, 0, 0, 0, 0, 0, 0]
keyScore = [0, 0.8, 0, 0, 0 ,0 ,0 ,0]
```

The next step is to find out whether the keyword has any multiplying or negating words in front of it. So the word *is* is checked and it is one of the linking words so the search for multiplying or negating words is extended by one. Also *weather* and *The* are checked, but they are not defined in the database, so *keyScore* is added to the *totalScore*.

```
The | weather | is | gloomy | she | is | very | unhappy |.
totalScore = [0, 0,8, 0, 0, 0, 0, 0, 0]
```

After that the loop goes back to $i = 3 + 1$ (one after the last keyword) and searches for the next keyword. In this example it is the word *unhappy*. Its' score gets assinged to the *keyScore*, but the word has an affix *un-* so, its score gets inverted. Therefore, 1 for *joy* turns into 1 for *sadness*.

```
The | weather | is | gloomy | she | is | very | unhappy |.
totalScore = [0, 0,8, 0, 0, 0, 0, 0, 0]
keyScore = [0, 1, 0, 0, 0, 0, 0, 0]
```

Then it is checked if *unhappy* had any multiplying or negating words in front of it and it this case it does. The word *very* has a multiplying value of 1.5, so each value in the *keyScore* get multiplied by 1.5.

```
The | weather | is | gloomy | she | is | very | unhappy |.
totalScore = [0, 0,8, 0, 0, 0, 0, 0, 0]
keyScore = [0, 1.5, 0, 0, 0, 0, 0, 0]
```

As there were no other words that could impact the *keyScore* value it gets added to the *totalScore*. Then it is the end of a sentence so the punctuation is checked, but it is not a *"!"* so it does not multiply the final *totalScore*, which then is returned.

```
The | weather | is | gloomy | she | is | very | unhappy |.
totalScore = [0, 2.3, 0, 0, 0, 0, 0, 0]
```

3.4.3 Topic Analysis

Topic analysis is an additional feature of the application that gathers overall sentiment around a specified topic. This may come in handy when there is a need to parse long texts or many of them. Aforementioned solution is not perfectly accurate, but it helps to get general opinions as it finds in what kind of context the chosen topic is discussed in. For example, one may want to use it to compare how different news sites talk about the same event or a person and find if any bias is shown or to find a topic that raised more negative emotions, than usual and may need extra attention. Another way to use this product could be opinion mining about a product or a service by analysing users reviews.

Sentiment gathering around a topic is achieved by looking for specified topic keywords within a given text. If such keywords are found, the emotional score of that sentence is added to topics' emotional score. Also each topic may have strong keywords that directly describe the topic for example first and last name of a person. Yet in the next sentence the same person could be referred to using their pronouns, thus making those weak keywords. Weak keywords only are only checked for in the neighbouring sentences of a sentence that contains a strong keyword, as otherwise they may be describing a completely different person.

Also this is more of an advanced feature and the word order in the sentence matters, VADER analysis model should be used. For example some topic keywords are found in a sentence but a different topic is also mentioned somewhere in the same sentence. In this case, only the word next to the defined topic should be accounted for when calculating its' score and not the other emotions associated with a different topic. Therefore, BoW would lead to significantly higher inaccuracies and should be avoided.

4 Implementation

4.1 Tools

For this project we chose to use C++ for several reasons. Due to C++ being quite low-level it offers fast performance with a lot of control over resource management, allowing us to optimise the analysis process. Moreover, C++ supports object-oriented programming and has many libraries and documentation, making it convenient to work with.

The application relies on several dependencies. For web scraping process, *cURL* in library form *(libcurl)* is used to download websites HTML code which is then scraped. This choice was made, because cURL has all the required functionality for downloading HTML and is a common tool, making the application more accessible. Furthermore, for creating formatted PDF output we chose *"libHaru"* library, which allows to create PDF documents with custom graphs, which fits right into the defined functional requirements.

For building the project we use automated building tool *CMake*. It was chosen for its robustness, widespread use and cross-platform compatibility. In addition, for testing we use *Google Tests framework*, since it allows for many different test types and integrates well with *CMake*.

4.2 Database

To implement a local database, we chose JSON format for its widespread use and readability. To handle loading and storing the data a *Database* class was created. Additionally, To help manage keywords and their scores, affixes, negations and amplifiers custom *Keyword*, *Multiplier* and *EmotionalScore* classes were implemented.

4.2.1 JSON Parsing

Once the application is running, it expect a 'database.json' file in the same directory as the executable. If the default database file is not found, the application prompts the user to input a path to another database file. Additionally, the database file can be specified using *-d* flag. If the user specified file is not found, the code throws an error, and the application exits.

To parse the JSON file, we chose *nlohmann/json* library for its wide capabilities and easy-to-use interface. Once the JSON file is found, it is fully read using *std::wifstream* and the text is passed to *nlohmann::json::parse()*, which returns a *nlohmann::json* object that holds all the database information. Created object is then used to iteratively go through data in the database where all the data is put into correct objects and data structures inside the Database class.

4.2.2 Keywords, Multipliers and Emotional Scores

To avoid having to manually use an array or a vector to store and modify emotional scores we implemented a *EmotionalScore* class. This class stores an *std::vector<float>* with eight float values, one for each emotion in this order: joy, sadness, trust, disgust, fear, anger, surprise, anticipation.

To allow for easy modification of the values using multipliers, there is an *operator** overload, which takes in a float and multiplies every emotions score by that multiplier. So after the execution of the following code:

```
EmotionalScore score({1, 0, 3, 2, 6, 0, 5, 3.5});
score = score * 1.5f;
```

the score will be {1.5, 0, 4.5, 3, 9, 0, 7.5, 5.25}. To handle negation and avoid having negative values, every time the score values are modified *invertValues()* is called. This method checks if there are any negative values. If there are any, it takes the absolute values of each emotion and swaps it with the opposite emotions score. So after the following code:

```
EmotionalScore score({1, 0, 0, 1, 1, 0, 0, 1});
score = score * (-1);
```

the final value of *score* will be {0, 1, 1, 0, 0, 1, 1, 0}.

To store a keyword root and its score, we employ *Keyword* class. This class stores a *wstring* for the root and *EmotionalScore* for the score. These values are assigned at construction and cannot be modified later on. For affixes we use *Multiplier* class that stores *wstring* for the affix itself and *float* for the multiplier. Like the Keyword class, the Multiplier values are assigned at construction and cannot be modified later.

4.2.3 Database Class Structure and Searching

The database is implemented as a *Database* class which contains these private fields:

- std::vector<Keyword> keywords - stores every keyword's root and their score. Stored inside a vector, because root finding requires iterative approach.
- std::vector<Multiplier> affixes - stores every affix that can affect the emotional scores, and their multipliers. Stored inside a vector, because affix detection requires iterative approach.
- std::unordered_map<std::wstring, float> negations - stores every negation mapped to their multiplier. Unordered_map allows for faster search.
- std::unordered_map<std::wstring, float> amplifiers - stores every amplifier and dampeners together with their multiplier.
- std::unordered_set<std::wstring> connectingWords - stores every connecting word and ignored words used for faster search.
- std::unordered_set<std::wstring> generalPrefixes - stores every English prefix, used to more correctly detect the root.

The database has a method *findKeyword()* which takes in a string and searches for it in the vector of keyword roots. A *for* loop goes through every keyword in the database and using *wstring::find()* checks if the root in the database is a part of the given word. Once a match is found, the system checks if the part before the root (if it is found in the middle of the word) is a known prefix by calling isRealRoot(), which searches for the prefix in the general prefixes. If the prefix is found, the root is considered to be the correct one and the *Keyword* object is returned. If no match is found, an empty *Keyword* is returned. The search complexity for the prefix in the keyword is O(n) in the worst case scenario, while the prefix search is O(1) since *std::unordered_set* and *std::unordered_map* use hash tables.

Affixes are searched for using the iterative approach similar to keyword search. The *findAffix()* checks if the affix is at the start or the end of the word and returns the affix's multiplier, otherwise it returns 1. Both *findNegation()* and *findAmplifier()* matches the word fully and returns its multiplier or 1 if nothing was found. Searching for connecting words with *isConnectingWord()* returns either

true or false. Since connecting words are stored in an *std::unordered_set*, the search complexity is O(1), thus it is faster to check if a word is a connecting word and skip it, instead of searching for it in the keyword list.

4.3 Gathering Input

The main point of input segment is to gather the text that the user wants to analyse, tokenise it and store it inside a *Text* class object. This module takes in a string (plain text, file name or web link) and produces a Text object, which then can be used by analysis module. To achieve this there are three main classes: *TerminalInput, FileInput, WebInput*.

4.3.1 Terminal, Text File and Web Page Input

To gather user input there is an abstract class *InputManager* which has an abstract method *getInput()* and static method *separateText()* (see Figure 3). This class has three derived classes for each type of input. When the user launches an application, the application prompts for an input. That input can be either plain text, path to a file or a link to a website and the application chooses accordingly which implementation to use and assigns a new object of the corresponding class to a pointer to *InputManager*. Use of polymorphism simplifies the code allowing to use the pointer further on in the application without knowing which implementation is being used. Additionally, the input type can be chosen using flags *-f* for file and *-u* for a URL.

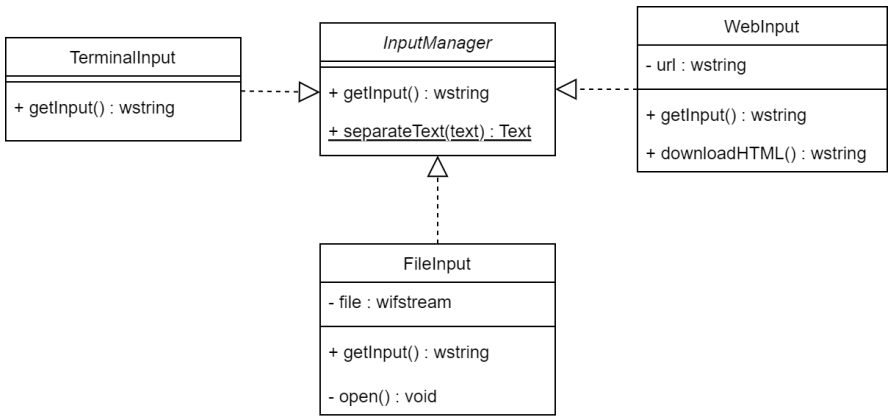

Figure 3. Input Management Class Diagram

The terminal input is most simple, once it is called, it prints out to the screen the prompt and using *std::getline()* reads from *std::wcin* into a string, which is then returned. This type of input is only meant for simple cases since it cannot handle new lines. For longer texts file input should be used. When instance of *FileInput* is being created, a file name must be passed to it, but if the application does not find or cannot open the file, an error is thrown. Once *getInput()* is called, full file contents are read into a *std::wstringstream* and constructed string is returned.

URL input is handled by *WebInput*. Object of this class takes in a URL and stores it. This implementation of *getInput()* calls *downloadHTML()*, which downloads websites HTML and then scrapes it. Scraping algorithms searches the HTML text for *<p>, <h1>* and other elements that contain text. The text is then extracted removing any left-over HTML symbols and returned. To download the website's HTML, we are using *libcurl* library which provides a robust and easy-to-use interface for downloading HTMLs and debugging. The only limitations of this approach is that it cannot handle dynamic content (e.g. social media pages that scroll forever).

4.3.2 Text tokenisation

Lexicon-based sentiment analysis requires to separate the text into tokens to simplify the analysis process. Therefore, once the input is gathered and stored in a string, it is passed to a static *Input-Manager* method *separateText()*. The tokenisation process separates the text into three different levels: paragraphs, sentences and words.

To store the tokenised text, we employ three related classes: *Text, Paragraph, Sentence* (see Figure 4). A *Sentence* object holds two *std::vector<std::wstring>* instances, one for every word of that sentence and other for all lower-case version of each word. Additionally, the last element in the vectors is the sentence end punctuation. Moreover, every object also holds an *EmotionalScore* object representing that sentence's emotional score, which initially is all set to zero and calculated value is assigned during analysis process. A *Paragraph* object holds a vector of sentences and its own emotional score. This score is set to zero at the start and calculated value, which is a sum of all sentences' scores, is assigned during analysis process. Finally, all paragraphs and the text's total score is stored inside a *Text* object. To avoid excessive value copying during separation process and later storage, every vector is actually a *std::shared_ptr<std::vector<...>>*. Using a smart pointer allows to pass large amounts of text by reference instead of copying the values and it also does automatic memory management.

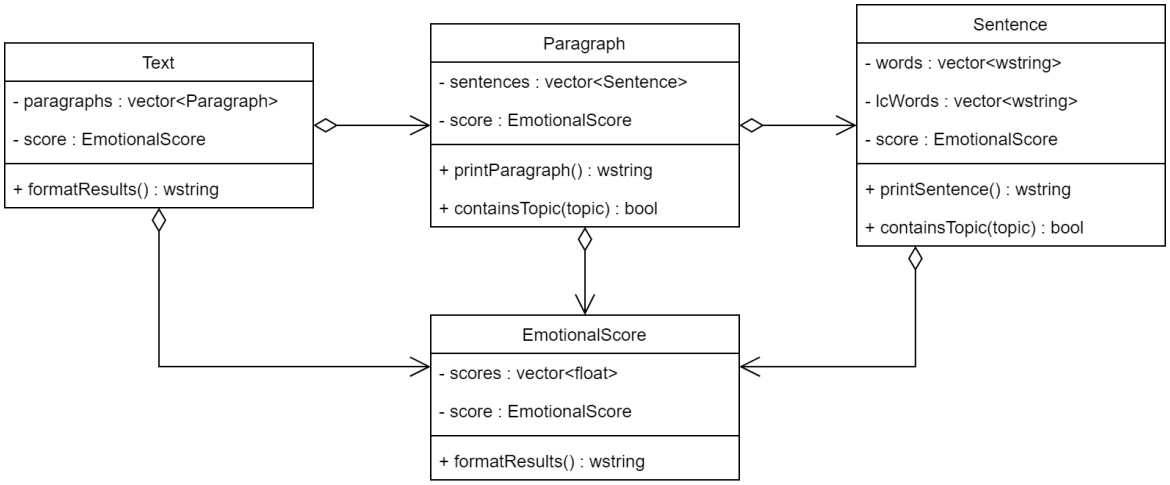

Figure 4. Text Storage System Class Diagram

Separation of the text by separateText() begins by creating empty vectors for words, sentences and paragraphs. The vectors here are also shared pointers to avoid value copying but will be referred to as vectors for simplicity. Once the variables are initialised, a for loop goes through the whole input text character by character, keeping the current index and index of last word found. Three additional methods are also used: isWordEnd(), isSentenceEnd() and isParagraphEnd(). Word end checks if symbol is a white space, sentence end checks if the symbols is a sentence ending punctuation, and paragraph end checks for new line '\n' or carriage return '\r'. At the start of the loop all three requirements are checked and if at least one of them is met, a new word is added to words vector using substr() and index is saved as last words index. After that only sentence end and paragraph end is checked and if at least one of them returns true, a new sentence is created from the words vector. Additionally, punctuation is also saved as the final word. Finally, only paragraph end is checked for, and if true, a new paragraph is constructed from the vector of sentences. Every time a new word, sentence, or paragraph should be created it is checked if they are not empty. Moreover, after adding words to a sentence or sentences to a paragraph, a new

pointer to an empty vector is assigned to words or sentences. Once the loop is finished by reaching the end of the text a new Text object is created from the vector of paragraphs and returned.

4.4 Analysis

To implement different analysis models and types we employ a similar structure to the one used in input gathering. There is an abstract class *Analyser*, which contains methods for analysing different tokens and a pointer to the database used for searching keywords (see Figure 5). This interface has two different implementations for two different analysis models: BoW and VADER. Moreover, class for topic analysis is derived from VADER analyser, because it uses the same analysis model, but only analyses the parts around a specific topic. During the runtime the correct model is chosen according to raised flags (VADER is the default option) and coresponding analyser is assigned to a *Analyser* pointer allowing to use the analyser further on without knowing which implementation was chosen. In all implementations *calculateScore()* overloads which take in *Text, Paragraph* or *Sentence* tokens iterate through every lower element (e.g. sentences in a paragraph) and calls the method for it. Once that methods returns a score it is stored inside the element for later use. BoW is the most simple analyser and it simply takes the aggregate value of every found keyword's score without taking anything else into account. This model was implemented because it offers less accuracy, but also a faster analysis.

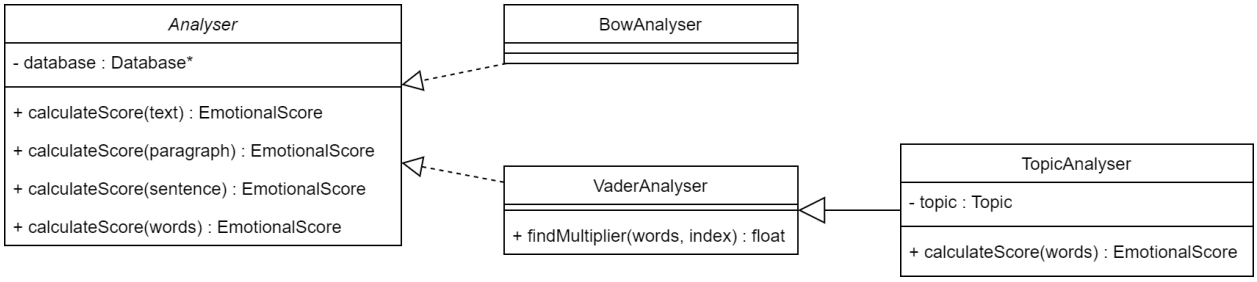

Figure 5. Analyser Class Diagram

4.4.1 Vader analyser

In the VADER analyser implementation main two differences is the override for *calculateScore(words)* and addition of *findMultiplier()*. They are implemented very similarly to the proposed design (see Algorithm 1). During the analysis of a sentence, the algorithm goes through every word one by one and checks if it is the database. Before that, it is checked if the word could be skipped using *isConnectingWord()* method in the database. If the word is not skipped and is a keyword, then the affixes of the keyword are checked using *findAffix()*, multiplying a copy of the keywords score by the returned multiplier. If nothing was found, the returned value is 1, so there is no need to check if anything was detected. After this, the code calls *findMultiplier()*.

The process of finding the multiplier involved going backwards through the sentence and searching for negations or amplifications. The initial value of the multiplier is 1 so to not damage the score if nothing is found through this process. The for loop goes backwards by up to two indexes, but this limit can be increased if a connecting word is found. Each word is checked for being an affix or an amplifier and the initial multiplier is multiplied. Once the start of the sentence is reached or index reaches the limit of how far it should look for, the multiplier is returned and found keyword's score is multiplied and added to the total. This repeats until the sentence end,

where the punctuation is checked for exclamation (which increases the score) and the calculated score is returned.

During this process all values that are passed between different analyser and database methods are passed by reference instead of value. This choice was made due to large amounts of data being unnecessarily being copied during analysis of longer texts. The only values copied are emotional scores. This was done to avoid modifying keyword's score values inside the database. Moreover, during the whole process a log is being written, which describes what keywords, affixes, negations and amplifications were found and how it affected the score.

4.4.2 Topic analyser

The main objective of the topic analyser is to detect sentiment around a specific topic. The user is able to define a topic by choosing strong keywords and optionally weak ones. They can be assigned when running the program by supplying strong keyword flag with parameters-*ts "keyword1 keyword2..."* and *-tw "keyword1 keyword2..."* for weak ones. Strong keywords are the ones that describe the topic directly like exact names of events or people while weak keywords may be alternative ways the topic could be described like pronouns or work position.

This process is done by *TopicAnalyser* (see Figure 5) which is derived from *VaderAnalyser*. With the difference being *calculateScore()* which only does the analysis on the paragraph if it contains strong topic keywords. Also *TopicAnalyser* has *Topic* class inside it, which stores strong and weak keywords and is able to check whether given word is one of them.

The analysis starts by going trough each paragraph and checking if of of its' sentences contains strong keywords using *containtsTopic(topic)*. If it does then the algorithm goes through each sentence looking for strong keywords or if the last one had a strong one, weak keywords are also accepted. If a sentence satisfied this condition its' score gets added to the *Paragraph* score, which later gets summed up to show overall sentiment around the topic.

4.5 Generating Output

The structure of output system is very similar to the input system (see Figure 3. There is an abstract class *OutputManager* which provides several overloads for *writeMessage()* method and there are implementations for terminal, text file and PDF outputs. At the start of the application the correct implementation is chosen according to default option and raised flags and then assigned to an *OutputManager* pointer. This allows to use polimorphysm and simplify the further execution.

The simplest way of output is the terminal and it is the default output destination. Additionally, *-f* flag can be used to write the results to a specified destination file. Both of these options allow only for text based results. There are several different ways of formatting the output, but the default output showcases a total score for the whole paragraph and look like this:

```
> Joy: 5 20%
> Sadness: 2 8%
> Trust: 6.5 26%
> Disgust: 1 4%
> Fear: 3 12%
> Disgust: 3 12%
> Surprise: 1 4%
> Anticipation: 0.5 2%
```

In each row a separate emotion is showcased and the first number is the aggregate score. The size of the score usually depends on the size of the text, since a larger text will contain more keywords. The second number is provided to help understand the results and it showcases what part of all emotions the particular emotion takes up.

In case the user wants more detailed results, there are verbose paragraph (*-vp*) and verbose sentence (*-vs*) flags, which instead show the results for each paragraph or sentence respectively. These results are formatted for human readability but in case the user wants to automate the analysis process and pass the results to other applications the *-csv* flag can be used. If this flag is raised, the outputs will be only the aggregate scores as comma-separated values. This way of outputting the results works very well with verbose paragraph or sentence flags.

In case there are any errors or weird results produced by the application, two different logs are written during the lifetime of the application. First one is for general workings of the application. It shows what input and how was it read, what database file was parsed, thrown errors and etc. The second log is meant for analysis process specifically. This log contains paragraphs, sentences and their scores, found keywords together with detected amplification/negation and its own score. The logs are written into files named after the date of execution in 'yyyy-MM-dd-hh-ss' format, but the analyser log additionally has a suffix for the type of analyser.

4.5.1 Emotional Flow

The primary goal of emotional flow analysis is to visualise how emotions change throughout the text. To achieve this, the text is divided into 10 equal segments, each represented as a vector of vectors with paragraph indexes: *std::vector<std::vector<int>>*. Subsequently, emotional scores for each paragraph within the index range are stored in *std::vector<int>* and summed up. At the end, the emotional flow is presented as *std::vector<EmotionalScore>*.

If the number of paragraphs is less than the default, the segment number will be adjusted to match the number of paragraphs. If the segments are not equal ($\frac{\text{ParagraphNumber}}{10} \notin \mathbb{N}$), then the last segment is smaller but its score is also adjusted accordingly. This is done by multiplying the segments score by $\frac{n}{m}$ where n is average amount of paragraphs per segment and m is amount of paragraphs in the current segment.

To visualize the emotional flow, users can enable the *-flow* flag. The output is a well-structured table that also indicates the emotional change in brackets compared to the previous line. Here is an example of the emotional flow displayed in the terminal:

```
> [3, 11, 0, 1, 11, 4, 0, 0]
> [2(-1), 4(-7), 0(0), 1.5(+0.5), 12(+1), 6(+2), 2(+2), 0]
> [1.5(-0.5), 8(+4), 5(+5), 0(-1.5), 5(-7), 6(0), 0(-2), -2(-2)]
> ...
```

4.5.2 PDF Output

If the user likes or requires a more visual representation of the results, they can opt to receive the results in a PDF format using *-pdf* flag. If this flag is raised, then a PDF file with user-chosen name is generated. In the PDF, all the text results are written like in the terminal, but additionally pie charts for score distribution are drawn. To implement PDF document creation, we use a low level library *libharu*, which provides a difficult-to-use but powerful interface. This library was chosen,

because it has many features and proper documentation, but also because there are not many other options.

To represent the total or intermediary scores, the *PdfOutput* draws a pie chart to represent the distribution of the emotions in that segment (see Figure 6). If there are several segments being showcased (e.g. *-vp* flag is raised), a separate pie chart is drawn for each The pie pieces are sorted in the order of largest to smallest and are color coded to an emotion to create an easy to understand chart. If there are no emotions found in the segment, the pie is grey. Additionally, when the pie

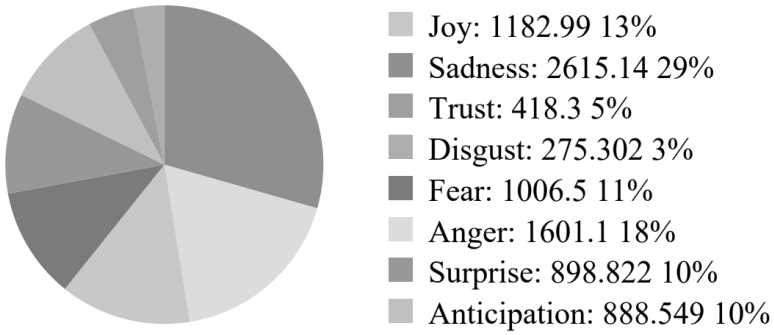

Figure 6. PDF Pie Chart Example

chart is being drawn, an emotional flow diagram is also drawn, independent of the *-flow* flag (see Figure 7). This diagram showcases how the emotions change through out the text by representing each emotions by its own color coded line. The vertical values represent individual emotion scores and horizontal values represent segments. This diagram helps more clearly visualise the change of the emotions through out the text.

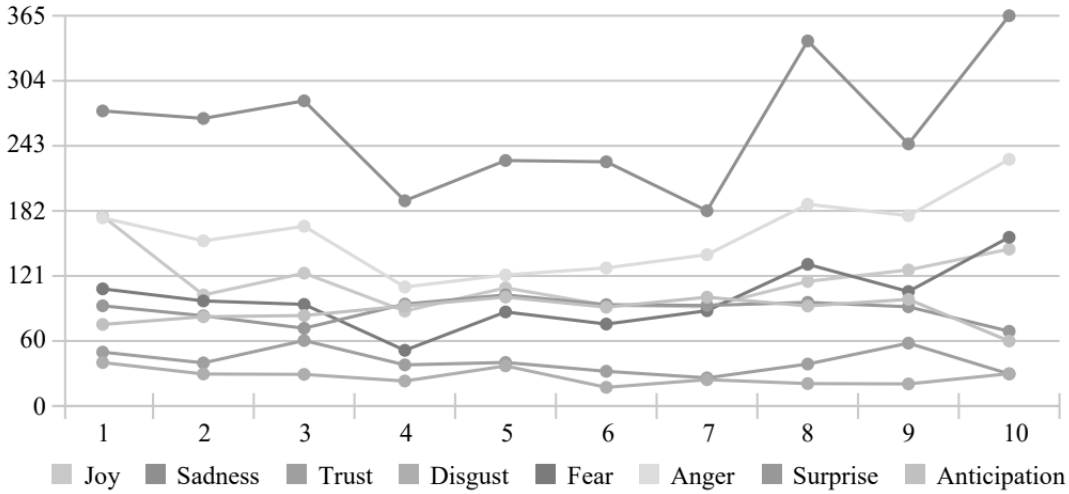

Figure 7. PDF Emotional Flow Diagram Example

5 Testing

This project uses the Google Tests framework for unit testing, with tests organized into separate chapters. Each chapter focuses on specific functionalities to ensure the accuracy and reliability of the implemented algorithms. The primary testing chapters are as follows: Analyser Tests, Input /

Output Manager Tests, Web Input Tests, Emotional Score Tests, Emotions Combinations Tests, Database Tests, File Input / Output Tests, Flag Tests, Topic Tests and Log Tests. Below are some examples of test cases:

1. **Reading Database File**

   Description: This test is designed to validate the accurate parsing of JSON-formatted data into specific fields of the Database object.

   Precondition: The *nlohmann/json* library is required to be installed. Additionally, ensure the availability of a specific Database constructor capable of accepting a JSON object as a parameter.

   Assumption: The test assumes that the JSON data file structure adheres to the expected format, and it will correctly scans the data into specific fields.

   Test Steps:

   (a) Create a *std::string jsonStr* representing the structure of the JSON data file.

   (b) Initialise a Database object, providing the parsed *jsonStr* as a parameter.

   (c) Verify that all fields (keywords, affixes, negations, amplifiers, connecting words, general prefixes) are correctly populated within the Database object.

   Expected Result: The Database object should reflect the expected values for each field, as specified in the provided JSON data.

2. **Searching in database**

   Description: This test verifies the functionality of the *Database* class in locating keywords, affixes, negations, amplifiers, and connecting words based on input text.

   Precondition: A *Database* constructor which instantiates with predefined keywords, affixes, negations, amplifiers, and connecting words. Input for searching must be written in lowercase.

   Assumption: The *Database* class using necessary methods can find if the given input contains keywords, negations, amplifiers, and connecting words or can detect affixes inside it.

   Test Steps:

   (a) Test the identification of keywords using *findKeyword* method. Verify that the correct keyword is found for exact and partial match case inputs.

   (b) Validate the handling of unknown keywords, ensuring the function returns an empty keyword object.

   (c) Test the identification of affixes using *findAffix* method. Check that the correct affix is found for various word forms.

   (d) Test the identification of negations using *findNegation* method. Ensure the correct negation is found in a fully similar word.

   (e) Test the identification of amplifiers using *findAmplifier* method. Ensure the correct amplifier is found in a fully similar word.

(f) Test the identification of connecting words using *isConnectingWord* method. Ensure the correct word is found in a fully similar word.

(g) Check if affix, negation, or amplifier is not found, then the result is 1.

Expected Result: The *Database* class should accurately find match keywords, affixes, negations, amplifiers, and connecting words based on the provided input text, using necessary methods.

3. **Analyse text using VADER analysis**

   Description: This test aims to validate the functionality of the Vader Analyser, ensuring accurate emotional score calculations based on predefined keywords, affixes, negations, and amplifiers.

   Precondition: The Vader Analyser relies on a Database object constructed with specific keywords, affixes, negations, and amplifiers.

   Assumption: Provided text inputs correctly follow the expected emotional score.

   Test Steps:

   (a) Initialise a Database object with predefined keywords, affixes, negations, and amplifiers, including a set of general prefixes.

   (b) Create various text input scenarios, each containing different emotional expressions.

   (c) Use the Vader Analyser to calculate emotional scores for each text input.

   (d) Compare each of the results with the expected.

   Expected Result: The Vader Analyser should accurately calculate emotional scores for the provided text inputs, considering on given linguistic modifiers.

4. **Separate text into Individual Words, paragraphs, and sentences**

   Description: This test verifies the functionality of the *separateTextIntoWords* method in the *IOManager* class. It ensures the correct tokenisation of input text into individual words. Also using another method *separateText* the input string will be parsed into sentences and paragraphs.

   Precondition: None.

   Assumption: The *separateTextIntoWords* method correctly handles various text structures, including newlines, punctuation, and multiple spaces.

   Test Steps:

   (a) Provide a simple sentence without special characters. Verify that the output matches the expected tokenisation.

   (b) Introduce newlines in the text. Validate that the method correctly separates words, disregarding newlines.

   (c) Include multiple spaces between words. Confirm that extra spaces are ignored, and words are correctly identified.

(d) Add punctuation at the end of sentences. Check that the method handles punctuation marks appropriately. Also, check if a new Sentence object was created and compare it with the expected.

(e) Test a paragraph with multiple sentences. Ensure that each sentence is correctly tokenised.

(f) Include empty lines between sentences. Verify that empty lines are treated as sentence separators.

(g) Test a more complex case with various sentence structures, punctuation, and newlines.

(h) Assess the handling of a sentence without an explicit end, followed by a new paragraph and sentence.

Expected Result: Successfully separated text into words using *separateTextIntoWords* method and created necessary Sentence and Paragraph objects using *separateText* method.

Conclusions and Future Work

We adapted the VADER model ideas to a sentiment analysis system based on *Plutchik's* model of eight emotions. The design proved to be capable of doing the analysis and providing mostly accurate results. The application is capable of analysing text files and web pages. The analysis can be either general or based around a specific topic. The results then can be presented text format through the terminal or text file, or it can be visual and presented in a nicely formatted PDF document with pie charts and line diagrams.

Although the proposed design and created system works, there are still many improvements and issues left to address. One of the issues that has to be addressed is the accuracy of the analysis system. There is still no mechanism to avoid some words being detected in others (e.g. 'war' in 'warm'). Moreover, the algorithm can only detect very direct negation or amplification, so the future work would be to create a system, which detect indirect modifications, especially literary tool like hyperbole or sarcasm. Moreover, the current database was created by small amount of individuals, therefore there is a need to design and implement a robust and reliant system for creating and filling a sentiment dictionary meant for analysis using *Plutchik's* model.

References

[1] Stefano Baccianella, Andrea Esuli, and Fabrizio Sebastiani. SentiWordNet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, May 2010.

[2] Erik Cambria, Björn Schuller, Bing Liu, Haixun Wang, and Catherine Havasi. Statistical approaches to concept-level sentiment analysis. *IEEE Intelligent Systems*, 28(3):6--9, 2013.

[3] K. R. Chowdhary. *Natural Language Processing*, pages 603--649. Springer India, New Delhi, 2020.

[4] Rabiyatou Diouf, Edouard Ngor Sarr, Ousmane Sall, Babiga Birregah, Mamadou Bousso, and Sény Ndiaye Mbaye. Web scraping: State-of-the-art and areas of application. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 6040--6042, 2019.

[5] Doaa e. Enhancement bag-of-words model for solving the challenges of sentiment analysis. *International Journal of Advanced Computer Science and Applications*, 7(3), 2016.

[6] Jack E. et al. Dynamic facial expressions of emotion transmit an evolving hierarchy of signals over time. In *Current Biology, Volume 24, Issue 2*, pages 187--192, 2014.

[7] C. Hutto and Eric Gilbert. Vader: A parsimonious rule-based model for sentiment analysis of social media text. *Proceedings of the International AAAI Conference on Web and Social Media*, 8(1):216--225, May 2014.

[8] Muhammad Taimoor Khan, Mehr Durrani, Armughan Ali, Irum Inayat, Shehzad Khalid, and Kamran Habib Khan. Sentiment analysis and the complex natural language. *Complex Adaptive Systems Modeling*, 4(1):2, February 3 2016.

[9] Walaa Medhat, Ahmed Hassan, and Hoda Korashy. Sentiment analysis algorithms and applications: A survey. *Ain Shams Engineering Journal*, 5(4):1093--1113, 2014.

[10] Fares Murhaf. Erg tokenization and lexical categorization: A sequence labeling approach. Master's thesis, 2013.

[11] Ekman P. and Friesen W. Constant across cultures in the face and emotion. In *Journal of Personality and Social Psychology, Vol. 17, No. 2*, pages 124--129, 1971.

[12] Robert Plutchik. The nature of emotions: Human emotions have deep evolutionary roots, a fact that may explain their complexity and provide tools for clinical practice. *American Scientist*, 89(4):344--350, 2001.

[13] Wisam A. Qader, Musa M. Ameen, and Bilal I. Ahmed. An overview of bag of words;importance, implementation, applications, and challenges. In *2019 International Engineering Conference (IEC)*, pages 200--204, 2019.

[14] Maite Taboada, Julian Brooke, Milan Tofiloski, Kimberly Voll, and Manfred Stede. Lexicon-Based Methods for Sentiment Analysis. *Computational Linguistics*, 37(2):267--307, 06 2011.

Appendices

A Example Database Data

This segment is meant to showcase example data that could be in the database for this project.

1. The **Keywords** consists of root of keyword and eight values from 0 to 1 for each primary emotions: joy, sadness, trust, disgust, fear, anger, surprise, and anticipation.

root	joy	sadness	trust	disgust	fear	anger	surprise	anticipation
disgust	0	0	0	1	0	0	0	0
anticipat	0	0	0	0	0	0	0	1
surpris	0	0	0	0	0	0	1	0
happy	1	0	0	0	0	0	0	0
amaz	0.4	0	0	0	0	0	0.5	0
gloom	0	0.8	0	0	0	0	0	0
terror	0	0.2	0	0	1	0.2	0	0
stress	0	0	0	0	1	0	0	0
doom	0	1	0	0	0.5	1	0.1	0.1
destr	0	1	0	0	1	1	0	0
lie	0	0.2	0	0	0	0.3	0.3	0

2. **Prefixes** and **suffixes** are part of the words which modifies the words meaning. The affix itself and multiplier is stored.

affix	multiplier
un	-1
less	-1
in	-1
ful	1.5

3. **Amplifiers**, **dampeners** and **negations** which invert, increase or decrease the keyword's emotional values.

word	multiplier
not	-1
no	-1
never	-1
extremely	2
very	1.5
extra	1.5
super	1.3
likely	0.8
little	0.5