Department of Management Science and Technology

M.Sc. in Business Analytics

M.Sc. Thesis

Hedge Detection: An Application on the Wikipedia Corpus

Agapiou Marios

F2821901

Thesis submitted to the Secretariat of the MSc in Business Analytics of the Athens University of Economics and Business, in partial fulfillment of the Requirements for the master's degree in Business Analytics.

May 2021

Abstract

The purpose of this thesis is to develop a system that automatically detects hedges in Wikipedia articles, using weasel tags. The motivation behind this research project was to tackle the issue of ambiguity in Wikipedia articles, which could lead to the promotion of misleading information to the reader. This paper provides the general overview of this task, including the extraction of the data, the classification methods that were used, as well as the evaluation metrics employed to examine the overall performance of these methods. In this thesis we experimented with machine and deep learning models to apply the text classification. We implemented Support Vector Machine and XGBoost classifiers, and developed neural networks, such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) with Long short-term memory (LSTM) architecture to complete this task. We then evaluated these systems against the best performing systems from previous studies that focus on this issue. Overall, we achieved notable results on our dataset, surpassing most hedge detection systems from previous studies, and thus proving the effectiveness of our methods.

Περίληψη

Σκοπός αυτής της διατριβής είναι να αναπτυχθεί ένα σύστημα που ανιχνεύει αυτόματα αμφισημίες σε άρθρα της Βικιπαίδειας, χρησιμοποιώντας ετικέτες weasel. Το κίνητρο πίσω από αυτό το ερευνητικό έργο ήταν να αντιμετωπιστεί το θέμα της ασάφειας στα άρθρα της Βικιπαίδειας, που θα μπορούσε να οδηγήσει στην προώθηση παραπλανητικών πληροφοριών στον αναγνώστη. Το παρόν έγγραφο παρέχει τη γενική επισκόπηση αυτού του έργου, συμπεριλαμβανομένης της εξαγωγής των δεδομένων, των μεθόδων κατηγοριοποίησης που χρησιμοποιήθηκαν, καθώς και των μετρήσεων αξιολόγησης που εφαρμόστηκαν για την εξέταση της συνολικής απόδοσης αυτών των μεθόδων. Σε αυτή τη διατριβή πειραματιστήκαμε με μοντέλα τόσο μηχανικής όσο και βαθιάς Μάθησης για να εφαρμόσουμε την ταξινόμηση κειμένου. Υλοποιήσαμε Support Vector Machine και XGBoost ταξινομητές, και αναπτύξαμε νευρωνικά δίκτυα, όπως τα Convolutional Neural Networks (CNNs) και τα Reccurent Neural Networks (RNNs) με αρχιτεκτονική Long short-term memory (LSTM) για την ολοκλήρωση αυτής της εργασίας. Στη συνέχεια, αξιολογούμε αυτά τα συστήματά συγκριτικά με τα καλύτερα συστήματα από προηγούμενες μελέτες που εστιάζουν σε αυτό το ζήτημα. Συνολικά, πετύχαμε αξιοσημείωτα αποτελέσματα, ξεπερνώντας τα περισσότερα συστήματα ανίχνευσης αμφισημιών από προηγούμενες μελέτες, αποδεικνύοντας έτσι την αποτελεσματικότητα των μεθόδων μας.

Acknowledgements

I would like to thank my supervisor Prof. Panagiotis Louridas for all the great advice and guidance he has provided. Additionally, I want to extend my gratitude to my family for their care and support during my postgraduate studies.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Task Description

The concept of hedging has been first coined by George Lakoff in 1973, who describes hedge words and phrases as "words whose job it is to make things more or less fuzzy"[1]. The term hedging is used as a means of specifying the ambiguous claims made by authors without any factual or specific information to prove their validity.

Indicative examples of linguistic hedging include the use of impersonal pronouns, concessive conjunctions (e.g., "although", "whereas"), modal verbs (e.g., "might", "may"), epistemic adverbs/adjectives (e.g., "allegedly", "supposedly", "probable"), the use of non-committal statements expressed in passive voice (e.g., "it is believed"), or the use of active voice with beliefs attributed to dummy subjects (e.g., "researchers believe")[2].

The effects of hedges can be both positive and negative in terms of their use. On occasion, the usage of hedge words and phrases allows authors to acknowledge the degrees of uncertainty in their statements, rather than claiming something that is not factual, meaning that they are more accurate and honest when interpreting scientific results. This helps the reader understand the level of the author's commitment to the reliability of their statements.

Nevertheless, using hedge words or phrases could dilute sentences' meaning or make them open to multiple interpretations. Authors using hedge words can easily mislead the reader as well as spread hearsay or their personal opinions instead of factual information, therefore making them dishonest. This would be very problematic, especially in academic literature since clarity in science is of the utmost importance. Usage of unclear language in academic writing, as well as presenting ambiguous scientific results could confuse the reader by promoting misleading information, thus impeding any scientific advancement. It is prudent for every author to be cautious in their statements and be able to distinguish between facts from unreliable or uncertain information.

Therefore, it is essential to detect ambiguity and subsequently improve an article by removing or correcting any statement that contains any form of hedging. This will in turn strengthen the authors' commitment to their statements and increase their overall trustworthiness. Many strategies for detecting hedges in academic texts within the linguistics community have focused on determining relevant linguistic markers. An example would be Ken Hyland's article on the use of hedging [3], where he describes in detail all the forms of linguistic hedges that were found in academic literature. Instead on focusing only on lexical patterns, in this thesis we will focus on the classification of speculative language from a Machine Learning perspective.

1.2 Thesis Structure

The employed steps are highlighted in each section of this paper.

In **Section 2** we investigate the related works to our task. These works are about hedge detection and how hedge detection has been achieved with different methods.

Section 3 describes in detail the process of data extraction, as well as data cleaning techniques that were used. Section 3 also mentions the creation of two separate datasets that will be used for text classification. The results from the first dataset will be compared with a baseline classifier, while the results from the second dataset will be presented side by side with the results from previous studies, which will be described in Section 2. Furthermore Section 3 provides a summary for each dataset.

In the lengthy **Section 4**, the full methodology for the classification is presented. This section describes the text preprocessing techniques that were used. It should be mentioned that various approaches were used in text preprocessing and are described in detail in this section. This section also introduces the basic concepts of each machine learning method that was used. In total, two different classifiers were trained, as well as three deep learning models. The classifiers that were trained are the following:

- XGBoost Classifier
- Linear Support Vector Machine

The deep learning models that were trained are listed below:

- CNN Model
- Bi-LSTM Model
- Bi-LSTM Model with self-attention mechanism

Finally, **Section 5** includes a presentation of all experiments and their results with graphic presentation, while **Section 6** sums up the steps of the project and more importantly its conclusions.

Notation

NLP	Natural Language Processing
BoW	Bag of Words Model
SVM	Support Vector Machines
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory
Bi-LSTM	Bidirectional LSTM
Att+Bi-LSTM	Bidirectional LSTM with self-attention
Tanh	Hyperbolic Tangent Function
ReLU	Rectified Linear Unit Function

Table 1 Notation Table

2 Related Work

At first, research on hedge detection in NLP has almost exclusively been focused on the biomedical text. Light et al. (2004) [4] used a straightforward technique, which classifies sentences to hedge and not-hedge sentences, based on the presence of specific hedge cue words. Examples of these hedge cue words were words such as “suggest”, “potential”, “likely”, “may”, etc. For classification, Support Vector Machines (SVM) were used in conjunction with Stemming and term weighting (TF-IDF weights).

Medlock & Briscoe (2007) [5] also developed a system for automatic classification of speculative language ('hedging'). Using weakly supervised machine learning they

created an automatic hedge detection system in biomedical literature. A SVM classifier was used to perform the classification.

Szarvas (2008) [6], inspired from the methodology used by Medlock & Briscoe (2007), extended it to also include n gram features. He also used a semi-supervised selection of the key word features to enrich the training dataset.

Hedge detection has also received considerable interest in recent years. The academic paper from Viola Ganter and Michael Strube (2007) [7] on hedge detection, changes the format of the previous studies in terms of data. While previous studies were restricted to analyzing only biomedical literature, Viola Ganter and Michael Strube used Wikipedia's articles as a source of training data for hedge classification. They utilized specific Wikipedia template tags to create a corpus of weasel-worded sentences.

One of the most important works on hedge detection remains the CoNLL-2010 shared task [8], which primarily concentrated on using machine learning models based on annotated data. Some participants used the Bio-Scope corpus of biomedical abstracts and articles for annotation (TaskB), while others used a Wikipedia corpus of already annotated sentences (TaskW). The CoNLL-2010 systems approach the task either as a standard sentence classification problem or as a token-by-token classification problem.

In our study, the main focus is the classification of sentences from Wikipedia articles. To be more precise, these sentences are collected by identifying distinctive tags that are added by Wikipedia editors, which indicate weasel-worded statements. It should be noted that Wikipedia's notion of weasel words/phrases is closely related to hedges[1]. This statement is confirmed in Viola Ganter's and Michael Strube's study on hedge classification, where they also used Wikipedia as a primary source for the collection of training data.

[1] https://en.wikipedia.org/wiki/Weasel_word

It is essential to highlight that in this thesis, related work will be primarily focused on the CoNLL-2010 shared task and approaches from the specific subtask named "Task1W". Considering that the results of the "Task1W" task will also be compared with the results of this paper, it would be useful to highlight some observations that were drawn from the CoNLL-2010 shared task:

- The top ranked systems of Task1B followed a sequence labeling approach while the best systems on Task1W applied a bag-of-words sentence classification. Note that Task1B refers to the hedge detection task on biomedical text, while Task1W uses only data from Wikipedia.

- Task1B yielded better results than Task1W. This could be attributed to the fact that biological sentences have relatively simple patterns, while Wikipedia weasel sentences tend to have an increased abundance of complicated patterns.

- Classification methods that were used by the participants of Task1W include Max Entropy classifiers (MaxEnt), Support Vector Machines (SVM), Conditional random fields (CRFs), the *k*-nearest neighbors algorithm (*k*-NN) and Naïve Bayes classifiers.

- Pre-processing steps that were used in Task1W typically consisted of Stemming, Lemmatization and POS-tagging.

In this thesis we used some of the previous methods of the CoNLL-2010 shared task and added new ideas to try and produce better results by using Deep Learning algorithms. In section 5 we presented our results and the results from the participants of Task1W alongside each other, to examine the performance of our classifiers.

3 Data Exploratory Analysis

Providing a large amount of high-quality data is a must for developing a good statistical model. Data collection and cleaning is a time-consuming process, and if not done right, it could lead to misleading results. In our research, training and testing data were retrieved from Wikipedia articles, as was mentioned in the previous Section of this thesis.

The first step of data extraction was to access the complete edit history of all Wikipedia articles. These data are stored in dumps, provided by Wikimedia. Since we wanted to parse all available Wikipedia articles, we selected the Wikimedia database dump on the 1st of February 2021 for data collection. This database dump is a complete copy of all Wikimedia wikis, in the form of Wikitext source and metadata embedded in XML. Wikimedia provides public dumps that store Wikipedia content and related metadata, such as search indexes and URL mappings[2] . These are used for archiving all available wiki articles, as well as any information regarding them, such as the timestamp of each article, author's name, etc. Each dump contains various compressed XML documents, each corresponding to the specific needs of the researcher. Some of these documents include the current version of each article, others include the articles along with the current discussion, and other types of documents contain the articles along with all past edits and discussion. In this case, the latter type of files will be downloaded, which contain all the revisions and all pages of all articles. Such files have «pages-meta-history» in their name, to distinguish them from the rest. Since Wikipedia provides a massive dump containing all edits on all articles, and to parse such a large file would be computationally very demanding, it is not optimal to use DOM API. It was necessary to use SAX, also known as the Simple API for XML, to parse XML documents. The process of SAX parsing is described in detail in Section 3.1.

The next paragraphs will primarily concentrate on everything involved in downloading, extracting, and cleaning a Wikipedia dump, to output the appropriate dataset.

[2] https://dumps.wikimedia.org/

3.1 Data Extraction

Since the first step is to download the data, it is of high importance that the correct files are downloaded in the Wikipedia dump. It was mentioned in the previous paragraph that specific files that include all past edits and the current version of articles have 'pages-meta-history' in their name to distinguish them from the rest. A python script, named «download_dumps.py», was created to download wiki dump files with a specified pattern automatically. In total, 694 compressed files were downloaded and parsed.

A new python script was created, to parse the 694 compressed files, named as «saxparser_1.py». The parsing process required the use of SAX, which is an event-driven algorithm for parsing XML documents, with an API developed by the members of the XML-DEV mailing list. As an alternative to the Document Object Model (DOM), SAX provides a mechanism for reading data from an XML document continuous stream. While SAX reads the continuous stream of a XML file, DOM works on the entire document, meaning the whole abstract syntax tree of an XML document. SAX parsers work on each piece of the XML document sequentially, issuing parsing events while making a single pass through the input stream. This approach makes SAX extremely efficient, allowing it to handle XML documents of nearly any linear time and near-constant memory size. Tree-based interfaces, such as the Document Object Model (DOM), are much more straightforward for developing but at a high cost in time and computer resources.

SAX parser searches through each compressed XML file while it uncompresses and extracts the information between specific XML elements. The most important part of this algorithm is the “ContentHandler” which controls how the parser's data is handled. The XML file is passed to the parser one line at a time, and the Content Handler extracts the relevant information from the XML elements that we choose to search.

The elements that are of interest in the XML files were the title of the article, the revision of the article when it contained statements with weasel words, the subsequent revision of the article when these statements were substantiated, as well as the

timestamps of each revision that was collected. It is noteworthy to mention that specific weasel tags identified these revisions of articles that contained weasel words. The Wikipedia style guidelines instruct editors to insert a weasel tag if they come across weasel words. It is usual for an editor to add a {{weasel}} tag to the top of an article or section to draw attention to the presence of weasel words. For this paper's case, though, it was more appropriate to use other tags that not only categorize an article or section but a specific sentence or phrase. Alternative weasel tags to the tag {{weasel}} that are added directly to the phrase in question are the tags {{weasel word}}, {{weasel inline}}, {{who}}, {{which}}, {{by whom}} and {{according to whom}}. These are the tags used in this paper to spot the article revisions containing weasel words. In order to give more generalized regular expressions for pattern matching and further broaden the search for weasel tags, RegEx was used.

Since SAX Parser is event-driven, it is vital to highlight the SAX events that appear during the parsing process:

- The SAX Parser starts parsing the XML file, and when it sees the start tag in the file, the startElement() method is called. Information about the tag is sent as a parameter to this method.

- The methodcharacters() is called with the text contents between the start and end of an XML element and is used to store the information of requested XML elements to the output.

- Lastly, the endElement() method is called when SAX Parser sees a closing tag while moving through the file as a stream. As in the startElement() method, the tag information is sent as a parameter in this method.

In the case of wiki dumps, whenever the parser encounters a revision by spotting the “text” element tag in each XML file, it will save the text if it contains the specific weasel tags and marks them as articles with specifically marked weasel-worded phrases. It continues the parsing on the article’s history until the next revision that

does not include any weasel tag. This revision is marked as the corrected/updated form of the article. Therefore, the data consists of the revisions of articles when they included weasel tags and the subsequent revisions of the same articles that had these tags removed.

It should be noted that some adjustments needed to be made during the parsing of each XML file so that the correct articles were selected. The edit history of an article may contain suspicious edits that signify vandalism. According to Wikipedia[3], vandalism has a very specific meaning. Editing deliberately intended to obstruct or defeat the project's purpose, through malicious removal of encyclopedic content, or the changing of such content beyond all recognition, verifiability, and no original research, leads to vandalism. Editors remove or undo vandalism in articles, but sometimes vandalism takes place on top of older, undetected vandalism. With undetected vandalism, editors may make edits without realizing the vandalism occurred. Vandalism in wiki articles is very harmful for our task, because during parsing there is a possibility to also store sentences subjected to vandalism to our data. There are two approaches that were used to minimize the problem of vandalism in our data. The first approach includes the action of ignoring revisions of articles, which are very small (one or two sentences long). This problem occurred almost exclusively on the corrected/updated revisions of our data. Therefore, while parsing the edit history of a historically weasel article for revisions that do not contain weasel tags, we ensured to only store the updated revision that is at least half the size of its equivalent weasel revision. The second approach is mentioned in Section 3.2, as it is implemented during the cleaning of the final data.

In order to speed up the process of parsing all 694 files, the GNU parallel shell tool was used for executing background jobs in parallel. The GNU parallel shell tool was created by Ole Tange [9] as a means to execute jobs in parallel using one or more computers.

Each background job referred to one file. The command that was run for each background job decompressed each file with 7z extension, while the SAX parser read

[3] https://en.wikipedia.org/wiki/Vandalism_on_Wikipedia

the file line by line and extracted the weasel-worded articles, as well as the corrected version of them. All the contents of the SAX Parser's output were stored in a CSV file.
The command that was run in the Linux terminal, to parse the wiki dump, is the following:

```
nohup parallel -j5 -k  '7z x -so {} | nice -n 10 python3 saxparser_1.py --output data.csv' ::: /wikidumps/enwiki-20210201*
```

Table 2 Linux command to run the Sax Parser on the Wikipedia Dump Files.

Since this paper aims to create automatic hedge detectors that attempt to identify specifically sentences that contain uncertain information, it is necessary to extract only the weasel-worded sentences and their rewritten versions that do not contain weasel tags. The CSV file created from the SAX parser needed to be parsed again to extract only the notable sentences. The python script «csvparser_2.py» was created to read the data from the CSV file, identify the weasel-worded sentences using the already mentioned weasel tags and their equivalent, updated versions extracted from the later revisions that do not contain weasel words.

Before the parsing of the new data is explained, it is worthwhile to describe the syntax and structure of the extracted Wikipedia pages or, better known as WikiText. According to Wikipedia[4], Wikitext, also known as Wiki markup or Wikicode, consists of the MediaWiki software's syntax and keywords to format a page. Wikitext is also comprised of various templates, which provide standard ways of recording the article's information and metadata. There are numerous templates for everything on Wikipedia. Still, the most relevant for this paper's purposes are the already mentioned weasel tags, such as {{weasel-word}} and {{weasel-inline}} that mark specific sentences instead of whole sections or articles. Using RegEx, we identified the marked sentences. The identification the corrected/non-weasel versions of these sentences was made by looking at the later revisions for similar sentences. For example, the older revision of the article with the title « Livonians » contains the weasel-worded sentence:

[4] https://en.wikipedia.org/wiki/Help:Wikitext

« It has been suggested that the first person to convert some Livonians to Christianity was the Danish archbishop Absalon, who supposedly built a church in the Livonian village today known as Kolka. »

To identify and store the updated/non-weasel version of this sentence, we need to read the updated revision of the article and compare all pairs of the weasel-worded sentence and all the sentences from the updated revision using the SequenceMatcher[5] that available in the python module named "difflib". This module will calculate the similarity ratios between the weasel-worded sentence and all the sentences of the updated revision. The sentence from the updated revision with the highest similarity ratio will be marked as the non-weasel version and added to the final dataset. In this case, the sentence:

«One of the first people to convert some Livonians to Christianity was the Danish archbishop Absalon, who supposedly built a church in the Livonian village today known as Kolka.»

was marked as the equivalent non-weasel version of the weasel sentence with a similarity ratio of 0.9.

Considering that the performance of classifiers needs to be compared with the CoNLL-2010 1st Shared Task results, it is crucial to work with a similar dataset to that of CoNLL-2010 1st Shared Task. According to Task1W of the CoNLL-2010 Shared Task, Task1W training data included 2186 paragraphs culled from Wikipedia archives (11111 sentences with 2484 uncertain ones). Furthermore, the test data included 9634 sentences, out of which 2234 were uncertain. Consequently, the ratio of uncertain and certain sentences in the test data of Task1W is roughly 1:4. To change the ratio of the dataset, additional non-weasel-worded sentences must be added.

Subsequently, our task will be divided in to two similar subtasks. Since it is preferable to compare our results to the results of Task1W of the CoNLL-2010 Shared Task, as

[5] https://docs.python.org/3/library/difflib.html

was mentioned before, we considered two different datasets to train and test our classification models.

- The first subtask only included the first dataset («clean_dataset»). It contained the weasel sentences as well as their equivalent corrected/non-weasel sentences and was used for the training and testing of our classification models. We then compared the performances of these models with each other and a baseline classifier. The ratio of weasel and non-weasel sentences in this dataset is 1:1.

- In the second subtask merged the first dataset («clean_dataset») with another dataset («extra_dataset») that contains only non-weasel sentences in order to change the ratio of weasel and non-weasel sentences to the appropriate amount of 1:4. The purpose of this action is to have a similar class ratios in our dataset with the dataset of Task1W of the CoNLL-2010 Shared Task. Hence, we were able to compare our results with the results of Task1W, with this dataset.

The command that was run in the Linux terminal, to run the python script «csvparser_2.py» is the following:

```
nohup sh -c 'python3 csvparser_2_new.py --input data.csv --output
clean_dataset.csv --extra extra_dataset.csv' &
```

Table 3 Linux Command for running the second parser.

The following table describes the types of sentences that were extracted, after we run the above command.

Types of sentences	Description	Label
Weasel Sentence	A sentence containing ambiguous claims	weasel
Non-Weasel Sentence	A sentence with factual information	

Corrected/Updated/Rewritten/Non-Weasel Sentence	A sentence that was rewritten from Wiki editors, to remove any uncertain statements. It is considered a Non-weasel sentence.	non-weasel

Table 4 Types of sentences that were extracted for both subtasks.

3.2 Data Cleaning & Exploration

Since the unfiltered text from wiki dumps contains wiki markups, it is essential to remove these markups to clean the text. We mainly used "mwparserfromhell"[6], a library built to work with MediaWiki content to remove templates and keep only text.

While the MediaWiki parser has access to the contents of templates, among other things, "mwparserfromhell" has some limitations. Syntax elements produced by a template "transclusion"[7] cannot be detected. When different syntax elements overlap, the parser gets confused and treats the first syntax construct as plain text. Additionally, the parser is not aware of localized namespace names, so file links (such as [[File:...]]) are treated as regular wikilinks. So, to remove the rest syntax elements that are not needed, custom functions were also made for each different template encountered during the cleaning after the MediaWiki parser was applied.

One of the critical points to correct hedging in text and improve weasel words in articles is to name the source for an ambiguous statement or opinion. Therefore, it is crucial to maintain the information of references in sentences since it can determine if a sentence is ambiguous. Using the "mwparserfromhell" python package and removing all templates cannot ensure that the knowledge of referencing a sentence is kept since all reference tags are deleted.

[6] https://github.com/earwig/mwparserfromhell
[7] https://en.wikipedia.org/wiki/Help:Transclusion

The tag [reference] was created to combat this problem. It was made to replace all reference and citation footnote templates of the text with a label that will not be removed during text cleaning.

Another data cleaning technique was the removal of weasel and non-weasel sentences, with length lower of 3 words, since these sentences do not actually represent complete sentences. A complete sentence contains, at the very minimum, three elements: a subject, verb, and an object. We also calculated the 95th percentile of all the lengths of all the sentences from our dataset and removed all weasel and non-weasel sentences with length greater than that number. These techniques were used for removing data that do not represent complete sentences, but instead are single words, letters or sentences subjected to vandalism. The following diagram shows the frequency distribution of the lengths of sentences after the removal of these data.

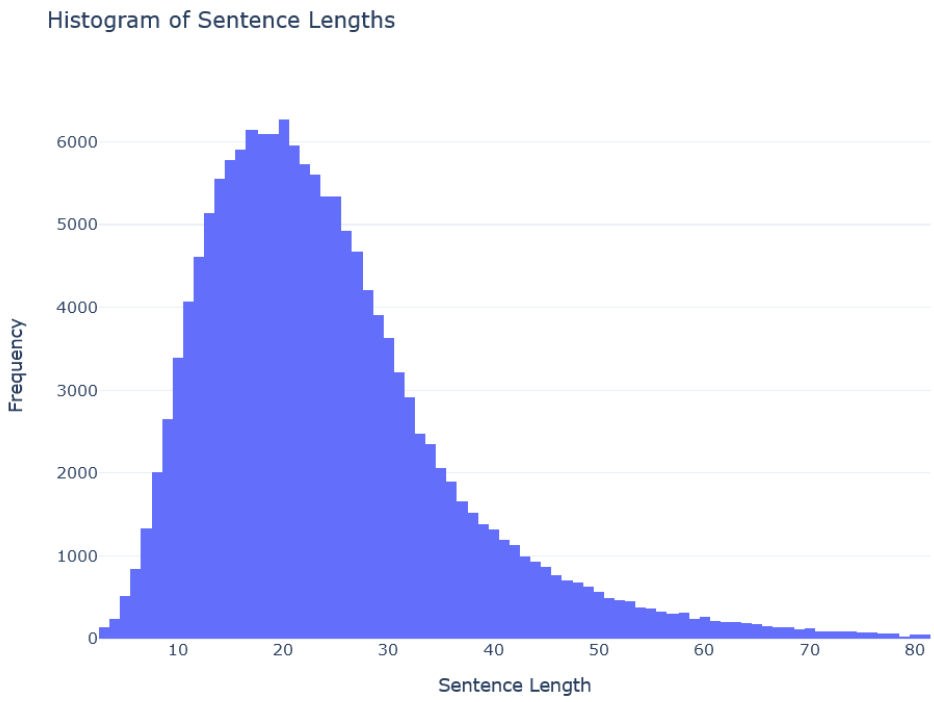

Figure 1 Frequency Distribution of Sentence lengths

Another cleaning procedure involved the similarities between weasel sentences and their non-weasel counterpart from the first dataset («clean_dataset.csv»). We decided

to remove the weasel sentences and their equivalent corrections that have a similarity percentage of 1. A similarity percentage of exactly one indicates that the same sentences are marked as weasel and non-weasel. The following histogram visualizes the frequency distribution of the similarity percentages of all “weasel” and “corrected/non-weasel” sentences.

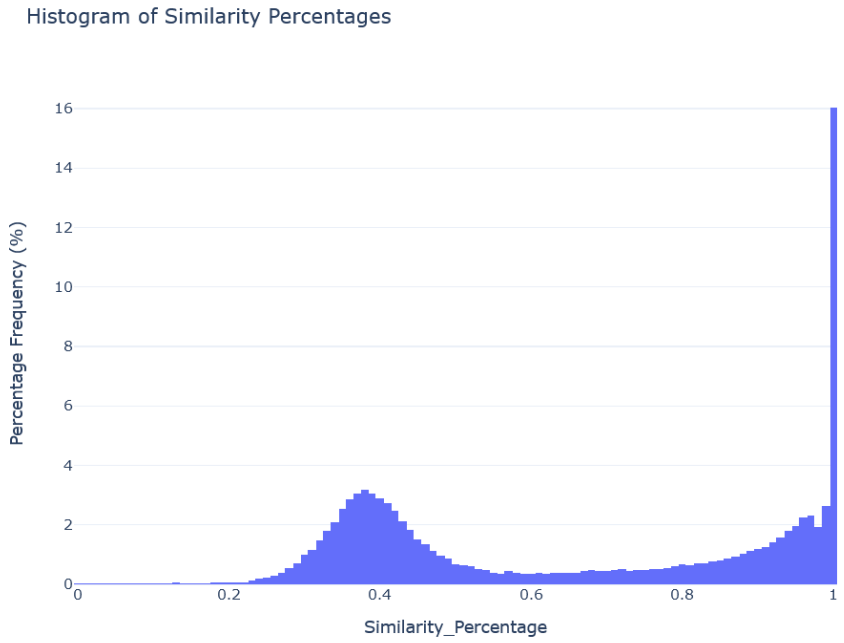

Figure 2 Frequency Distribution of Similarity percentages between "weasel" sentences and their "non-weasel" versions

The total percentage of instances where the weasel sentences and their corrected versions are exactly the same reaches 16% in our dataset. It is necessary to remove this percentage of our data, so that it does not undermine the classification.

Other cleaning techniques, included the removal of symbols, numbers, and other unwanted characters.

After the implementation of these text cleaning techniques, another plot was created from the first dataset. This plot presents the 10 most frequent bi-grams, trigrams, 4-grams and 5-grams that are found in the weasel sentences.

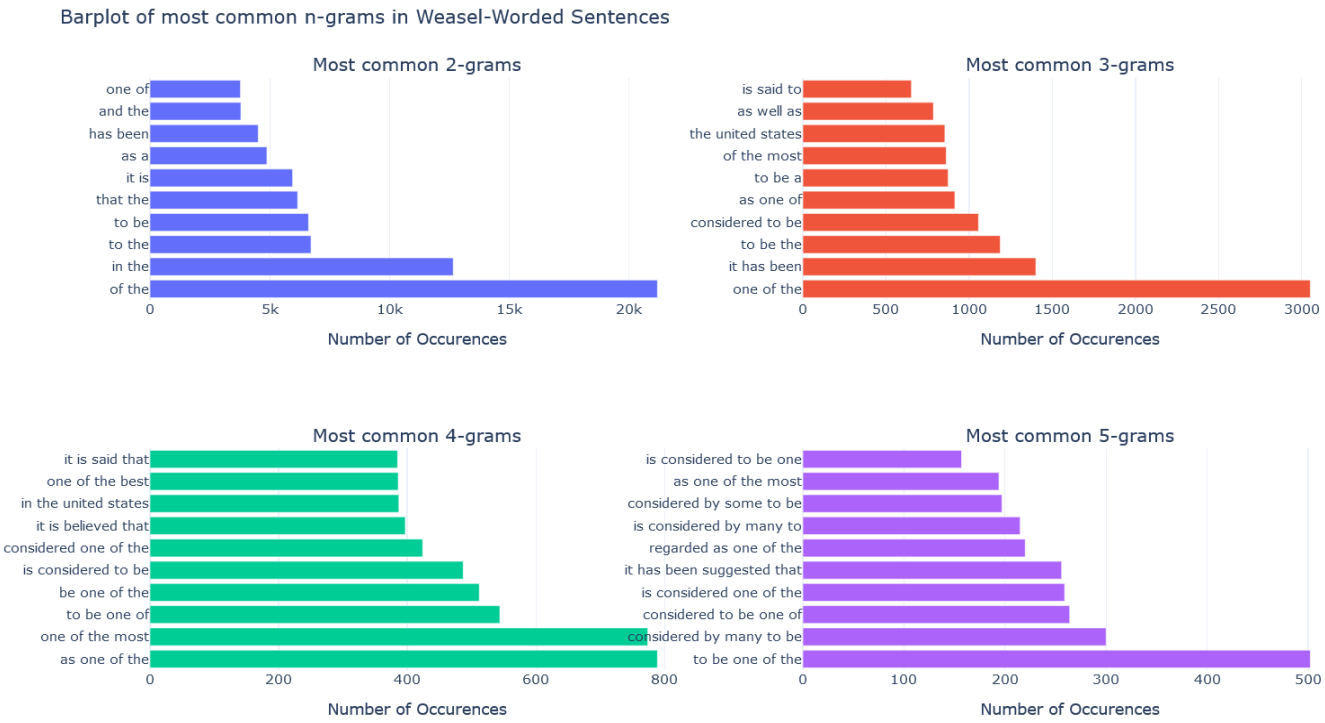

Figure 3 Barplot of 10 most frequent n-grams in "weasel" sentences

Many phrases such as “considered by many to be”, “one of the most”, “it is said that”, etc., that appear on the above diagrams, are presumably key words that define if a sentence contains a hedge statement. It is interesting to also plot the 10 most frequent phrases for the corrected/non-weasel-worded versions of the weasel sentences.

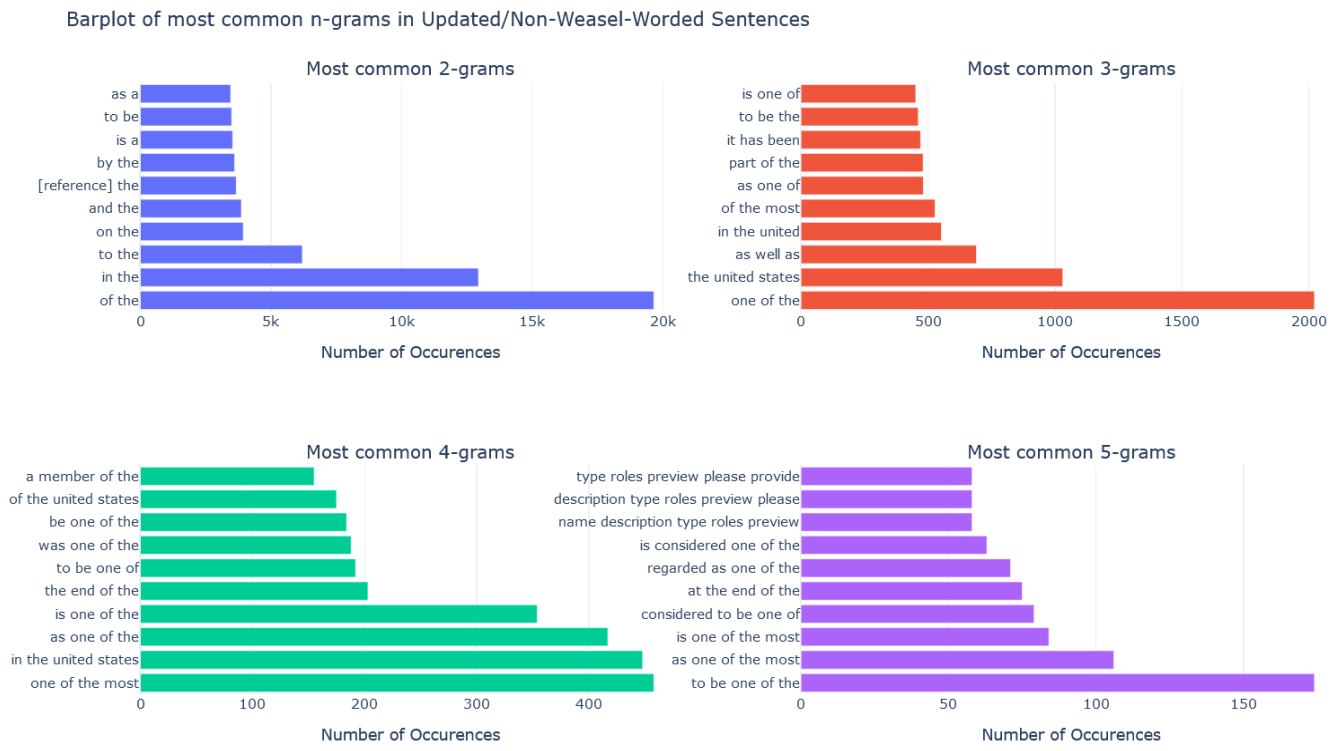

Figure 4 Barplot of 10 most frequent n-grams in "non-weasel"/corrected sentences

Although most phrases that appear on the above diagram do not appear on the previous one, some frequent phrases from the non-weasel versions of sentences are also frequent to weasel sentences, such as the phrases “is considered one of the”, “regarded as one of the” and “is one of the most”.

This observation demonstrates the level of difficulty and complexity of our task since it is not possible to detect hedges from sentences by looking for specific “key-words”. It is important to also capture the context of words and phrases inside sentences. This will be achieved through the implementation of various methods that are highlighted in Section 4 of this thesis.

3.3 Datasets

The first dataset consists of 151470 sentences in total, where 50% of them are marked as «weasel», meaning that they are subjected to hedging. The rest 50% of them are

marked as «not-weasel» and they are the rewritten version of the «weasel sentences», which do not contain ambiguous statements, i.e., hedgings. This dataset was used for Subtask 1.

The second dataset is comprised of 328777 sentences in total, where 23% of them are marked as «weasel sentences», and 77% as «not-weasel ». This dataset was used for subtask 2.

The data on both the first and second dataset were divided into three subsets, named Train Data, Validation Data, Test Data. It is important to mention that during the splitting, all created subsets preserved the percentage for each class. The following table presents the results after the splitting of the two datasets for Subtask 1 and 2.

		Class		
Subtask	Split Data	weasel	Not-weasel	Total
Subtask 1 (100%)	Train (64%)	48470	48470	96940
	Validation (16%)	12118	12118	24236
	Test (20%)	15147	15147	30294
Subtask 2 (100%)	Train (64%)	48470	161946	210416
	Validation (16%)	12118	40487	52605
	Test (20%)	15147	50609	65756

Table 5 Dataset Statistics for subtask 1 and 2

4 Methodology

In this section, the theoretical background of all implemented methods on Text Vectorization and Text Classification will be described.

- For Text Vectorization, the following vectorization techniques were applied.

I. Word Frequency Indexing using Sklearn CountVectorizer[8]

II. Using the Sklearn TfidfTransformer[9] method for better representations of words

III. Converting text data into vectors using Keras Tokenizer[10] text to sequence

IV. Adding Pre-Trained word embeddings by using the word embedding method known as GloVe

The methods used in the Text Vectorization of the two datasets will be described in further detail in Section 4.1.

- For Text Classification, the following classification models were used:

  I. Dummy Classifier (Baseline Model) (Section 4.2)

  II. XGBoost Algorithm (Section 4.3)

  III. Linear SVM (Section 4.4)

  IV. Convolutional Neural Network (Section 4.5)

  V. Bi-LSTM Neural Network (Section 4.6)

  VI. Bi-LSTM with self-Attention mechanism (Section 4.7)

  It should be mentioned that for Parameter Tuning, the "RandomizedSearchCV" technique was used to improve the performance of the XGBoost and Linear SVM classification model. Furthermore, Parameter Tuning was also applied to the rest neural network models during training by

[8] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
[9] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
[10] https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer

monitoring validation or test loss. Many hyperparameters were optimized for maximum performance, such as the learning rate, batch size e.t.c. The hyperparameters that were adjusted on all classification models in each subtask are further elaborated in Section 4.11.

4.1 Text Vectorization Techniques

When using machine learning algorithms on text data, it is essential to convert all text features to numbers. For the classification of documents, such as sentences, each document that is an input must be converted to a fixed-length vector of numbers. A simple and effective model for executing this important task in machine learning is called the Bag-of-Words Model, or BoW.

The Bag-of-Words model is used primarily for document classification task where it captures the frequency of each unique word from the text data. These frequencies are used as features for training the classifiers. It should be noted that the Bag-of-Words model is very limiting since it considers only the occurrence of words in the document, and not their position in said document. The Bag-of-Words model operates by assigning a unique number to each word. So, any document in the dataset is encoded as a fixed-length vector. This vector has a size equal to the length of the vocabulary of recognized words.

It should be noted also that different approaches for building the BOW model were applied for the more traditional classifiers and the more advanced neural networks.

For the most traditional classifiers, such as the XGBoost and Linear SVM classification model, the function called CountVectorizer was first used, which was provided from sklearn. The CountVectorizer is a simple tool for tokenizing a collection of text documents and building a vocabulary of unique words. This tool also performs encoding of these documents with that vocabulary. It manages to count the frequency of words, selecting the most frequent words as features. The main issue of counting only the occurrence of each word is that some words, specifically stopwords such as “the”, appear many times on the corpus, and they would be considered more important than other less known words. Because of their frequency,

they will be given the highest weight in the classification model, skewing that model. An alternative to CountVectorizer, and more effective is a method called TF-IDF. This is an abbreviation for "Term Frequency – Inverse Document". TF-IDF Score combines the definitions of term frequency and inverse document frequency, to produce a composite weight for each term in each document. According to Manning, Raghavan, and Schuetze (2009) [10], term frequency represents the number of occurrences of a term in a specific document. The inverse document frequency of the term across a set of documents signifies how common or rare that word is in the entire document set. The closer it is to zero, the more common the word. Subsequently, TF-IDF word frequency scores attempt to highlight more interesting words, such as those that are frequent in a document but not across documents.

In our approach, we also applied the Sklearn TfidfTransformer to create more accurate representations of the input for the training of the XGBoost and SVM classifiers. The input consisted of encoded vectors with 35000 features for subtask 1 and 60000 features for subtask 2.

For the more advanced neural network models, the Keras tokenizer was used to vectorize the text data. Although the CountVectorizer function has more options in terms of modifying the vocabulary, such as adding n-grams, the Tokenizer from Keras library can be combined with the pad_sequences() function from the Keras deep learning library. This function can be used to pad variable-length sequences to a fixed length. It is essential to use this function since all neural networks require inputs that are of the same shape and size.

The inputs for the neural networks were encoded vectors with 35000 features for subtask 1 and 60000 features for subtask 2.

Another important technique that was used for neural networks for a better representation of the data, is the addition of Pre-Trained word embeddings by using the word embedding method known as GloVe. Word embedding is a term used in natural language processing (NLP) to describe the representation of words for text analysis, typically in the form of a real-valued vector that encodes the meaning of the word. This encoding ensures that the relationship of specific words is also maintained. The main advantage of word representation through embeddings instead of one-hot vectors appears when the vocabulary of the data is relatively big. This is true in this case since the vocabulary of the first dataset contains 35000 features to be learned,

while the second dataset contains 60000 unique features. The standard tokenizers would represent each word as a 35,000-dimensional (and more) vector. With such a large vocabulary, this sparse representation would not be efficient at all. Ideally, similar words should have similar representations, making it easy for the model to generalize what it learns about a word to all similar words.
The bag-of-words (BoW) model, compared to word embeddings, does not capture the position in the text, semantics, co-occurrences in different documents, etc. Furthermore, TF-IDF extracts only lexical level features and cannot capture semantics, compared to word embeddings. Therefore, pre-trained embeddings were used during the implementation of the neural network models.

To load the pre-trained vectors, it is necessary to first create a dictionary that will hold the mappings between words, and the embedding vectors of those words. In this study, the glove.60B.200d.txt was used. which contains 6B Tokens, 400K known words, and their corresponding embedding vectors of size 200. These pre-trained word embeddings were drawn from the GloVe project website[11].

Every word present in the dataset is embedded with the GloVe downloaded text vectors and an embedding matrix is created containing words with their respective vectors.

4.2 Baseline

In this section, the baseline model will be described that was used for the experiments. This is the dummy classifier.

The dummy classifier provides a measure of "baseline" performance, i.e., the success rate by simply guessing. It is a type of classifier that does not generate any insight about the data and instead classifies it using only simple rules. The classifier's behavior is completely independent of the training data since the trends in these training data are not taken into consideration. It is only used as a simple baseline for the other classifiers, implying that any other classifier should outperform it on the given dataset. It is especially useful for datasets where there is a strong likelihood of a

[11] https://nlp.stanford.edu/projects/glove/

class imbalance. It is based on the premise that any analytic approach to a classification problem should outperform a random guessing approach.

4.3 XGBoost Algorithm

XGBoost stands for Extreme Gradient Boosting. XGBoost is a specific implementation of the Gradient Boosting technique, designed to have enhanced performance and speed. Gradient boosting refers to a machine learning technique that can be used for classification and regression predictive modeling problems. It generates a prediction model in the form of an ensemble of prediction models, most commonly decision trees. Thus, the XGBoost algorithm is an ensemble learning method that combines the predictive power of multiple boosted decision trees.

It was created by Tianqi Chen, now with contributions from many developers and belongs to a broader collection of tools under the umbrella of the Distributed Machine Learning Community[12].

4.4 Linear Support Vector Machine

A Support Vector Machine (SVM) is a highly effective and versatile Machine Learning model that can perform linear or nonlinear classification, regression, and even outlier detection. It was developed in 1995 at AT&T Bell Laboratories by Vladimir Vapnik [11].

The fundamental idea behind Support Vector Machines or SVMs is that these algorithms search for the best hyperplane that can be used to classify new data based on the already labeled data. SVMs can perform linear classification, as well as non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces. It should be noted that in general SVMs, and more specifically Non-linear kernel SVMs are not suitable for the classification of

[12] https://github.com/dmlc/xgboost

large data sets, because the training complexity of the SVM is very high. Linear SVMs are less computationally demanding than Non-linear kernel SVMs and are much faster to train on a large dataset. Thorsten Joachims in his paper about text classification with Support Vector Machines [12], states that in the case of a high dimensional, sparse problem with few irrelevant features, linear SVMs achieve great performance while being faster to train and simpler to implement. Therefore, due to the large dataset that was obtained during the parsing, a linear SVM was trained for the classification.

4.5 Convolutional Neural Network

Neural networks (NNs) are a collection of algorithms that are loosely modeled after the human brain and are designed to recognize patterns. They recognize patterns from the input sentences, which are represented as arrays of numbers. Neurons, connections, weights, and propagation functions comprise Neural Networks (NNs).

There are numerous types of neural networks that are available for application and experimentation. They can be classified based on their structure, data flow, the number of neurons used and their density, layers and their depth, activation filters, and so on. Three of the most common NNs are Multi-layer Perceptrons (MLPs), Recurrent Neural Networks (RNNs), and Convolutional Neural Networks (CNNs) (CNNs).

The first deep learning architecture that was employed for sentence classification is a Convolutional Neural Network (CNN).

At the center of any convolutional neural network is the convolutional layer which gives its name to the network. This layer performs a process known as “convolution”. A convolution is a linear operation that, in the context of a convolutional neural network, involves the multiplication of a set of weights with the input, just like in a traditional neural network.

Because this technique was designed for two-dimensional input, the multiplication is performed between the input data, which is presented as a two-dimensional array of numbers, and a two-dimensional array of weights known as a filter or a kernel.

Originally CNNs were built for image processing and have been very successful on image classification. The CNN's immense success on image classification and object recognition can be certified by the results of the annual ImageNet Large Scale Visual Recognition Challenge of 2014 [13], where the Google Net architecture achieved the best results.

Although originally built for image processing, CNNs have also been effectively used for text classification. The model architecture of our convolutional network (CNN) is based on the work of Yoon Kim [14] on sentence classification. Kim's CNN architecture consisted of multiple convolutional neural networks with different hyperparameters, which were used in parallel to extract various features from the input. Kim also experimented with static and dynamic (updated) embedding layers in his paper.

In our approach, we focused on the use of different kernel sizes. Multiple versions of the standard model with different sized kernels were used in a multi-channel convolutional neural network for document classification. This allows the document to be processed at various n-grams (groups of words) at the same time, while the

model learns how to best integrate these interpretations. The architecture of this model is presented on the following figure.

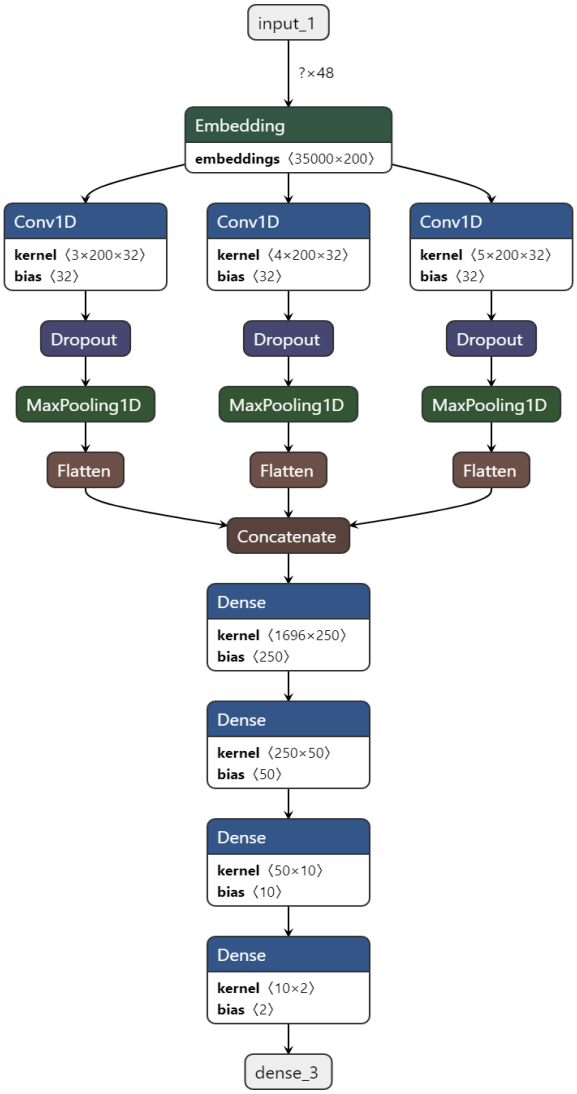

Figure 5 Architecture of CNN model.

Firstly, the input is passed through the embedding layer, with fixed word embeddings, transforming it from a vector to a two-dimensional array of numbers. The transformed input is then passed to three CNN models, named channels, which are responsible for processing 3-grams, 4-grams, and 5-grams of the input.

Each channel is comprised of a one-dimensional convolutional layer with 32 filters and a kernel size whose value is either 3,4 or 5. What follows is a dropout layer which is a regularization method that prevents overfitting, a max-pooling layer which

manages to consolidate the output from the convolutional layer and lastly a flatten layer to reduce the two-dimensional output to one-dimensional for concatenation.

In the following paragraph we will analyze the elements of each channel, that were mentioned, except for the dropout technique, which is described in detail in Section 4.8.

- **1-D Convolution Layer**

After the input is passed through the Embedding layer it is transformed into a sequence of words $w_{1:n} = w_1, \dots, w_n$, where each is associated with an embedding vector of dimension d.

A 1D convolution of width-k is the result of moving a sliding-window of size k over the sentence/input, and applying the same convolution filter or kernel to each window in the sequence, i.e., a dot-product between the concatenation of the embedding vectors in each window and a weight vector u, which is then often followed by a non-linear activation function g.

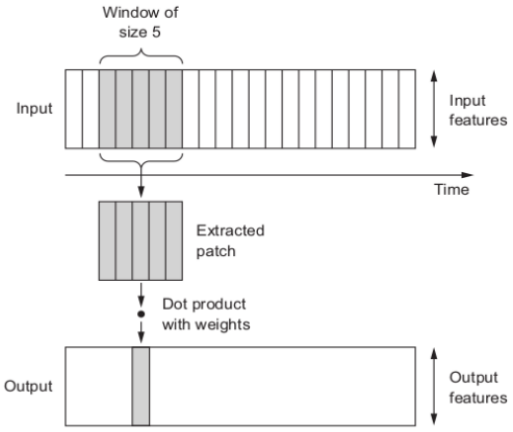

Figure 6 Example of 1D convolution of width 5 on text input.

Considering a window of tokens $w_i, \dots, w_{i+k}$:

$$x_i = [w_i, w_{i+1}, \dots, w_{i+k}] \in R^{k \times d}$$

The convolution filter is applied to each window, and as a result returning scalar values r_i, each for the i_{th} window:

$$r_i = g(x_i * u) \in R$$

In practice we typically apply more filters to each window. This can then be represented as a vector multiplied by a matrix U and with a bias term b:

$r_i = g(x_i * U + b)$, with $r_i \in R^1$, $x_i \in R^{k \times d}$, $U \in R^{(k \times d) \times 1}$ and $b \in R^1$

- **Max Pooling layer**

This layer is responsible for down sampling the feature maps created from the 1-D Convolution layer. It summarizes the presence of features of the feature map in patches. The Maximum Pooling layer calculates the maximum value for each patch of the feature map. The advantage of using pooling layers is that they progressively reduce the number of parameters and computation in the network.

- **Flatten layer**

The flatten layer' purpose is the transformation of the input's shape to one dimensional, so that it can be passed through dense layers or the output layer.

- **Concatenation of channel's outputs**

The three channels' outputs are then concatenated into a single vector and processed by three Dense layers. The activation function, known as ReLU is implemented in the hidden layers of Neural network.

The outputs from the Dense layers are passed through a fully connected sigmoid layer, that creates a probability distribution over all possible target classes. For optimization, the Adam optimizer (Kingma and Ba 2014) was used. In Section 4.11 more information is provided about the Adam optimizer that was used in all neural network models.

4.6 Bidirectional LSTM Model

Long Short-Term Memory (LSTM) networks are a modified version of Recurrent Neural Networks (RNN). Prior to the description of the LSTM architecture, it is important to introduce the mechanisms of the RNNs.

4.6.1 Recurrent Neural Networks

Recurrent Neural Network is a generalization of feedforward neural network and its main feature is its internal memory. This neural network trains on sequential data or time series data, which is optimal for our task since sentences can be considered as sequences of tokens. RNN is recurrent in nature since the output that it produces depend on the prior elements within the sequence input. After the output is generated from each input, it is copied and returned to the recurrent network for latter use. To make a decision, the RNN processes the current input as well as the output that was learned from the previous input. Overall, in a standard neural network, all inputs are independent of one another, however, in RNN, all inputs are related to one another. RNNs are more advanced than other standard neural networks, such as the feedforward artificial neural networks, due to this mechanism. They could also yield better results.

Figure 7 visualizes the Recurrent Neural Network unrolled in time.

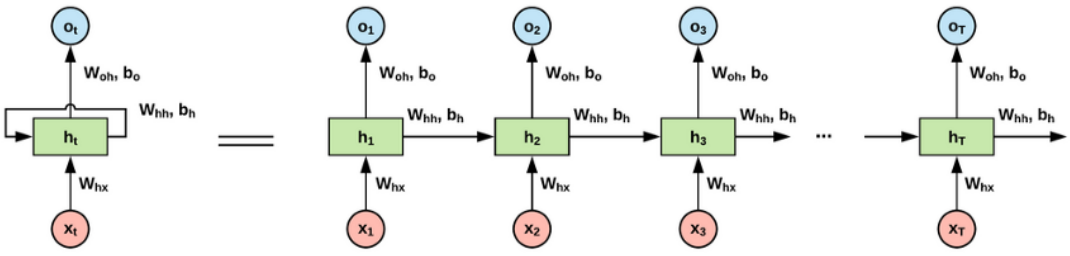

Figure 7 RNN unrolled in time. [13]

[13] https://medium.com/deeplearningbrasilia/deep-learning-recurrent-neural-networks-f9482a24d010

Consider a sequence of input $[x_1, x_2, \dots, x_T]$ Each element x_i, where $i \in [1, T]$, is fed sequentially to the model, implying that at t=1, x_1 is given as an input, at t=2, x_2 is given as an input, and so forth. At each time step t, this model receives two inputs, namely x_t as well as its own output from the previous time step, o_{t-1}. For example, the state from the first-time step (t=1) is remembered and given as input during the second time step (t=2) along with the current input at that time step.

The figure 7, shows all the components of the simplest possible RNN, composed of just one neuron. They are as follows:

- The x_t is the input at step t. It represents the t-th token of the input sentence as a dense numerical vector, since each token is represented by a specific word embedding.

- The h_t is the hidden state at time step t. It represents the memory cell of the RNN. It represents the history of the sentence up to the word x_t. The formula for the current state h_t is $h_t = f(h_{t-1}, x_t)$.

- The o_t is the output of the RNN neuron at the time t.

- Each recurrent neuron has two sets of weights. These weights are presented by the diagram as W_{hx}, W_{hh}. W_{hx} represents the weight for the inputs x_t while W_{hh} depicts the weight for the outputs of the previous time step, h_{t-1}.

The formula of h_t is represented as follows: $h_t = f(W_{hh} * h_{t-1} + W_{xh} * X_t + b_h)$, where h_{t-1} is a single hidden vector representing the previous hidden state, b_h is the bias term, W_{hh} is the weight at previous hidden state, W_{hx} is the weight at current input state. The activation function f can be linear or non-linear, such as a sigmoid, tanh or a ReLU function.

The following formula defines the output at time t: $o_t = g(W_{oh} * h_t + b_o)$, where g is the output layer function, which can be another tanh, sigmoid, or ReLU function, and the weight matrix W_{oh} which is associated with the output.

Because text is naturally sequential, RNN's architecture is more efficient compared to other standard architectures, such as CNNs, when dealing with textual data. Therefore, this technique is a powerful method for text, string, and sequential data classification.

4.6.2 Long Short-Term Memory

Although RNNs are a powerful tool when dealing with sequential data, they suffer from the problem of vanishing gradients, which hampers learning of long data sequences.

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network that can learn long-term dependencies. Hochreiter and Schmidhuber first introduced them in 1997. Their default behavior is to remember information for longer periods of time. It is important to concentrate on the architecture of the LSTM network since they are the primary models for our classification task.

A typical LSTM network is made up of many memory blocks known as cells. Each cell can be trained to decide whether to save, propagate to the output, or discard information from the sequence.

Figure 8 illustrates the architecture of a typical LSTM cell.

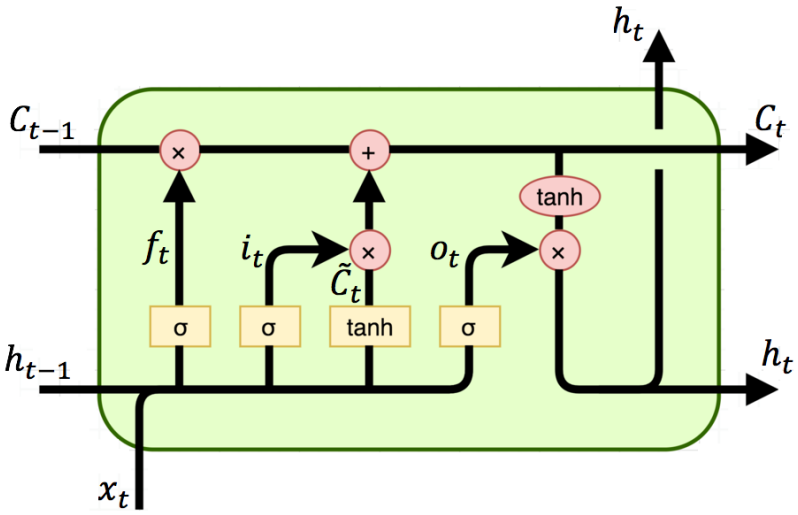

Figure 8 Basic LSTM cell [14]

An LSTM, like a simple RNN, has a hidden state where h_{t-1} represents the previous timestamp's hidden state and h_t represents the current timestamp's hidden state. LSTMs also have a cell state for the previous timestamp represented by $C_{t-1}$and a cell state for the current timestamp which is shown by the figure as C_t.

The key to LSTMs is the cell state, which is represented by figure 8 as the horizontal line running from C_{t-1} to C_t on the top of the diagram. The unique functionality of the LSTM compared to a basic RNN, is namely that it takes as input the memory cell C_{t-1} from the previous unit and determines action to remove or add information to the cell state. This action is carefully regulated by specific structures, named gates.

The function of these gates is to decide what information passes through the memory cell. Each of them contains a sigmoid layer, which is depicted as the letter σ in the figure and a pointwise multiplication operation, which is portrayed as an $\times$. The purpose of the sigmoid neural net layer is to produce a number ranging from zero to one, indicating how much of the information from the previous timestamp should be allowed through. For example, if this layer outputs zero, then it will not keep the memory from the previous unit. If the layer outputs one, then it will store all the memory from the previous unit.

[14] https://towardsdatascience.com/grus-and-lstm-s-741709a9b9b1

An LSTM is comprised of three such gates, to protect and control the cell state. These gates as named as the input gate, the output gate, and the forget gate.

The first gate of the LSTM cell is used when it is fed the inputs x_t and h_{t-1}. Its main purpose is to remove information from the cell state that is no longer needed or has no relevance in understanding data. As it was mentioned before, the forget gate contains a sigmoid layer called the "forget gate layer". It is the mechanism that decides whether to remove information or not. It takes x_t and h_{t-1}, as inputs, and outputs a number between zero and one. The forget gate is defined by the following equation.

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f)$$

x_t is the input of the current timestamp, h_{t-1} is the hidden state of the previous timestamp, W_f is the weighted matrix between the forget and input gate and b_f is the connection bias at timestep t.

The next gate of the LSTM cell that is activated is called the input gate. It is responsible for controlling the amount of old information that is going to be added to the new information and stored in the cell state. This process consists of two parts. First, a sigmoid layer called the "input gate layer" is activated to decide which values to update. Next, a tanh layer is applied on the input to create a vector of new candidate values, $\widetilde{C}_t$, that are included into the cell state. The element-wise multiplication of the outputs of these two layers is then conducted to update the cell state.

These two layers are represented by the following two equations.

$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i)$$

$$\widetilde{C}_t = tanh(W_C * [h_{t-1}, x_t] + b_C)$$

The 1st equation calculates i_t, which refers to the input gate at t. W_i is the weight matrix of the sigmoid operator between input gate and output gate. $\widetilde{C}_t$ is the value

generated by tanh. W_C is the weight matrix of tanh operator between cell state and output, and b_i and b_C are bias vectors.

Now that the network has enough information gathered from the forget gate and the input gate, its priority is to decide and store the information from the new state in the cell state. The previous cell state C_{t-1} gets multiplied with forget vector f_t. Next, the network takes the output vector from the input gate and performs point-by-point addition, which updates the cell state. The result of this operation is a new cell state C_t. In short, the new cell state is calculated from the following equation.

$$C_t = f_t * C_{t-1} + i_t * \widetilde{\mathrm{C}}_\mathrm{t}$$

Lastly, the LSTM cell utilizes the final gate to produce the final output. The output gate's main function is to determine the output of the hidden state, as well as the input for the next hidden state. The output gate firstly receives the values of the current state as well as the previous hidden state as input. Note that this hidden state contains information on previous inputs. The previous hidden state and the current input, which comprise the input, are passed into a third sigmoid function. This function decides what parts of the cell state are the output. Meanwhile, the new cell that is generated from the cell state, is now passed through the tanh function.
The tanh output is then multiplied point-by-point with the output of the sigmoid function to decide what information the hidden state should carry. The final output that is produced, is the hidden state. Finally, the new cell state and new hidden state are carried over to the next time step. The calculations that result to the final output are the depicted below.

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh\,(C_t)$$

o_t represents the output gate at t. W_o is the weight matrix of the output gate, and b_o represents the bias. Finally, h_t is the output of the LSTM cell.

To conclude, the forget gate determines which relevant information from previous steps is required. The input gate decides what relevant information from the current step can be added. Lastly, the output gate finalizes the next hidden state.

4.6.3 Bidirectional LSTM Model Architecture

Bidirectional recurrent neural networks (BRNN) were first introduced in 1997 by Schuster and Paliwal [15], as an improvement over RNNs. A typical Bidirectional recurrent neural network (BRNN) is comprised of two instead of one RNNs. Both RNNs receive the same input sentence, but one trains on the input sequence as-is and the second RNN on a reversed copy of the input sequence. This can provide additional context to the network and result in faster and even fuller learning on the problem. This approach has also been used to great effect with Long Short-Term Memory (LSTM) Recurrent Neural Networks, mainly for speech recognition tasks. One prime example of such task is the works of A. Graves and J. Schmidhuber on the task of framewise phoneme classification [16], with the use of Bidirectional LSTM Networks.

In the present paper, we construct two different Bidirectional LSTM networks. The first one is a Stacked Bidirectional LSTM model, meaning that it consists of more than one Bidirectional LSTM layers, and a Bidirectional LSTM model with self-attention layer (Section 4.7).

The architecture of the 1st Bidirectional LSTM model is presented in the following figure.

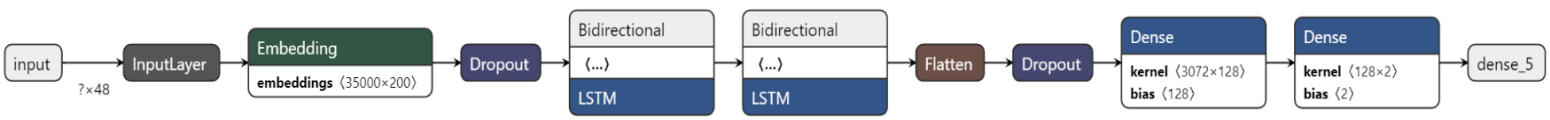

Figure 9 Architecture of Stacked Bi-LSTM model

It contains two bidirectional layers that process the input, which has been first passed through the embedding layer. A flatten layer is then added to transform the shape of the bidirectional layers' output to one dimension, so that it can be passed to a Dense layer that consists of 128 neurons. The last layer is the output layer.

4.7 Bidirectional LSTM Model with Self-Attention Mechanism

As mentioned in the previous section, the third neural network model that was constructed, contained the self-attention layer. To describe the functionality of the self-attention layer, it is important to introduce the attention mechanism.

In natural language processing, the attention mechanism first emerged on Machine Translation tasks. It was an improvement over the encoder-decoder based neural machine translation system. Later, this mechanism and its variants, were used in other applications such as computer vision, speech processing, as well as text classification tasks. This mechanism was first introduced by Bahdanau et al, who proposed the first Attention model in 2015 [17]. Before the introduction of the attention mechanism, encoder decoder-based models were struggling to capture long-range dependencies in text. Although the introduction of LSTMs brought success to minimizing this problem, it persisted in certain cases. Another issue is that there is no way to prioritize some of the input words over others when translating the sentence.

The Attention mechanism manages to fix these issues, by giving more weight to "important" words, while ignoring the rest. It is also an improvement to word embeddings, since word embeddings have a limited meaning for each word identified in the text data. Embeddings are associated with tokens by a straight dictionary lookup, which means that the same token always gets the same embedding regardless of its context in each sentence. This would cause issues, especially when we encounter homonyms in our data. During training, the attention mechanism manages to change the default embeddings, so that the values of each token's embedding are more representative of the token and its context.

As it was mentioned before, the attention mechanism can also be implemented in text classification problems, through self-attention layers. The self-attention module works by comparing each word in the sentence to every other word in the sentence, including itself, and reweighting the word embeddings of each token to add contextual relevance. In our study, we created a second bidirectional LSTM model with one layer. In order to enhance this classification process, we added a self-attention layer for processing sequential data. In our case, the self-attention processes the output of the bidirectional LSTM layer. By applying the self-attention mechanism, the model will look back at previous states, but on a weighted combination of all the LSTM's hidden states. Consider the output of the bidirectional LSTM as a sequence of vectors $[h_1, \dots, h_T]$, where each vector is a word representation. The self-attention mechanism is comprised of three functions.

$$u_t = \tanh(W_h * h_t + b_w), u_t \in [-1,1]$$

$$dt = \frac{\exp(u_t^T * u_w)}{\sum_t \exp(u_t^T * u_w)}, \sum_{i=1}^{T} dt = 1$$

$$s = \sum_t d_t * h_t$$

Each hidden word representation h_i is passed through a dense layer with $tanh$ activation. The resulting vector is multiplied with a context vector u_w. The output of this operation is the attention score d_t of h_t. Next softmax function is applied to all attention scores, so that their sum is equal to one. The final representation s of the sentence is calculated by multiplying the new weights dt with their respective hidden word representations, that were produced from the Bidirectional LSTM layer.

Note that u_w, $W_h$and $b_w$are learned during the training of the model.

The architecture of the 3rd model is displayed on the following figure.

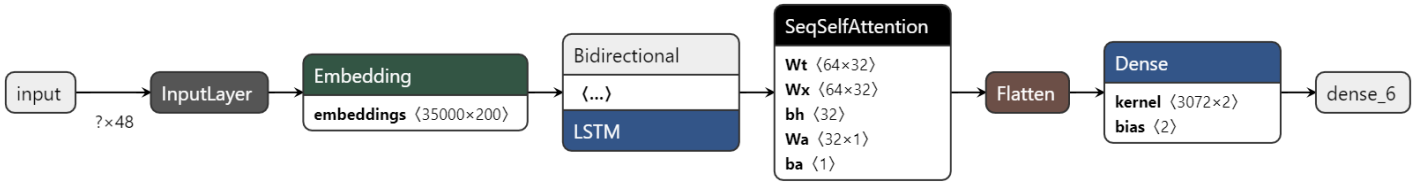

Figure 10 Architecture of Bi-STM model with self-attention mechanism

In this thesis, we created a second bidirectional LSTM model with one layer. We then added the keras self-attention layer[15] to process the outputs produced by the bidirectional LSTM layer. This is a minor variant of the self-attention mechanism that is specialized in processing sequential data and are very effective when in conjunction with LSTM layers. The code for this self-attention module was created and maintained by GitHub user "CyberZHG".

4.8 Regularization

Neural network models, during training are prone to overfitting. Overfitting occurs when a model memorizes the training data to the point where it negatively impacts the model's performance on new data. To prevent overfitting, we apply dropout. Dropout is implemented per-layer in a neural network. It is compatible with most layers, including dense fully connected layers, convolutional layers, and recurrent layers like the long short-term memory network layer. Dropout can be implemented on any or all the network's hidden layers, as well as the input layer. A dropout layer's main function is the action of ignoring or "dropping out" units in a neural network with a certain probability. To better illustrate, consider a simple dense layer. During the process of training, a number outputs from this dense layer are randomly ignored or "dropped out" with a probability of p. This has the effect of making each dense layer be treated as a layer with a different number of nodes. This indicates that the neural network is forced to find new ways to classify the data, thus reducing the chance of overfitting. Without any dropout, our neural networks exhibit substantial overfitting. To further prevent overfitting, we decided to save each model at the epoch where it achieved the best performance. The performance is measured using the loss function, which will be described in the section 4.9.

[15] https://pypi.org/project/keras-self-attention/

4.9 Optimization

For the optimization of the neural network models, we used the Adam optimizer. Introduced by Kingma and Ba (2014)[18], the Adam optimizer is a variant of Stochastic Gradient Descent. It is noteworthy that the architecture of the neural network models and the optimizer used were the same for each subtask. The optimizer's objective during the training of each network, is to update the weights of the network to minimize the cost. Since we were training neural networks for binary text classification, we used the binary cross-entropy / log loss as the loss function.

Cross Entropy loss function:

$$L(p, y) = \sum_{i=1}^{N} y_i \log (p_i)$$

y_i is the true label of the class for a specific sentence. The label values i are either zero (not-weasel) or one (weasel). p_i is the predicted probability distribution of class i from the model. To combat the problem of overfitting during the models' training, we also decreased the value of the learning rate. The learning rate parameter controls how quickly or slowly the neural network model learns the text data. A smaller learning rate may provide us with a model that can learn a more optimal set of weights, although it may need significantly more epochs to train.

4.10 Class Balancing

This section focuses only on the second subtask since the dataset that was used suffers from class imbalance. Having added additional not-weasel, the target class's frequency is highly imbalanced. More precisely the percentage frequency of weasel sentences in our second dataset is approximately 23% of the whole dataset, while the rest data points are non-weasel sentences. This problem, if not resolved, will cause our models and classifiers to have a clear bias towards the majority class, i.e., the non-weasel sentences. Hence, we add weights to the loss function to penalize the misclassifications of the minority class. The class weights were computed

automatically using the compute_class_weight[16] function from the sklearn library in python.

4.11 Hyperparameters

We used Random Search[17] for the parameter tuning of the XGBoost classifier and the Linear SVM. It is a faster technique than standard Grid Search, since it uses random combinations of the hyperparameters to find the best combination for the built models.

The hyperparameter that was tuned for the XGBoost model was the size of Decision trees in that model. Meanwhile, the hyperparameters that were adjusted for Linear SVM were the tolerance parameter (Tol), the maximum number of iterations to be run within the solver (max_iter), as well as the penalty parameter of the error term (C).

The value of hyperparameters for both the traditional classifiers and the neural network models that were adjusted to have an optimal performance appear on the following table.

		Range	
Classifier/Model	Hyperparameter	Subtask 1	Subtask 2
XGBoost	max_depth	10	10
Linear SVC	Tol	0,0001	0,0001
	max_iter	1000	1000
	C	0,1	0,1
CNN Model	batch size	32	32
	Number of epochs	35	35
	Dropout rate	0,5	0,5
	Number of kernels	32	32
	Kernel size	[3,4,5]	[3,4,5]
	Learning rate	0,00001	0,00001
	batch size	64	128

[16] https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html
[17] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

Bi-LSTM Model	Number of epochs	15	15
	Dropout rate	0,5	0,3
	Unit size	32	64
	Learning rate	0,0001	0,0001
Att+Bi-LSTM Model	batch size	64	128
	Number of epochs	20	20
	Dropout rate	0,5	0,5
	Unit size	32	64
	Learning rate	0,0001	0,0001

Table 6 Hyperparameters applied to each model for each subtask.

5 Experiments & Results

5.1 Evaluation Measures

Here the evaluation measures for Hedge Detection are presented for both subtasks. We will give more on the F score, particularly on the 2nd class "weasel".

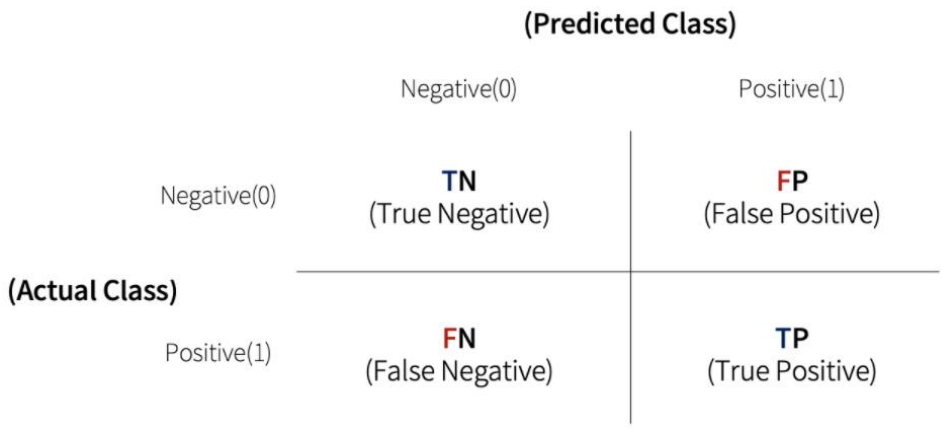

Figure 11 Example of a confusion matrix

The above figure illustrates a basic template for a confusion matrix. Class 1 represents the "weasel" class, while class 0 the "not weasel" class. The components of the confusion matrix are the following:

True Positives (TP): Predicted as positive instances given that, they were positive.

True Negatives (TN): Predicted as negative instances given that, they were negatives.

False Positives (FP): Predicted as positive given that, they were negative instances.

False Negatives (FN): Predicted as negative given that, they were positive instances.

The evaluation of the classifiers was done through Accuracy, Precision, Recall and F-score.

The accuracy metric has the following definition:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

The formula for the precision metric is the following:

$$Precision = \frac{TP}{TP + FP}$$

The recall metric can be calculated as:

$$Recall = \frac{TP}{TP + FN}$$

Finally, the F_1 Score metric is defined as the harmonic mean of the precision and recall metrics.

$$F_1\ score = 2 * \frac{(Precision * Recall)}{(Precision + Recall)}$$

5.2 Experimental Setup

To implement the models, we used Google Collaboratory. The classifiers and models were trained on an Intel(R) Xeon(R) CPU. We provide the source codes that we used for our task in GitHub[18].

5.3 Evaluation

5.3.1 Evaluation for Subtask 1

Before the results on the test data are presented, it is important to examine how much each neural network learned during training and diagnose their performances. For that task we reviewed the learning curve of the validation loss of each model during training. For each epoch, a machine learning algorithm completes a pass of the entire training dataset and outputs its performance using the validation dataset, that we provided. In subtask 1 we trained the CNN model for 35 epochs, the Bi-LSTM model for 15 epochs and the Bi-LSTM model with self-attention for 20 epochs, while evaluating their progress by measuring the loss function on the validation dataset. The progress of the validation loss per epoch on the validation data for each neural network model are illustrated in the following plots:

[18] https://github.com/AgapiouMarios/Hedge-Detection

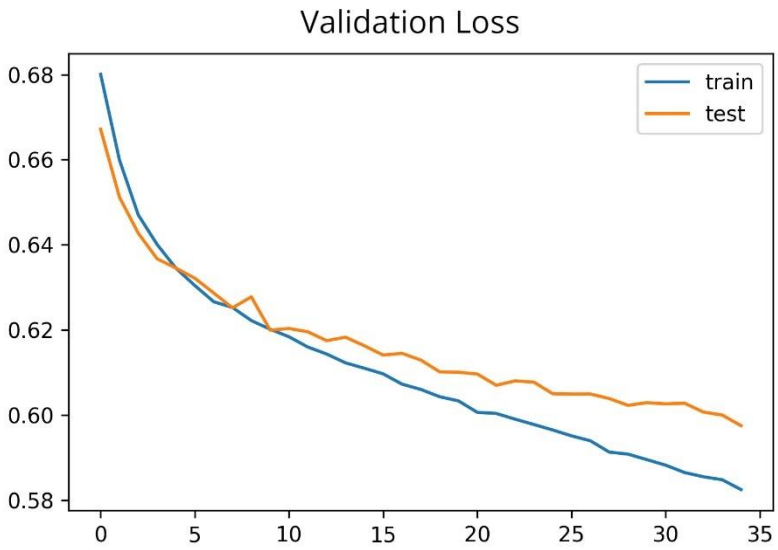

Figure 12 CNN Loss on validation data, subtask 1

The first plot illustrates the progress of the CNN model during training for 35 epochs. This model achieved the optimal value for validation loss in the 35th epoch.

Figure 13 Stacked Bi-LSTM Loss on validation data, subtask 1

It is clear from the above diagram that the Bi-LSTM model that was created seems to overfit. To combat this problem, we used callbacks to save the model at the 13th epoch when it had the lowest validation loss value.

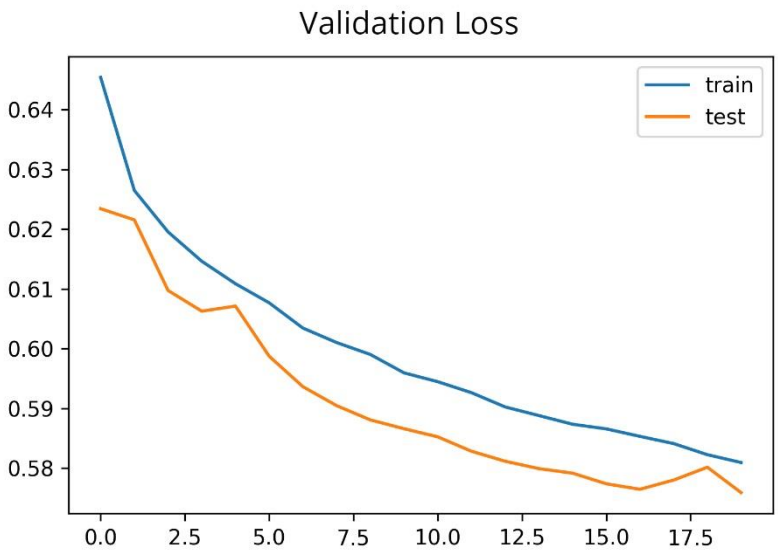

Figure 14 Att+Bi-LSTM Loss on validation data, subtask 1

The Bi-LSTM model with self-attention achieved the lowest validation loss score in the 18th epoch.

The validation results for all classifiers and neural network models are shown in the following figure.

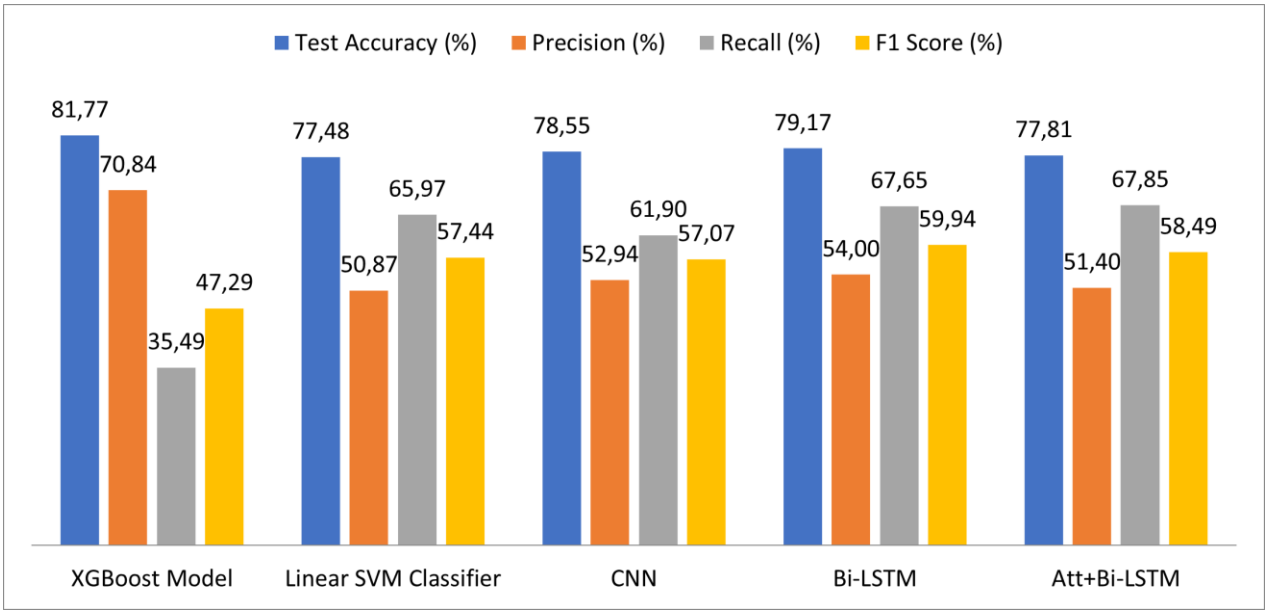

Figure 15 Final percentage results of the five classifiers on the test dataset, subtask 1

The Stacked Bi-LSTM model achieved the highest value of $F_1\ score$ with 70.80%, followed by the second Bi-LSTM model with self-attention.

5.3.2 Evaluation for Subtask 2

For subtask 2, we faced many challenges when training the data, especially overfitting. We had to deal with an imbalanced dataset, which undermines the performance of the models, specifically for the minority class. Therefore, we added class weights to penalize the models more, when they misclassify data points that belong to the minority class, namely the "weasel" class. The following diagrams display the neural network model's performances during training. We evaluated the performance of the models by measuring the loss function on the validation dataset.

Figure 16 CNN Loss on validation data, subtask 2

The CNN model produces the lowest validation loss score 0,51 in the 28th epoch. We used a callback which saves the model in the 28th epoch to avoid overtraining.

Figure 17 Stacked Bi-LSTM Loss on validation data, subtask 2

The stacked Bi-LSTM model struggled during training and started overfitting the dataset immediately after the 3rd epoch. It achieved 0,48 validation loss in the 11th epoch.

Figure 18 Att+Bi-LSTM Loss on validation data, subtask 2

The last model also showed signs of overfitting during the training. From the above plot we observe that the validation loss rapidly increases to 0,67 after the 2nd epoch. It managed to achieve 0,51 validation loss in the 18th epoch.

Finally, we present the performances of all models and classifiers on the testing dataset.

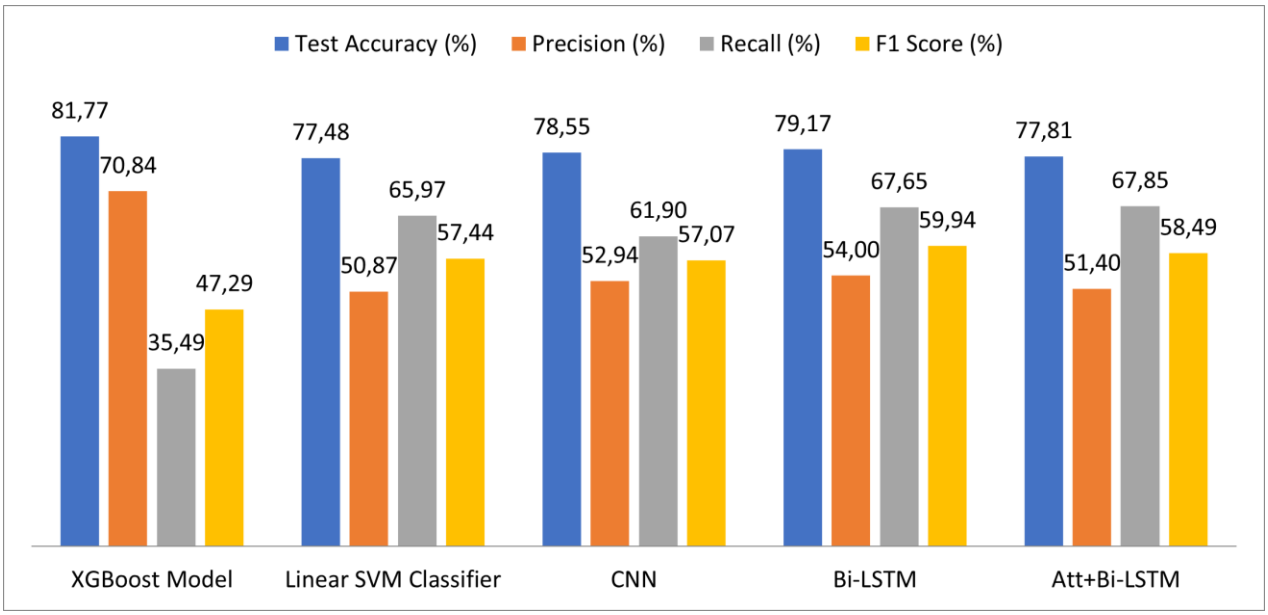

Figure 19 Final percentage results of the five classifiers on the test dataset, subtask 2

For subtask 2, the Stacked Bi-LSTM model also achieves the highest F_1score of 59,94%. The second highest F_1score was produced from the Bi-LSTM model with self-attention. Since subtask 2 was initiated so that we have comparable results with the CoNLL-2010 Shared Task, and more specifically the subtask "Task1W", we provided another diagram which compares our results with the results of all participants of Task1W. Note that we have excluded the XGBoost algorithm from the final plot, since it did not yield good results, due to very low recall score (35%).

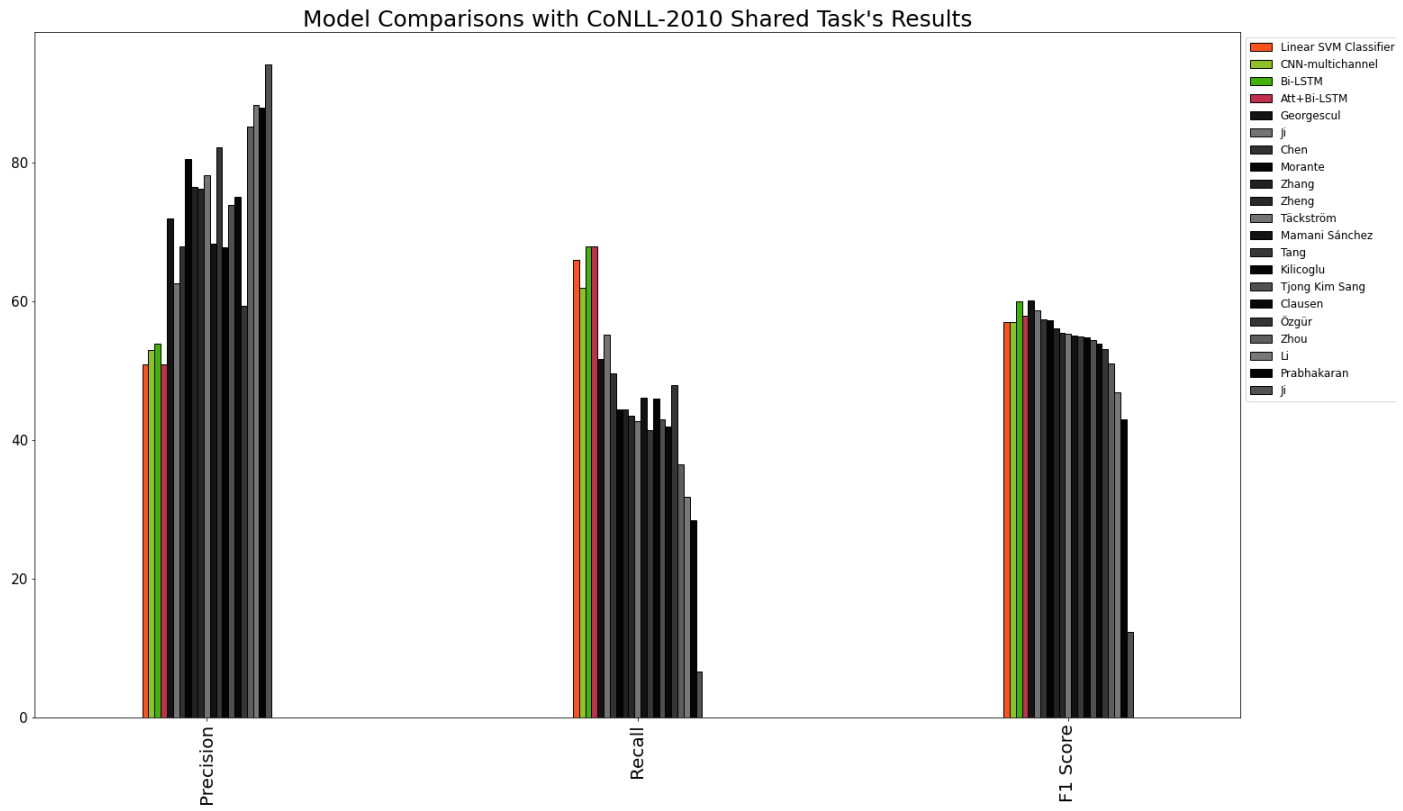

Figure 20 Classifiers' performance comparisons with CoNLL-2010 Shared Task's results

Overall, the results of all four classifiers achieve great results, especially the Bi-LSTM model. Nonetheless, our model's performance could not exceed Georgescul's SVM performance, which produced a F_1 score of 60,20%.

6 Conclusions

This thesis focused on the task of successfully detecting hedges in sentences from Wikipedia articles. We selected the Wikimedia database dump on the 1st of February 2021 to extract sentences tagged with specific weasel templates that signify hedging strategies. We examined the revisions of articles that contained these weasel worded sentences and their adjacent revisions that did not contain weasel templates. We then

extracted the rewritten sentences from these revisions that correspond to each weasel worded sentence and marked them as “not-weasel” sentences.

We then created two different datasets. The first one contained the weasel worded sentences as well as their corresponding non-weasel version. In the second dataset we added additional non-weasel sentences, to change the ratios of the two classes and consequently create a dataset that is similar with CoNLL-2010 shared task’s dataset. The task was split to two subtasks, each referring to one dataset.

Regarding the detection task, after preprocessing the data, we used five different classifiers. The first two classifiers were the XGBoost algorithm and Linear Support Vector Machine. The last three classifiers were neural network models, specifically a CNN model, a Stacked Bi-LSTM model and a Bi-LSTM model combined with a self-attention layer. The Stacked Bi-LSTM model had overall the best scores for both subtasks.

The results from the second subtask were then compared with the results of Task1W from the CoNLL-2010 shared task. Although, all classifiers achieved great results in the second subtask, they did not manage to exceed the highest score achieved from the CoNLL-2010 shared task. This is attributed to possible overfitting of the neural network models, but also to the different dataset used in this thesis and in the CoNLL-2010 shared task. Nevertheless, the above results of the hedge detection system achieved state-of-the-art performance for both subtasks.

7 Bibliography

[1] Lakoff G. Hedges: A study in meaning criteria and the logic of fuzzy concepts. Journal of Philosophical Logic; 1973, p. 458.

[2] Bruce F. Pragmatic Competence: The Case of Hedging. Boston University; 2009.

[3] Hyland K. Talking to the Academy: Forms of Hedging in Science Research Articles. Written Communication; 1996. Written communication, 13(2), pp.251-281.

[4] Marc Light, Srinivasan P, Xin Ying Qiu. The language of bioscience: facts, speculations, and statements in between; 2004 In Proceedings of the HLT-NAACL 2004 Workshop: Biolink 2004, Linking Biological Literature, Ontologies and Databases, pp. 17–24.

[5] Ben Medlock, Ted Briscoe. Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics. Prague, Czech Republic; 2007, pp. 992-999

[6] Szarvas G. Hedge Classification in Biomedical Texts with a Weakly Supervised Selection of Keywords. In Proceedings of ACL-08: HLT, Columbus, Ohio; 2008, pp.281-289.

[7] Ganter V, Strube M. Finding Hedges by Chasing Weasels: Hedge Detection Using Wikipedia Tags and Shallow Linguistic Features; 2009.

[8] Farkas R, Vincze V, Mora G, Csirik J, Szarvas G. The CoNLL-2010 Shared Task: Learning to Detect Hedges and their Scope in Natural Language Text. In Proceedings of the Fourteenth Conference on Computational Natural Language Learning---Shared Task; 2010, pp.1-12. Association for Computational Linguistics.

[9] Tange O. GNU Parallel: The Command-Line Power Tool; 2011.

[10] Manning C, Raghavan P, Schuetze H. Introduction to Information Retrieval; 2009, p. 118-119.

[11] Cortes C, Vapnik V. Support-Vector Networks; 1995, Kluwer Academic Publishers, Boston, pp.273-297.

[12] Joachims T. Text Categorization with Support Vector Machines: Learning with Many Relevant Features.; November 1997, vol. 18.

[13] Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S, et al. ImageNet Large Scale Visual Recognition Challenge.; 2015.

[14] Kim Y. Convolutional Neural Networks for Sentence Classification.; 2014.

[15] Schuster M, Paliwal KK. Bidirectional recurrent neural networks. IEEE Trans Signal Process; 1997, vol. 45, p. 2673–81.

[16] Graves A, Schmidhuber J. Framewise phoneme classification with bidirectional LSTM networks. Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., vol. 4, Montreal, Que., Canada: IEEE; 2005, p. 2047–52.

[17] Bahdanau D, Cho K, Bengio Y. Neural Machine Translation by Jointly Learning to Align and Translate.; 2016.

[18] Kingma DP, Ba J. Adam: A Method for Stochastic Optimization.; 2017.