Project Report S8 - Data Science Project Division

Andreï Radlovic
Antoine Marquis
Ouissal Boutouatou
Hala Chafik
Damla Hatice Selçuk

NLP-Driven Tag Prediction for Sentence Classification in Scientific Abstracts

06/02/2024 - 10/06/2024

Supervisors :

Xujia ZHU (L2S, CentraleSupélec)
Emannuel VAZQUEZ (L2S, CentraleSupélec)
Gilles CHARDON (DPT_SIC, CentraleSupélec)

Contents

1 Introduction

The proliferation of scholarly articles, numbering over 50 million [5] and steadily increasing, especially in fields such as Biomedicine and Computer Science, presents a formidable challenge. While this vast reservoir of knowledge holds great promise for advancing research and understanding, its sheer scale complicates the task of effectively accessing pertinent information. Consequently, there is a pressing need for technological solutions that can expedite the process of information retrieval, thereby optimizing researchers' time and efforts.

During literature review processes, researchers frequently rely on abstracts to quickly assess the relevance of papers. Structured abstracts, featuring well-defined sections outlining objectives, methods, results, and conclusions, facilitate this evaluation process. However, a significant portion of abstracts lacks such organization, impeding efficient information retrieval. Herein lies the importance of sequential sentence classification, which enables the systematic categorization of abstract sentences, streamlining the process of identifying relevant information.

Beyond its immediate utility for researchers, sequential sentence classification holds broader implications for various applications, including automated text summarization, information extraction, and enhanced search functionalities. By simplifying access to scholarly literature, this approach not only enhances research efficiency but also promotes broader engagement with and utilization of available knowledge resources.

2 Exploratory data analysis

The dataset used in this project was constructed through a rigorous process outlined in the paper [6]. The first task into building a comprehensive set of classified abstract sentences in computer science applied to social media is to retrieve a raw set of abstracts from the *arXiv* platform, leveraging its API for efficient data retrieval. The search was focused on articles containing terms related to social media in the title within the computer science category. Subsequently, the collected articles were filtered to include only those with English abstracts, considering the predominant use of the English language in scientific literature. To classify the abstracts, a crowdsourcing solution was implemented using the Amazon Elastic Compute Cloud platform. Each sentence was classified into five main categories: **BACKGROUND, METHOD, RESULT, OBJECTIVE** and **CONCLUSION**.

The crowdsourcing platform, accessible via a web interface, enabled users to classify unclassified abstracts according to predefined categories. To incentivize participation, a gamification component was integrated, rewarding users with points for each classified abstract and offering different levels based on accumulated points.

Despite concerted efforts to encourage engagement, participation remained uneven, with a minority of users contributing significantly, resulting in a dataset with the following distribution :

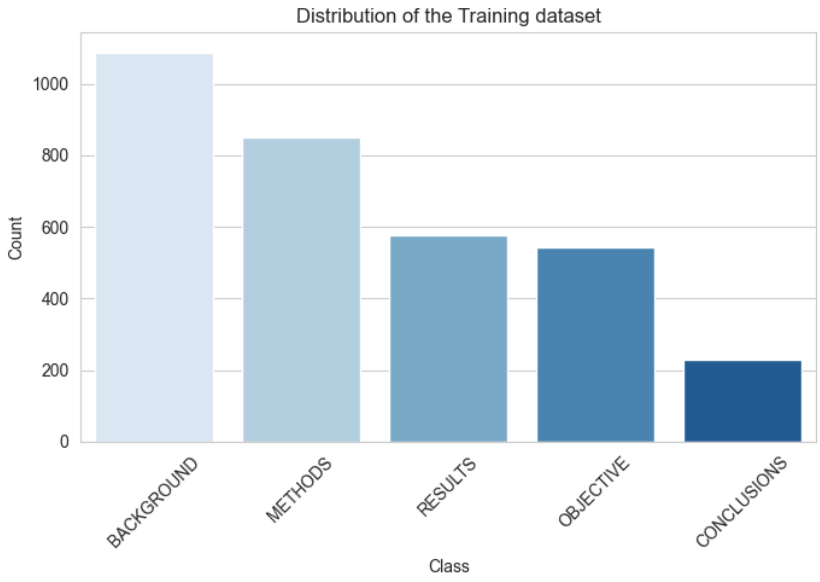


Figure 2: Target variable distribution - Train Data

As shown in Figure 2, Figure 3 and Figure 4, the different train, validation and test datasets have similar class distributions. This balanced distribution is critical

Figure 3: Test data partition

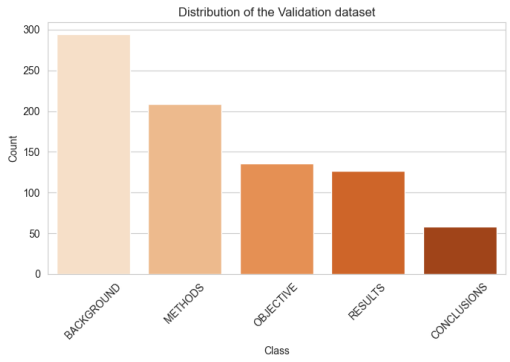


Figure 4: Validation data partition

for avoiding performance inconsistencies in practical applications.

2.1 Data loading and initial exploration

The initial phase of our project focused on the manipulation and understanding of the provided data, essential for subsequent processing and analysis.

Our dataset was delivered in a .txt file format containing various types of information: each abstract was grouped, and each sentence within the abstract was tagged with its position (e.g., 1st sentence, 2nd sentence, etc.) and labeled according to one of the five categories mentioned in the previous section.

To transform these text files into a usable tabular format, we utilized the **Pandas library** [8], a staple in Python for handling such data. The Pandas library allowed us to efficiently read, manipulate, and analyze the data, facilitating our workflow. The following code snippet demonstrates how we converted the raw text data into a structured dataframe:

```
train_df = pd.read_csv('path_to_file.txt', delimiter='\t', header=None,
                       names=['text_position', 'label', 'sentence'])
```

As a result, we obtained a dataframe with three columns, each representing an aspect of the sentence data: the **normalized position** of the sentence within the abstract, the label denoting its category, and the sentence text itself. This structured format enabled us to easily navigate and manipulate the dataset, setting the stage for more advanced data processing and analysis tasks.

N.B : Using the normalized position of a sentence within an abstract rather than its actual position provides a standardized metric across documents of varying lengths. Abstracts differ in their total number of sentences, making absolute positions unreliable for consistent comparison. Normalizing the sentence position to a proportion of the total number of sentences ensures that this feature reflects a comparable stage across all abstracts, enhancing model robustness and generalizability.

2.2 Data correction and label refinement

In our dataset, when we applied various machine learning and deep learning algorithms discussed in the upcoming sections 5 6, we observed that the algorithm rarely detected sentences classified as conclusions. This anomaly led us to suspect potential mislabeling within the dataset. Upon further analysis, we discovered, for example, instances where sentences that would more appropriately be labeled as background information were instead classified as conclusions. This mislabeling issue was detrimental to the performance of our machine learning models. To address this, we considered relabeling our dataset by dividing the task among different team members. However, due to time constraints, we were unable to undertake this relabeling process, which could have significantly improved the accuracy and reliability of our models.

2.3 Label encoding

In our preprocessing workflow, we applied the **'LabelEncoder'** from the library **sklearn.preprocessing** [9] to transform the categorical class labels in our datasets into numerical form. This encoding is essential for the compatibility of our machine learning algorithms with categorical data. After fitting the encoder on the training data to establish a consistent mapping, we applied this mapping to convert labels in the training, testing, and validation datasets into a new numerical column, **label_num**. This process ensures that all datasets use the same encoding scheme, facilitating a straightforward evaluation and comparison across different model training stages.

2.4 Summary statistics

2.4.1 Text Length Analysis

In our analysis, we utilized Kernel Density Estimates (KDE) to visualize the distribution of text lengths across different classes in the training dataset. This approach allows us to assess the variability and distribution tendencies of text length within each class. By overlaying the density plots for each class, we can directly compare their text length characteristics. This visualization helps in identifying any class-specific patterns or anomalies in text length, which are crucial for optimizing our feature engineering and enhancing model performance. Understanding these distributions also aids in better interpreting how text length could influence the classification process.

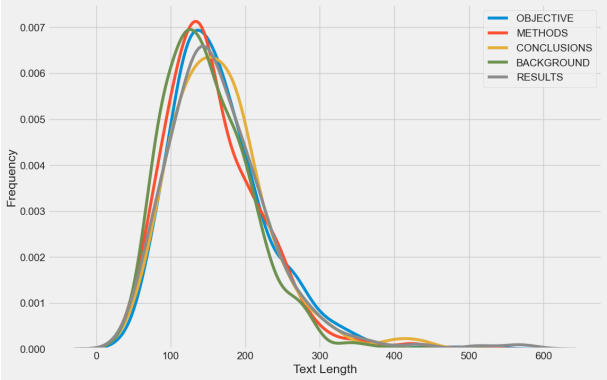


Figure 5: Distribution of Text Length by Category in the Training dataset

The graph 5 depicting the distribution of text lengths across different abstract sections— Objective, Methods, Conclusions, Background, and Results—reveals through the similar shapes and significant overlap of the density plots that text length may not be a robust feature for distinguishing between these categories, as all sections exhibit a central tendency around 100 to 200 characters.

2.4.2 Text Readability Analysis

Moreover, we have extended our exploration of textual features within the dataset by incorporating the Flesch Reading Ease score , a widely recognized readability metric. The Flesch Reading Ease (FRE) Score Formula is given by:

$$\text{FRE} = 206.835 - 1.015\left(\frac{\text{total words}}{\text{total sentences}}\right) - 84.6\left(\frac{\text{total syllables}}{\text{total words}}\right)$$

This score provides insights into the readability of the text, with higher scores indicating text that is easier to read. By plotting the distribution of these scores for each class, we aim to uncover any patterns in readability that might differentiate between the sections, such as whether certain sections tend to be more complex or simpler in their language usage. This visualization could highlight unique readability characteristics of each section, potentially serving as an effective feature in improving our text classification model's performance.

The graph 6 illustrating the distribution of Flesch Reading Ease scores across different sections of abstracts reveals closely aligned distribution curves with a peak around 50. This peak suggests a moderate level of readability common to all sections, indicative of academic language which generally targets an educated audience. Just like the previous section, the close alignment and overlap of the curves across all categories indicate that, in terms of readability, the text across different sections does not vary significantly. This uniformity in readability suggests that factors other

than Flesch scores might be necessary to effectively differentiate between these sections for classification purposes.

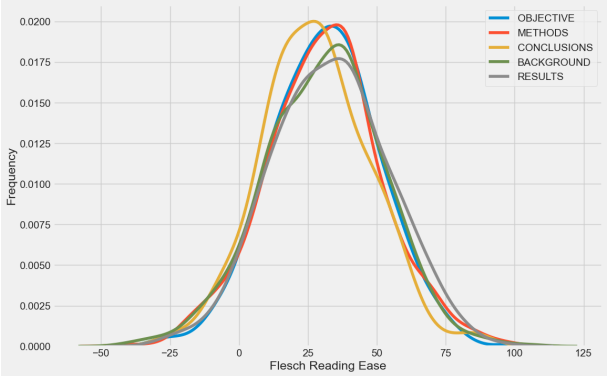


Figure 6: Distribution of Text Readability by Category in the Training dataset

3 Data preprocessing

Text preprocessing is a crucial step in natural language processing (NLP) and text analysis, involving various techniques to transform raw text data into a more structured and analyzable format. This process is essential because text data often contains noise, inconsistencies, and irrelevant information that can hinder effective analysis and model performance. Preprocessing tasks like noise reduction help remove punctuation, special characters, and irrelevant symbols, making the text cleaner and easier to work with. Normalization techniques, such as stemming and lemmatization, standardize different forms of words, ensuring that words conveying the same meaning are treated consistently. Tokenization breaks down text into smaller units like words or phrases, facilitating further analysis steps like feature extraction. Stopword removal eliminates common words that add little semantic value, thereby reducing noise and improving analysis efficiency. The aforementioned techniques represent the essential steps of data preprocessing in NLP, which can help creating data that can effectively be used to train machine learning algorithms and enhance results.

3.1 Tokenization

Tokenization is the technique of dividing a string of text into smaller parts such as words, phrases, sentences, or symbols, known as tokens. This method is fundamental to preparing text for further processing and analysis in natural language tasks. In other words, tokenization transforms a text document from a continuous string of characters into a structured, numerical data format, which is immediately usable in machine learning. The tokens derived from a document can be directly used as a vector to represent the document in numerical form.

There are numerous open-source tools available that facilitate the tokenization of text, enabling the extraction of meaningful units for computational analysis. In this project, we focus on the tools provided by the NLTK library**??**.

- **Word tokenizer :**
  The **'word_tokenize'** function from NLTK's **'nltk.tokenize'** module is adept at breaking down text into individual words and punctuation.

```
from nltk.tokenize import word_tokenize
s = "I can accept failure, everyone fails at something. But I
    can't accept not trying."
word_tokenize(s)
>>>["I", "can", "accept", "failure", ",", "everyone", "fails",
    "at", "something", ".", "But", "I", "can", "'", "t", "
    accept", "not", "trying", ",","said","Micheal","Jordan","."
    ,"3.88"]
```

- **Punctuation-based tokenizer :**
  This tokenizer splits the sentences into words based on whitespaces and punctuations.

```
from nltk.tokenize import wordpunct_tokenize
s = "I can accept failure, everyone fails at something. But I
    can't accept not trying, said Michael Jordan. 3.88"
wordpunct_tokenize(s)
>>>["I", "can", "accept", "failure", ",", "everyone", "fails",
    "at", "something", ".", "But", "I", "can", "'", "t", "
    accept", "not", "trying", ",","said","Micheal","Jordan","."
    ,"3",".","88"]
```

- **Treebank word tokenizer :**
  This tokenizer applies a range of standard rules for English word tokenization. It separates phrase-terminating punctuation such as (?!.;,) from adjacent tokens and preserves decimal numbers as single tokens. Additionally, it includes rules to correctly handle English contractions.
  For example 'don't' is tokenized as ['do', 'n't'].

```
from nltk.tokenize import TreebankWordTokenizer
s = "I can accept failure, everyone fails at something. But I
    can't accept not trying, said Michael Jordan. 3.88"
tokenizer = TreebankWordTokenizer()
tokens = tokenizer.tokenize(sentence)
>>>["I", "can", "accept", "failure", ",", "everyone", "fails",
    "at", "something", ".", "But", "I", "ca", "n't", "accept",
    "not", "trying", ",","said","Micheal","Jordan.","3.88",]
```

These tokenization methods provided by the NLTK library illustrate different approaches to breaking down text for natural language processing. In this project, we experimented with various tokenizers, including the simple Word Tokenizer and the RegexpTokenizer initialized with a pattern, which captures both words and numbers, including decimals.

3.2 Stemming & Lemmatization

For grammatical reasons, documents are going to use different forms of a word, such as write, writing and writes. Additionally, there are families of derivationally related words with similar meanings. The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form.

Stemming usually refers to a process that chops off the ends of words in the hope of achieving goal correctly most of the time and often includes the removal of derivational affixes.

Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base and dictionary form of a word

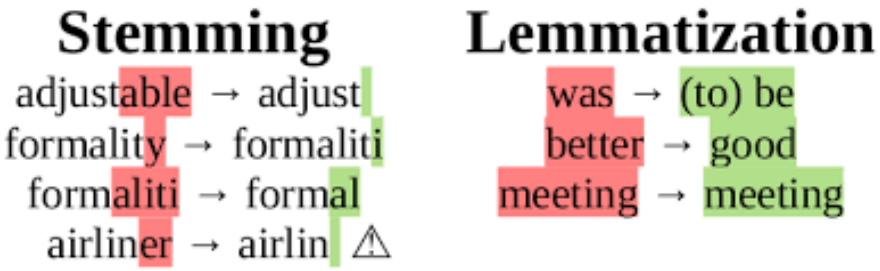


Figure 7: Stemming versus lemmatization. Source: Kushwah 2019 [6]

3.2.1 Stemming algorithms

There are several stemming algorithms implemented in **NLTK Python library** [1]:

- **PorterStemmer**: Utilizes Suffix Stripping for stem generation, known for its simplicity and speed. It applies a set of 05 rules in phases, disregarding linguistic principles. Consequently, it may not consistently produce valid English stems, as it lacks a lookup table and relies solely on algorithmic rules to determine suffix stripping.

- Python's NLTK offers **SnowballStemmers**, allowing users to create stemmers for non-English languages by defining custom rules.

- **LancasterStemmer (Paice-Husk stemmer)**: An iterative algorithm employing externally stored rules. It employs a table of approximately 120 rules indexed by the last letter of a suffix. Each iteration attempts to find a rule based on the word's last character, with rules specifying deletion or replacement of endings. The process terminates under certain conditions, such as when a word begins with a vowel and has only two remaining letters, or when a word starts with a consonant and has only three characters left.

3.2.2 Lemmatization algorithms

- **WordNet Lemmatizer**: Utilizes lexical knowledge base WordNet to find the lemma (base form) of a word. It maps inflected forms to their corresponding lemma by considering morphological relationships between words.

- **SpaCy Lemmatizer**: Built on statistical models, SpaCy's lemmatizer analyzes word context to determine the most likely lemma. It employs deep learning techniques to generate lemmatization results.

- **Stanford CoreNLP Lemmatizer**: A rule-based lemmatizer that employs part-of-speech tagging and morphological analysis. It applies linguistic rules and patterns to transform words into their base forms.

3.3 Data cleaning

3.3.1 Lowercasing and Punctuation removal

Lowercasing is a text preprocessing step where all letters in the text are converted to lowercase. This is done to ensure that the algorithm treats identical words consistently, regardless of their case in different contexts.

Punctuation removal involves stripping all punctuation marks, such as periods, commas, exclamation marks.., from the text. This simplification helps to emphasize the words themselves, ensuring the processing focuses solely on textual content.

3.3.2 Stop-words removal

In text processing, stopwords are words that are filtered out before or after the processing of text. These are usually the most common words in a language, such as "and", "the", "a", and similar articles, conjunctions, and prepositions. They are called "stopwords" because they stop the processing algorithm from focusing on words that carry more meaning and are more relevant to the analysis. The rationale behind eliminating stopwords is to reduce the dataset to its most informative components, thus enhancing the efficiency of natural language processing tasks, such as our sentence classification.

Moreover, removing stopwords from the corpus significantly reduces the data size due to their high frequency. This reduction in size facilitates quicker computations on text data and enables the text classification model to work on fewer features, which enhances its robustness and efficiency.

The NLTK library includes a collection of stopwords which can be utilized to filter out these words from our text.

3.3.3 Numbers replacement

In text preprocessing, replacing numbers with a special character or token is a common step to normalize the treatment of numerical data within text. This process involves substituting all numeric characters or sequences with a designated placeholder, such as "@". This technique is particularly useful because numbers often do not carry intrinsic meaning in the context of linguistic analysis and can introduce noise into the dataset.

This replacement not only simplifies the textual data but also aids in reducing model complexity by eliminating the potentially vast array of unique numerical values that the model would otherwise need to interpret. This step ensures that numerical

data does not disproportionately influence the algorithm's learning process, thereby maintaining a focus on more relevant textual features.

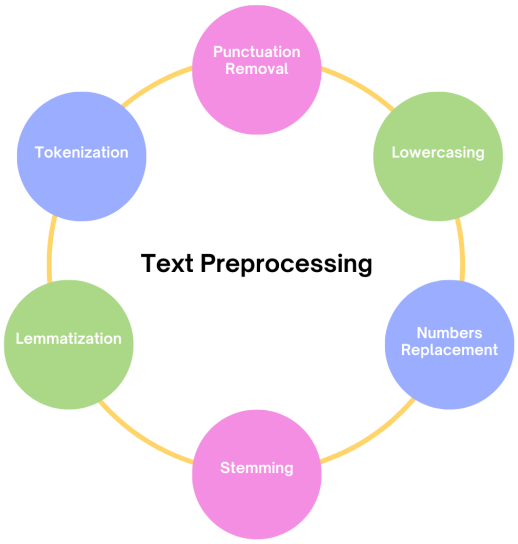


Figure 8: Text Preprocessing Techniques for NLP

3.4 Tailored preprocessing for different algorithms

In our project, we took a thoughtful approach to text preprocessing by tailoring our methods to suit different machine learning algorithms. For models like Bag of Words, CountVectorizer, and TF-IDF, we applied the techniques mentionned in the previous paragraphs; stemming, lemmatization, stopword removal, punctuation removal, and replacing numbers with a special character. These techniques help in reducing dimensionality and noise for such algorithms, which rely heavily on word frequency and occurrence patterns.

However, for algorithms that handle sequential data well, like LSTMs, we kept things simpler by just replacing numbers with a special character. This way, we preserved the natural flow and context of the text, which is crucial for these models to capture patterns over time.

4 Data vectorization - Feature encoding

Modeling text can be intricate, given that machine learning algorithms require structured inputs and outputs rather than raw text. Consequently, textual data needs to be converted into numerical vectors to encapsulate linguistic characteristics. This process involves feature extraction or encoding to form vectors suitable for machine and deep learning algorithms post tokenization of sentences.

" Within language processing, vectors x are generated from textual data to represent different linguistic attributes present in the text. *This process is known as feature extraction or feature encoding.* " [2]

Once we have transformed our sentences into a token list, it becomes essential to convert each token or sentence into vectors for effective utilization by machine and deep learning algorithms.

4.1 Bag-of-words

A bag-of-words model, or BoW for short, is a method of extracting features from text for use in modeling, such as with machine learning algorithms.

The approach is straightforward and adaptable, suitable for various ways of extracting features from documents.

A bag-of-words is a representation of text that indicates the occurrence of words within a document. It consists of:

- A vocabulary of known words.

- A measure of the presence of these known words.

This method is referred to as a "bag" of words because it disregards any details about the order or structure of words in the document. The model simply focuses on whether known words appear, rather than their placement within the document.

The underlying idea is that documents with similar content are alike. Moreover, solely based on the content, we can derive insights about the document's meaning.

The bag-of-words can be as uncomplicated or intricate as you wish. The complexity arises from determining the design of the known word vocabulary and how to evaluate the presence of these known words. In our scenario, the vocabulary is predetermined by the tokens present in the training corpus.

4.1.1 Countervectorizer

The primary idea here is to use a bag-of-words method. Each sentence is assigned a vector with dimensions equal to the total unique words in the dataset. This method

converts textual data into a matrix displaying token frequency as illustrated by 9. To achieve this, we will utilize SciKit Learn's CountVectorizer.

This can be visualized as a 2D matrix. One dimension signifies the entire vocabulary (1 row for each word) and the other dimension represents the sentences, with each sentence in a column.

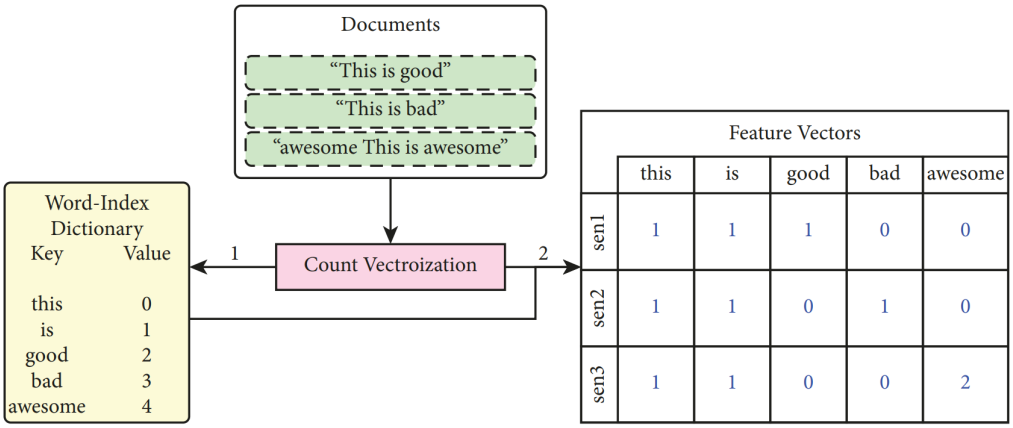


Figure 9: illustration of count vectorization

4.1.2 Tf-idf Vectorizer

Scoring word frequency poses a challenge as common words tend to overshadow rarer, potentially more informative ones. One solution is to adjust word frequency based on their occurrence across all documents, penalizing common words like "the" that are prevalent overall. This strategy, known as Term Frequency – Inverse Document Frequency (TF-IDF), evaluates:

- **Term Frequency**: the word's frequency in the current document.
- **Inverse Document Frequency**: how uncommon the word is across documents.

Mathematically speaking, TF-IDF scores are calculated as the product of the following two quantities :

$$TF(t,d) = \frac{\text{frequency of term t in document d}}{\text{total number of words in the document}}$$

$$IDF(t,d) = log(\frac{\text{total number of documents in the dataset}}{\text{number of documents containing the term t}})$$

This method assigns weights to words, acknowledging that not all words hold equal importance or interest. The resulting scores emphasize distinct words that

offer valuable information within a specific document. Consequently, rare terms have a high IDF, while common terms have a lower IDF.

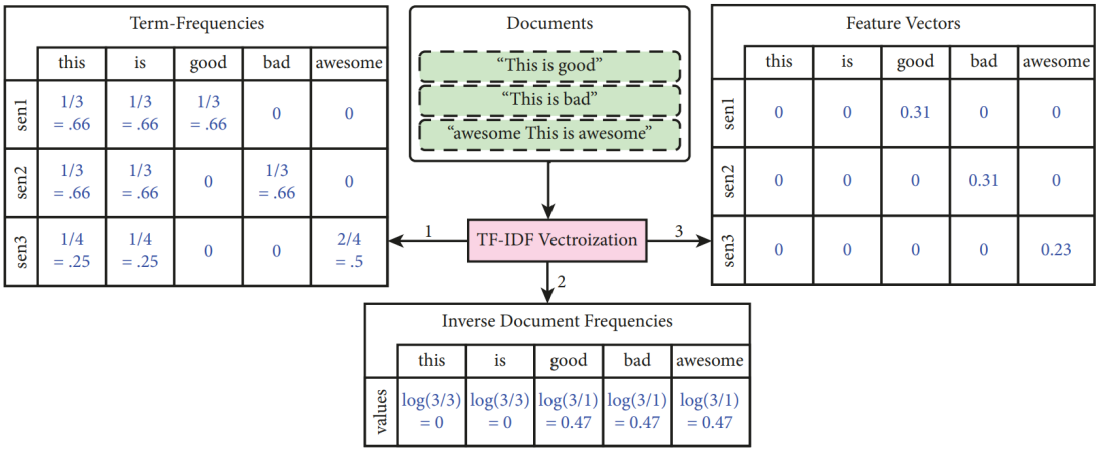


Figure 10: Illustration of Tf-Idf vectorization

4.1.3 Limitations of bag-of-words

The bag-of-words model is very simple to understand and implement and offers a lot of flexibility for customization on your specific text data.

It has been used with great success on prediction problems like language modeling and documentation classification.

Nevertheless, it suffers from some shortcomings, such as:

- **Vocabulary:** The vocabulary requires careful design, most specifically in order to manage the size, which impacts the sparsity of the document representations.

- **Sparsity:** Sparse representations are harder to model both for computational reasons (space and time complexity) and also for information reasons, where the challenge is for the models to harness so little information in such a large representational space.

- **Meaning:** Discarding word order ignores the context, and in turn meaning of words in the document (semantics). Context and meaning can offer a lot to the model, that if modeled could tell the difference between the same words differently arranged ("this is interesting" vs "is this interesting"), synonyms ("old bike" vs "used bike"), and much more.

4.2 Word embedding

Word embeddings are a technique that translates words into numerical vectors, enabling machines to understand and process human language. By mapping words from a vocabulary to vectors of real numbers, word embeddings position these vectors in an n-dimensional space. This spatial arrangement captures the meanings and relationships between words, placing those that appear in similar contexts closer together. This transformation into numeric values is crucial as many machine learning models require vector representations to function effectively.

4.2.1 Glove

GloVe (Global Vectors for Word Representation) is a well-known word embedding approach that produces efficient word representations by training on aggregated global word-word co-occurrence data from a corpus. This unsupervised, count-based model combines the local context window technique with global matrix factorization, taking into account both immediate word context and overall co-occurrence statistics. This method allows GloVe to effectively obtain semantic links and evaluate linguistic patterns on a large scale.

GloVe is particularly effective for finding similar words. By calculating the Euclidean distance or cosine similarity between word vectors, it can determine the linguistic or semantic closeness of different terms. This approach can sometimes highlight uncommon but pertinent words that might not be familiar to most people. For instance, when considering the word "king," GloVe identifies the closest words as:

- king
- queen
- monarch
- crown
- prince
- ruler
- emperor
- sovereign

Additionally, using a single number to measure word similarity can be limiting, as it doesn't capture the detailed relationships between words. For instance, "strong" and "stronger" share a comparative relationship where "stronger" is a more intense

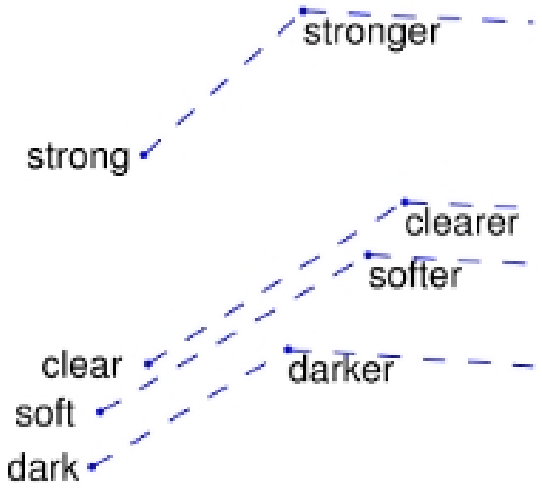


Figure 11: Comperative - Superlative Links

form of "strong." To better capture these nuances, GloVe uses vector differences between word pairs.

By representing the difference between the vectors for "strong" and "stronger," GloVe can effectively capture the comparative meaning. This approach allows GloVe to understand specific distinctions and relationships between words, providing a richer representation than simple similarity measures. This design helps the model to more accurately process and interpret language.

4.2.2 Word2Vect

Word2Vec is another word embedding approach that creates word embeddings from dense representations using two key model architectures: continuous bag-of-words (CBOW) and skip-gram. Both designs rely on neural prediction models. CBOW predicts a target word based on its context, whereas skip-gram predicts context words based on the target word. This predictive technique assignes probability to words, making it particularly effective for word similarity tasks. Word2Vec successfully converts an unlabeled corpus into labeled data by mapping each target word to its context words. However, it does not automatically use global information and does not clearly specify sub-linear relationships. More insight on how those two architectures work as follows:

The Continuous Bag of Words (CBOW)

The Continuous Bag of Words (CBOW) model predicts a target word by utilizing its surrounding context words. Essentially, it leverages the words around a specific word to predict the word in the middle. The model combines all the context words into a single vector, which is then used to predict the target word.

For example, in the sentence “The weather is nice today,” if "nice" is the target word, the CBOW model will use "The", "weather", "is", and "today" as context to predict the word "nice". This model is particularly useful for smaller datasets and is faster than the Skip-Gram model.

Continuous Skip-Gram Model

The Skip-Gram model predicts the neighboring context words from a target word. Instead of using multiple words to predict one, it utilizes a single word to forecast its neighboring words. For instance, in the sentence “The weather is nice today,” the Skip-Gram model would take "nice" as the input and predict “The”, “weather”, “is”, and “today”.

This model is particularly effective with large datasets and infrequent words. While it demands more computational resources than the CBOW model because it predicts multiple context words, it offers several advantages. These include a superior ability to capture semantic relationships, handle rare words, and adapt to diverse linguistic contexts.

4.2.3 Fasttext

FastText is an open-source library created by Facebook AI Research, aimed at delivering scalable solutions for text classification and representation. It excels in processing large datasets both swiftly and accurately, and it builds on the word2vec model with notable enhancements.

FastText addresses some limitations of word2vec by treating each word as a series of character n-grams rather than as single, indivisible entities. While word2vec creates a single vector for each word, FastText breaks down words into smaller character n-grams, capturing more granular information.

For example, consider the word “language” with character trigrams (n = 3):

Character trigrams:

<lan, ang, ngu, gua, uag, age>

In FastText, the input for the word “language” includes both these character n-grams and the word itself:

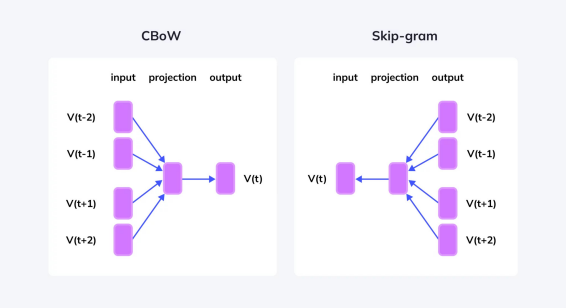


Figure 12: Word2vec Architecture

<lan, ang, ngu, gua, uag, age> and <language>

This method allows FastText to produce embeddings for words not present in the training corpus by combining the embeddings of their n-grams. This overcomes a key limitation of traditional word2vec models. Thus, FastText not only utilizes the skip-gram and continuous bag-of-words (CBOW) models from word2vec but also enhances them by incorporating subword information, leading to improved handling of rare and out-of-vocabulary words.

4.3 Sentence embedding

In this section, we are exploring an other approach to perform the embedding of the abstracts. Indeed, we previously focused on the words of each sentence, and vectorized them thanks to multiple methods in order to predict the label of the whole sentence.
Another approach would be to see the sentence as the object we're performing our algorithms on, and not like a product of the words it contains.
Therefore the goal of this section is to find a way to transform the sentence as a vector in a n-dimensional space to predict its label through the use of ML algorithms.
The first intuitive method would be to use the embedding of the words to produce the embedding of the sentence. Indeed, the vector representing the sentence would be a n_0-dimensional vector, n_0 being the maximum of a vector amongst the ones representing the words contained in the sentence. The components of the sentence vector would then be the mean (weighted or not) of the components of the words it contains, where 0-filling method would be used to eliminate dimensionality issues within words.
However, this method is not as precise as we would like it to be and above that it implies that we already have an embedding of the words contained in the sentences.
In the following subsections, we will explore two more complex methods which allow us to vectorize sentences. This transformation is then used to perform the classical ML algorithms and labelize the sentences.

4.3.1 SBERT

In order to understand how the Sentence-BERT model works, one first needs to be familiar with BERT. BERT (Bidirectional Encoder Representation of Transformers) networks are formed by stacking multiple encoder parts of Transformer networks used to vectorize words, and capture their meaning : the vectorization is really efficient as it is based on context. However, if we wanted to use them in the context of this section, there would be two main problems : firstly, the aggregate sentence vector obtained from the words vectors of BERT (via a mean pool technique f.e.) isn't of good quality, and we would rather use GloVe embedding of the words to obtain a mean sentence vector which would be of better quality (see table below). The second problem is that BERT compares the sentence to every other sentence

of its database to determine the similarity score, thus leading to extremely long processing times. For example, finding in a collection of 10 000 sentences the pair with the highest similarity requires approximately 50 millions computations, which would take roughly 65 hours to compute.

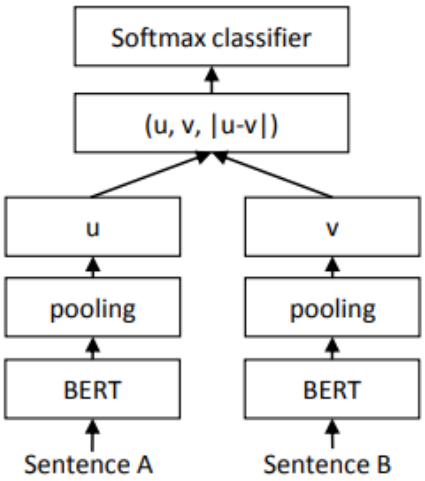
Model	STS12	STS13	STS14	STS15	STS16	STSb	SICK-R	Avg.
Avg. GloVe embeddings	55.14	70.66	59.73	68.25	63.66	58.02	53.76	61.32
Avg. BERT embeddings	38.78	57.98	57.98	63.15	61.06	46.35	58.40	54.81
BERT CLS-vector	20.16	30.01	20.09	36.88	38.08	16.50	42.63	29.19
InferSent - Glove	52.86	66.75	62.15	72.77	66.87	68.03	65.65	65.01
Universal Sentence Encoder	64.49	67.80	64.61	76.83	73.18	74.92	**76.69**	71.22
SBERT-NLI-base	70.97	76.53	73.19	79.09	74.30	77.03	72.91	74.89
SBERT-NLI-large	72.27	**78.46**	**74.90**	80.99	76.25	**79.23**	73.75	76.55
SRoBERTa-NLI-base	71.54	72.49	70.80	78.74	73.69	77.77	74.46	74.21
SRoBERTa-NLI-large	**74.53**	77.00	73.18	**81.85**	**76.82**	79.10	74.29	**76.68**

Table 1: Spearman rank correlation ρ between the cosine similarity of sentence representations and the gold labels for various Textual Similarity (STS) tasks. Performance is reported by convention as $\rho \times 100$. STS12-STS16: SemEval 2012-2016, STSb: STSbenchmark, SICK-R: SICK relatedness dataset.

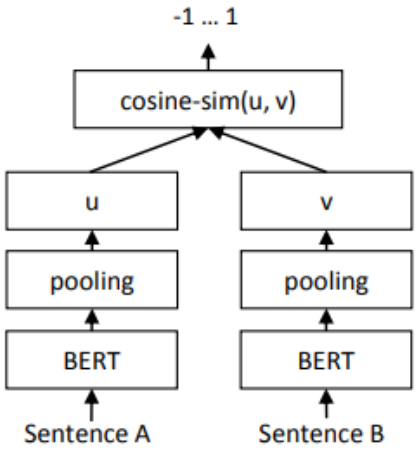
Figure 13: Sentence embedding models comparison

To obtain accurate sentence vector representations, SBERT uses siamese and triplet networks. To train the SBERT network, two sentences are passed through BERT networks to obtain sentences embedding. Multiple pooling operations can be used such as using the output of the CLS token, the mean of the output vectors of BERT or computing a max-over-time of the output vectors. Therefore, the two sentences are firstly embedded into two vectors u and v. Then, these embeddings are compared through an objective function. The three most used ones are the following :

- Classification objective function : The embeddings are passed through a Soft-Max classifier



- Regression objective function : Similarity between the two sentences is calculated via a cosine similarity score



- Triplet objective function : given three sentences, triplet function tunes the network so that the distance between an anchor sentence and a positive sentence is smaller than the distance between an anchor sentence and a positive sentence.

$$max(||s_a - s_p|| - ||s_a - s_n|| + \epsilon, 0)$$

Thus, in order to be trained, SBERT needs a huge database. For instance, the SNLI (Stanford Natural Language Inference) consists of 570,000 pairs annotated with the labels contradiction, eintailment, and neutral is a good way of training the SBERT model as it gives the most interesting results (see the table above).

Once the SBERT model is trained, we have accurate representation vectors of our sentences, and we can calculate the similarity between them using a cosine similarity function. Since our task is to assign labels to new sentences, we will use a k-NN classifier, using a cosine similarity function as the distance function. Indeed, a new sentence will be vectorized through SBERT, and a label will be assigned to it by comparing its cosine similarity to the sentences of the training dataset.

4.3.2 Universal sentence Encoder

The objective of the Universal Sentence Encoder (USE), developed by Google Research and explained in the "Universal Sentence Encoder" article [**13**], is to encode sentences into high-dimensional (512-dimensional) vectors that effectively capture their semantic meaning.

USE offers two primary models: one based on the transformer architecture and the other on a deep averaging network (DAN). The transformer model creates

context-aware representations using attention mechanisms, aggregating word vectors into a fixed-length vector. This approach excels in capturing nuanced semantics but requires significant computational resources. In contrast, the DAN model averages input word embeddings and passes them through a feedforward neural network, providing efficient processing but with slightly reduced accuracy compared to the transformer model.

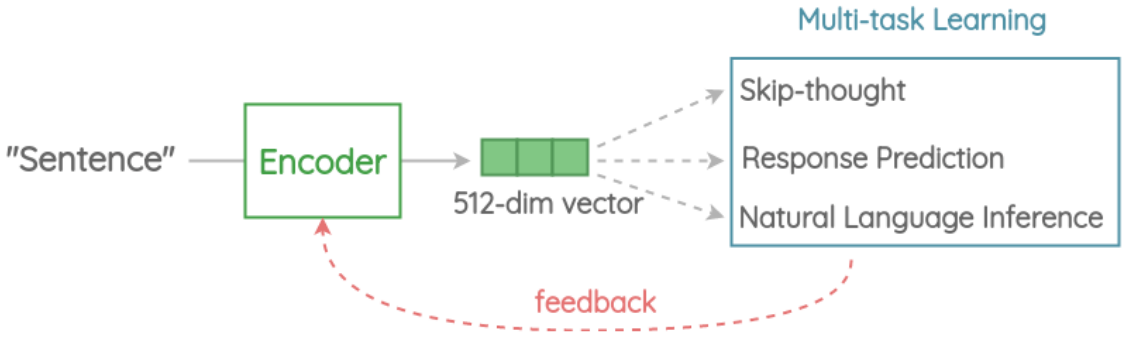


Figure 14: Illustration of USE encoder

Trained on a diverse corpus, including Wikipedia, web news, and discussion forums, USE combines unsupervised and supervised learning for generalizable sentence embeddings. Its effectiveness is demonstrated through strong performance across various tests, showing high correlation with human judgments and consistently outperforming traditional word-level embeddings in tasks like sentiment analysis and sentence classification. This is why we chose to try this method for our goals.

To implement it in Python, we used the "https://tfhub.dev/google/universal-sentence-encoder/4" model on Kaggle, which includes the latest updates and uses the DAN architecture, allowing us to run it on our computers.

For some of the classical algorithms such as logistic regression or SVM, we added a standard scaler to the matrix containing the sentences to prevent some components from gaining undue significance.

After this is done, we can implement our algorithms in the usual manner.

5 Machine Learning Techniques

Now that we have preprocessed our data and embedded the sentences, we delve into the machine learning techniques that we initially explored for our classification task. These algorithms are central to machine learning and have proven to be effective across a wide range of applications. By starting with these established methods, we aimed to build a baseline for our model's performance and to understand the strengths and limitations of different approaches. This section aims to explain these algorithms and how they are used to be performed on our data. The results we obtain from it are discussed in the last part of the report.

5.1 Logistic Regression

Logistic Regression is a classification method for analyzing datasets where one or more independent variables determine an outcome which is often binary. Logistic regression models the dependent variable using a logistic function, also known as the sigmoid function, which outputs probabilities and is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

where z is the linear combination of the input features, which is for us the embedded vectors.

Obtaining the best-fitting model involves estimating the coefficients for each variable to maximize the likelihood of observing the given data. The optimization of weights is typically achieved using techniques like gradient descent.
Logistic regression can be extended to handle multiple classes using two main approaches: Multinomial Logistic Regression and One-vs-Rest Logistic Regression.

Multinomial Logistic Regression: Multinomial Logistic Regression is a direct extension of binary logistic regression that handles multiple classes. Instead of a single logistic function, the model uses a softmax function to calculate the probabilities of each class which is defined as:

$$P(y = j|x) = \frac{e^{\theta_j \cdot x}}{\sum_{k=1}^{K} e^{\theta_k \cdot x}}$$

where θ_j are the parameters for class j, x is the input feature vector, and K is the number of classes (5 for us). The model is trained to maximize the likelihood of the observed class labels by adjusting the parameters θ. This approach allows the model to directly predict the probabilities for all classes and label it as the most likely.

One-vs-Rest Logistic Regression: One-vs-Rest Logistic Regression, also known as One-vs-All, is another method for extending binary logistic regression to multiclass problems. In OvR, a binary classifier is trained for each class, where the outcomes are being from this class or not. This results in multiple classifiers, each specialized in recognizing a single class. After that, for prediction, each classifier produces a confidence score for its respective class, and the class with the highest score is chosen as the final prediction.

In our implementation, we used the LogisticRegression function from the `sklearn` library. Two primary parameters we tuned are the solver and the C parameter. The solver parameter specifies the algorithm to use for optimization, such as 'liblinear', 'lbfgs', 'newton-cg', or 'sag'. The choice of solver can impact the efficiency and convergence of the model. The C parameter is the inverse of regularization strength and

controls the trade-off between achieving a low training error and a low testing error. A smaller value of C results in stronger regularization, which prevents overfitting.

5.2 Complement Naive Bayes

Complement Naive Bayes is a variant of the standard Naive Bayes classifier designed to handle imbalanced datasets more effectively. Standard Naive Bayes assumes that the presence of a particular feature in a class is independent of the presence of any other feature, an assumption that often does not hold true in practice. CNB addresses this by focusing on the complements of the classes, which helps to mitigate the bias introduced by imbalanced class distributions.

In our application, CNB is used for classification by treating each sentence as a collection of features (words or n-grams). The model estimates the probability of a sentence belonging to each class by considering the frequency of words in the complement of each class. This approach tends to be more robust when dealing with skewed data distributions, which is common in text classification tasks like ours. By using the complementary information, CNB provides a more balanced classification that can improve accuracy, especially for underrepresented classes.

5.3 Support Vector Machine

Support Vector Machine (SVM) is a powerful classification algorithm that constructs a hyperplane in a high-dimensional space to separate different classes. The goal of SVM is to find the hyperplane that maximizes the margin, i.e., the distance between the hyperplane and the nearest data points from each class, known as support vectors.

For sentence classification, SVM is particularly effective due to its ability to handle high-dimensional data and its robustness to overfitting. The kernel trick allows SVM to perform non-linear classification by mapping the input features into higher-dimensional spaces where a linear separator can be found. This flexibility makes SVM suitable for capturing complex relationships between the features and the target classes.

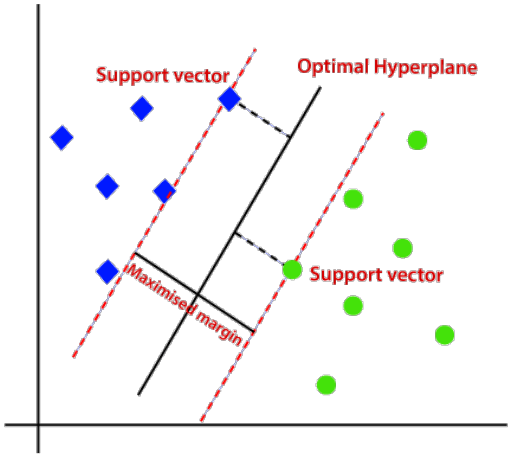


Figure 15: Illustration of Support Vector Machine

In our implementation, we used the SVM function from the `sklearn` library. The choice of kernel and hyperparameters is crucial for the performance of the model, and we employed cross-validation to select the best configuration. The primary parameter we adjusted is the kernel function (linear, polynomial, radial basis function) to explore different ways of mapping our sentence features. Additionally, we tuned the constant C, which controls the balance between a wide margin (low C) and correctly classifying as many data points as possible (high C). SVM's strength in handling high-dimensional spaces and its ability to generalize well on unseen data make it one of the best candidates for our task.

5.4 K-Nearest Neighbors

The k-Nearest Neighbors (k-NN) algorithm is simple, it classifies a data point based on how its neighbors are classified. K-NN assumes that similar data points are close to each other in the high dimensional space. The algorithm works as follows: for a given data point, it finds the k nearest neighbors in the training dataset and assigns the most common class among these neighbors to the data point.

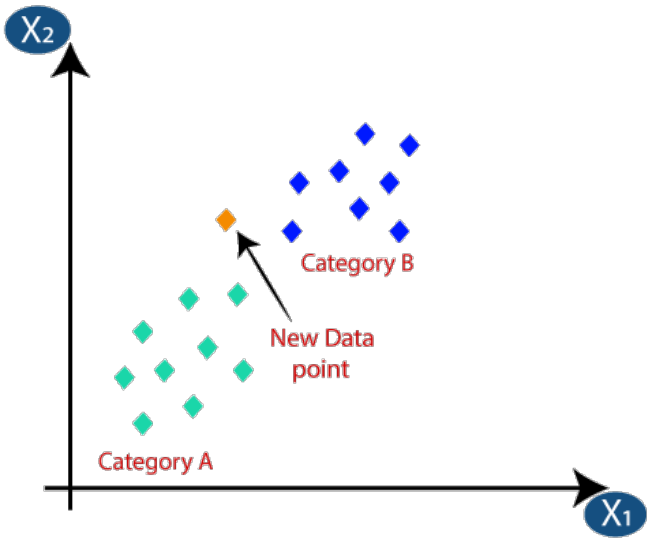


Figure 16: Illustration of k-nn

In our implementation, we used the `KNeighborsClassifier` from the `sklearn` library. This function allows us to specify parameters such as the number of neighbors k or the distance metric to use.

The number of neighbors, k, represents the number of nearest neighbors to consider when making a classification decision. A small k can lead to a model that is sensitive to noise in the training data, as it bases its classification on the closest single neighbor. Conversely, a larger k smooths out the decision boundary and makes the model more robust to noise but can also cause it to lose the finer details of the data distribution.

The distance metric determines how the similarity between data points is measured. Common distance metrics include Euclidean distance, Manhattan distance, and Minkowski distance. Euclidean distance is the most commonly used metric and calculates the straight-line distance between two points in a multi-dimensional space. The choice of distance metric affects the performance of the classifier, especially in our high-dimensional space.

6 Deep Learning Techniques

In recent years, deep learning has made significant advancements. Architectures like convolutional neural network (CNN), LSTM, and GRU have shown competitive results in competitions such as computer vision, signal processing, and natural language.

6.1 Convolutional neural networks

The CNN is a network mainly composed by convolutional layers. The purpose of the convolutional layers is to extract features that preserve relevant information from the inputs [7]. To obtain the features, a convolutional layer receives a matrix as input, to which a matrix with a set of weights, known as a filter, is applied using a sliding window approach and, at each of the sliding window steps, a convolution is calculated, resulting in a feature. The size of the filter is a relevant hyperparameter.

Although CNNs have been widely used in computer vision, they can also be used in sentence classification [11]. The use of convolutional layers enables the extraction of features from a window of words, which is useful because word embeddings alone are not able to detect specific nuances, such as double negation, which is important for sentiment classification. The width of the filter, represented by h, determines the length of the n-grams. The number of filters is also a hyperparameter, making it possible to use multiple filters with varying lengths [11]. The filters are initialized with random weights, and, during network training, weights are learned for the specific task of the network, through backpropagation. Since each filter produces its own feature map, there is a need to reduce the dimensionality caused by using multiple filters. A sentence can be encoded as a single vector by applying a max pooling layer after the convolutional layer, which takes the maximum value for each position, from all the feature maps, keeping only the most important features.

The figure 17 illustrates a CNN architecture for sentence classification.Here we depict three filter region sizes: 2, 3 and 4, each of which has 2 filters. Every filter performs convolution on the sentence matrix and generates (variable-length) feature maps. Then 1-max pooling is performed over each map, i.e., the largest number from each feature map is recorded. Thus a univariate feature vector is generated from all six maps, and these 6 features are concatenated to form a feature vector for the penultimate layer. The final softmax layer then receives this feature vector as input and uses it to classify the sentence; here we assume binary classification and hence depict two possible output states.

6.2 Reccurent neural networks

When it comes to sequential or time series data, traditional feedforward networks cannot be used for learning and prediction. A mechanism is required to retain past or historical information to forecast future values. Recurrent neural networks, or RNNs for short, are a variant of the conventional feedforward artificial neural networks that can deal with sequential data and can be trained to hold knowledge about the past.

Ordinary feedforward neural networks are only meant for data points that are independent of each other. However, if we have data in a sequence such that one data point depends upon the previous data point, we need to modify the neural network to incorporate the dependencies between these data points. RNNs have

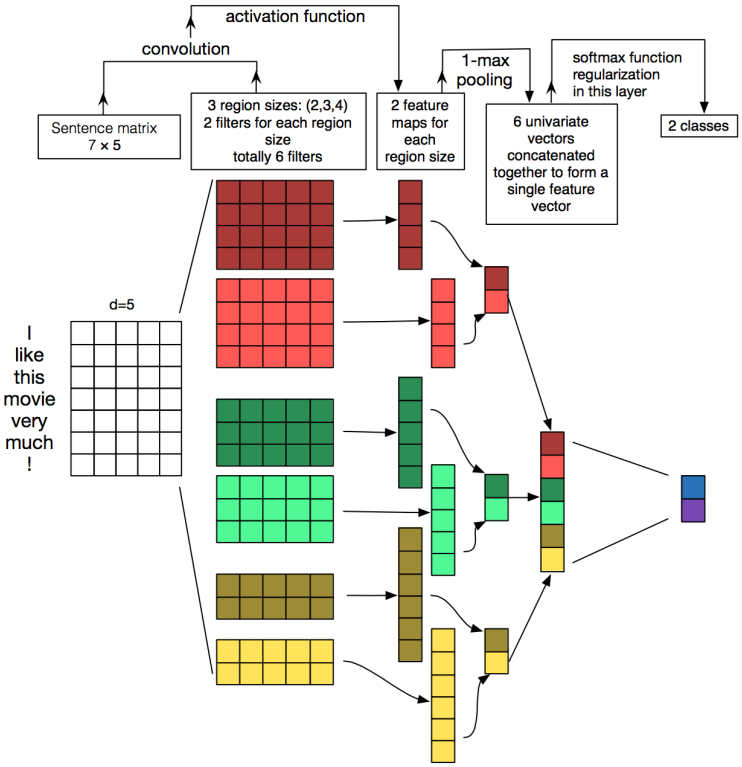


Figure 17: Illustration of a Convolutional Neural Network (CNN) architecture for sentence classification [**7**]

the concept of "memory" that helps them store the states or information of previous inputs to generate the next output of the sequence.

Another distinguishing characteristic of recurrent networks is that they share parameters across each layer of the network. While feedforward networks have different weights across each node, recurrent neural networks share the same weight parameter within each layer of the network. That said, these weights are still adjusted in the through the processes of backpropagation and gradient descent to facilitate reinforcement learning.

Recurrent neural networks leverage backpropagation through time (BPTT) algorithm to determine the gradients, which is slightly different from traditional backpropagation as it is specific to sequence data. The principles of BPTT are the same as traditional backpropagation, where the model trains itself by calculating errors from its output layer to its input layer. These calculations allow us to adjust and fit the parameters of the model appropriately. BPTT differs from the traditional approach in that BPTT sums errors at each time step whereas feedforward networks do not need to sum errors as they do not share parameters across each layer.

Through this process, RNNs tend to run into two problems, known as exploding gradients and vanishing gradients. These issues are defined by the size of the gradient, which is the slope of the loss function along the error curve. When the gradient is too small, it continues to become smaller, updating the weight parameters until they become insignificant—i.e. 0. When that occurs, the algorithm is no longer learning. Exploding gradients occur when the gradient is too large, creating an unstable model. In this case, the model weights will grow too large, and they will eventually be represented as NaN. One solution to these issues is to reduce the number of hidden layers within the neural network, eliminating some of the complexity in the RNN model.

6.2.1 BRNN

Bidirectional recurrent neural networks (BRNN) are a variant network architecture of RNNs. While unidirectional RNNs can only drawn from previous inputs to make predictions about the current state, bidirectional RNNs pull in future data to improve the accuracy of it. If we take the example of the sentence "feeling under the weather" , the model can better predict that the second word in that phrase is "under" if it knew that the last word in the sequence is "weather."

6.2.2 LSTM

This is a popular RNN architecture, which was introduced by Sepp Hochreiter and Juergen Schmidhuber as a solution to vanishing gradient problem. In their paper [3], they work to address the problem of long-term dependencies. That is, if the previous state that is influencing the current prediction is not in the recent past, the RNN model may not be able to accurately predict the current state. As an

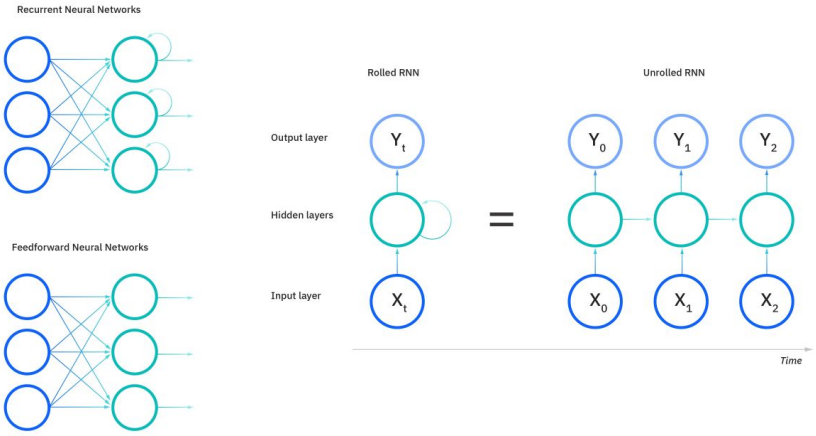


Figure 18: Illustration of the difference between recurrent and feedforward neural networks [4]

example, let's say we wanted to predict the italicized words in following, "Alice is allergic to nuts. She can't eat peanut butter." The context of a nut allergy can help us anticipate that the food that cannot be eaten contains nuts. However, if that context was a few sentences prior, then it would make it difficult, or even impossible, for the RNN to connect the information. To remedy this, LSTMs have "cells" in the hidden layers of the neural network, which have three gates–an input gate, an output gate, and a forget gate. These gates control the flow of information which is needed to predict the output in the network. For example, if gender pronouns, such as "she", was repeated multiple times in prior sentences, you may exclude that from the cell state.

As illustrated in figure 19, in each computational step, the current input `x(t)` is used, the previous state of short-term memory `c(t-1)`, and the previous state of hidden state `h(t-1)`.

These three values pass through the following gates on their way to a new Cell State and Hidden State:

- In the so-called **Forget Gate**, it is decided which current and previous information is kept and which is thrown out. This includes the hidden status from the previous pass and the current input. These values are passed into a sigmoid function, which can only output values between 0 and 1. The value 0 means that previous information can be forgotten because there is possibly new, more important information. The number one means accordingly that the previous information is preserved. The results from this are multiplied by the current Cell State so that knowledge that is no longer needed is forgotten since it is multiplied by 0 and thus dropped out.

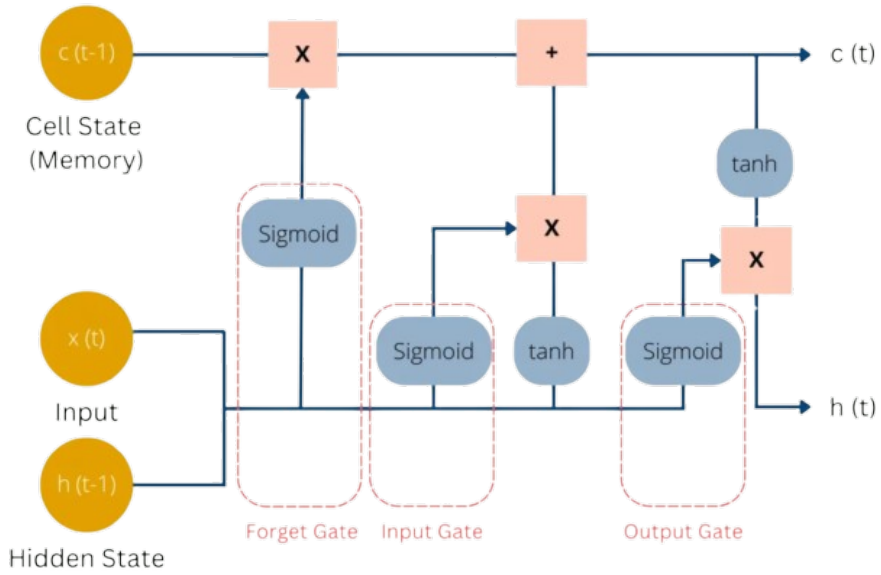


Figure 19: LTSM cell architecture

- In the **Input Gate**, it is decided how valuable the current input is to solve the task. For this, the current input is multiplied by the hidden state and the weight matrix of the last run. All information that appears important in the Input Gate is then added to the Cell State and forms the new Cell State c(t). This new Cell State is now the current state of the long-term memory and will be used in the next run.
- In the **Output Gate**, the output of the LSTM model is then calculated in the Hidden State. Depending on the application, it can be, for example, a word that complements the meaning of the sentence. To do this, the sigmoid function decides what information can come through the output gate and then the cell state is multiplied after it is activated with the tanh function.

6.2.3 GRU

This RNN variant, Gated Reccurent Unit, is similar the LSTMs as it also works to address the short-term memory problem of RNN models. Instead of using a "cell state" regulate information, it uses hidden states, and instead of three gates, it has two—a reset gate and an update gate. Similar to the gates within LSTMs, the reset and update gates control how much and which information to retain.

7 Building and evaluating a classification model

In this section, we focus on testing the performances of our classification models. We highlight the importance of careful evaluation using the right metrics and validation

methods to ensure the model's performance is reliable and can be generalized to a wider range of abstracts. This approach not only confirms how effective the model is but it also helps identify areas where it can be improved.

7.1 Cross validation

Cross-validation is a common method for assessing how well our classification model performs. It involves dividing the dataset into several parts to train and test the model multiple times, ensuring that every data point gets a chance to be in the test set. This technique helps prevent overfitting, where the model performs well on training data but poorly on new, unseen data, and provides a more accurate measure of the model's performance.

We used k-fold cross-validation, where the dataset is split into k smaller sets or folds. The model is trained on k-1 folds and tested on the remaining one. This process is repeated k times, with each fold being used as the test set once. The results are then averaged to get a more reliable performance metric. This method ensures a thorough evaluation and reduces the bias that might result from a single train-test split.

Figure 20: Illustration of 5-fold cross validation

By using cross-validation, we make sure our model is tested and validated on different parts of the data, making it more generalizable and better at handling new data. This step is vital for ensuring the model's accuracy and effectiveness in applications that could go beyond our dataset.

7.2 Evaluation metrics

In multiclass tasks, a classifier often outputs a class probability and the highest probability class is assigned as the predicted class label. Using these predicted labels, classification accuracy is often measured by building a confusion matrix, which maps predicted versus desired labels. From this matrix, several metrics can be computed, such as [10]: Precision, Recall, F1-score.

For a class c, these metrics are obtained using:

$$\text{Precision}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FP}_c} \tag{1}$$

$$\text{Recall}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FN}_c} \tag{2}$$

$$\text{F1-score}_c = 2 \times \frac{\text{Precision}_c \times \text{Recall}_c}{\text{Precision}_c + \text{Recall}_c} \tag{3}$$

where TP_c, FP_c, FN_c denote the number of true positives, false positives and false negatives for class c.

To combine all five class confusion matrices results into a single measure, we adopt two aggregation methods: macro-averaging and weight-averaging. The macro-averaging computes first the metric (e.g., Precision using Eq. 1) for each class and then averages the overall result. The weight-averaging is computed in a similar way except that each class metric is weighted proportionally to its prevalence in the data.

7.3 Neural newtork architecture

For our neural network models, we used a customized preprocessing function as detailed in 3.4. Once the sentences are tokenized, we create a dictionary that associates each word with an index, beginning at 1 for the most frequent word in the training dataset. (TensorFlow Tokenizer is used in this step)

7.3.1 Managing variable-length input sequences

Most (if not all) neural networks require input sequence data with the same length, which is why padding is necessary: to truncate or pad sequences (usually with 0s) to the same length.

Figure 21 demonstrates how padding and truncating function. In our case, the `max_length` is set to match the length of the longest sentence in the training set, and zero padding is applied. To address variable-length input sequences and disregard 0s from padding, a masking layer is utilized.

7.3.2 Handling out of vocabulary words

Handling out-of-vocabulary (OOV) words is a critical aspect of natural language processing, especially in neural network setups. OOV words denote terms not encountered during training, which can detrimentally impact model performance.

Embedding layer builds the bridge between the token sequences (as input) and the word embedding representation (as output) through an embedding matrix (as weights) as illustrated in figure 22.

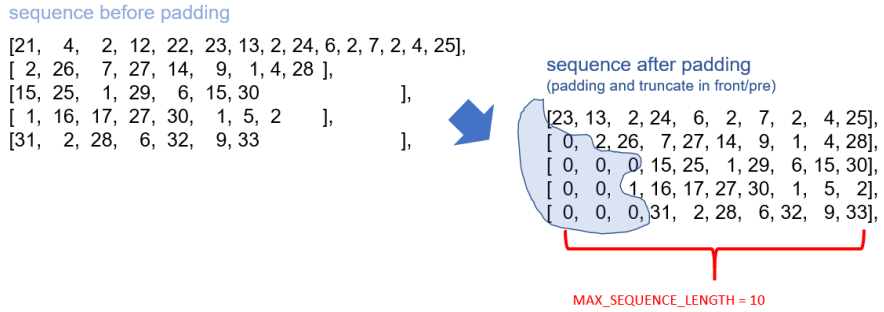


Figure 21: Illustration of padding and truncating for managing variable-length input sequences

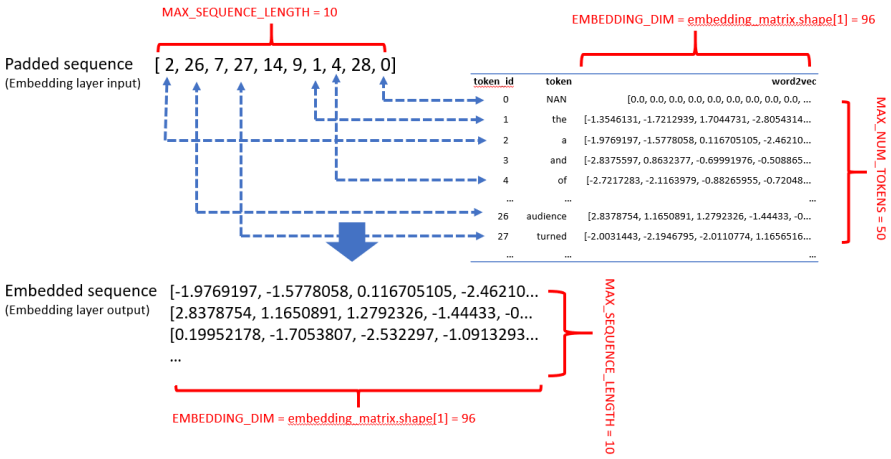


Figure 22: Illustration of how embedding layers work

- **Input of an Embedding layer:** the padded sequence is fed in as input to the Embedding layer; each position of the padded
- **Weights of an Embedding layer:** by looking up into the embedding matrix, the Embedding layer can find the word2vec representation of words(tokens) associated With the token-id. Note that padded sequence use zeros to indicate empty tokens resulting zero-embedding-vectors. That's Why we have saved the first row in the embedding matrix for the emotv tokens.
- **Output of an Embedding layer:** after going through the input padded sequences, the Embedding layer "replaces" the token-id With representative vectors(word2vec) and output embedded sequences. sequence is designated to a token-id.

In our scenario, we utilized a common strategy to handle OOV words by leveraging pre-trained word embeddings 4.2. Within our neural network architecture, we incorporate two embedding layers. The first caters to known words with pre-trained

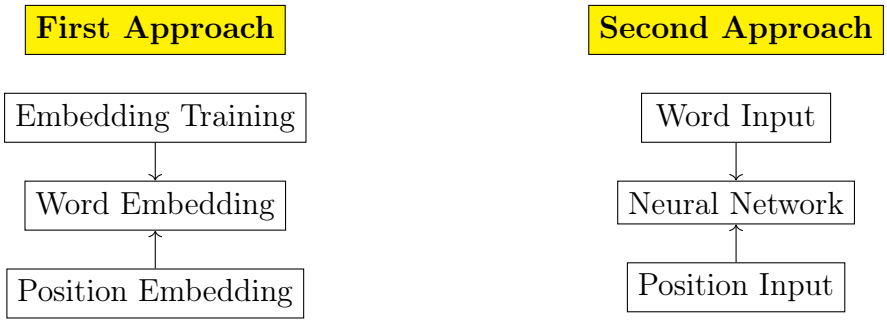


Figure 23: Diagram illustrating the two approaches for incorporating sentence position information

embeddings, whereas the second handles OOV words initialized with the average of all word embeddings in the corpus. This layer undergoes training in the training phase to enhance test accuracy.

The outputs of these embedding layers are summed element-wise to form a combined embedding representation for each word.

7.3.3 Integrating sentence position information

Two approaches are used to incorporate sentence position information as illustrated in figure 23. The *first approach* adds an extra dimension to word embeddings to store the sentence position while training the embedding vectors. This helps distinguish between the same words appearing in different sentence positions. The *second approach* involves treating sentence position information as an additional input to the neural network, separate from the words forming the sentence.

7.3.4 proposed architecture

The proposed architecture integrates word embeddings, position embeddings, convolutional layers, and bidirectional LSTM. It is illustrated in Figure 24.

We assume that each abstract has i sentences $(S_1, \ldots, S_i)$ and each individual sentence has n words $(x_1^1, \ldots, x_n^i)$, where x_n^i is the n-th word from the i-th sentence. Additionally, we incorporate positional information p_i for each sentence.

Initially, the words in the sequence are mapped to their corresponding word embeddings. from the GloVe model ('glove-wiki-gigaword-300')[10] as described in 7.3.2.

To integrate positional information, we repeat the position input to match the sequence length and concatenate it with the combined word embeddings, resulting in a matrix $E \in \mathbb{R}^{n \times (d+1)}$, where d is the dimensionality of the word embeddings.

Next, a masking layer is applied to handle any padding in the input sequences, ensuring that the model does not consider padded values. This is followed by a convolutional layer with a filter size of 3, which slides over the embeddings to capture

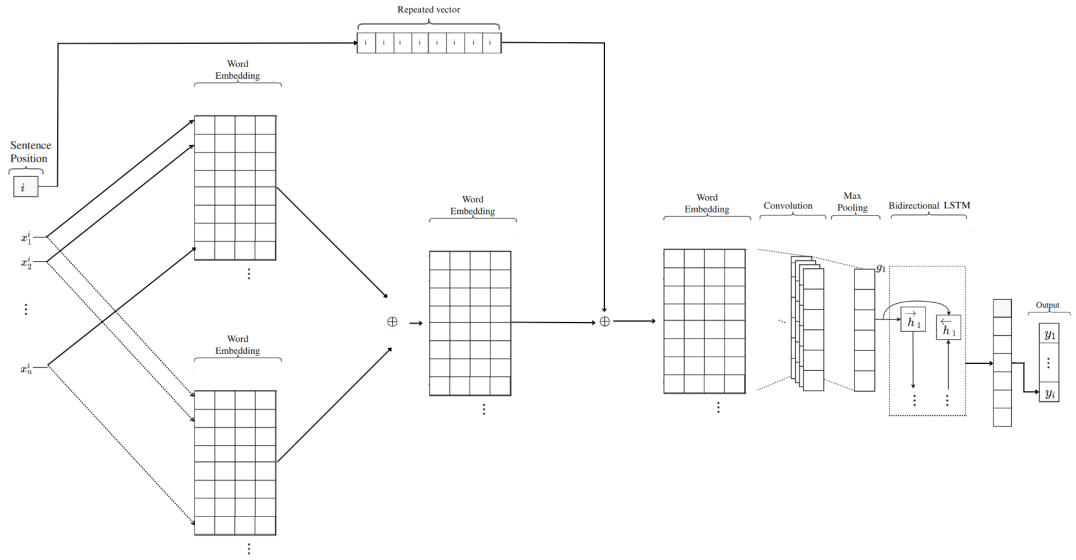


Figure 24: Schematic of the proposed Word-BiLSTM deep learning architecture

local features within the sequence. The output of the convolutional layer is then passed through a max-pooling layer to reduce its dimensionality and highlight the most significant features.

Subsequently, a bidirectional LSTM layer processes the features extracted by the convolutional layer, capturing dependencies in both forward and backward directions. A dropout layer is included to prevent overfitting by randomly setting a fraction of the input units to zero during training.

Finally, the processed features are passed through a dense layer with 32 units and ReLU activation, and an output layer with softmax activation produces the final classification.

The model is trained using categorical cross-entropy loss, the Adam optimizer, and is evaluated on its accuracy. Early stopping is employed to monitor the validation loss, halting training if there is no improvement for 3 consecutive epochs and restoring the best weights.

8 Numerical results discussion

In this section, we will analyze the numerical findings from our study, starting with machine learning methods and advancing to deep learning techniques.

8.1 Assessment of machine learning models

As described previously, for the development of our machine learning algorithms discussed in 5, we began with data preprocessing, as outlined in section 3.4. Following that, labels were encoded for machine learning model comprehension, and sentences were vectorized using bag-of-word approaches. A 5-fold cross-validation procedure was then utilized to enhance performance, as detailed in 7.1.

The final results are displayed in Table 1.

Model	**ML model**	**Vectorizer**	**Macro F1-score (%)**	**Micro F1-score (%)**
ml_model_1	CNB	CountVect	45.47	56.06
ml_model_2	CNB	Tf-IDF Vect	44.55	55.57
ml_model_3	Multinomial LR	CountVect	46.68	55.90
ml_model_4	OVR LR	Tf-IDF Vect	55.00	57.70
ml_model_5	SVM	CountVect	45.72	58.00
ml_model_6	Multinomial LR	USE	52.00	60.00
ml_model_7	SVM	USE	52.00	61.00

Table 1: Comparison of machine learning models

In order to assess the performance of the machine learning models, two evaluation metrics were used: the micro F1-score and the macro F1-score, in addition to confusion metrics. The main difference between these two evaluation metrics is that the micro F1-score considers all instances equally, while the macro F1-score calculates the average F1 score for each class.

The results show varying performance across different models and vectorization techniques. The micro F1-score, which is more reflective of overall performance across all instances, ranges from 55.57% to 61.00%. The macro F1-score, which gives equal weight to each class, ranges from 44.55% to 55.00%.

The macro F1-scores are generally lower than the micro F1-scores, indicating that the models may be performing unevenly across different classes. This discrepancy is likely due to the "conclusion" class being weakly predicted. This is a common issue in imbalanced datasets where some classes have fewer instances than others, leading to poorer performance in those underrepresented classes.

Among the models, the OVR LR with Tf-IDF vectorization shows the highest macro F1-score at 55.00%, indicating it performs relatively better across all classes compared to the others. However, the highest micro F1-score of 61.00% is achieved by the SVM with Universal Sentence Encoder, suggesting it has the best overall performance when considering all instances equally.

The confusion matrix for ‘`ml_model_5`‘ (SVM with CountVect) is shown in Table 2. This matrix helps visualize the performance of the model by displaying the number of correct and incorrect predictions for each class.

The confusion matrix reveals that while the model performs well in predicting the "BACKGROUND" and "METHODS" classes, it struggles with the "CONCLUSIONS" class, which corroborates the lower macro F1-score. The poor values of macro F1-score are due to the fact that the conclusion class is weakly predicted.

	BACKGROUND	CONCLUSIONS	METHODS	OBJECTIVE	RESULTS
BACKGROUND	177	0	19	9	5
CONCLUSIONS	20	0	5	3	12
METHODS	49	0	84	13	11
OBJECTIVE	33	0	19	52	3
RESULTS	28	0	29	2	46

Table 2: Confusion matrix for '`ml_model_5`' (SVM with CountVect)

In conclusion, while some models show promising results, there is room for improvement, particularly in enhancing the prediction performance for underrepresented classes. Further tuning of the models and possibly exploring advanced techniques such as ensemble methods or deep learning could help in achieving better performance.

8.2 Assessment of deep learning models

Moving on to the assessment of deep learning models, we experimented with various configurations to understand their performance in classifying text data. Table 3 presents a comparison of different deep learning models based on their precision, recall, and F1-score.

For our deep learning models, we employed techniques such as word embedding and positional information to capture semantic meaning and context within the text.

	Position Info (None, 1st, 2nd)	**1st Embedding (Trainable, Non-trainable)**	**2nd Embedding (None, Trainable, Non-trainable)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**
Model_1	None	non-trainable	None	64.55	68.50	66.12
Model_2	None	trainable	None	64.64	68.34	66.13
Model_3	None	non-trainable	trainable	64.33	67.53	65.45
Model_4	1st	non-trainable	None	73.86	70.6	68.60
Model_5	1st	trainable	None	71.60	71.57	71.27
Model_6	1st	non-trainable	trainable	71.61	72.05	71.15
Model_7	2st	non-trainable	None	69.87	70.27	69.51
Model_8	2st	trainable	None	65.65	69.47	67.05
Model_9	2st	non-trainable	trainable	69.41	70.11	68.69

Table 3: Comparison of deep learning models

Model_1 through Model_3 experimented with different combinations of embeddings. Model_1 utilized non-trainable embeddings exclusively, resulting in a precision of 64.55% and an F1-score of 66.12%. Model_2, on the other hand, employed trainable embeddings, yielding similar performance with a precision of 64.64% and an F1-score of 66.13%. Model_3 combined both non-trainable and trainable embeddings, achieving slightly lower precision and F1-score compared to Model_1 and Model_2.

Incorporating positional information significantly impacted the performance of

the models. Models such as Model_4 through Model_9 utilized positional information, using either the 1st or 2nd approach as described in 23.

Model_4, employing positional information and non-trainable embeddings, demonstrated improved precision at 73.86% and an F1-score of 68.60%. This indicates that incorporating positional information aids in capturing context, resulting in better classification performance.

Similarly, Model_5 and Model_6, with trainable embeddings and positional information, also exhibited enhanced precision and F1-score compared to models without positional information.

However, there's a slight drop in performance observed in models with positional information from the 2nd approach (Model_7 to Model_9) compared to those focusing on the 1st approach. This suggests that the importance of positional information might vary based on the way they are incorporated in the model.

Overall, these results highlight the importance of experimenting with different configurations of embeddings and positional information in deep learning models to achieve optimal performance in text classification tasks. Further exploration and tuning of these models could potentially lead to even better results.

8.2.1 Ensemble learning for robustness

To mitigate the effects of randomness and enhance the robustness of our results, we applied ensemble learning techniques independently to Model_4 and Model_6.

This approach involves creating multiple instances of each model with different random initializations and then combining their predictions to form a final, more reliable output. By doing so, we aim to smooth out variability and improve overall performance.

For each model, we created an ensemble of 5 instances. The final prediction for each instance was determined by averaging the predictions of the individual models. This method helps to reduce overfitting and captures a broader range of data patterns.

The performance of the ensemble models is summarized in Table 4.

Model	**Precision (%)**	**Recall (%)**	**F1-score (%)**
Model_4 (Single)	73.86	70.60	68.60
Model_4 (Ensemble)	72.47	72.37	70.47
Model_6 (Single)	71.61	72.05	71.15
Model_6 (Ensemble)	72.37	72.37	70.85

Table 4: Performance of Individual and Ensemble Models

These results demonstrate that the ensemble approach generally helps to maintain or slightly improve performance compared to individual models.

The slight decrease in precision for Model_4's ensemble could be due to the smoothing out of some of the more extreme predictions made by the individual models. However, the overall improvement in recall and F1-score suggests that the ensemble method is indeed capturing a more robust set of patterns present in the data.

For Model_6, we observe a similar trend with a slight increase in precision and F1-score for the ensemble version. This consistency across both models further validates the effectiveness of ensemble learning in enhancing model generalization and reducing the impact of randomness in the training process.

8.2.2 Two-Phase Training

To enhance the performance and stability of the model, a two-phase training strategy was implemented. This approach involves training the model in two distinct phases, each serving a specific purpose.

In the first phase, both embedding layers were trained simultaneously for a fixed number of epochs. This phase aimed to allow the model to learn meaningful representations from the input data while considering both in-vocabulary and out-of-vocabulary words. During this phase, the entire model architecture, including the embedding layers, was trainable.

Following the initial phase, the weights of the second embedding layer were retrieved, and the layer was subsequently frozen. By freezing the second embedding layer, its weights became static and were no longer updated during training. This step effectively preserved the learned representations of out-of-vocabulary words.

In the second phase, the model was recompiled after freezing the second embedding layer, and the training process continued. With the frozen embedding layer, the model underwent further training to refine its parameters while keeping the representations of out-of-vocabulary words unchanged. This phase allowed the model to focus on fine-tuning its other layers, such as the convolutional and recurrent layers, without affecting the learned embeddings.

This two-phase training strategy aimed to strike a balance between learning informative representations and maintaining stability during training. By separating the training process into distinct phases and selectively freezing layers, the model could leverage both in-vocabulary and out-of-vocabulary information effectively, ultimately improving its overall performance and generalization capabilities.

The results obtained from the two-phase training strategy are as follows for 'model_6' :

- Precision: 0.6726
- Recall: 0.6543
- F1 Score: 0.6597

Contrary to expectations, the results obtained from the two-phase training strategy for 'Model_6' exhibited a drop in performance compared to the single-phase training. The precision decreased from 71.61% to 67.26%, the recall decreased from 72.05% to 65.43%, and the F1 Score decreased from 71.15% to 65.97%. This unexpected outcome suggests that the freezing of the second embedding layer might have limited the model's ability to adapt to the data during the second phase of training, leading to a degradation in performance.

These results indicate that while the two-phase training strategy aims to enhance performance and stability, its effectiveness may vary depending on the specific characteristics of the dataset and model architecture. Further investigation and refinement of this approach are warranted to better understand its implications and potential benefits.

9 Conclusion

In this project, we explored various machine learning and deep learning techniques for the classification of abstract sentences in computer science papers. Our approach began with a detailed exploratory data analysis, data preprocessing, and the application of several machine learning models including Complement Naive Bayes, Multinomial Logistic Regression, and Support Vector Machines using bag-of-words and TF-IDF vectorization techniques. The performance of these models was evaluated using micro and macro F1-scores.

The results indicated that while the models performed adequately, there was a notable discrepancy between the micro and macro F1-scores, particularly highlighting the challenge of predicting the "CONCLUSION" class. This suggests that our models are less effective at handling underrepresented classes in an imbalanced dataset. One potential factor contributing to this issue is the quality of the data. During our analysis, we discovered instances of mislabeling, especially in the "CONCLUSION" category, where sentences that would be more appropriately labeled as "BACKGROUND" were sometimes classified incorrectly. This mislabeling likely detracted from the overall performance of our models.

In our exploration of deep learning models, we experimented with word embeddings and positional information to enhance model performance. These models showed improved precision, recall, and F1-scores compared to traditional machine learning methods, with the incorporation of positional information proving to be particularly beneficial.

Looking ahead, there remains significant room for improvement. Enhancing the prediction performance for underrepresented classes is crucial. Future work should focus on model tuning and the exploration of advanced techniques such as ensemble learning or more sophisticated deep learning architectures. Moreover, incorporating transformers and fine-tuning pre-trained transformer models like BERT or GPT could offer substantial improvements. Transformers are known for their exceptional performance in various NLP tasks and could potentially address some of the challenges identified in this project. Additionally, addressing class imbalance through data augmentation or more effective sampling strategies could further enhance the robustness and accuracy of our models. Ensuring the accuracy and consistency of data labeling will also be essential in improving model performance.

In conclusion, this project provides a foundational approach to abstract sentence classification, demonstrating the potential and limitations of both machine learning and deep learning techniques. Continued refinement and innovation, including the exploration of transformers, will undoubtedly contribute to more efficient and effective information retrieval from the ever-growing body of scholarly literature.

References

[1] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media, Inc., 2009.

[2] Yoav Goldberg. *Neural Network Methods in Natural Language Processing*. 2017, p. 65.

[3] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural Computation* 9 (1997).

[4] IBM. "What are recurrent neural networks?" In: (). URL: https://www.ibm.com/topics/recurrent-neural-networks.

[5] Arif Jinha. "Article 50 million: An estimate of the number of scholarly articles in existence". In: *Learned Publishing* 23 (July 2010), pp. 258–263. DOI: 10.1087/20100308.

[6] Devendra Kushwah. "What is difference between stemming and lemmatization". In: *Quora* (May 2019).

[7] Farabet C LeCun Y Kavukcuoglu K. "Convolutional networks and applications in vision". In: *Proceedings of 2010 IEEE international symposium on circuits and systems* (2010), pp. 253–256.

[8] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[9] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[10] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162.

[11] Kim Y. "Convolutional neural networks for sentence classification". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (2014), pp. 1746–1751.