INSTITUT FÜR INFORMATIK
Dialog Systems and Machine Learning
Universitätsstr. 1 D–40225 Düsseldorf

Heinrich Heine Universität Düsseldorf

Semantic search on visited websites

Modar Alkanj

Bachelorarbeit

Date of issue: 22. Juni 2023
Date of submission: 22. September 2023
Reviewers: Dr. K. Völkel
Dr. P. Arndt

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 22. September 2023

Modar Alkanj

Abstract

In this paper, we implement a semantic search engine using various algorithms such as Word2Vec, fastText and TF-IDF.
These algorithms assist with converting words into valued vectors, called embeddings. With these embeddings, we can perform different applications, such as document retrieving.
Converting words in vectors relies on the foundation of word embedding. This is a term used to describe words in vectors which capture various meanings of a word. Using this approach, we maintain the semantics of the words, and we can process them in a more comprehensive manner.
The search engine performs search queries on websites. Therefore, we integrate a Chrome browser plugin and a web server to assist with extracting contents from the websites.
For the document retrieval task, we use the cosine similarity metric which provides a value to determine the similarity between two documents.
Before applying these algorithms, the extracted texts need to be cleaned and prepared for the search functionality. Therefore, we consider splitting the text into tokens (words) and removing numerals, stop words and non-alphabet characters. Also, we transform words to the root form with a method called lemmatization, and we remove any redundant tokens.
For the evaluation of the search engine, we use a dataset from BBC news. For the documents in the BBC news, we generate queries using TF-IDF. This method extract keywords from the documents that contribute to the context.
In later chapters of this paper, we consider replacing the Word2Vec model with the fastText model to evaluate the differences in the performance.

Contents

1 Introduction

The goal of this project is to create a semantic search engine for performing search queries on websites.
The search engine does not perform a traditional lexical search, which matches the query terms with the exact keywords in the documents. Semantic search goes beyond lexical search to process the query in a way to find a relevance to a certain document.
Semantic search is about utilizing natural language processing, machine learning and other AI-based techniques and algorithms. These algorithms are used to analyze queries and documents in a more comprehensive way. As we will discuss in chapter 3, Word2Vec is one of the most common algorithms in natural language processing. Word2Vec provides a way to convert words into vectors, called embeddings.
These embeddings allow various applications and computations such as the calculation of similarity metrics between vectors. Word2Vec considers the information held by words or documents to provide more relevant and precise search outcomes. It aims to process the meaning and the content of a query to match it with the most relevant document, rather than returning results with the highest keyword matches. The relevance is measured using mathematical metrics such as the cosine similarity metric, which will be discussed in chapter 3.
In this paper we will explain the functionality and the implementation of Word2Vec, and embed this method in our search engine in chapter 4. Eventually, we will test the performance of the implemented search function in chapter 6, and based on the evaluation results we do changes and try other approaches for the purpose of improvement. Those approaches consider changing the way we process texts and utilizing other models such as fastText, another approach of Word2Vec to achieve more reasonable search results.
The search engine is created to retrieve documents saved by visiting websites. Therefore we create a Chrome browser plugin to assist in saving and processing visited websites.
The primary objective of the search engine is to search for a certain website which has been visited by the user. In case the user loses the link to the website, he can access it again by typing a query into the input field of the extension. The search engine executes the search algorithm on the saved websites and returns the most relevant results.
The user is asked to provide a query to the input area of the extension. With this information provided by the user, the search engine can look up for the right website. The user interface shows a list of outcomes. Every result is a link to a website. The links are sorted based on their relevance to the query. At the top, is the most relevant result the engine can find and at the bottom the most irrelevant.
To demonstrate the process of the search engine, here is an example: The user visits various websites. One of them is a page about travelling and vacations (https://www.tripadvisor.com/TravelersChoice-ThingsToDo). The plugin sends the page with the whole content of the HTML to the server. In the future, the user remembers he once searched for vacation destinations and found promising options, but he doesn't remember where. Searching again in the browser history or again in the search machine could be time-consuming. All he needs to do is to provide a description, for example: "places to spend vacation next summer". With this description, the search engine is able to find the website immediately and return the link to the user.

2 The extension

This extension has been developed for Chrome browsers version 115.0.5790.171 and is compatible only with Chrome.

2.1 Server

We integrate a web server to facilitate the processing of the websites, which makes it simpler handling user queries. This approach involves forwarding each website visited by the user automatically to the webserver.
To prevent complexity of developing a server using the built-in packages in Python, we use Flask, a Python framework designed to create servers with less complexity. The server architecture comprises three primary methods: `index()`, `search()` and `save()`.
In the `index()` method, we allow only queries with HTTP GET method. The index method has one main functionality, which is to retrieve the index.html, the homepage. The homepage is the user interface. The user interface holds the title (TimeTraveller), the input field for the search query and a button for sending the query as shown in figure 1.

```
@app.route('/', methods=['GET'])
def index():
    return render_template('index.html')
```

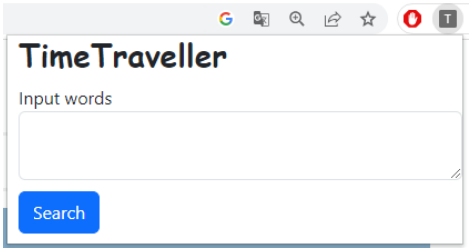

Figure 1: screen shot of the user interface

Also, in the search function we only allow the GET method of HTTP, because we do not aim to change any existing or adding new data. The search function is responsible for unpacking and passing the data delivered by the query from the user to the function `website_search()`, which executes the search algorithm. This function is implemented in another Python file and will be explained in chapter 4. Subsequently, the search results are packaged in JSON format and transferred back to the user.

```
@app.route('/search', methods=['GET'])
def search():
    data = request.args.get("input")
    res = [r[0] for r in website_search(data,2)]
    return jsonify(results = res)
```

The most interesting functionality and the main complexity in terms of semantic search resides in the method `save()`. Unlike the other two functions (`index()` and `search()`), we allow here two HTTP methods, namely POST and OPTIONS. The reason we allow these methods is that we need to do changes on the side of the server. These changes consider adding new documents to the collection. The OPTIONS method is required because when we send a POST query with JavaScript, it will first send an OPTIONS query and then the POST query. If the OPTIONS method is not allowed, the POST query will fail.
The `save()` function is responsible for processing the data received from the user. Initially, it strips away the HTML tags from the received content, retaining only the text relevant for the search engine. In this case we use the library BeautifulSoup, which fits our needs, as it effectively extracts text from HTML content.

```
data = request.get_json()
soup = BeautifulSoup(data["html"], 'html.parser')
data["html"] = soup.get_text().replace("\n", "␣")
```

The extracted text is then stored as a dictionary with following key attributes: "`address`" (includes the website's URL), "`html`" (consists of the extracted HTML text), and "`hash_code`" (generated by hashing the URL using sha256). The "`hash_code`" serves as a tool in finding existing websites to prevent redundancy. Therefore, we used the Python library `hashlib`. From the `hashlib` we call the function `sha256()`, which takes a byte-like object. We can now call the function `hexidigest()`, which returns the generated hash-code as a string. The following code snippet shows the exact way of generating the hash-code using `hashlib`.

```
hashed_address = hashlib.sha256(
            data["address"].encode('utf16')).hexdigest()
```

The `data["address"]` contains the URL to the website, which we receive from the extension. The output of this code for the URL "www.facebook.com" looks like this:

```
'f26ef57573acafe46986cebe2c29412350f2ef0aef6b08efcbe804beb2ba8b77'
```

Finally, these processed data are passed into the `prepare()` method. This method does further modifications on the text, such as tokenizing or removing unnecessary words. This will be discussed in chapter 4.2.

2.2 User interface

The previously mentioned index.html includes a script.js, a JavaScript file which sends HTTP requests and receives outcomes from the server. This file initiates a query directed to "localhost:5000/search?input=" incorporating the user input, and processes the response of the server. The index.html contains a `<div>` tag with `id="search-results"`. The following HTML snippet shows the relevant part for listing the search results.

```
<form class="myform" action="http://localhost:5000/search"
                                                    , method="get">
      <div class="form-group">
        <label for="input">Input words</label>
        <textarea
          type="text"
          class="form-control"
          id="input"
          name="input"
        ></textarea>
      </div>
      <button type="submit" class="btn␣btn-primary">Search</button>
      <div id="search-results">
        <h2>Search Results:</h2>
      </div>
    </form>
```

The JavaScript file replaces the content of the tag with the received results from the response of the server, as the code snippet shows.

```
fetch("http://localhost:5000/search?input=" + input)
      .then(function (response) {
        return response.json();
      })
      .then(function (data) {
        var resultsDiv = document.getElementById("search-results");
        resultsDiv.innerHTML = "";
        var resultsList = document.createElement("ul");
        data.results.forEach(function (result) {
          var listItem = document.createElement("li");
          var link = document.createElement("a");
          link.href = result;
          link.textContent = result;
          link.target = "_blank";
          listItem.appendChild(link);
          resultsList.appendChild(listItem);
        });
          resultsDiv.appendChild(resultsList);
      }
    });
```

These results are displayed on the user interface as links leading to the websites as shown in figure 2.

Additionally, another JavaScript file called content.js is responsible for extracting the HTML content from the website. It then sends the content alongside the URL of the website to the server on “localhost:5000/save”.

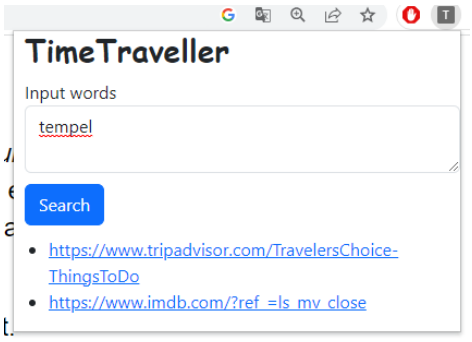

Figure 2: Listing of search results

```
const url1 = window.location.toString();
logCurrentUrl();
const url = "http://localhost:5000/save";
fetch(url, {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify({
    address: url1,
    html: content,
  }),
}).then((response) => response.text());
```

3 Search engine

The search engine is based on the idea of the semantic search. This approach can be realized using deep learning algorithms.
As an initial step, we use an established and common algorithm known as Word2Vec. This serves as a foundation for our search engine, offering room for fine-tuning and enhancements to yield optimal results. We chose Word2Vec for its simplicity and uncomplicated substitutability.

3.1 Word embedding

The objective of word embedding is to present words in the form of valued vectors that allows further processing for textual analysis. This approach effectively addresses the challenge of processing text-based content and is a fundamental component of the natural language processing [Turing, 2023].
Through this methodology, words or entire documents are converted into numeric vectors, which can be later fed to deep learning algorithms. These vectors aim to encapsulate the essence of real words.

They also have a certain mathematical relationship to other vectors. The vectors have a degree of similarity among themselves, shaped by the training set we use. In this way, word embedding provides a basis for manipulation of words and documents.
Multiple algorithms can be applied to realize the previous thoughts, such as TF-IDF which we use in chapter 5.2, GloVe or Word2Vec.

3.2 Word2Vec

Word2Vec is based on the idea of embedding vectors. We can consider a vector composed of two initial values, both set to 0: v=(0,0). Each dimension of this vector corresponds to a distinct feature. These dimensions (features) are abstract mathematical representations that capture various aspects of a word's meaning shaped by the training set and varies between the words. In our scenario, the vector consists of two features, currently both are 0. Word2Vec creates embeddings in a vector space, which has all the words in a corpus of a text presented by vectors. In this vector space, We find vectors of two words, such as those that are similar, to be near each other. The similarity is determined by the quality of the trained data.
Using this vector space, delivered by the embeddings of Word2Vec, we can now do for example similarity checks to see if two words or documents are relevant to each other which allows predictions or search functionalities.
For illustration, we can take an example of two words, "man" and "woman". Logically, "man" and "woman" are similar words. The cosine similarity of the two vectors of "man" and "women" is 0.7664. This is delivered by the Word2Vec google-news training set with 300 dimension, which we will use in chapter 4.1. The similarity implies that the two vectors face similar directions.
The following code shows the cosine similarity of "man" and "woman":

```
1-cosine(training_set["man"], training_set["woman"])
 -> 0.7664012312889099
```

Using this knowledge, we extend our spectrum of dimensions to achieve more precise results.
In this way, we have a first approach of describing words with numeric values. We can apply diverse deep learning algorithms or mathematical computations on the vectors. An especially common usage is comparing two vectors, effectively equating to comparing two words. This capability is important for natural language processing. Therefore, many methods and metrics are used for comparison. Most widely the cosine similarity, Euclidean distance or Jaccard similarity.

3.3 Word2Vec implementation

Word2Vec consists of two essential variants: CBOW (continuous bag of words) and Skip-gram [Ria Kulshrestha, 2019].
Continuous bag of words aims to combine the representations of neighboring words to

anticipate the word in the middle. The objective of skip-gram is to predict the context using the input word. Figure 3 shows the difference between both methods.

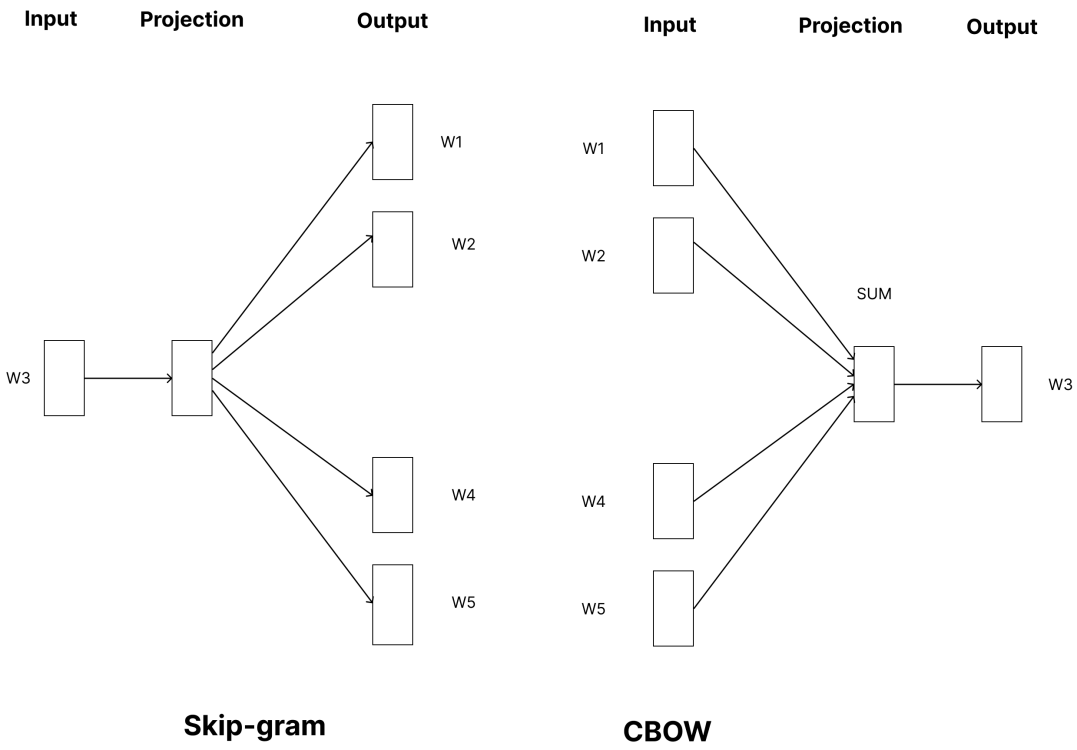

Figure 3: CBOW and skip-gram

We can take the following sentence for illustration: "NASA launched rocket last month." In case of CBOW, if we choose the word "rocket" then we can predict it using the surrounding words, for example "launched" and "last". With skip-gram, we predict the surrounding words using the word in the middle. Meaning, through the word "rocket", we predict the two surrounding words "launched" and "last".
The algorithm tells us the probability for every word of being the nearby word. For instance, in the previous example the word "rocket" is very likely to be nearby the word "launched", in contrary the word "month" is unlikely to be nearby "launched". The accuracy values and probabilities differ depending on the used dataset and the size of the vectors (dimensions).
For training purposes, we use a neural network of three layers. To create the embeddings, we can create a fake problem, which has the "side effect" of generating the embedding vectors. This fake problem is about finding the nearby words and predicting their probabilities. We can do this by feeding the neural network word pairs from the training document. Consequently, the neural network will learn the statistics from the frequency of occurrences for each pair [Chris McCormick, 2016].
In our case, the "NASA launched rocket last month." is the training document. Based on the choice of the window size of two, we generate following pairs: (NASA, launched), (NASA, rocket), (launched, NASA), (launched, rocket), (launched, last), (rocket NASA),

(rocket, launched), (rocket, last), (rocket, month), (last, launched), (last, rocket), (last, month), (month, rocket), and (month, last).
To prepare each word from the vocabulary for input, we can use one-hot encoding. As discussed earlier, every word has an embedding vector, initially random values. These embeddings are in the hidden layer of the neural network. If we have 3 words in the vocabulary and every word is presented by an embedding vector of length 5, then we would have a 5x3 matrix. Through multiplying the one-hot vector of a word with the matrix, we get the embedding-vector corresponding to the word out of the matrix as shown in the equation.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{bmatrix} = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}$$

It becomes now possible to feed the neural network with inputs in form of one-hot vectors. We apply a Softmax function on the output of the network to turn the output values into probabilities, which sum up to 1 [Thomas Wood, 2023]. The output at the first execution will not be correct because we chose random weights for the initial embeddings (the matrix in the hidden layer). After feeding the words to the neural network, we compare the predicted outputs with the actual results and apply a loss function, then back propagate to update the weights. This process is repeated until a certain accuracy is achieved, or convergence occurs.
When we use the word "NASA", we expect the word "launched" to likely be the nearby word (we take one word for simplicity. In reality the output is two vectors since we have two pairs (NASA, launched) and (NASA, rocket)). So, we expect an output like this:

$$\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

But the output will look more like this (these are random values only for illustration) :

$$\begin{pmatrix} 7 \\ 4 \\ -4 \\ 0.4 \\ 1 \end{pmatrix}$$

After applying the Softmax function, we get:

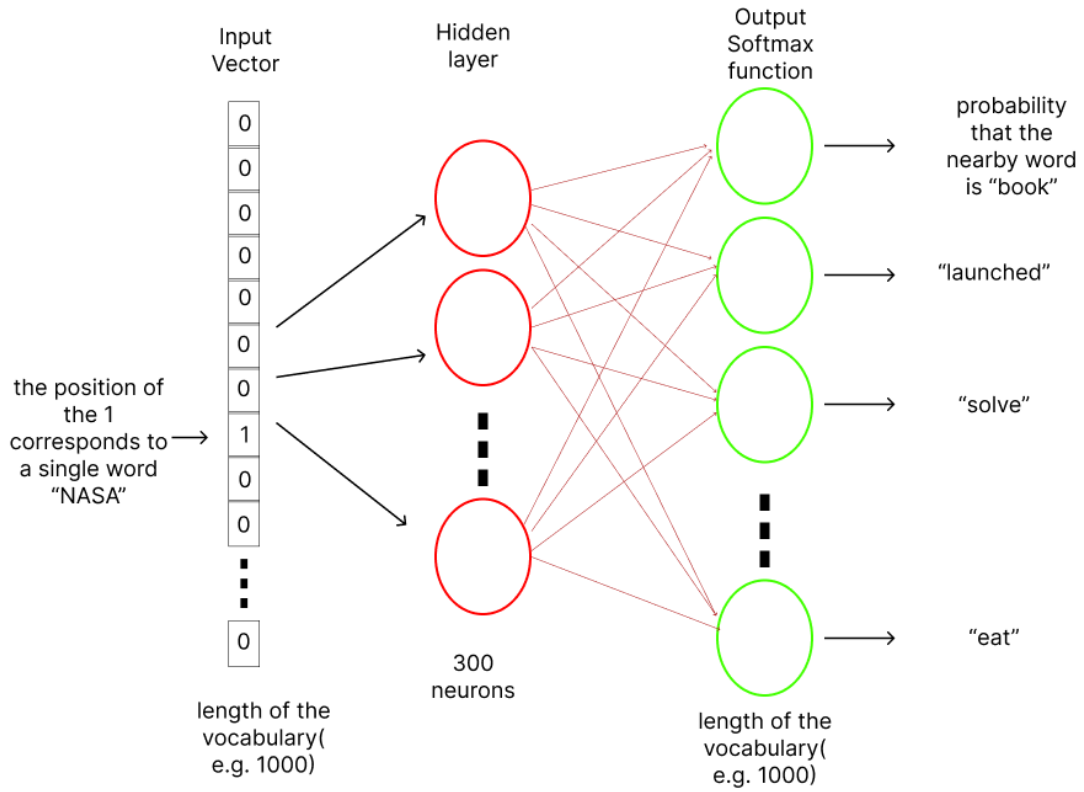

Figure 4: neural network

$$\begin{pmatrix} 0.49 \\ 0.047 \\ 1.58e-5 \\ 0.0012 \\ 0.00235 \end{pmatrix}$$

We want to increase the second value (0.047) and decrease the other values, so that we can match the prediction with the actual result. Therefore, we apply a loss function, then back propagate to update the values in the hidden layer of the network. As we mentioned, in the hidden layer resides the matrix, which holds the embedding of every word in our vocabulary. When updating the values of the matrix through the loss function, we are updating the embedding vectors, which correspond to the training data. After applying a huge amount of inputs on the neural network and doing the previous steps, we create more accurate embeddings.
We will use the pretrained Word2Vec model from Google news articles since it is trained on a massive corpus of texts which provides high quality embeddings. Additionally, we use this model as a starting point for our present task, which provides us the space for improvements and fine-tuning.
Having Word2Vec introduced, we carry on by making use of libraries that can assist

saving time and effort since they provide pretrained models and various other helpful functionalities.
This will be discussed and explained in the next chapter.

4 Implementation

4.1 Google-news dataset

As explained in the first chapter, when the data arrives to the server, it is passed to the method prepare. In this step the data consists of three entries, "`address`, "`html`", and "`hash_code`". These three attributes are later extended to hold more information to the document.
Initially, the HTML document is cleaned from HTML tags, which biases the model and do not contribute to the semantics of the text. The produced document needs further processing to achieve a final state, which can be used in the model.
First, throughout the implementation, we will make use of the Google news Word2Vec pretrained model. It has been trained on about 100 billion words from Google news. The model holds 3 million words and phrases, each presented by a 300-dimensional vector (300 dimensional embeddings) [Google Code, 2013]. This model suits our requirements, and we will be discussing the performance of the model later in chapter 5.
We use a framework called `genism`, which provides a variety of pretrained embeddings. Also, the Google news dataset is included. The following code snippet shows how the data is imported and offers an overview of what it provides for our purposes.

```
import gensim.downloader
training_set = gensim.downloader.load('word2vec-google-news-300')
training_set[1] ## output -->  array([ 0.0703125 ,  0.08691406,
0.08789062,  0.0625 , ..... , -0.0625 ]
```

The word vectors are represented by a standalone structure called "KeyedVectors". The structure is a mapping between keys and vectors .

```
type(training_set)
-> gensim.models.keyedvectors.KeyedVectors
```

The key is the word and the vector is the embedding vector. We can call the embedding vector by passing the word to the trained model as follows:

```
training_set["in"]
->array([ 0.0703125 ,  0.08691406,  0.08789062,
0.0625, ... , -0,0625])
```

Also, we can call the vector of a word by passing the index of that word. The output of the following code is the embedding vector of the second index, which corresponds to the key "in".

```
training_set.index_to_key[1] ## output --> "in"
```

4.2 Text cleaning

Before converting our words to vectors, we need to process them to clean any unnecessary words. We know from chapter 2 that the received data is passed to the prepare function, which is in another python file. The passed data is in a dictionary form. To make it easier to manipulate the data, we use data frames from `pandas`.
We create a data frame with following columns: `"address"`, `"html"`, `"hash_code"`, `"embedding_index"`, and `"mean_vector"`.

```
df = pd.DataFrame(columns=
                  ["address","html",
                   "hash_code",
                    "embedding_index",
                     "mean_vector"])
```

The data frame will be appended to a CSV file at the end. We make sure that the CSV file does exist. In case it does not exist, we create a new one. If the file does exit, we read it to append the new data later.

```
if not os.path.isfile("modified_data.csv"):
        df.to_csv("modified_data.csv", index = False)
df = pd.read_csv("modified_data.csv")
```

Then, we make sure that the received data does not already exist in the CSV file to prevent redundancy. By comparing the hash-codes of the URLs of the received website with the saved data, we can know if the website has been already saved. If it is true, we stop the process and do nothing further, otherwise we proceed.

```
hashes = df["hash_code"].tolist()
    if d["hash_code"] in hashes:
        return
```

The `d` is the dictionary passed from the server.
Now, we call a method named `read_data(d, df)`. This method adds the data received from the server to the pandas data frame. It initially adds a new row and sets the columns "`address`", "`html`", and "`hash_code`" corresponding to the dictionary from the server leaving the columns "`embedding_index`" and "`mean_vector`" with empty lists. Here, we start the first step of cleaning and preparing the text by turning all the words to lower case. Converting the text to lower case is useful because we can achieve uniformity between words. Words like "banana" and "Banana" should be considered the same in terms of their meaning. In this way, we avoid multiple variations of the same word.

```
def read_data(d,df):
     df.loc[len(df)] = [d["address"],
                                d["html"],
                                d["hash_code"],[],[]]
     df.at[len(df)-1,"html"] = df["html"][len(df)-1].lower()
```

We want to convert every word to a vector. To do this, we need to extract the words from the text.

Instead of having the whole text as a string, we tokenize it to a list of words (tokens). Therefore, we use the `word_tokenizer` from the library `nltk`, a tool to preprocess texts for further analysis.
For the sentence "NASA launched rocket last month", we get the following tokens: "NASA", "launched", "rocket", "last", and "month".

```
from nltk.tokenize import word_tokenize
df.at[len(df)-1, "html"] = clean_tokens(
                           word_tokenize(
                           df["html"][len(df)-1]),
                           df)
```

In the previous code snippet, we save the tokens in the "'html" column. We use `df.at[len(df)-1, "html"]` because the new data is placed in the last row.
The tokenized text gets passed to the method `clean_tokens()`. In this method, we first remove all non-alphabet letters (such as: (space)+-# etc.). Here we call the built-in method `isalpha()` (returns true if the string contains only alphabet characters, otherwise false) on every token and take only the tokens which return true. We remove the non-alphabet letters because they don't contribute to understanding the semantic of the text. The presence of non-alphabet characters can introduce noise to the data. The removal of such characters is useful in terms of achieving better accuracy.

```
tokens = [token for token in tokens if token.isalpha()]
```

Another thing we want to get rid of is stop words. Stop words are words like "and", "the", "at", "or" etc. These words appear frequently in a text and don't contribute to the processing of the underlying semantics of the text. By removing stop words, we also reduce the noise in the data and focus more on the content-words, which hold the actual meaning of the text. Removing them also offers a faster processing. When working with a huge amount of data (huge HTML contents), stop words reduce the processing speed because they appear much frequently than other words without adding any meaningful insights.
For the removal of such words, we use the module `nltk.corpus` from the library `nltk`. This module has a list of English stop words and other languages. We first download the list of stop words for the English language, then we call the function `stopwords.words("english")` from `nltk.corpus`. This returns a list of the stop words.

```
from nltk.corpus import stopwords
import nltk
nltk.download('stopwords')
stop_words = set(stopwords.words("english"))
```

Now that we have a list of stop words, we iterate through the previously generated tokens and remove any stop words occur.

```
tokens = [token for token in tokens if token not in stop_words]
```

Another thing in terms of preprocessing the text, is to reduce words to their base or root form. By replacing the words with their roots or with their stems, we reduce variation. Words can exist in different forms due to grammatical variations.

We want to allow better grouping of words with the same meaning, for instance the words “eating”, “ate” or “eats” contribute the same meaning in terms of the semantic processing. By using the techniques to reduce words to their stem, we reduce redundancy which contributes to the performance of the search engine.
To achieve this, we can use two variants of reducing words, stemming or lemmatization [Kurtis Pykes, 2023]. Stemming reduces the inflected word to its word stem. For example, “swimming” and “swimmer” will be reduced to “swim”.
Lemmatization reduces inflected words to their base or dictionary form, called the lemma. Lemmatization takes into consideration the context surrounding the inflected word and determines the appropriate lemma. In contrary to stemming, lemmatization is more accurate in terms of maintaining the semantics of the word.
Since we want to preserve the semantics of the words in the text, we used lemmatization. We can use the `WordNetLemmatizer` from the module `nltk.stem`.

```
from nltk.stem import WordNetLemmatizer
nltk.download('wordnet')
lemmatizer = WordNetLemmatizer()
tokens = [lemmatizer.lemmatize(token, pos = "v") for token in tokens]
```

The final `clean_tokens()` method looks like this:

```
def clean_tokens(tokens):
    tokens = [token for token in tokens if token.isalpha()]
    stop_words = set(stopwords.words("english"))
    tokens = [token for token in tokens if token not in stop_words]
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(token,pos="v") for token in tokens]
    return list(set(tokens))
```

After cleaning the text, the final output of our first example “NASA launched rocket last month” is as follows:

```
clean_tokens(word_tokenize("NASA launched rocket last month"))

-> ['nasa', 'launch', 'rocket', 'last', 'month']
```

Finally, the tokens are saved in the data frame, so we can access them again in chapter 4.3.

4.3 Converting tokens to vectors

A very important step of the preprocessing is to convert the tokens created in the last chapter to the corresponding embeddings.
Initially, we had empty lists in the columns `"embedding_index"` and `"mean_vector"` in the data frame.
One embedding vector has 300 values. Saving the vectors for all tokens consumes too much storage. To prevent this, we can only save the index of the vector in the trained set. In order to accomplish this, `genism` offers a function which turns the keys (words) into indices, which we can access by providing the key.

Table 1: data frame after all modifications

adress	html	hash_code	embedding_index	mean_vector
https://www.tripadvisor.com	['tripadvisor', 'travelers',..	28fb5af43 ..	[4943, 1324, ..	[0.013861185, 0.04280759,..
https://www.imdb.c...	['imdb', 'review',..	7a108e..	[422171, 60482,..	[0.08782417, 0.0068536378,..
https://www.goodreads.c...	['goodreads', 'meet',..	685b1..	[584, 147,..	[0.051466472, -0.006594352,..
https://foodnetwork.c...	['food', 'network',..	881a3..	[560, 751,..	[-0.012898685, 0.04590153,..

For example, the token "nasa" has the index 290335.

```
training_set.key_to_index["nasa"]
-> 290335
```

In our `prepare()` method, we call the function `add_embedding_indices(df)`. In this function we take the last row, where the new data is saved, and iterate over the tokens we created previously. We extract the index corresponding to the token and append it to a lis. Finally, we save the list in the data frame.

```
def add_embedding_indices(df):
    i = len(df)
    idxs = []
    for word in df["html"][i-1]:
        if training_set.__contains__(word):
            idxs.append(training_set.key_to_index[word])
    df.at[i-1, "embedding_index"] = idxs
```

The last step of the preparation is to compute the mean vector of all embeddings of the tokens of a document and save it in the data frame. Therefore, we create the method `compute_mean_vectors(df)`. Here, we first extract the embedding indices of the last row in the data frame, then use the function `mean()` from `NumPy` to compute the mean vector. This mean vector is important to perform comparisons in the vector space. We will utilize them in chapter 4.6.

```
def compute_mean_vectors(df):
    vectors = df["embedding_index"][len(df)-1]
    df.at[len(df)-1, "mean_vector"] = list(
                                      np.mean(
                                      [training_set[index]
                                      for index in vectors],
                                      axis=0))
```

After applying all these modifications on the original HTML content, we can now save the data frame to a CSV file, and the final result is showed in table 1.

4.4 Visualizing search functionality with mock data

As we discussed earlier, Word2Vec allows word embedding, where we can apply various computations and algorithms in the vector space. This is the basis for comparing existing documents with incoming queries.
Every document consists of many vectors (words). These vectors create clusters in the vector space, based on the subject of the document.

Through these clusters, we can define boundaries, which separate the documents from each other.
Since our vectors have 300 dimensions, it becomes challenging to visualize them. To come over this challenge, we can create mock points in a 3-dimensional vector space, which makes it possible to visualize the clusters in the vector space. These clusters can be compared to the real clusters in a 300-dimensional vector space, where the same algorithms apply.
From `sklearn.datasets` we make use of the `make-blobs()` function, which generates data based on our needs. Therefore, we generate two set of points corresponding to two documents and 300 samples in each set in a 3-dimensional vector space. These two set of points form two clusters in the vector space. We intentionally make the two clusters separated to simulate the actual case (in most of the cases the clusters are separated. This depends on the quality of the model).

```
n_samples = 300
n_features = 3
n_clusters = 2
random_state = 0
cluster_std = [1, 1.0]
data, labels = make_blobs(n_samples = n_samples,
                          n_features = n_features,
                         centers = n_clusters,
                           cluster_std = cluster_std,
                           random_state = random_state)
```

Figure 5 shows a top and side view of the 3-d plot of the generated vectors.

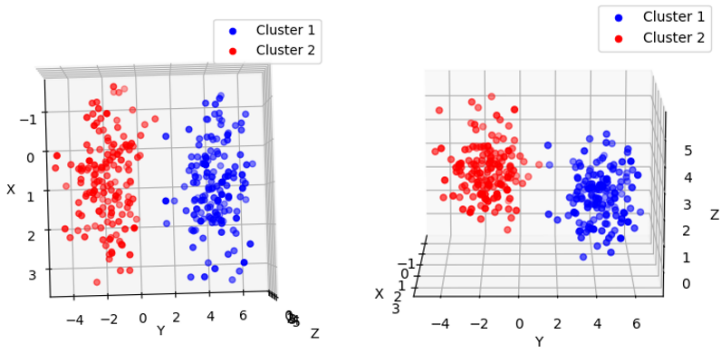

Figure 5: Top and side view of the generated blobs

We can observe the boundaries every cluster creates. In higher dimensions, the clusters also look similar to the ones in figure 5, depending on how much the documents are dissimilar.
We can imagine that every cluster is a document (HTML content). When a query arrives, we can determine which document (cluster) the query is closer to.
The query can also be considered as a document, and can also be visualized on this plot. We take an example of a query consisting of five words. These words correspond to five vectors in our 3-dimensional vector space. The query vectors have also a certain

relationship to each other, so they also create a cluster, unless we have a random query. Since we want to represent real queries, we create queries with less deviation, as shown in figure 6.

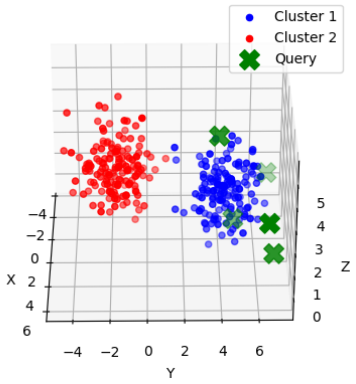

Figure 6: generated clusters including a manually generated query

The green points represent the words of the query. By observing the points in figure 6, we can be certain that the search query returns the first document (cluster1) since all the points of the query are closer to cluster 1, and we only have 2 documents. To take this decision, we measure the distance between the clusters by using the cosine distance. Lower values of the cosine distance imply that the two vectors tend to point in the same direction. The distance is computed using this formula:

$$\text{Cosine Similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \cdot \|\mathbf{B}\|}$$

$$\text{Cosine Distance} = 1 - \text{Cosine Similarity}$$

A and B are vectors, and $\|\mathbf{A}\|$ and $\|\mathbf{B}\|$ are the Euclidean norms of the vectors A and B. Cosine distance ranges from 0 to 2, where 0 means the two vectors are pointing in the same direction and 2 means that the vectors are pointing in exactly the opposite directions.
We take the mean vector of every document, including the query. These mean vectors serve as representatives of the document.
The distance is then measured between the query's mean vector and each document's mean vector.
In our case, the cosine distance between the mean vector of the query's cluster and the mean vector of cluster 1 delivers 0.009351, and the distance between the query and cluster 2 is 1.139979. These values indicate that document 1 is more relevant to the query than document 2.
To deliver better representation of the similarity between documents, we use the cosine similarity metric. The cosine similarity is computed by subtracting 1 from the cosine distance:

$$\text{Cosine Distance} = 1 - \text{Cosine Similarity}$$

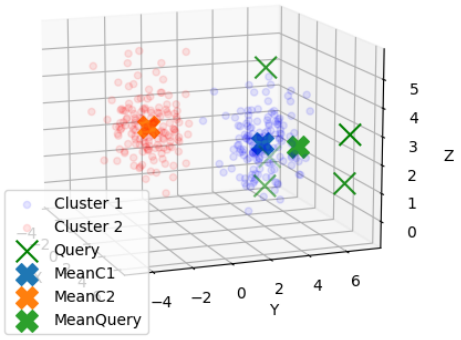

Figure 7: Centroids of the clusters of the documents and the query

The cosine similarity has values range from -1 to 1, where -1 indicates maximal dissimilarity and 1 maximal similarity.

4.5 Visualizing search functionality with real data

As mentioned in the previous chapter, visualizing the real data is challenging. To explain this difficulty, we take two documents as an example and try to plot them in the same way we did with the mock data. First, to achieve maximal separation between the clusters, we need to find two documents that have a low cosine similarity value.
This can be done by taking any document from the saved data and computing the cosine similarity between this document and all other existing documents. Then we take the document with the lowest similarity value.
For example, we take the document extracted from the NASA website ("https://www.nasa.gov/press-release/nasa-expands-options-for-spacewalking-moonwalking-suits-services/") and extract the mean vector, which we already calculated in previous steps. In this example, we compare the cosine similarities with all the documents from BBC news which we will use in evaluating the model in chapter 5.
We calculate the cosine similarity between the mean vector of the NASA document and the mean vectors of all remaining documents, then we sort them from the lowest value to the highest, and we take the first one which is the most dissimilar document.

```
sorted([
    (1-cosine(df["mean_vector"][2229],
              df["mean_vector"][i]),
              i) for i in range(len(df))],
              key = lambda x : x[0])[0]

Output --> (0.36121327024053107, 1607)
```

The document with index 1607 has the lowest similarity value.
Since every word is presented by a 300 dimensional embedding, we need to find a way to reduce them to at least 3 dimensions to be able to show them in the plot.

Therefore, we use principal component analysis (PCA). This is a method to transform data into variables called principal components. These components are ordered by their influence on the variance in the data.
With this method, we can remove features from the matrix that do not hold meaningful information and do not influence the data. This has its limits since removing too many features causes loss of information. Because we have 300 dimensions, we expect to lose too much information which causes inaccuracy. Another problem is reducing the dimensions to 3 and plotting them mostly lead to different shapes since using only three components could not be enough to present the whole data.
To illustrate the problem with visualizing the documents in a 3D plot, we proceed with our initial example and apply PCA on both documents. First, we need to reform the vectors to be suitable for the PCA function:

```
def reform_data(indices):
    data = np.array([trained_set[x] for x in indices])
    return data
```

Instead of having the vector in the column vector presentation, we represent every vector as a sample in a row vector presentation. After applying the `reform_data()` function, the matrix of one document will look like this:

$$\begin{bmatrix} 0.123 & 0.456 & 0.789 & \dots & \dots & \dots \\ 0.234 & 0.567 & 0.890 & \dots & \dots & \dots \\ 0.345 & 0.678 & 0.901 & \dots & \dots & \dots \\ \vdots & \vdots & \vdots & \ddots & & \\ \vdots & \vdots & \vdots & & \ddots & \\ \vdots & \vdots & \vdots & & & \ddots \end{bmatrix}$$

In this presentation, the rows are the samples (words) and the columns are the features (embeddings). The matrix has 300 columns since we are dealing with 300 dimensional vectors. This matrix can now be passed to the `PCA()` function as follows:

```
from sklearn.decomposition import PCA
nasa = reform_data(df["embedding_index"][2229])
dissimilar_doc = reform_data(df["embedding_index"][1607])
PCA_dissimilar_doc=PCA(3).fit_transform(dissimilar_doc)
PCA_nasa=PCA(3).fit_transform(nasa)
```

`PCA(3)` means we are taking the first three principal components to plot them in a 3 dimensional vector space.

As we can see in figures 8 and 9, we can hardly define any boundaries between the clusters. Although we intentionally chose two documents that are relatively dissimilar, the two clusters seems to be inside each others which means the two documents should have higher similarity value. In reality, we would expect a clear separation between the two documents in the vector space, which we were not able to visualize in this scenario.
Also, plotting the queries does bot bring us to any conclusion. The reason is, even if the plot of the queries made sense, we will not be able to determine to which document the query is directed to only by visual observation since we can not define separated clusters.

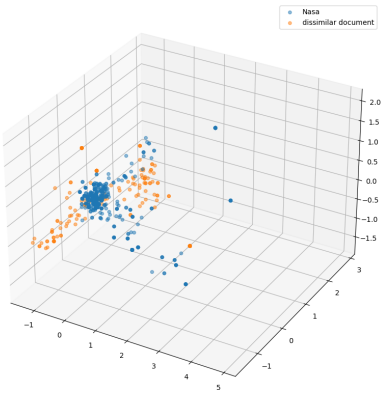

Figure 8: side view of real documents in the vector space

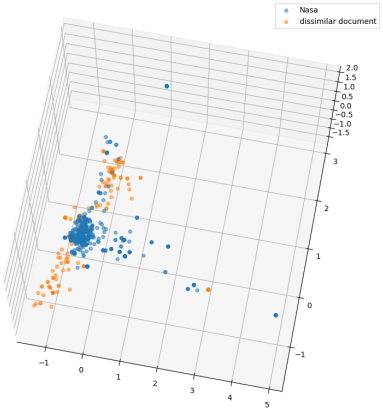

Figure 9: top view of real documents in the vector space

4.6 Search function implementation

In the same manner, the search function is constructed. First, we implement a function, which computes the mean vector of the query. It takes the query as a string and splits it into words (tokens). We check if the tokens are included in the vocabulary of the trained dataset. In case the token does not exist, we drop it. We use the function `mean()` included in `NumPy` to calculate the mean vector.

```
def mean_vector(data):
    keys = [x for x in data.split()
            if training_set.__contains__(x)]
    meanVec = np.mean(np.array(list(
                                map(lambda x :
                                    training_set.get_vector(x),
                                    keys))),axis=0)
    return meanVec
```

The search function iterates over all saved documents and extracts the mean vector of everyone which we have already computed.

The query's mean vector is then compared to the mean vectors of the documents by computing the cosine-similarity.
The results are sorted from the highest similarity value to the lowest as a tuple consisting of the similarity value and the index of the document in the data frame. We additionally add a variable to the function to control the number of results we want to show.

```
def search(input,nresults):
    dataframe = pd.read_csv("modified_data.csv")

    sims = [(i, 1-cosine(mean_vector(input),
                  dataframe["mean_vector"][i]))
                  for i in range(len(dataframe))]
    sims.sort(key = lambda x : x[1], reverse = True)
    links = [(dataframe["address"][i[0]], i)
                              for i in sims[0 : nresults]]
    return links
```

Through the indices in the tuples, we can extract the links leading to the websites and add them to the tuple in the links list.
By calling the function `search("launching rockets",3)`, we get the following output:

```
[('www.nasa.gov', (4, 0.3641294170638002)),
('www.tripadvisor.com', (0, 0.22559111531594778)),
('www.bbc.com', (5, 0.2171817715642993))]
```

5 Performance

5.1 Testing on a small amount of websites

As an initial step for testing if the model is working as we expect, we start by testing the model on 10 saved websites, with two queries for each website. We label every query with the index of the website in the data frame to compare them with the predicted labels. The queries are executed on the following websites where every website has a different subject, and some of them also have some elements in common:
https://www.tripadvisor.com, https://www.imdb.com, https://www.goodreads.com, https://foodnetwork.co.uk, https://www.nasa.gov, https://www.bbc.com (Europe news), https://www.bbc.com (Football news), https://www.nationalgeographic.com, https://www.kaggle.com, https://huggingface.co.
These links have indices corresponding to their order.
We generate following queries with their labels to the corresponding websites:

```
queries = [
    ("Top␣things␣to␣do␣according␣to␣Travelers␣Choice",0),
    ("movies␣and␣TV␣shows",1),
    ("books␣recommendations␣",2),
    ("Delicious␣recipes␣",3),
    ("spacewalking␣and␣moonwalking",4),
    ("Latest␣news␣Europe",5),
    ("Football␣updates␣",6),
    ("mummy␣excavation␣",7),
    ("Datasets␣competetions",8),
    ("Word2Vec␣model␣",9),
    ("Things␣To␣Do␣vacation",0),
    ("actor␣celebrity",1),
    ("review␣book",2),
    ("recipes␣hunger",3),
    ("space␣physics",4),
    ("western␣news",5),
    ("club␣sport",6),
    ("egyptian␣civilization",7),
    ("datasets␣for␣data␣science",8),
    ("word␣embedding␣language",9)
]
```

After calling the search function, we compare the predicted labels with the actual labels, and set a 1 if both labels are equal or 0 otherwise and return a list of zeros and ones.
By passing the previous queries to the search function, we get the following list:
`[0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1].`
In the list, we have five zeroes which means that five queries out of 20 referred to the wrong website. But since we want to show more than one result on the user interface, we can proceed with our testing by checking if the real website is in the first 3 results. Executing the same previous test, we get for every query a list of three indices referring to three different websites:

```
[[2, 0, 1],[1, 2, 5],
[2, 8, 9],[3, 2, 9],
[4, 1, 0],[5, 1, 6],
[6, 8, 9],[0, 4, 7],
[8, 9, 4],[9, 8, 4],
[2, 0, 1],[1, 2, 0],
[2, 9, 8],[2, 0, 5],
[4, 8, 0],[5, 1, 0],
[6, 0, 2],[2, 7, 0],
[8, 4, 9],[9, 7, 2]].
```

We can now check if the true index is in the list for every query. Again, if the index is included in the list we return 1 else a 0:
`[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1].`
The result looks much better now since we allowed more space for failure. We can see that in 19 out of 20 queries the true website appears in the first three results, and only one query did not retrieve the correct website.

For further experimenting, we will use more than 10 websites and see how this affects the accuracy of the search engine.

5.2 Introduction to TF-IDF and query generating

Visiting a bigger number of websites and saving them through the plugin in a short period of time is challenging. To tackle this challenge, we can use a dataset from BBC news. The dataset consists of 2225 documents from the BBC news website collected from 2004-2005. The documents are divided into five categories (business, entertainment, politics, sport, and tech). Every document in this dataset can serve as the content of a website [D. Greene and P. Cunningham, 2006].
To test the search engine on these documents, we need to have queries corresponding to the documents. Since we have 2225 documents, forming the queries manually consumes too much time and effort. To solve this problem, we use a technique called "term frequency and inverse document frequency", or shortly TF-IDF which will be explained in the next chapter.

5.2.1 TF-IDF

Using the same model on our data to extract keywords that are relevant to the document to form queries does not help us to evaluate the model. If we use the same model to generate queries, we tell the model to give us vectors, that are nearly the same as the embedding vectors of the document (since we want the query to be relevant to the document). In this way, we already know that the query returns the right document (most of the time) because we intentionally choose queries with similar vectors to the vectors of the document. This means, we are maximizing the cosine-similarity between the document and the query.
To avoid this, we can extract keywords that contribute the most to the context of the document using other techniques such as TF-IDF. This is a machine learning algorithm used in natural language processing. It is used to evaluate how relevant a word is to a document. It serves the same purpose as Word2Vec by transforming words into numerical vectors, but using an alternative approach.
TF-IDF consists of two fundamental terms, the term frequency and the inverse document frequency. Both terms should be calculated and multiplied with each other to get the final result.
Term frequency gives the frequency of a word in a document, and is calculated by dividing the number of repetitions of a word in one document over the number of words in the same document.

$$\text{Term frequency} = \frac{\text{number of repetitions of a word}}{\text{number of words in the same document}}$$

For example, "fast car", "fast bike", and "car bike fast" are sentences. We can calculate the term frequency of every word in every sentence using the previous formula as shown in table 2.

Table 2: Term Frequency Table

Term	Document		
	"fast car"	"fast bike"	"car bike fast"
"fast"	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{3}$
"car"	$\frac{1}{2}$	0	$\frac{1}{3}$
"bike"	0	$\frac{1}{2}$	$\frac{1}{3}$

The inverse document frequency is the frequency of a word across all documents. IDF tells us how common or rare a word is in the whole document. The IDF value ranges from 0 to 1, and the closer it is to 0, the more common a word is. In contrary, the closer it is to 1, the rarer the word is. The IDF is determined by dividing the total number of documents by the total number of documents containing the word, then computing the logarithm of the division. For the previous example, we get the IDF values shown in table 3.

Table 3: IDF Values for Words

Word	IDF-value
"fast"	0
"car"	$\log(\frac{3}{2})$
"bike"	$\log(\frac{3}{2})$

After calculating the two terms, we multiply them with each other [Bruno Stecanella, 2019]. Through the multiplication, we get the vector of the document, in which every word of the vocabulary is presented by its TF-IDF value. The values in the vectors are not the same in every document because a word doesn't have the same TF value in each document. If a word doesn't exist in a document, the corresponding value in the vector is 0 (tf=0 -> tf*idf=0).

Table 4: TF-IDF Values for Words in Documents

Word	"fast car"	"fast bike"	"car bike fast"
"fast"	$\frac{1}{2}\log(\frac{3}{3}) = 0$	$\frac{1}{2}\log(\frac{3}{3}) = 0$	$\frac{1}{3}\log(\frac{3}{3}) = 0$
"car"	$\frac{1}{2}\log(\frac{3}{2})$	0	$\frac{1}{3}\log(\frac{3}{2})$
"bike"	0	$\frac{1}{2}\log(\frac{3}{2})$	$\frac{1}{3}\log(\frac{3}{2})$

In the first sentence "fast car", we can see a higher value for car, which means the word "car" is receiving more importance than the word "fast". In this way, we represent every document with a vector where the values represent the words in the document and their importance in terms of holding the context of the document.

5.2.2 Generating queries with TF-IDF

As mentioned earlier, generating the queries manually for each document is challenging. Therefore, we used the previously explained algorithm. TF-IDF fits our need of generating document related queries since it extracts keywords from the document that are contributing the most to the context of the document. By combining these keywords, we can form the queries.
We used the implementation of TF-DF included in the module `sklearn.feature_extraction.text` from `sklearn`. From this module we import the `TfidfVectorizer`, to which we pass the documents, and it will generate the vectors.
Before performing TF-IDF on the documents, we need to clean the documents from any words that don't have any impact on the context of the document. We first start removing any numerals occurring in the text since we don't want to have them in the queries. To accomplish this, we write a simple regular expression which will remove all the numerals from the text. As we did in chapter 4.2, we remove stop words using `nltk`. Therefore, we tokenize the document and remove any words that are considered as stop words. Finally, we join the tokens together into one document.

```
def clean_text(sent):
    cleaned_sentence = re.sub(r'\b\d+\b', '', sent)
    tokens = sent.split()
    stop_words = set(stopwords.words("english"))
    no_stop_words = [token for token in tokens
                     if token not in stop_words
                       and token.isalpha()]
    cleaned_sentence = "␣".join(no_stop_words)
    return cleaned_sentence
```

Since we want to extract keywords from each document from the dataset, we handle every text independently. It means, we split the text into sentences, and these sentences are considered as documents for the TF-IDF. For illustration, if the text was "US and UK, raising millions of dollars for African famine relief. The re-release also marks the 20th anniversary of the original recording.", we receive the two sentences: "US and UK, raising millions of dollars for African famine relief" and "The re-release also marks the 20th anniversary of the original recording.". The TF-IDF algorithm considers the sentences as two documents, and takes the words in the two sentences to calculate the values.
We can now call the function `fit_transform()` on a list of sentences corresponding to one text from the dataset. After transforming the sentences, we retrieve the generated vectors :

```
v = TfidfVectorizer()
transformed = v.fit_transform(doc)
vectors = transformed.toarray()
```

The variable `vectors` contains the vectors for every sentence of the text. For the previous text, we get the following output for the `vectors` variable:

```
array([[0.37796447, 0.        , 0.        , 0.37796447, 0.37796447,
        0.        , 0.37796447, 0.        , 0.37796447, 0.        ,
        0.37796447, 0.        , 0.37796447],
       [0.        , 0.40824829, 0.40824829, 0.        , 0.        ,
        0.40824829, 0.        , 0.40824829, 0.        , 0.40824829,
        0.        , 0.40824829, 0.        ]])
```

As we can see, it consists of two different vectors corresponding to 2 different sentences. Every vector contains all the words in the vocabulary, and every word in the vocabulary has an index in the vector.

```
v.vocabulary_
-->
{'us': 12,
 'raising': 8,
 'millions': 6,
 'dollars': 3,
 'african': 0,
 'famine': 4,
 'relief': 10,
 'the': 11,
 'also': 1,
 'marks': 5,
 'anniversary': 2,
 'original': 7,
 'recording': 9}
```

In the first vector, we have the value 0.37796447 with index 0. Index 0 corresponds to the word "african", which is in the first sentence but not in the second one. That is why we have a 0 at index 0 in the second vector.
The idea is to choose the words which contribute the most to the context of the text. Meaning, we can take the words with the highest values in the vectors. But a word can occur in more than one sentence. The same word can have different values in every vector because the word has different importance degree in every sentence. To solve this problem, we can take all the values of the word from each vector and compute the average. This will give us the importance of the word in respect to the whole document. We then sort the words from the words with the highest mean value to the lowest.

```
v = TfidfVectorizer()
    transformed = v.fit_transform(doc)
    vectors = transformed.toarray()
    means = []
    keys = list(v.vocabulary_.keys())
    for word in keys:
        values = [vectors[i][v.vocabulary_.get(word)]
         for i in range(len(vectors))]
        means.append((word,sum(values)/len(values)))

     means = sorted(means, key = lambda x:x[1],reverse = True)
```

Now, we have a list of the words sorted from the most relevant words to the least relevant.

To choose words for our query, we take the first 20 words and choose randomly 10 of them. These ten words compose our query.

```
query_samples = [means[r][0] for r in
                                    random.sample(range(21), 10)]
```

After applying the previous steps for every document, we take one last action which is computing the mean vector of the query using our original Word2vec model. Computing the mean vector in advance makes the searching functionality runs easier when we do it in parallel, which will be discussed in chapter 5.3. As a final step, we save the generated queries in a CSV file. It consists of three columns. The column `document` (important for labelling the queries and comparing them with the predictions). The document is simply the name of the text file combined with its category (e.g. "business001.txt"). Additionally, we have the columns `query` which holds the generated queries and `mean_vec` which contains the mean vectors of the queries.
The final look of the data is shown in table 5.

Table 5: Insight inside the generated queries

Document	Query	Mean Vector
business001.txt	rose profits internet quarter...	[-0.038938735, 0.08812269, ...]
business002.txt	thursday could current ...	[-0.008850098, 0.03701477,...]
business003.txt	loan assets company ...	[0.065804616, 0.049595423,...]
business004.txt	ba prices turnover ...	[-0.03770981, 0.09681396, ...]
business005.txt	allied lifts shares ...	[0.04499054, 0.0413208, ...]

5.3 Executing the queries on the BBC news dataset

The dataset has 2225 text files. Executing the search engine on each query on all data will consume too much time and is less efficient. To encounter this issue, we can parallelize the process. Since we are working on one machine (not distributed) we can take advantage of the existing cores and run the search function on all queries in parallel.

5.3.1 Ipyparallel

`Ipyparallel` is a python library which allows us to run processes on multiple cores in parallel. To be able to preform the search in parallel, we need to do some modifications on our search function, but only temporary. First, we need to avoid using the Word2Vec trained model inside the function and try to pass it from outside. The reason behind this is that the model needs to be passed to each working engine which consumes too much memory and takes too much time.
In the search function, the Word2Vec model is used to compute the mean vector of the query. In the previous chapter, we computed it already and saved it in the CSV file.
We still need to pass the mean vectors of the queries from outside the function. Therefore, we put the query together with the corresponding mean vector in a tuple.

Later in the search function, we extract them for further usage.
In addition, we also extend the tuple to hold the name of the document (the column "document" in the data frame) and the number of queries we want to perform. The name of the document is needed to directly compare the predictions with the true labels (the name of the text file) to save another comparison step.
The function now returns a 0 if the predicted label is the same as the actual label. The number of queries is required to adjust the number of documents we want to consider in the search. If the number of the queries was 20, then we only take 20 documents from the data frame. We will also change this later as we do further experiments. Also, we pass the data frame from outside the function to avoid loading the CSV file inside the function.
The final modified search function looks like this:

```
def search1(inp,nresults,dataframe):
    from scipy.spatial.distance import cosine

    sims = [(i,1-cosine(inp[1],
                        dataframe["mean_vector"][i]))
           for i in range(inp[3])]
    sims.sort(key = lambda x:x[1],reverse = True)
    links=[dataframe["address"][i[0]]
            for i in sims[0:nresults]]
    return 1 if inp[2] in links else 0
```

The search function is ready for parallelization, and we can define the `parallel_search()` function which makes use of the library `ipyparallel`. The `parallel_search()` function takes the number of queries as a parameter and returns a list of ones and zeros where zero means a false prediction and one a true prediction. Before implementing the function, we initialize the engines by calling the `Client()` function and loading the direct view:

```
from ipyparallel import Client
rc = Client()
dv = rc[:]
```

In the `parallel_search()` function we call the `div.map_sync()` and pass the modified search function alongside the parameters:

```
def parallel_search(nqueries):
    search_result = dv.map_sync(search1,
    [(dfq["query"][i],dfq["mean_vec"][i],dfq["document"][i],nqueries)
     for i in range(0,nqueries)],[3 for i in range(0,nqueries)],
     [dataframe_documents for i in range(0,nqueries)])
    return search_result
```

Executing the search function on all 2225 queries and all documents takes about 17 seconds with 16 cores. Running the same queries on the original search function without parallelization is expected to take more than 15 minutes.

5.4 Results of the parallelized search queries

First, we take 20 queries from the previously generated queries and execute them using the search engine. After every execution of the queries on the search engine, we increase the number of websites by 10. Initially, we have 1 website and 1 query which means we reach an accuracy of 100%, then the accuracy starts to drop as we increase the number of documents. Every time we increase the number of documents, we compute the ratio (number of correct labels)/20. Figure 10 shows how the ratios change across different number of websites.

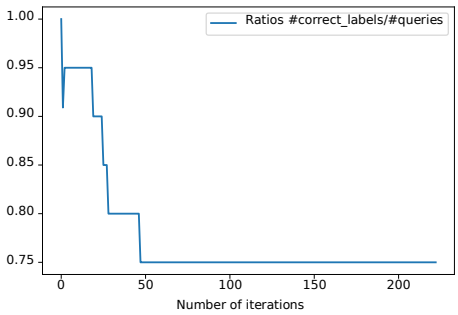

Figure 10: Accuracy of 20 queries with increasing number of documents

In figure 10, the x-axis shows how often we increase the number of documents by the factor 10. For example, 50 means that we increased the documents 50 times by 10, that is 500 documents.
The y-axis shows the computed ratios across different number of documents.
As we can see, the accuracy first drops rapidly, but then keeps a constant value of 0.75 which means from the 20 queries we get 5 false labels.
Particularly, this test shows a consistent value of 0.75 after 500 documents. This is because the subject of the news changes after 500 documents, from business to entertainment. The change of the subject creates more distinction between the queries and the new documents which causes more dissimilarity to the new texts.
To experiment further, we increase the number of the documents and the queries at the same time. Also, here we increase them every time by 10 queries and documents.
As we can see in figure 11, the accuracy keeps dropping at the beginning, but then slows down and maintains a value of approximately 0.65. The average of all ratios is approximately 0.71.
With these tests, we get different accuracy values compared to the beginning of the chapter when we tested on small data and got values around 0.9. This difference shows us the impact which the quantity of the documents has on the performance of the search engine.
These are reasonable results since we also chose a challenging dataset to test the model on. As discussed earlier, the news consists of 5 categories. When generating queries in the same categories, we expect overlaps because there is a high chance of finding two texts that handle the similar subject. Yet, the model was able to keep retrieving the true document in almost 70% of the cases.

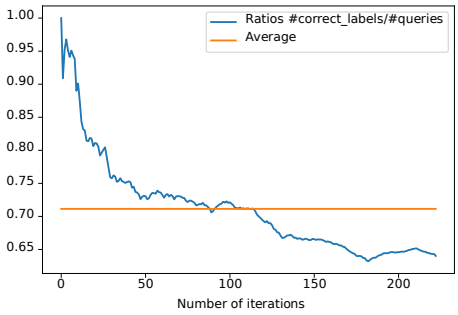

Figure 11: Accuracy of increasing the number of queries and documents

We can simulate a real life scenario where the user performs distinct set of queries while the quantity of the websites is increasing. To realize this case, we increase the number of documents by ten after executing a specific number of queries, but this time we choose random queries every loop, unlike the first test where we had fixed queries. We first start with one document and one query, and every time we increase the number of documents we increase the number of queries by one until we reach 20 queries. Every time we add new documents, we choose new queries at random from the previously generated queries which correspond to the current documents. Implementing the previous thoughts results in the ratios shown in figure 12.

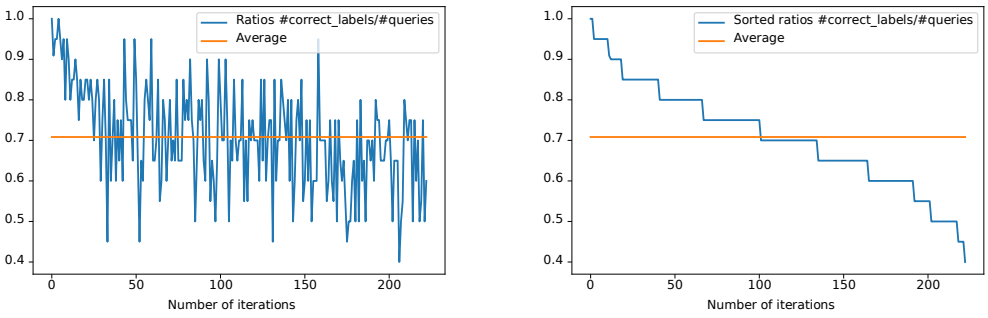

Figure 12: Executing 20 random 10 words queries with increasing number of documents

It is noticeable that the ratios differ significantly depending on the queries. We also notice that there is a significant variation between the ratios of the single executions, ranging from 0.4 to 1. The model achieved an average accuracy of approximately 0.71 which is similar to the previous test.
We can challenge the model more by reducing the number of words in the query to 5 words. We execute the previous test with the same settings, only this time with queries consisting of 5 words. Therefore, we generate the queries using TF-IDF, the same method we used to generate the queries with 10 words.
In figure 13, we can observe how the model is starting to suffer.

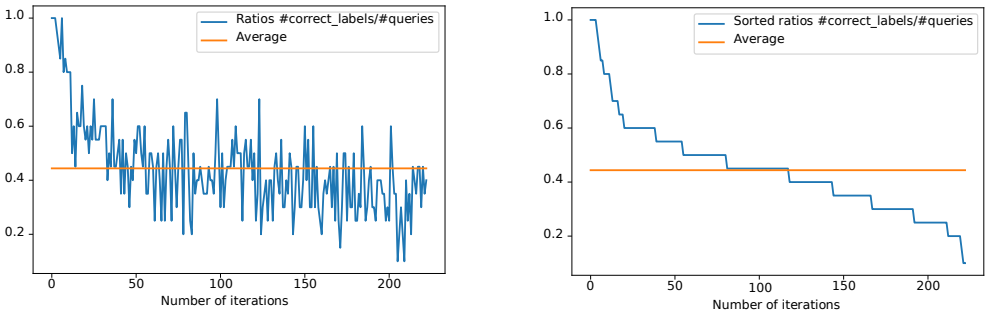

Figure 13: Executing 20 random 5 words queries with increasing number of documents

We have approximately 0.5 accuracy on average.
This low value is due to the less information a single query holds. Additionally, we generate the queries automatically using the TF-IDF method, which does not produce high quality and more precise queries as the user itself.

6 Improving the model

6.1 Influence of lemmatization

As we explained in chapter 4.2, we reduce words to their root form (lemma) using the lemmatization method. This can contribute to the performance of the model and reduce the noise in the data to achieve better accuracy. Depending on the model, lemmatization can either contribute positively or negatively, or it does not change anything.
To test this assumption on our model, we run the tests from chapter 5 again with 10 words queries. This time on the same dataset from BBC news, but without lemmatization. By executing the same test with increasing number of documents and queries, we get the results shown in figure 14.

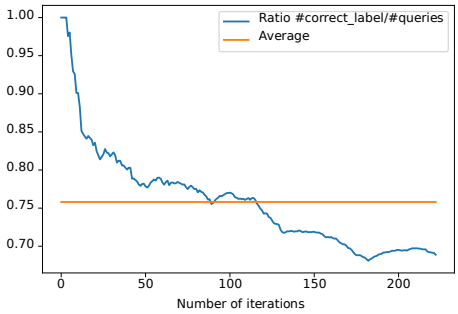

Figure 14: Ratio of correct labels without lemmatization

We definitely get better accuracy values without lemmatization.
We managed to achieve an average ratio just above 0.75 meanwhile with lemmatization we get an average of approximately 0.7. It is not a big difference, but still an improvement.
The reason behind this small improvement is that in our model derivative words appear to be the most similar words. For example, calling the function `most_similar("eat")` on our trained set, we receive the following output:

```
[('eating', 0.7529403567314148),
 ('ate', 0.7013993859291077),
 ('eaten', 0.6724975109100342),
 ('eats', 0.6589087843894958),
 ('munch', 0.6417747735977173),
 ('eat_healthfully', 0.6315395832061768),
 ('eat_fatty_foods', 0.6280142068862915),
 ('consume', 0.6184970140457153),
 ('Nutritionists_recommend', 0.6183844804763794),
 ('overeaten', 0.6109130382537842)]
```

The most similar words to “eat” according to the model are “eating” followed by “ate”, “eaten”, and “eats”. Keeping all these words in the document is beneficial, as they drive the mean vector more to their side. For instance, in a document focused on food, if the words “eat”, “eaten”, and others are included, they contribute to the context all together much better than only the lemma.

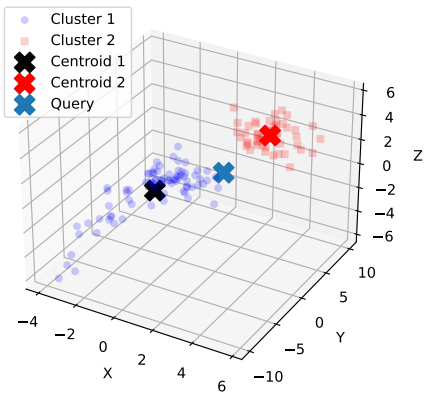

Figure 15: Two clusters in the vector, one with a significant number of outliers

Cluster 1 in figure 15 is a good example. We can observe the spreading among the points. The points far from the centroid are outliers and do not contribute much to the actual meaning of the document, but they have an influence on the values of the mean vector. The points near the centroid (most of the points) have more influence on the mean vector values because of their proximity to each other. If the word "eat" is among these points, the similar words will also be around the word “eat”.

Removing the derivative words will cause the centroid to move toward the scattered points.
We can verify this assumption by taking a look at the similarity values before and after the lemmatization.

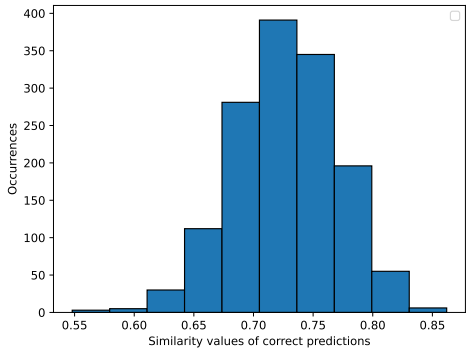

Figure 16: Similarity values of correct labels with lemmatization

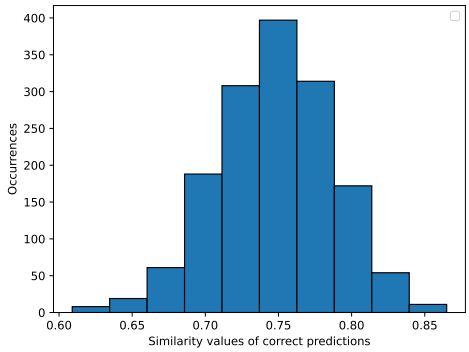

Figure 17: Similarity values of correct labels without lemmatization

Figure 16 shows the similarity values of the correct predictions from 2225 executed queries with lemmatization where figure 17 shows the same queries, but without lemmatization.
With lemmatization, the lowest similarity value starts at approximately 0.55, where without lemmatization at 0.6. The average values among all similarities differ slightly, with 0.72 in figure 16 and 0.74 in figure 17. We conclude, without lemmatization we achieve better accuracy percentage and even better similarity values.
Although the differences are not significant, but they drive us to another modification in terms of improvements which will be discussed in the next chapter.

6.2 Retaining redundancy

As we discussed earlier in chapter 6.1, keeping derivative words is beneficial in terms of achieving better accuracy. If keeping similar words contributes to better similarity values, then repetition of the same word also do the same job since two same words have the highest similarity value.
Having the same word twice or more impacts the values of the mean vector more than considering the word only once. This assumption can also have a negative impact. Repeating a word that is out of the actual context drives the mean vector to the wrong side. We can take this risk and test this theory by doing some modifications on the code. Originally, after the tokenization and the removal of unnecessary tokens, we save the generated tokens in a set. Saving the tokens in a set removes any repetitions of a token since every element occurs one time in a set. At this point, we keep the generated tokens in the list without converting it into a set. This is implemented in the `clean_tokens()` method, where we also removed the lemmatization step:

```
def clean_tokens(tokens):
    tokens = [token for token in tokens if token.isalpha()]
    stop_words = set(stopwords.words("english"))
    tokens = [token for token in tokens if token not in stop_words]
    #lemmatizer = WordNetLemmatizer()
    #tokens = [lemmatizer.lemmatize(token, pos="v") for token in tokens]

    return list(set(tokens))
```

We change the return statement to:

```
return list(tokens)
```

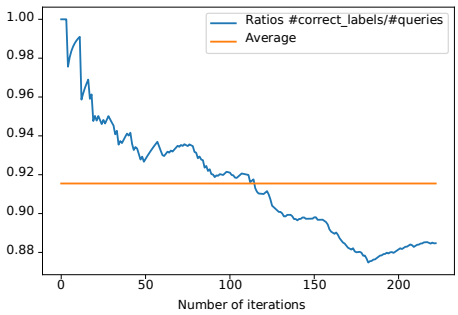

Figure 18: Ratio of correct labels with redundant tokens

In figure 18, we notice a considerable improvement in the accuracy. We achieved approximately 20% more than the original average accuracy in figure 11. Keeping redundant words does not result in an increase in the search function's processing time since we compare only the mean vectors. It may require more storage, but this is manageable because the amount of storage it consumes is not significant.
With the previous modifications, we were able to improve our model and achieve an average accuracy of 91%.

6.3 Better tokenization

Since we extract text from HTML, we do not expect the words to be perfectly separated.

There is a good chance that words could be merged with each other. Because of that, we encounter unexpected tokens such as “restaurantbeach”. This token should be separated into two single tokens, “restaurant”, and “beach”. As explained in chapter 4.3, we only consider tokens which are contained in the trained-set. Tokens like “restaurant-beach” are not contained in the trained-set. These tokens will be ignored and will not be considered in the calculation of the mean vector.
Utilizing a better tokenizer than the one we used from NLTK can be challenging since accomplishing such a task requires bigger models and more resources. A better way to tackle the out-of-vocabulary problem, is to use a modified version of Word2Vec called fastText.
In chapter 3.3, we explained the functioning of Word2Vec and mentioned the two essential variants, CBOW and skip-gram. Both of these variants take words as inputs. FastText operates in the same manner, but instead of taking whole words as inputs, we break down words into subwords or character n-ngrams.
It means, we train the neural network on these character-level n-grams. The value of n is a hyperparameter and can be adjusted to achieve better results. For instance, if we set n=3, the tokens “restaurant” and “beach” would be broken down into “res”, “est”, “sta”, “tau”, “aur”, “ura”, “ran”, “ant”, and “bea”, “eac”, and “ach”.
Instead of using the tokens “restaurant” and “beach” for the input, we take now the previous n-grams and use them as inputs for the neural network. This is beneficial, because if the token “restaurantbeach” appears in a document, it would no longer be considered out-of-vocabulary since we trained the model on the character-level n-grams, and the subwords appearing in the token "restaurantbeach" are part of the vocabulary [Kal, 2021].
To confirm our assumption, we can compare the accuracy of the two models in terms of retrieving the right documents for certain queries. Using the same dataset from BBC news from chapter 5 does not bring us to any conclusion, because the words in the documents are correctly separated. Instead, we collected 140 documents by visiting the most visited Wikipedia pages [Wikipedia, 2023].
The documents are modified and saved using the original `prepare()` function which still lemmatizes the words and removes redundant words. We also generate the queries using TF-IDF like in chapter 5.2.2.
The Wikipedia articles are much larger than the BBC news articles, with approximately 30000 words on average. These large articles make generating precise queries pretty challenging. Therefore, instead of generating queries with only 10 words, we decide to generate queries with 100 words. This is still fair because the BBC news articles have only 2000 words on average and we used 10 or 5 words queries, meanwhile, the Wikipedia articles are about 15 times larger than the BBC news articles. Using 10 or 5 words as a query will lead to very low accuracy values (down to 0.1) since they are not enough to describe the document.
Because controlling the quality of the query’s words is challenging, we increase the chance to catch more meaningful words by increasing the number of chosen words to 100.

For the fastText model we use the pre-trained model "cc.en.300.bin.gz" which is trained on english words and has 300 dimensional embeddings [Mikolov et al., 2017]. The model can be easily loaded using the `fasttext` library in Python.

```
import fasttext
model = fasttext.load_model("cc.en.300.bin")
```

FastText also provides a function (`get_word_vector()`) to get the embedding vector of a word which makes it simpler to calculate the mean vector for the whole document.

```
df["mean_vector"] = df["html"].apply(lambda x:
                                     np.mean(
                                         [model1.get_word_vector(w)
                                          for w in x],
                                         axis=0))
```

Similarly, we compute the mean vectors for the queries.
Figures 19 and 20 show the ratios after executing the search queries with increasing number of queries and documents using the fastText model and the original Word2Vec model.

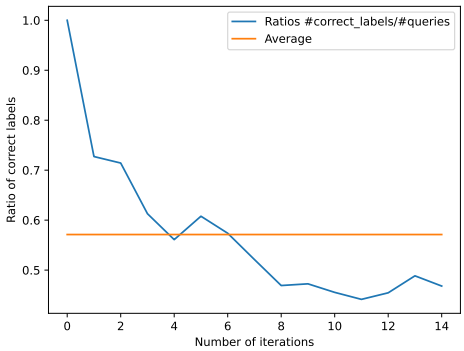

Figure 19: Accuracy values using Word2Vec model

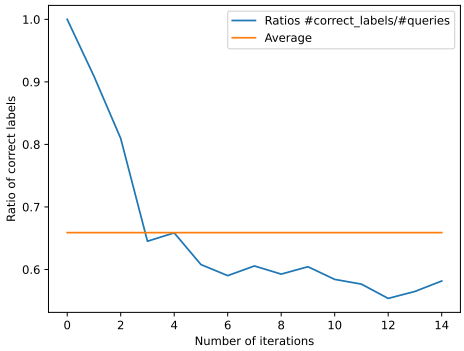

Figure 20: Accuracy values using fastText model

We can see that we achieved a noticeable improvement. The average accuracy went from approximately 0.55 to 0.66.

7 Conclusion

Our initial thoughts were to build a search engine which relies on retrieving documents in a more comprehensive way and combine it with a browser plugin. The purpose was to find the document which has more relevance to the query and try to avoid key-word matching algorithms. Therefore, we first started with the Word2Vec algorithm which provides a unique approach to convert words into valued vectors, embeddings.
By looking at the charts from chapter 5, we can notice that in 70% of the queries we retrieve the true document. Eventually, we were able to drive the accuracy up to 90% by doing some modifications and improvements in chapter 6.
The weakness of this evaluation is that we tested our model on a dataset (BBC news) which is relatively small, and the documents do not contain a significant number of corrupted words which do not require further actions as we experienced in chapter 6.3. In that chapter we also noticed how the accuracy drops down to approximately 0.55 (figure 19) when we tested the original Word2Vec model without the improvements on a real life scenario with the Wikipedia websites. If we compare it with the test in figure 11, we notice a difference of about 15%.
We can confirm the assumption that corrupted tokens have a noticeable impact by testing the fastText model on the BBC news dataset, which has less corruption in the words with the same queries used for figure 11. If the test shows same or lower values than the values we reached in the second test (figure 11), we can suppose that corrupted tokens have some influence.

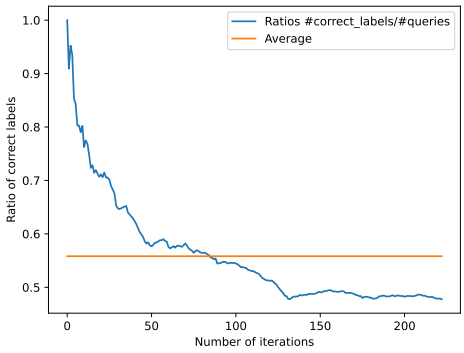

Figure 21: Accuracy values using fastText on BBC news

Figure 21 shows an average value of approximately 0.56, which is much lower than the values we achieved with Word2Vec in figure 11 with 0.71. This confirms that the quality of the words have a significant impact since we achieved better accuracy using the fastText model on a dataset with a significant amount of corrupted tokens.

Consequently, we conclude that it is necessary to test the model on a dataset similar to the Wikipedia dataset but much larger.
Since we do not have a larger amount of websites, which should simulate the real approach of using the search engine, we can not determine the actual performance of the search engine.
The value we achieved in figure 20 could not be accurate since we tested the search engine only on 140 websites, where in reality we would have much more, in thousands.
It is also worth mentioning that in the 140 Wikipedia websites we noticed a proper amount of shared words among the documents. These are the words from the navigation bar, options to change language, option menu, etc. For example, we can see the words "jump to content" in almost all documents. We can try to find words that are common in most of the documents and remove them, but this will need more processing power and resources, and we can not be sure if this modification is effective in terms of reaching more accurate search results.
It also might be beneficial to keep such words since they might give some uniqueness for a website. Also testing this approach was not possible due to the absence of a proper dataset with variety among the websites.
Another thing we could have implemented, is to add a save button to the extension. Only when this button is clicked the website will be transmitted to the server. This feature was pretty challenging to implement due to the privacy restrictions in Chrome browsers. Also, we could add an option to change language or an auto-language-detector. Those remain features and would not affect the performance of the search engine.
There remains a considerable issue. For a fair evaluation of the search engine, we expect queries with higher quality and more precise. We have observed the impact of the quality of the queries in chapter 5.4 when we reduced the number of words in the query to 5 and the average accuracy dropped from 0.7 to approximately 0.4. This is a massive difference with 30% less accuracy.
The issue with the queries complicates the task of evaluating the true performance of the search engine and makes it pretty challenging. A better approach would be to construct queries manually which takes too much time and effort. This approach may be the most effective way to evaluate the search engine, but the absence of a proper dataset with manually created queries, makes applying this approach a bit challenging.

References

Bruno Stecanella (2019). *Understanding TF-ID: A Simple Introduction*. URL: https://monkeylearn.com/blog/what-is-tf-idf/.

Chris McCormick (2016). *Word2Vec Tutorial - The Skip-Gram Model*. URL: http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/.

D. Greene and P. Cunningham (2006). *"Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering", Proc. ICML 2006*. URL: http://mlg.ucd.ie/datasets/bbc.html.

Google Code (2013). *word2vec*. URL: https://code.google.com/archive/p/word2vec/.

Kal (2021). *Word Embedding Using FastText*. URL: https://medium.com/@93Kryptonian/word-embedding-using-fasttext-62beb0209db9.

Kurtis Pykes (2023). *Stemming and Lemmatization in Python*. URL: https://www.datacamp.com/tutorial/stemming-lemmatization-python.

T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, and A. Joulin (2017). "Advances in Pre-Training Distributed Word Representations". In: *arXiv*.

Ria Kulshrestha (2019). *NLP 101: Word2Vec — Skip-gram and CBOW*. URL: https://towardsdatascience.com/nlp-101-word2vec-skip-gram-and-cbow-93512ee24314#:~:text=Continuous%20Bag%20of%20Words%20Model,Languages%20for%20Machine%20Translation%20paper.

Thomas Wood (2023). *Softmax Function*. URL: https://deepai.org/machine-learning-glossary-and-terms/softmax-layer (visited on 2023).

Turing (2023). *A Guide on Word Embeddings in NLP*. URL: https://www.turing.com/kb/guide-on-word-embeddings-in-nlp.

Wikipedia (2023). *Wikipedia:Popular pages*. URL: https://en.wikipedia.org/wiki/Wikipedia:Popular_pages (visited on 2023).