ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Natural Language Processing

Graph-Based Keyword Extraction from Scientific Paper Abstracts using Word Embeddings

Author:
Dinno Koluh

Supervisor:
Prof. Paolo Torroni

Co-Supervisor:
Dr. Federico Ruggeri

Bologna,
December 2023.

Abstract

In the era of information overload it became essential to efficiently extract concise, precise and quality information from large texts. One aspect of information extraction is keyword extraction where large texts are represented as sets of tokens i.e. keywords. This prospect of keyword extraction is paramount to researchers as they deal with huge numbers of scientific papers, and having a good and concise representation of those papers is essential for them. This thesis paper addresses that problem in the realm of natural language processing (NLP).

Using core concepts of NLP and modeling texts as graphs, we are going to build a model for the automatic extraction of keywords. This is done in an unsupervised manner as the importance of a word is calculated through the position and weights associated with respective words in the graph. The first metric used to calculate the graph weights are co-occurrence matrices and the other metric are word embeddings. Word embeddings became a crucial way of representing the semantic information of words as dense vectors.

The results of this paper were compared with keywords that were provided by authors of scientific papers in the area of computer science which act as the ground truth, but crucially are not a component in the model construction, but just serve as a verifier of the model's accuracy.

Keywords: NLP, keyword extraction, scientific papers, graphs, word-embeddings

Contents

List of Figures

List of Tables

1 Introduction and Motivation

Natural Language Processing (NLP) is a subfield or Artificial Intelligence (AI) and linguistics that has a focus on the interaction between computers and human languages. This is mostly restricted to written language as other fields (like Speech Processing) deal with spoken language (using audio instead of textual features). In the past decade NLP has seen huge attention with the introduction of some key concepts like *word-embeddings* [1] (which we are going to use) and *transformers* [2]. At the moment of writing of this paper, NLP is the subfield of AI that has the most resources invested into it, mostly in research and development, with new large language models (LLMs) coming out on a daily basis. Our focus will be a bit shifted from Deep Neural Network (DNN) models and more to traditional Machine Learning (ML) models. NLP has many subfields and application areas such as:

- Text classification
- Information retrieval
- Automatic translation
- Speech analysis
- Question answering
- Conversational agents
- Sentiment analysis

The field we are going to be working on will be **information retrieval**, more precisely *keyword extraction*. Keyword (keyphrase) extraction is the automatic selection of important and topical phrases from the body of a document [3]. Scientific papers are usually the area where keywords are frequently used as researchers use them for a quick overview of papers and also sorting papers into different categories. This is the topic we are going to be working on as well. The keywords are going to be extracted from the abstracts of the scientific papers. One detail we should address that will be important later is the distinction between *keywords* and *keyphrases*.

Keywords would be single words or at most MWEs (Multi-Word expressions, i.e. "deep learning") while keyphrases are more complicated entities comprised of several words and they act as a single unit (e.g. the phrase "scientific paper" would be a keyphrase). When doing keyword extraction we might have as an output a combination of both, keywords and keyphrases. Usually keyphrases carry more information than keywords but a combination of both as an output is most representative. From now on, when referring to *keywords* it will also include *keyphrases*, if not explicitly stated otherwise.

The model to be used for keyword extraction is **graph-based**. The motivation behind using graphs is that graphs are a well-studied and understood concept and many practical problems can be represented with graphs. There are also a plethora of algorithms which are going to be useful for the case of instance ranking.

The idea is to model words in a paper abstract as nodes of a graph. Not all the words in the abstract should be included, as keywords are usually composed of a combination of nouns and adjectives (e.g. scientific[ADJ] paper[N]). This means that some preprocessing of the raw text will be required, especially when dealing with MWEs. We will speak about this in the next chapter.

As stated, the nodes of the graph will be modeled as words, but the question comes on how to model the edges of the graph?

It starts from the assumption that words that occur in the same context tend to carry a similar meaning. The idea is to use a **sliding window** of some predefined size and words that are in that window are connected with and edge. In that way the final graph carries semantic (the meaning of words) information of the input text. The number of occurrences when sliding the window over the entire text will give us the initial graph weights. To solidify this relationship another metric that we will use are word embeddings. Word embeddings are essentially a way of representing words as vectors of numbers. We will dive more deeply into how word embeddings are computed ans used but for now we can assume that word embeddings are vectors of numbers and that these vectors carry semantic information of the corresponding word. This means that words that have a similar meaning (whatever that might mean in some general picture) also have similar vector representations and that we can use the usual mathematical tools on these vectors like the dot product which means that we can actually measure the similarity between words.

We now have the complete representation of the input text as a graph. Now we need to somehow rank the nodes according to the graph edges and weights. We can refer to the ranking procedure as to finding the importance of each node. There are several algorithms which can find the importance of nodes, and we will speak in more detail about them later, but for now can assume we have a black box which takes in the graph representation of the abstract as described before and gives us all the nodes (word) ranked by their importance.

We now have a list of the most important words, but there is one more step that we can do to get a more robust output. The given output would be a list of words, but we might want to have phrases instead of keywords (e.g. the phrase "scientific paper" is more representative instead of just "paper"). We can use the words we got out from the graph and traverse the initial text for phrases in which they appear. Then based on those phrases in the initial text and the importance of the words which they are made of, we can get alongside the ranked words also the phrases in which they appear. In this way we can also generalize the problem by letting a phrase be only made up of one keyword. If it is made up of more than one keyword we can average it out, and in the end get the ranking of the specific phrases which appear in the input text.

This was a rough explanation of the procedures which should give us an overview of the steps involved in developing a pipeline for the extraction of keywords from paper abstracts. In the next few chapters we are going to look in more detail in the inner workings of the steps involved. We will start with the NLP preprocessing pipeline.

2 NLP Pipeline

In this chapter we are going to explain the preprocessing steps needed to transform the raw input text into something that can be modeled with a graph. These preprocessing steps are going to be carried out with concepts from NLP like tokenization, sentence splitting, lemmatization, etc. All these NLP concepts used for preprocessing can be stacked into a pipeline hence the name of the chapter, **NLP pipeline**.

We will explain these concepts giving examples and at the end show the whole pipeline as a schema. In the end we will have single instances of processed words which are going to be the graph nodes. Let us start!

2.1 Tokenization

Tokenization is the basic and most important NLP concept. Let us firstly define what a token is. A **token** is a sequence of characters grouped together as a semantic unit used for text processing [4]. Depending on the use case, tokens can be equivalent to words, but they don't have to. They can even be single characters but that depends on the tokenizer model used. An example of words and tokens not being equivalent is:

I'd → [I, 'd]

In this case the word "I'd" was broken into two tokens. Depending on the specific tokenization method used, words can be broken into lower units (subwords):

unlucky → [un, lucky]

Here, we have that the adjective "unlucky" was broken into its prefix "un" and the adjective "lucky". In this example tokenization provided a way to work with text on a granular level, breaking down words into units which might not even be real words, but in this way we are making text processing tasks more cohesive for machines.

The process of **tokenization** would be the process of breaking down an input text into single tokens. As the input text inherently has whitespaces and newlines, those are usually excluded from the obtained tokens. We can look at an example sentence broken down into tokens:

She didn’t have time. → [She, did, n’t, have, time, .]

When dealing with whitespaces, tokenization is a trivial process, but when dealing with punctuations, tokenization becomes a hard problem. The reason is that sometimes punctuations should be kept, but sometimes they should be separated. In the example above, the token “time.” was separated into “time” and “.”, but let’s take a look at this example:

She didn’t have time for Mr. Nobody. → [She, did, n’t, have, time, for, Mr., Nobody, .]

In this example the period for the token “Mr.” was not separated while for “Nobody.” it was. This is the reason why tokenization can be ambiguous and hard to compute correctly. And there are tens of other example of ambiguities apart from punctuation marks. One solution to the problem we’ve seen is to keep a **lexicon** of possible ambiguous tokens so we know how to separate them.

2.2 Sentence splitting

Sentence splitting is the process of splitting up text into sentences. This steps is going to be important when computing co-occurrence matrices with the sliding window. The reason is that when using the sliding window we don’t want to capture the end of one sentence and the beginning of the next one in the same context as they might not carry the same meaning as sentences are naturally boundaries that also we, humans, respect and take into account when reading and processing text.

Sentence splitting has inherently similar problems as tokenization does as punctuation signs can be ambiguous depending on the position and the tokens surrounding the punctuation sign. Apart from punctuation signs, sentences in quotation marks inside sentence cause problems as such sentences are basically nested sentences. But for our purposes we don’t expect to have many such examples in scientific paper abstracts.

2.3 MWE

MWE or Multi-Word Expressions are sequences of words that when grouped together carry one meaning, but when standing alone carry another. An example of an MWE is “deep learning”. We see that the word “deep” and “learning” on their own carry other meanings in contrast when putting them together into a single phrase. For the purposes of this paper we need to keep these expressions together and not split them. The reason is that we are extracting keywords and keyphrases, and the keyphrase to be extracted from a paper that is written on the topic of “deep learning” is “deep learning”, not “deep” and “learning”.

When dealing with scientific papers, we expect that they are going to have many MWEs and possibly new, non-existent MWEs. This makes the tokenization problem ever more harder because these tokens should be kept together. One solution is to again keep an MWE lexicon. Another solution, which was observed while inspecting the dataset is that novel MWEs are usually written with upper-case letters and they have an abbreviation after the MWE in parenthesis (e.g. “Introduction to Large Language Models (LLMs)”) and this fact can be leveraged when dealing with MWE that are not present in the MWE lexicon.

Let us see an example for MWE tokenization:

We are discussing Machine Learning (ML) models in San Francisco. →
[We, are, discussing, Machine Learning, (, ML,), models, in, San Francisco, .]

2.4 Token removal based on PoS tags

This step is specific for our task, but PoS (Part of Speech) is a regular NLP concept. **Part of Speech** tagging is the process of giving a grammatical category tag to words based on their syntactic and semantic meaning. Basically this step assigns syntactic tags to words i.e. telling if a word is a noun, verb, adjective, etc. The question is why do we need this step?

The reason is the nature of the task we are doing. Namely, keywords are usually a phrase of an adjective and verb or at most a single noun. Taking this into consideration we can substantially reduce the graph size if we just keep nouns and adjective as the possible graph nodes. There are several ways on getting PoS tags of words, like rule-based methods, statistical methods, hybrid methods, deep learning methods, etc. We are not going into the inner workings of these methods, but we will just use an existing model for getting PoS tags. One of the problems when computing PoS tags is they highly depend on the context in which they occur as we can have words which are *homographs* (they are spelled the same but have a different meaning) but carry different PoS tags. For example the word "developed" can be a verb and an adjective at the same time as show below:

Japan[Noun] developed[Verb] into[Prep.] a[Det.] developed[Adj.] nation[Noun].

This is the reason why context is very important when assigning PoS tags. Deep learning and statistical methods are usually good at these context specific examples but they require quite large training datasets.
An example of this step can be seen below:

Japan developed into a developed nation. → [Japan, developed, nation, .]

2.5 Stopword removal

Stopwords are common words occurring in a language that are usually filtered out during NLP tasks as they do not carry much value for the specific task. For our purposes stopword removal is essential as we are going to have many of them, and in the final ranking they will not be present in the keyphrases. The usual method of stopword removal is to build a stopword lexicon and just remove those words from the input text. An example of stopwords are: "the", "and", "in", etc. Let us see an example for stopword removal on a sentence:

The quick brown fox jumped over the lazy dog. → [quick, brown, fox, jumped, lazy, dog, .]

This step will come after the token removal based on PoS tags, as stopwords could carry context specific information for PoS tagging. Even though PoS tagging should take care of stopwords, we will still include this step in case some stopword slip thorough the previous step, as stopword removal is a cheap operation.

2.6 Lemmatization

Lemmatization is an NLP technique used to reduce words to their base form, also known as their *root* or *lemma*. This means that words that might be morphologically different have the same root and we want to transfer those words to their root form. Some examples of lemmatization is show below:

am, are, is → be
horses, horse, horses', horse's → horse
develops, developed, developing → develop

For lemmatization to be accurate, it usually uses PoS as context is important to determine the grammatical role of a word in a sentence. This step is important as the lemma of a word may vary depending on its PoS. Lemmatization will usually transform verbs, adjectives and adverbs to their base form, but if a noun is present among a set of words (e.g. "developer" in the last example) it will keep the noun how it is as nouns are usually lemmas themselves (but it will of course strip possessive or plural forms). With the use of PoS tags and grammatical rules and dictionaries, words are mapped to their lemmas.

One similar concept to lemmatization is stemming, where in the stemming process words are stripped from their suffixes and prefixes using a predefined set of rules. The problem with stemming is that it might produce non-existent words (e.g. "replacement" → "replac") which is not practical for our purposes.

The reason we need lemmatization is that words that might have different forms have to be treated as the same word as in the other case we would have two different nodes in the graph that actually represent the same word. Let us see an example of this:

Graph models were used for modeling words. → Graph model be use for model word.

2.7 Normalization

Normalization is the process of putting words with different forms into a single normalized form. A good example of this step is the lowercasing of all words in a sentence, otherwise we

will have words which are the same but they are essentially represented as two different words. We could also look at lemmatization and stemming as parts of normalization as they also map words with different forms into a single one, but we will keep them as two separate steps.
One essential normalization step will be normalizing MWE expressions into single forms. We can look at the example below:

U.S.A., US, United States, United States of America, USA → USA

All of the words on the left essentially mean the same thing, so we can represent it by one word. We will encounter a great deal of MWE expressions in scientific papers that use acronyms and their long forms, so it is necessary to represent them as a single word.
In the normalization process we will also include the removal of punctuation signs and other non-alphanumeric sign (e.g. #, (,), ?), and the expansion of clitics into their expanded form (e.g. I'd → I would).

2.8 The pipeline

Let us now combine all the steps of the NLP preprocessing pipeline into a single schema in Figure 1 for better readability.

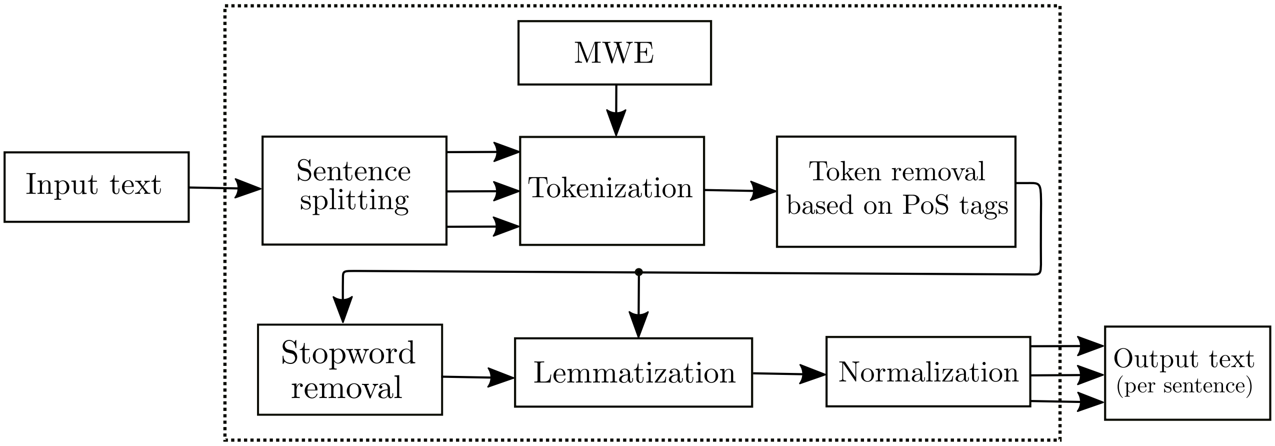


Figure 1: The NLP pipeline

Let us just address briefly some possible points in the figure. Since sentence splitting produces sentences from an input text, the three arrows represent multiple sentences going into the tokenization block. As already said, it is important to know the sentence boundaries when computing the co-occurrence matrix.

The tokenization block takes as an input the MWE block since it uses the MWE lexicon and other MWE specific features that we mentioned (e.g. abbreviations after the MWE) to correctly tokenize the input.

The lemmatization block which takes in two inputs, the flow of tokens from the previous block and, to function properly, it also needs the PoS tags from the token removal block.

The last point to mention is that after the normalization block, the output is not a single stream of tokens, but rather a list of lists of tokens, where each list represents one sentence that the sentence splitting block separated. This figure will be expanded once we talk about other steps required to build the whole model.

3 Word Embeddings

3.1 How to Represent Words as Numbers?

The big challenge of any text processing on a computer is how to represent words as numbers? This is a fundamental question in NLP as computers inherently work with numbers and NLP works with words. In contrast to other areas of computer science like image processing and audio processing whose working medium (images and waveforms) can be and are represented with numbers, there is no apparent way on how to represent the working medium of NLP (words) as numbers.

Of course, the naive approach would be to map each word in a language to a single number, but, for example, the word "banana" being number 134 and the word "orange" being number 54 doesn't tell us anything about those words or the relation between them. The thing is that a single number cannot meaningfully represent the essence of a word among thousands of other words. The idea is to somehow represent the features of words. For example features of "orange" are that it is a fruit, round, orange colored, etc. In contrast a "banana" is also a fruit but it is curved and yellow. So in that sense, we could encode the features of words as numbers, and measure, for instance, how much is an object round or yellow and so on. The next natural step is to encode those word features as *feature vectors* and this is the point where we come to word embeddings.

Word embeddings are representations of words as real-valued dense vectors such that words which are closer in such a vector space tend to have a similar meaning [1]. Word embeddings have revolutionized the field of NLP by enabling computers to interpret and understand words. Each entry of the embedding represents a feature of the word. These vectors are in high-dimension spaces as there are tens and even hundreds of features that words have. The higher dimension the better representation of words we have, but there is the computational trade-off, as longer vectors required more resources to process and store. Let us look at an example of a 2D subspace of a larger, high-dimensional space. We can see how words in such a space are behaving in terms of their embeddings in Figure 2.

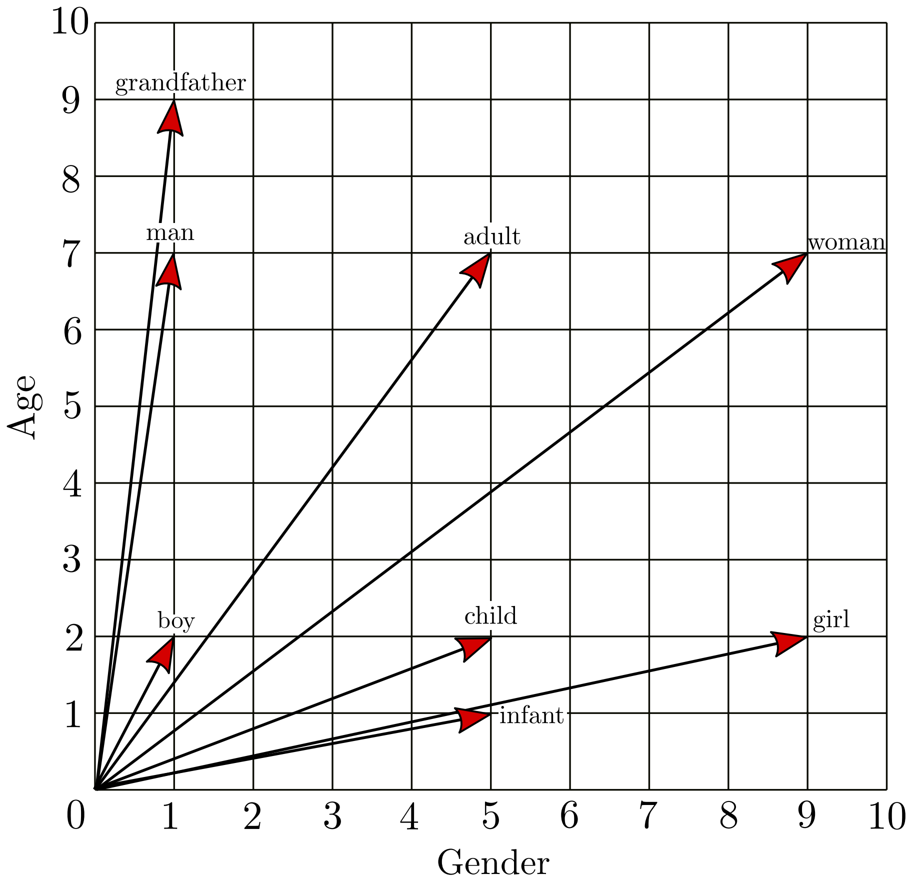


Figure 2: Word embeddings in the Gender-Age subspace

The numbers on the Gender and Age axes are just for presentation purpose as word embeddings are usually real-valued which means they are also continuous. The reason for such a representation instead of an integer or one-hot representation is that continuous and dense vectors capture more accurate and subtle semantic relations between words. The other reason is that real-valued embeddings are more efficient for computation than other sparse representations. The vector elements are also usually in the range $[-1, 1]$ for normalization purposes when using them in neural networks.

As for the example in Figure 2 we can see that for the Gender axis the lower values are regarded as more male (grandfather, man, boy), the higher values are regarded as more female (woman, girl) and the values in between have no specific gender (adult, child, infant). As for the Age axis, we have the same idea, based on the age of person the words have higher or lower values. We can see the power of embeddings, as the vector representation of a word does really carry meaning in such a space. And if a new word would be mapped in this space, e.g. "lady" to the vector $(8, 6)$ based on how much the word is "old" or "young" and "male" or "female". We also see that words that are more similar are closer to each other in this space, and they form **clusters**. This is just a $2D$ example of a space, but a space with, say 50 dimensions, will have many more refined features and we can form more refined clusters (e.g. cluster of fruits which are yellow). In Figure 3 we can see an example of different word clusters. One of the ways of obtaining these clusters is using the K-means clustering algorithm (in fact, it is the perfect example of using K-means).

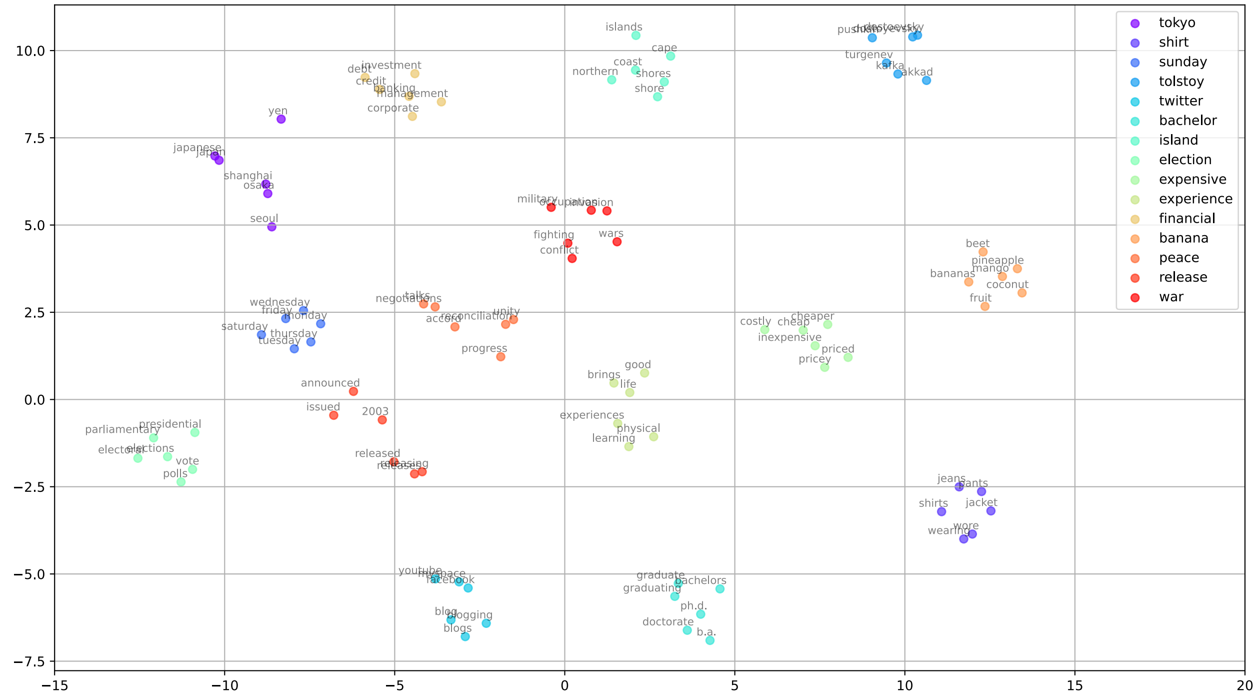


Figure 3: Clusters of words using word embeddings

The real power of these vectors is that we can perform standard vector operations on them, like the addition and the cosine similarity. This enables us to numerically measure the similarity between words with the cosine similarity like:

$$S_C(A,B) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} \tag{3.1}$$

We can test this on the figure with the similarity between "grandfather" and "man" versus "grandfather" and "girl":

$$S_C(\text{grandfather}, \text{man}) = \frac{(1,9)\cdot(1,2)}{\|(1,9)\|\|(1,2)\|} = 0.94$$
$$S_C(\text{grandfather}, \text{girl}) = \frac{(1,9)\cdot(9,2)}{\|(1,9)\|\|(9,2)\|} = 0.32$$

We observe the expected, that "grandfather" and "man" are more similar (both male and older) than "grandfather" and "girl" (different genders and ages). The other operation we mentioned was vector addition. Adding and subtracting word embeddings means that we basically add and subtract the features associated with words. Let us show that on an example (where WE means the word embedding of a word):

$$\text{WE}(\text{woman}) - \text{WE}(\text{man}) + \text{WE}(\text{boy}) = (9,7) - (1,7) + (1,2) = (9,2) = \text{WE}(\text{girl})$$

The intuition behind the addition and subtraction for this example is that we subtracted the age from "woman" with the age of "man" (which gives 0 age), and gender from "boy" with the gender of "man" (which gives 0 gender) and added the age from "boy" and gender from "woman" which gives us something that has the age of a boy and the gender of a woman. That something is a girl, which is exactly what we got. Doing this in higher dimensions we can get the closest cluster to the resulting vector and get a set of words which are similar to the desired output.

3.2 How are Word Embeddings Computed?

We have seen what word embeddings are, their functionalities and how to use them, but how do we actually compute them, in an automated way?

To core concept of obtaining word embeddings is that similar words occur in the same context. This means that words that are close together in a text also tend to be similar to each other. This is also known as the **distributional hypothesis**. This is a strong hypothesis but it has been proven to be viable. The most successful models for calculating word embeddings are neural network based using the context assumption. The neural networks are usually shallow as the network weights are used as the embeddings. The idea is to use a **skip-gram** model, which takes a target word and tries to predict the surrounding context words. A sliding window is used for the size of the context. Then the target word is used as the input to the neural network (actually the one hot encoded sliding window) and the preceding and succeeding words in the context are predicted as the output (which we know). The network is then trained to minimize the difference between the predicted word and actual word in the context. The network weights are adjusted during the training to improve the accuracy of the predictions using the backpropagation algorithm. The resulting weights of the hidden layer are used as the word embeddings as they capture the semantic relationships between words. This skip-gram architecture is used in Google's **word2vec** word embeddings toolkit [5] developed in 2013.

The other way of getting the embeddings is using global statistical information of the entire corpus constructing a co-occurrence matrix. That matrix represents the frequency of word pairs occurring together in a given context window. This is exactly the way we will get the pairwise connections of nodes in the graph we will build. The obtained matrix is then factorized to the final embeddings (the process has some more details but we won't go into detail). This co-occurrence matrix architecture is used in the **GloVe** model [6].

When the word embeddings are obtained, they are usually saved in a data object similar to a dictionary where the key is the word and the value is the word embedding. This is an efficient way of getting the embedding values for a specified word. One can train their own model on their own data, but there are many pre-trained models, and the embeddings can be used directly.

Some apparent problems using word embeddings as they were described are:

1. How to embed phrases (or MWE)?

   When training the network on target words, we can define *target phrases* instead of words. There is actually a model called phrase2vec exclusively build for phrase embedding. There are also models which can embed sentences as vectors, so we can treat sentences as phrases.

2. How to deal with homonymy (words with different meaning but written the same)?
   To overcome this problem, *multi-sense* embeddings can be used, where a word can be mapped to more embeddings considering a word's context [7].

Let us also explain how are we going to use word embeddings in the graph model. Once the graph is built, with words as nodes and the edges as connections between the words (gotten from the co-occurrence matrix which will be explained in the next chapter), we can alter the weights of the edges by multiplying them with the cosine similarity of the embeddings between two nodes. In this way we aim to strengthen (or loosen) the semantic link between words.

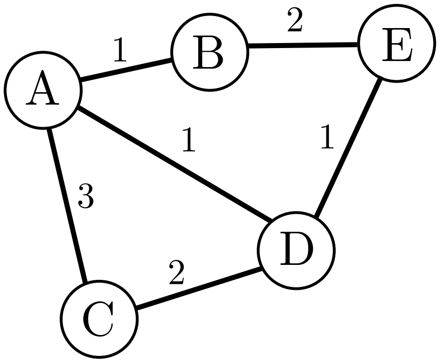
4 Graph Construction

In this section we will dive into the construction of the graph model. A graph $\mathcal{G}$ is defined as an ordered pair:

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}) \tag{4.1}$$

where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges, the links between adjacent nodes [8]. We will model the text to be analyzed as a graph. The set of nodes are going to be the tokenized words. The more important question is what are going to be the edges? Well, we need to somehow encode the links (semantic relation) between words. We are going to use the same hypothesis word embeddings use, that words that occur in similar contexts tend to have similar meanings. If words occur in the same context, we will link them with an edge, and in such a way we will encode the links between words. An important note is that the graph is going to be *undirected* which means that the edges are bidirectional i.e. they don't have an orientation. In this way we are not preserving the word order, as we will not need it, but for some other tasks it could be useful to have it.

There are different data structure representations of graphs, e.g. as an *adjacency list*, *adjacency matrix*, etc. In the Figure 4 we can see a graphical and matrix representation of a weighted graph. Each entry in the matrix corresponds to an edge between two nodes and its respective weight (weight of 0 means the nodes are not connected).



$$\begin{array}{c c} & \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} & \begin{bmatrix} 0 & 1 & 3 & 1 & 0 \\ 1 & 0 & 0 & 0 & 2 \\ 3 & 0 & 0 & 2 & 0 \\ 1 & 0 & 2 & 0 & 1 \\ 0 & 2 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Figure 4: Example of weighted graph and its adjacency matrix

We are going to use an adjacency matrix representation for visualization purposes as it is a good way to translate between graphs and text but in the actual implementation we will use an adjacency list as adjacency matrices are usually sparse. This is especially true as we will express the *distributional hypothesis* as a co-occurrence matrix which we will explain in the next subsection.

4.1 Co-occurrence matrix

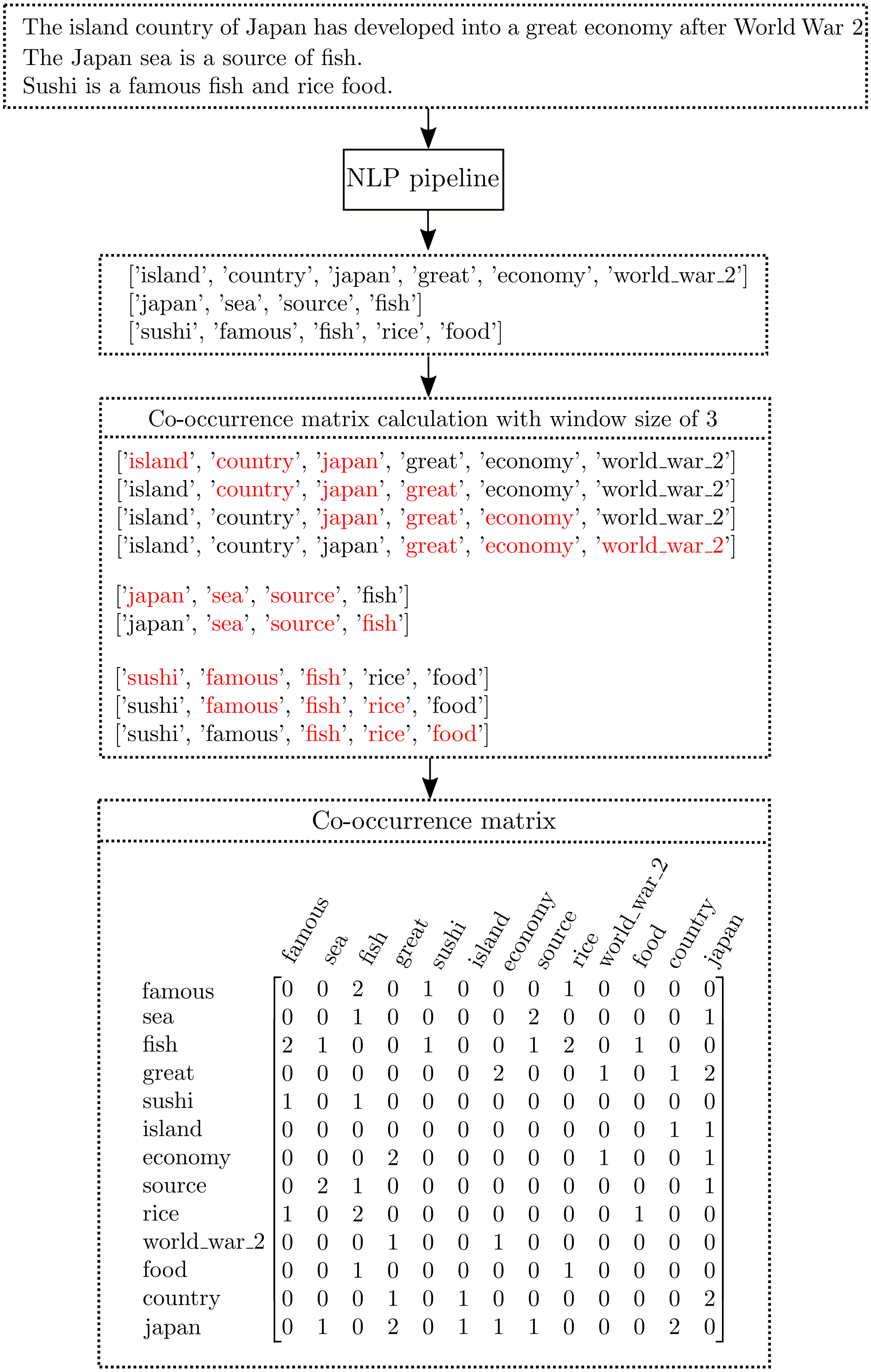
A **co-occurrence matrix** in NLP is a matrix that represents the frequency of word co-occurrences within a specified context window for a given text. The idea is to capture how often words appear together in similar contexts. For a given context window size we see how many pair-wise combinations there are. Each row and column of the matrix corresponds to a unique word in the vocabulary [1] and the matrix elements contain the count of pair-wise word occurrences.

One thing to address, is that will respect sentence boundaries, so that the context window cannot be present across two or more sentences but just one. The reason is that adjacent sentences might not refer to the same context so it is better to analyze them independently. In Figure 1 the output text is assumed to be a set of tokenized sentences, which means that for each sentence after the sentence splitting module we perform all the steps and get all the tokens per sentence.

Let us look at an example set of sentences, for which we are going to perform the NLP pipeline, perform the context window sliding with a window size of 3 words and show the co-occurrence matrix.

We can see in the diagram 5 that we have an input of three sentences which go through the NLP pipeline and we obtain the tokenized sentences (which went also through all the other steps of the NLP pipeline). The next step is the co-occurrence calculation. The sliding window is visualized as the context words are marked red and the window is sliding until it reaches the end of the sentence after which it goes to the next sentence. The last step is the aggregation of the number of pair-wise co-occurrences into the co-occurrence matrix.

[1] Set of unique words in a corpus also knows as *types*



The island country of Japan has developed into a great economy after World War 2.
The Japan sea is a source of fish.
Sushi is a famous fish and rice food.

['island', 'country', 'japan', 'great', 'economy', 'world_war_2']
['japan', 'sea', 'source', 'fish']
['sushi', 'famous', 'fish', 'rice', 'food']

Co-occurrence matrix calculation with window size of 3

['island', 'country', 'japan', 'great', 'economy', 'world_war_2']
['island', 'country', 'japan', 'great', 'economy', 'world_war_2']
['island', 'country', 'japan', 'great', 'economy', 'world_war_2']
['island', 'country', 'japan', 'great', 'economy', 'world_war_2']

['japan', 'sea', 'source', 'fish']
['japan', 'sea', 'source', 'fish']

['sushi', 'famous', 'fish', 'rice', 'food']
['sushi', 'famous', 'fish', 'rice', 'food']
['sushi', 'famous', 'fish', 'rice', 'food']

Co-occurrence matrix

$$\begin{array}{l|ccccccccccccc}
 & \text{famous} & \text{sea} & \text{fish} & \text{great} & \text{sushi} & \text{island} & \text{economy} & \text{source} & \text{rice} & \text{world\_war\_2} & \text{food} & \text{country} & \text{japan} \\
\text{famous} & 0 & 0 & 2 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\text{sea} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 1 \\
\text{fish} & 2 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 2 & 0 & 1 & 0 & 0 \\
\text{great} & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 1 & 2 \\
\text{sushi} & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{island} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\text{economy} & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
\text{source} & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\text{rice} & 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\text{world\_war\_2} & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{food} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\text{country} & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
\text{japan} & 0 & 1 & 0 & 2 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 2 & 0
\end{array}$$

Figure 5: Co-occurrence matrix construction

The last step left to do it to transforms the co-occurrence matrix to a graph. The co-occurrence matrix is in a one-to-one relationship to a graph's adjacency matrix so we can get a graph representation directly from the co-occurrence matrix. One thing to note is that the co-occurrence matrix in the Figure 5 is sparse, so for memory purposes, we actually store the adjacency matrix as a dictionary (a version of the adjacency list) where the keys are the pair-wise word occurrences and the values are the number of co-occurrences (the graph weights). In Figure 6 we can see the graph representation of the example above.

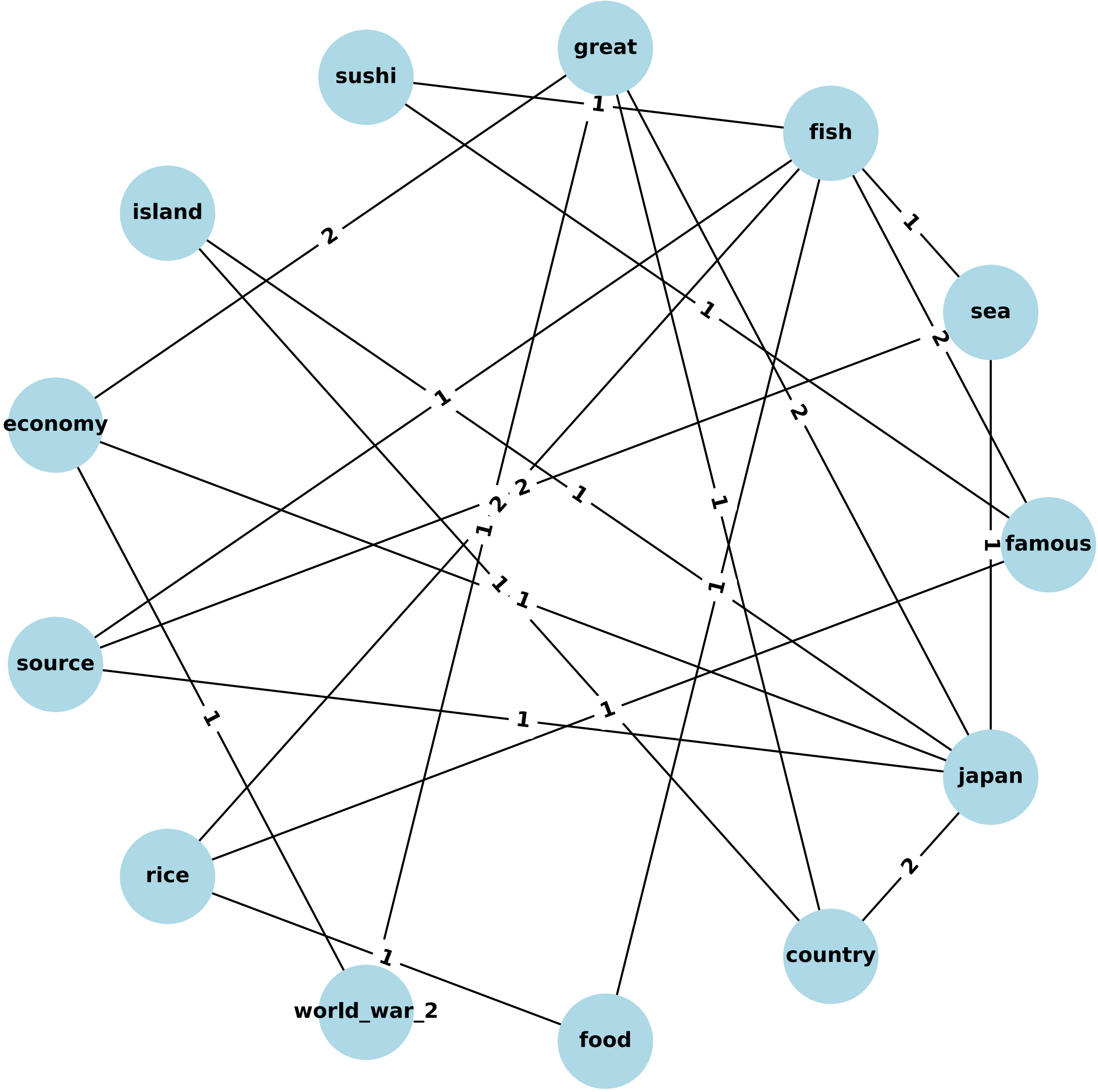


Figure 6: Graph representation of the co-occurrence matrix

Just by looking at the graph we can observe that "japan" has the most edge connections. We can also see the semantic relationship between words and their respective strength like (japan, country, 2), (sushi, fish, 1), (rice, food, 1), etc.

The last step in the graph construction is to encode the word-embedding values in the graph. It is enough to find the cosine-similarity between adjacent nodes and multiply the edge weight with the value of the similarity between words. We can now visualize the whole keyword extraction pipeline.

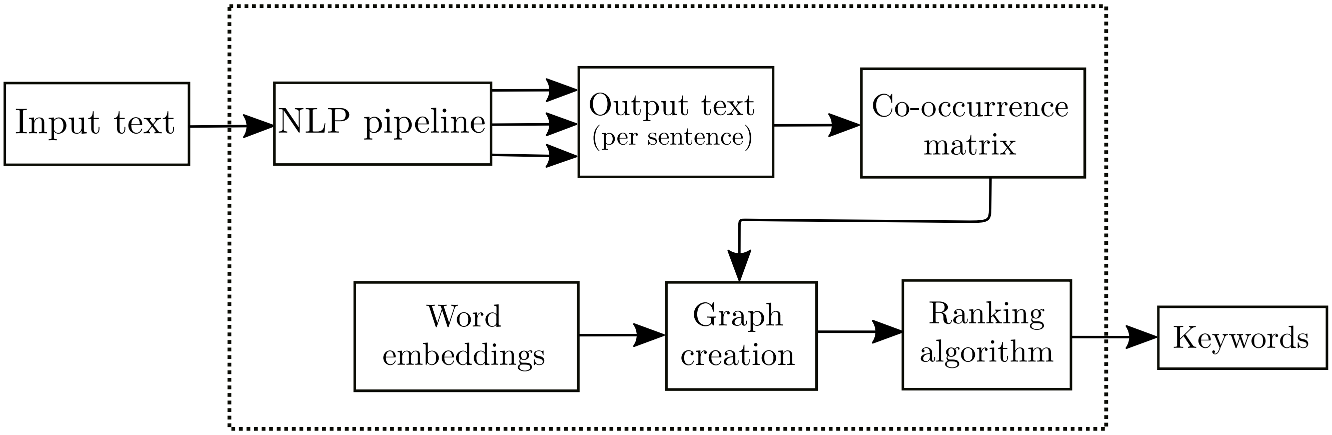


Figure 7: Full pipeline of keyword extraction

We can see that the last block before obtaining the keywords is the *ranking algorithm*. In the next chapter we will discuss how to rank nodes in a graph based on different graph features, whose output are going to be the keywords.

5 Implementation of Keyword Extraction from Graphs

The last, and frankly, most important step is the actual ranking of the words in the graph. Based on this ranking we will get the most important nodes i.e. keywords, based on some metrics. There are many different ranking algorithms to rank graph nodes, which take into account node connections, edge number, edge weights, etc. We will take a look at some of them and in the next section we will carry out tests so we can compare how they perform. Let us now explore these ranking algorithms.

5.1 Degree centrality

Degree centrality is a measure that quantifies the importance of nodes in a graph based on the number of edges connected to them [9]. The number of edges connected to a node in a graph is called it's *degree*. The degree of a node can also be thought of as the number of non-zero values in the row or column of the given node in the adjacency matrix. The formula for degree centrality is given as:

$$C_d(v) = \frac{\deg(v)}{|\mathcal{V}| - 1} \tag{5.1}$$

We divide the node degree by the number of nodes minus 1 to normalize the degree centrality, so that all the values are in the range $[0, -1]$. In the Figure 6 we can check the degree centralities of a few different nodes like

$$C_d(\text{japan}) = \frac{\deg(\text{japan})}{|\mathcal{V}| - 1} = \frac{6}{13 - 1} = 0.50$$

$$C_d(\text{sea}) = \frac{\deg(\text{sea})}{|\mathcal{V}| - 1} = \frac{3}{13 - 1} = 0.25$$

We can see that the edge weights are not considered in this ranking algorithm. From the standpoint of text, words with high degree centrality would be the ones which are frequently present in the context windows of other words. However this might not capture the full semantic meaning as we are not looking at the weights of the edges. Nevertheless, this algorithm gives us still a simple but effective way to rank words in a text.

5.2 Closeness centrality

Closeness centrality is a measure that quantifies how close a node is to other nodes in a graph [10]. This notion of closeness is the defined as the inverse of the sum of the shortest distances from a node to all the other nodes in the graph. It usually contains the factor of $|\mathcal{V}| - 1$ to normalize the formula, so it has the following form:

$$C_c(v) = \frac{|\mathcal{V}| - 1}{\sum_{u=1}^{|\mathcal{V}|} d(u, v)} \tag{5.2}$$

where $d(u, v)$ is the shortest distance between nodes u and v. When talking about the distance between two adjacent nodes, that distance is the edge weight. So, the shortest distance between any two nodes is the set of edges which connect the two nodes such that the sum of their weights is minimal. The most famous algorithm used for this task is Dijkstra's algorithm. In the case that the graph in not completely connected, the closeness centrality would be computed for each connected part seperately and then scaled by the size of that part.

In Figure 4 the shortest distance between nodes A and E is 2 along the path $A-D-E$ even though from the figure it might look like it is the path $A - B - E$ but that path has a distance of 3. We can calculate the closeness centrality for "japan" in Figure 6:

$$\begin{aligned}
C_c(\text{japan}) &= \frac{|\mathcal{V}| - 1}{d(\text{source}, \text{japan}) + \cdots + d(\text{sushi}, \text{japan}) + \cdots + d(\text{world\_war\_2}, \text{japan})} \\
&= \frac{12}{|\text{source, japan}| + \cdots |\text{sush, fish, sea, japan}| + \cdots + |\text{world\_war\_2, economy, japan}|} \\
&= \frac{12}{1 + \cdots + 3 + \cdots + 2} \\
&= \frac{12}{26}
\end{aligned}$$

In this manner the closeness can be calculated for all the other edges. The meaning of the obtained closeness score is that higher scores are assigned to nodes that are, on average, closer to all the other nodes in the graph. An intuitive example is when cities and the roads between them are modeled as graphs. Then, the cities which are well connected to other cities and located centrally in a country have higher closeness scores (e.g. Madrid vs. Barcelona in Spain).

When it comes to text, words with high closeness scores are on average closer to other words which means that they are central to the concept and the overall flow of information in text. Such words are potentially important for understanding the core meaning or theme of the text, as they are well-connected to other words and might play a crucial role in conveying the context. They could also act as bridges between different parts of the text (e.g. in example Figure 5, Japan connects the first two sentences and fish connects the second and third sentence).

5.3 Betweenness centrality

Betweenness centrality is a measure of node's centrality based on its part in facilitating the flow of information between other nodes. The flow of information in a graph is governed by the shortest paths between respective nodes. The betweenness centrality for each node is the number of these shortest paths that pass through that node. This is defined mathematically as:

$$C_B(v) = \sum_{s,t \in \mathcal{V}} \frac{\sigma(s,t|v)}{\sigma(s,t)} \tag{5.3}$$

where $\sigma(s,t)$ is the number of shortest paths between nodes s and t, and $\sigma(s,t|v)$ is the number of those paths that pass through node v. If $v \in s,t$ then $\sigma(s,t|v) = 0$ [11]. This expression is usually normalized by multiplying it by the factor $\frac{2}{(|\mathcal{V}|-1)(|\mathcal{V}|-2)}$. One thing to note is that, the shortest path between two nodes is not unique, there might be more than one path that has the minimum distance between two nodes (e.g. in Figure 4 the distances of paths $A-C$ and $A-D-C$ between nodes A and C are the same).

Betweenness centrality quantifies the number of times a node serves as a bridge along the shortest paths between pairs of other nodes in the graph. Nodes with high betweenness centrality play a crucial role in maintaining efficient flow and connectivity within a graph. Calculating the betweenness score can be quite tiresome when doing it by hand for larger graphs, so instead of the example in Figure 6 we will show how the example in Figure 4. Let us firstly list all the pairs of nodes that have more than one shortest paths:

A to C: $A-C$, $A-D-C$

A to E: $A - D - E$

B to C: $B - A - C, B - A - D - C$

B to D: $B - A - D$

C to E: $C - D - E$

We see that only nodes A and D appear as a part of the shortest paths so all the other nodes have 0 betweenness centrality. Let us see the scores for the nodes A and D:

$$\begin{aligned}C_B(A) &= \sum_{s,t\in\mathcal{V}} \frac{\sigma(s,t|A)}{\sigma(s,t)} = \frac{\sigma(A,C|A)}{\sigma(A,C)} + \frac{\sigma(A,E|A)}{\sigma(A,E)} + \frac{\sigma(B,C|A)}{\sigma(B,C)} + \frac{\sigma(B,D|A)}{\sigma(B,D)} + \frac{\sigma(C,E|A)}{\sigma(C,E)} \\ &= \frac{0}{2} + \frac{0}{1} + \frac{2}{2} + \frac{1}{1} + \frac{0}{1} = 2 \\ C_B(D) &= \sum_{s,t\in\mathcal{V}} \frac{\sigma(s,t|D)}{\sigma(s,t)} = \frac{\sigma(A,C|D)}{\sigma(A,C)} + \frac{\sigma(A,E|D)}{\sigma(A,E)} + \frac{\sigma(B,C|D)}{\sigma(B,C)} + \frac{\sigma(B,D|D)}{\sigma(B,D)} + \frac{\sigma(C,E|D)}{\sigma(C,E)} \\ &= \frac{1}{2} + \frac{1}{1} + \frac{1}{2} + \frac{0}{1} + \frac{1}{1} = 3\end{aligned}$$

And normalizing the results we get $C_B(A) = 1/3$ and $C_B(D) = 1/2$.

When it comes to text, words with high betweenness scores act as “bridges” or “connectors” between different parts of text, as they might appear in position which link distinct text concepts. On the other hand, words with low scores may be isolated or play a more local role on a sentence or phrase level rather than a text level.

5.4 Eigenvector centrality

Eigenvector centrality is a measure of the importance of a node in a graph, taking into account both its direct connections to other nodes, and the connections of its neighbors. A node has a high eigenvector centrality score if it is connected to other nodes that have a high score. The eigenvector score of a node *i* is defined as the sum of the scores of its neighbors multiplied by the edge weights and scaled by a factor λ. Let $\mathbf{A}$ be the adjacency matrix of a graph $\mathcal{G}$, where the entry $a_{v,t}$ of $\mathbf{A}$ is the edge weight between nodes v and t then the centrality score x_v of node v is defined as:

$$x_v = \frac{1}{\lambda} \sum_{t\in\mathcal{V}} a_{v,t} x_t \tag{5.4}$$

doing this for all the nodes in the graph, the equation can be rewritten in matrix notation as:

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{x} \tag{5.5}$$

which the definition of an eigenvector, hence the name eigenvector centrality. The scaling factor λ is the eigenvalue assigned to the eigenvector. The number of eigenvalues for the adjacency matrix $\mathbf{A}$ is going to be the number of nodes $|\mathcal{V}|$, so which one do we choose?

The centrality score of the *i*-th node is obtained as the *i*-th component of the eigenvector $\mathbf{x}$ with the highest eigenvalue λ (also called the *dominant eigenvector*). This condition of using the dominant eigenvector for the scores comes from the fact that the scores have to be positive. [2] The eigenvector is defined up to the eigenvalue λ, so only the ratios of the centralities of the nodes are well defined. We can go a step further to make sure that the scores are in the range $[0, 1]$ by using the unit vector $\mathbf{x}/||\mathbf{x}||$. Let us demonstrate this ranking algorithm on the graph in Figure 4 where we get that the dominant eigenvector is:

$$\mathbf{x} = \begin{bmatrix} 2.738 & 1.087 & 2.887 & 2.183 & 1 \end{bmatrix}^T, \text{with eigenvalue of } \lambda = 4.358$$

and scaling $\mathbf{x}$ with $||\mathbf{x}||$ we get the scores as:

$$\frac{\mathbf{x}}{||\mathbf{x}||} = \begin{bmatrix} 0.574 & 0.228 & 0.605 & 0.457 & 0.209 \end{bmatrix}^T$$

We can check if the scores of the neighboring nodes add up as stated in equation 5.4 for the node C as:

$$x_C = \frac{1}{\lambda}(a_{C,A}x_A + a_{C,D}x_D) = \frac{3 \cdot 0.574 + 2 \cdot 0.457}{4.358} = 0.605 = x_C$$

From the graph in Figure 4 we can see that even though the node C is not central it has a high score as it is connected to nodes which have a high score and are central.

[2] By the Perron-Frobenius theorem a real square matrix with positive entries has a unique eigenvalue of largest magnitude and the eigenvalue is real. The corresponding eigenvector can be chosen to have strictly positive components.

The usual way to calculate eigenvalues are using the *characteristic polynomial* as $\det(A - \lambda I) = 0$ but we do not need all the eigenvalues, just the one with the largest magnitude. Taking this into account, in practice when calculating the dominant eigenvector the **power iteration** algorithm is used. It is an iterative algorithm which returns the eigenvalue λ which is the greatest eigenvalue of A in magnitude and the eigenvector $\mathbf{x}$ assigned to it. The iterative recurrence relation is defined as:

$$\mathbf{b}_{k+1} = \frac{\mathbf{A}\mathbf{b}_k}{||\mathbf{A}\mathbf{b}_k||} \tag{5.6}$$

where $\mathbf{b}_k$ is the eigenvector in the *k*-th iteration. The stopping condition can be defined in various ways such as the maximum number of iterations or a threshold in the change of the eigenvector in consecutive iterations.

When it comes to text, words with high eigenvector scores have either direct connections to important words or are themselves important words. On the other hand, low scores indicate that the words are less central to the main topic of the text and that they are more local to specific sentences.

5.5 PageRank (TextRank) algorithm

PageRank is a ranking algorithm, originally used by Google's search engine to rank web pages in the search engine results [12]. The internet can be modeled as a graph where the web pages are the nodes and the hyperlinks between them are the edges. This idea can be applied to text as well, where the variation of the algorithm is called **textrank** [13] but we will keep the name PageRank as the ranking algorithm.

The basic idea behind the algorithm is the notion of *casting votes*. When one node links to another is casts a vote for the that other node. The higher the number of votes for a node, the higher the importance for that node. The importance of the vote that was casted by the initial node is influenced by the importance of that node itself. For a weighted graph, the vote of node j to node i is scaled by the ratio of the (i, j) edge weight to the sum of all the edge weights of node j. In this way only a part of the j node's vote is casted towards node i. Casting all the votes toward the node i we get the following recurrence relation for the PageRank (PR) score:

$$PR(v_i) = \sum_{v_j \in N(v_i)} \frac{w_{j,i}}{\sum_{v_k \in N(v_j)} w_{j,k}} PR(v_j) \tag{5.7}$$

where $N(v_i)$ is the set of all the neighbors of node i and $w_{i,j}$ is the edge weight between nodes i and j. The mentioned weight ratio can be obtained from the adjacency matrix if the matrix is converted to a *left stochastic matrix* [3] One thing to note is that the maximum magnitude of the eigenvalue of a stochastic matrix is 1 to which the dominant eigenvector is associated.

Let us call the left stochastic matrix of the adjacency matrix $\mathbf{P_A}$ and the vector containing the PageRank values for all the nodes $\mathbf{R}$, then the equation 5.7 can be rewritten in matrix notation as:

$$\mathbf{R} = \mathbf{P_A}\mathbf{R} \tag{5.8}$$

which is the same form as the equation 5.5 where $\lambda = 1$ for 5.8 as already said. So, the PageRank algorithm is reduced to the eigenvector centrality. The eigenvector $\mathbf{R}$ is also called the *stationary probability vector* as its values (scores) sum to 1 and the stationary part means that the vector does not change under the application of the $\mathbf{P_A}$ matrix.

Let us show the PageRank algorithm on the example from Figure 4 by getting the left stochastic matrix first:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 3 & 1 & 0 \\ 1 & 0 & 0 & 0 & 2 \\ 3 & 0 & 0 & 2 & 0 \\ 1 & 0 & 2 & 0 & 1 \\ 0 & 2 & 0 & 1 & 0 \end{bmatrix}, \mathbf{P_A} = \begin{bmatrix} 0 & 0.33 & 0.6 & 0.25 & 0 \\ 0.2 & 0 & 0 & 0 & 0.67 \\ 0.6 & 0 & 0 & 0.5 & 0 \\ 0.2 & 0 & 0.4 & 0 & 0.33 \\ 0 & 0.67 & 0 & 0.25 & 0 \end{bmatrix}$$

The dominant eigenvector after normalization has the following value:

$$\mathbf{R} = \begin{bmatrix} 0.25 & 0.15 & 0.25 & 0.20 & 0.15 \end{bmatrix}^T, \text{with eigenvalue of } \lambda = 1$$

[3] A left stochastic matrix is a square matrix, with each column summing to 1.

Let us confirm this result by calculating the PR score from equation 5.7 for the node A:

$$\begin{aligned}
PR(A) &= \sum_{v_j \in \{B,C,D\}} \frac{w_{j,A}}{\sum_{v_k \in N(v_j)} w_{j,k}} PR(v_j) \\
&= \frac{w_{B,A}}{w_{B,A} + w_{B,E}} PR(B) + \frac{w_{C,A}}{w_{C,A} + w_{C,D}} PR(C) + \frac{w_{D,A}}{w_{D,A} + w_{D,C} + w_{D,E}} PR(D) \\
&= \frac{1}{1+2} 0.15 + \frac{3}{3+2} 0.25 + \frac{1}{1+2+1} 0.20 \\
&= 0.25 = PR(A)
\end{aligned}$$

When it comes to text, words with high PageRank scores are central concepts and they are likely to be connected to other important words in the text (the same concept as web pages and web search) [13].

5.6 Katz centrality

Katz

g

g

g

g

5.7 Comparing the ranking algorithms

To conclude this section let us apply all the ranking algorithms on the example graph from Figure 6 that models the sentences from Figure 5 in the form of a table.

Table 1: Ranking values for the example in Figure 6.

Degree centrality		Closeness Centrality		Betweenness Centrality		Eigenvector Centrality		Pagerank		Katz Centrality	
Word	Ranking	Word	Ranking	Word	Ranking	Word	Ranking	Word	Ranking	Word	Ranking
japan	0.5	japan	0.462	japan	0.566	japan	0.515	fish	0.148	japan	0.374
fish	0.5	source	0.444	fish	0.505	great	0.431	japan	0.148	fish	0.354
great	0.333	sea	0.444	source	0.227	country	0.33	great	0.111	great	0.327
country	0.25	fish	0.429	sea	0.227	economy	0.309	famous	0.074	country	0.278
famous	0.25	island	0.353	economy	0.129	fish	0.279	rice	0.074	source	0.277
rice	0.25	economy	0.353	food	0.068	source	0.27	source	0.074	sea	0.277
source	0.25	food	0.333	sushi	0.068	sea	0.27	sea	0.074	economy	0.273
sea	0.25	sushi	0.333	island	0.064	island	0.171	country	0.074	famous	0.268
economy	0.25	great	0.3	great	0.03	famous	0.164	economy	0.074	rice	0.268
food	0.167	world_war_2	0.293	country	0.023	rice	0.164	food	0.037	island	0.214
sushi	0.167	country	0.293	famous	0.015	world_war_2	0.15	sushi	0.037	food	0.211
island	0.167	famous	0.273	rice	0.015	food	0.09	island	0.037	sushi	0.211
world_war_2	0.167	rice	0.273	world_war_2	0.011	sushi	0.09	world_war_2	0.037	world_war_2	0.209

We can see that word rankings are mostly consistent throughout all the algorithms. The words “japan”, “fish”, “great”, “sea” and “country” are the words with the highest ranks and they do really convey the meaning of the example sentences. The words “japan” and “fish” were acting as some kind of connectors between sentences they get high rankings. Other words that happened to be in the same context as those high scoring words have also gotten higher rankings (e.g. “great” and “sea”).

With this overview we can see the potential of modeling text as graphs, especially as the methods are unsupervised and fairly easy to implement. In the next chapter we will test the ranking algorithms on real data and see how they perform.

6 Experiments and Results

In this section we will perform tests on a dataset of scientific paper abstracts. We will define the metrics used to evaluate the model and in the end we will see the obtained results and discuss them. Let us firstly look at the dataset used.

6.1 Dataset

6.2 Evaluation metrics

Slide05 in Turkey's NLP slides. Macro average , precision, recall

6.3 Testing

7 Conclusion

8 Literature

[1] Almeida, Felipe, and Geraldo Xexéo. "*Word embeddings: A survey*". arXiv preprint arXiv:1901.09069 (2019).
URL: https://arxiv.org/pdf/1901.09069.pdf

[2] Lin, Tianyang, et al. "*A survey of transformers*". AI Open (2022).
URL: https://www.sciencedirect.com/science/article/pii/S2666651022000146

[3] Zu, Xian, Fei Xie, and Xiaojian Liu. "*Graph-based keyphrase extraction using word and document embeddings*". IEEE International Conference on Knowledge Graph (ICKG) (2020).
URL: https://ieeexplore.ieee.org/abstract/document/9194571

[4] The Stanford NLP Group
URL: https://nlp.stanford.edu/

[5] Word2Vec project information
URL: https://code.google.com/archive/p/word2vec/

[6] GloVe: Global Vectors for Word Representation
URL: https://nlp.stanford.edu/projects/glove/

[7] Ruas, T., Grosky, W. and Aizawa, A. "*Multi-sense embeddings through a word sense disambiguation process*". Expert Systems with Applications, (2019).
URL: https://arxiv.org/pdf/2101.08700.pdf

[8] Jurić Ž. "*Diskretna matematika za studente tehničkih nauka*". ETF Sarajevo, UNSA, (2017).

[9] NetworkX Library documentation
URL: https://networkx.org/documentation/stable/reference/algorithms/

[10] Bavelas, Alex. "*Communication patterns in task-oriented groups*". The journal of the acoustical society of America (1950).
URL: https://hal.science/hal-03266728/file/bavelas1950groupefmr.pdf

[11] Freeman, Linton C. "*A set of measures of centrality based on betweenness*". Sociometry (1977).
URL: https://www.jstor.org/stable/3033543?seq=7

[12] Brin, Sergey, and Lawrence Page."*The anatomy of a large-scale hypertextual web search engine*". Computer networks and ISDN systems (1998).
URL: https://www.sciencedirect.com/science/article/abs/pii/S016975529800110X

[13] Mihalcea, Rada, and Paul Tarau. "*Textrank: Bringing order into text*." Proceedings of the 2004 conference on empirical methods in natural language processing (2004).
URL: https://aclanthology.org/W04-3252.pdf