Ministry of Education and Science of Ukraine

National University “Kyiv-Mohyla Academy”

Faculty of Informatics

Mathematics Department

Coursework

educational level – master

on the topic: “**DEVELOPMENT OF THE SYSTEM FOR PLAGIARISM CHECKING OF UKRAINIAN TEXTS**”

By: 1-st year student

of the educational program “Applied Mathematics”,
specialty 113 Applied Mathematics

Bikchentaev Mykola Oleksiiovych

Supervisor: Hlybovets A. M.,

Doctor of Engineering

Reviewer________________________

(surname and initials)

The coursework was defended
with a grade _______________________

EC secretary ____________________
«____» _________________ 2022 р.

Kyiv – 2022

CONTENTS

Introduction ... 3
1. Types of plagiarism ... 4
2. Plagiarism detection approaches ... 5
3. Models used for plagiarism detection ... 5
3.1. Word2Vec ... 5
3.1.1. Continuous Bag of Words (CBOW) ... 5
3.1.2. Skip-gram ... 6
3.1.3. Word2Vec architecture ... 7
3.1.4. Word2Vec training ... 8
3.1.5. Negative sampling ... 9
3.1.6. Word2Vec training with negative sampling ... 11
3.1.7. Limitations of Word2Vec ... 13
3.2. Transformers ... 13
3.2.1. What is a transformer? ... 13
3.2.2. Embedding and position encoding ... 15
3.2.3. Self-attention ... 16
3.2.4. Multi-headed attention ... 19
3.2.5. Decoders ... 19
3.3. BERT ... 21
3.3.1. What is BERT? ... 21
3.3.2. BERT use cases ... 21
3.3.3. BERT origins ... 22
3.3.4. BERT architecture ... 24
4. Using Word2Vec and BERT for plagiarism detection ... 26
5. Plagiarism detection in Ukrainian texts ... 27
5.1. Overview of BERT and Word2Vec implementations ... 27
5.2. Application for plagiarism detection ... 28
Conclusion ... 33
References ... 34

INTRODUCTION

Plagiarism is usually defined as the passing off someone else's ideas as your own. As the Internet becomes more and more accessible every day, a huge amount of data becomes available to people. Nowadays, it is quite easy to find a suitable study and plagiarize it instead of developing one's own from scratch.

Plagiarism undermines the efforts of the researcher whose work has been plagiarized and gives the plagiarist the opportunity to over-praise himself; such a person can be detrimental when appointed to an important position.

Many fields of life are susceptible to plagiarism, including research and education. Plagiarism can also take many forms: from straight up copy-paste to paraphrasing and sentence restructuring. This makes plagiarism a rather complex problem, where methods, such as longest common subsequence or n-grams, based on finding shared words between documents[3], might not work. Therefore, we might consider applying deep learning to the problem of plagiarism detection.

So, the **aim of this work** is to review two machine learning models called BERT and Word2Vec, determine how can they be used in plagiarism detection, and develop an application where users can check texts for plagiarism.

Object of study: plagiarism, BERT and Word2Vec models.

Research methods: analysis of scientific literature.

Objectives of the study:

1. Study the concept of plagiarism and its types;
2. Review BERT and Word2Vec models;
3. Using BERT and Word2Vec develop an application for plagiarism detection.

The work consists of an introduction, five chapters, conclusions and a list of references.

In the *first chapter*, we study the concept of plagiarism and its types.

In the *second chapter*, we review modern approaches to plagiarism detection.

The *third chapter* is devoted to the study of BERT and Word2Vec models, their architecture, and possible use cases.

The *fourth chapter* deals with how BERT and Word2Vec can be used specifically in plagiarism detection.

The *fifth section* is dedicated to the development of the application for plagiarism detection.

Scientific novelty of the obtained results: the paper presents one of the possible ways to utilize machine learning models, like BERT and Word2Vec, in order to solve a problem of plagiarism detection. Reviews the architecture of BERT and Word2Vec models.

The practical significance of the results obtained: the paper might be of use to the ones who are interested in the problem of plagiarism detection.

1.Types of plagiarism

With the appearance and further development of the Internet, lots of texts have become available to people. And with that, plagiarism has become a common phenomenon. It can occur in many fields of life, including research and education [1, p. 81].

According to the Cambridge Dictionary, *plagiarism* – is the process or practice of using another person's ideas or work and pretending that it is your own. We can divide plagiarism into several types[2]:

a) *Direct* or *verbatim plagiarism*. The author copies text word to word from the source and pastes it into his work;

b) *Source-based plagiarism*. The author references non-existent resources or does not reference all used resources;

c) *Paraphrasing plagiarism*. The author changes the structure of the sentences of the original text by rearranging or deleting words, replacing them with synonyms, etc.;

d) *Mosaic* or *patchwork plagiarism*. It is a more advanced variation of the previous type. The author interweaves the original text (probably paraphrased) with many different sources, including his ideas and perspective;

e) *Plagiarism of ideas*. The most hard-to-detect plagiarism. The author uses (without acknowledging) ideas or conclusions from other works as a foundation for his work.

All these types of plagiarism usually boil down to changing the text vocabulary or its syntactic or semantic representation.

Vocabulary changes involve the addition, deletion, or words replacement. We can detect changes of this type by methods like the longest common subsequence or n-grams. The more shared terms documents have, the more similar they are[3, p. 1-2].

Synthetic changes imply changes in the structure of the sentence, for example, the rearrangement of words and phrases or changing the sentence grammar. To detect such changes, we make use of the text's syntactical units. For instance, we may use POS tags to find similar documents[3, p. 1-2].

Semantic changes include the changes of two previous types, as well as text paraphrasing. To detect these changes, we conduct a semantic analysis of the texts. For instance, some methods use synonyms, antonyms, hypernyms, and hyponyms to identify changes and compare texts[3, p. 1-2].

Apart from various kinds of plagiarism, we can divide plagiarism detection itself into two categories. These are *external* and *intrinsic* plagiarism detection[3, p. 2].

External plagiarism detection compares an input document with other available documents. *Intrinsic plagiarism* finds parts of the input document that another author has written.

2. Plagiarism detection approaches

Plagiarism detection is a part of Natural Language Processing (NLP). Currently, there are many solutions for lexical or syntactical plagiarism based on NLP techniques, including concept extraction using corpus like WordNet[3, p. 82].

In recent years, deep learning approaches have become rather popular in NLP. Deep learning is a branch of machine learning. It focuses on the extraction of high-level (abstract) features of the data. Also, deep learning networks do not require any complicated featuring engineering or labeled data. These factors make deep learning models a good fit for the problem of plagiarism detection[1, p. 3].

In the subsequent chapters, we will look at one deep learning network called *BERT* and a two-layer neural network called *Word2Vec*.

3. Models used for plagiarism detection

3.1. Word2Vec

Word2Vec is a two-layer neural network. It takes as input a large corpus of words and produces a vector space of several hundred dimensions. Each vector represents a single word from the corpus. Words that have a similar meaning (or context) have vectors that are close to each other. Vectors produced by the Word2Vec are called *word embeddings*[4].

We can implement Word2Vec in two ways: via *Continuous Bag of Words* (CBOW) or *Skip-gram*.

3.1.1. Continuous Bag of Words (CBOW)

CBOW tries to predict a target word (for example, "porch") from the surrounding context or, simply speaking, from the surrounding words ("a dog sits on a"). Skip-gram, on the other hand, tries to predict the surrounding context from the target word[4].

Consider a quote from the sci-fi book Dune: "Thou shalt not make a machine in the likeness of a human mind."

CBOW will begin its work by generating a dataset from the input text (in our case is just a sentence) using a *sliding window*. The sliding window has a fixed size. It defines the number of words we will extract from the text[5].

We will take a sliding window with the size of three. The sliding window will start on the first three words (Fig. 1).

The first three words will constitute our first dataset sample. The first two words are features the third is a target that we will try to predict.

Thou shalt not make a machine in the likeness of a human mind

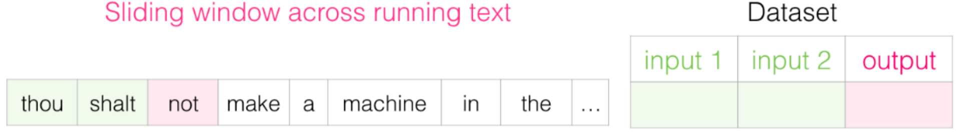


Figure 1. Word2Vec sliding window – Step 1 [5]

We then continue to slide our window by one word to the right.

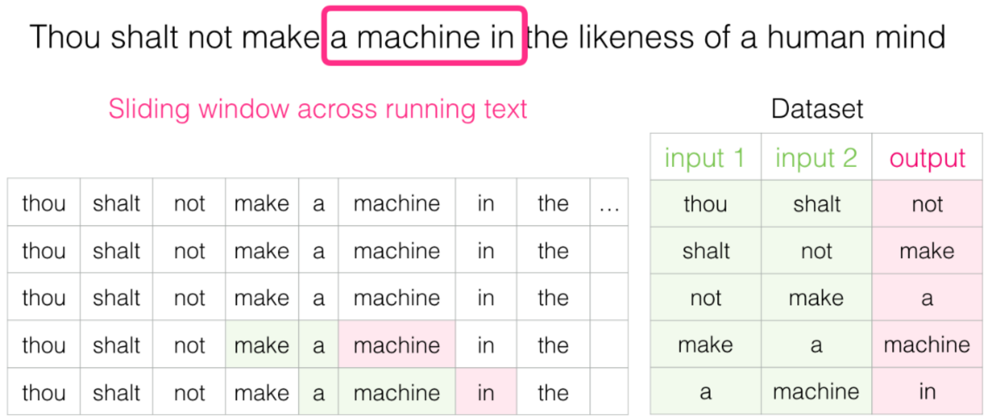


Figure 2. Word2Vec sliding window - Step 2 [5]

We can also improve this approach if we consider words that come before and after the target. Thus, for every "window slide," we will have four features and one target.

3.1.2. Skip-gram

Since skip-gram tries to predict the surrounding context from the target word, it uses a different method for dataset organization[5].
Consider the following picture:

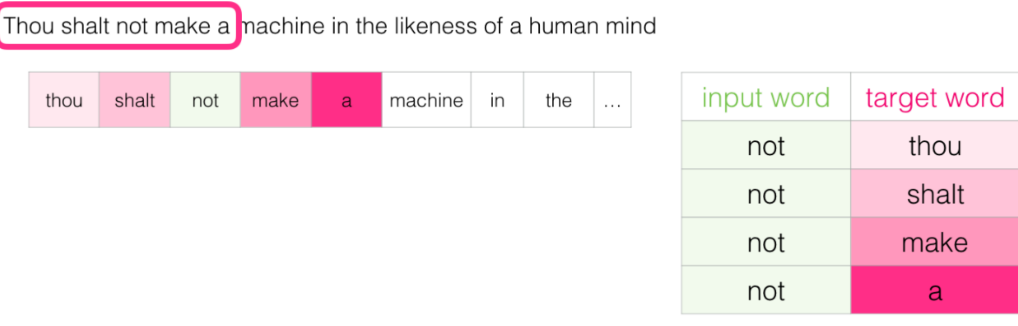


Figure 3. Word2Vec skip-gram – Step 1 [5]

Unlike the approach proposed for CBOW skip-gram generates four samples instead of one.

Moving one position will produce four new samples:

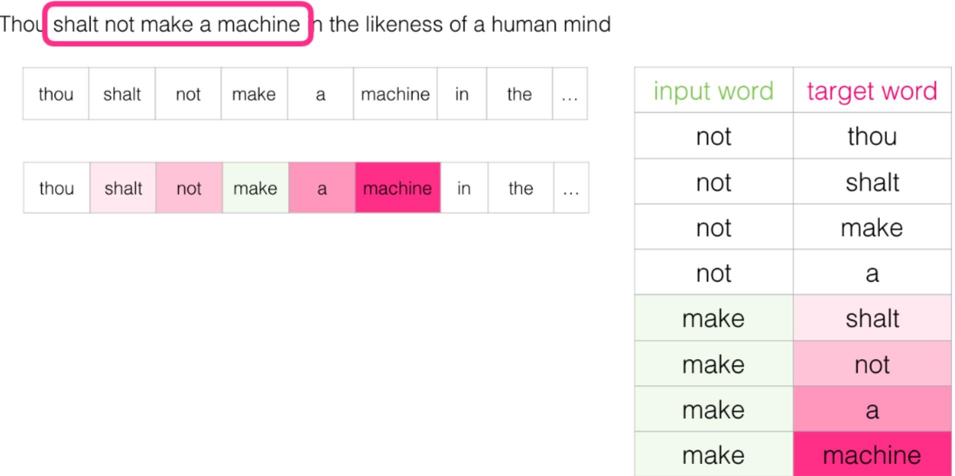


Figure 4. Word2Vec skip-gram – Step 2 [5]

3.1.3. Word2Vec architecture

After we have generated a dataset, we proceed to model training. It is very similar for both CBOW and Skip-gram. Thus, in the example below, we will consider model training in the case of Skip-gram.

To begin with, that's how the model architecture looks like:

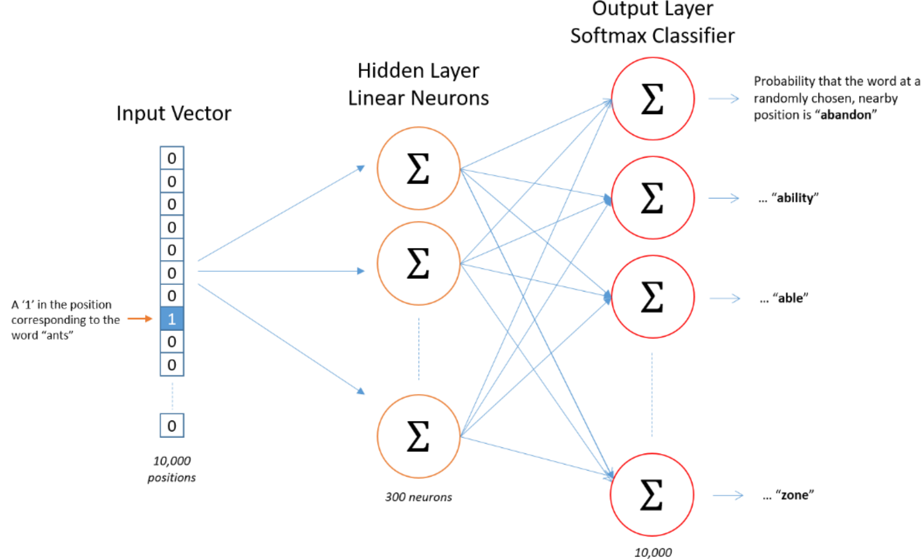


Figure 5. Word2Vec architecture [4]

The input vector is a vector representation of a single word. Its length is equal to the size of the vocabulary. It has zeroes everywhere, except for the index that corresponds to the input word[4].

The hidden layer is a standard fully connected layer, which weights are the word embeddings. The output layer gives us the probability for other words to be the neighbor of the input word.

To get the word embeddings, we remove the output layer after having trained the model. Thus, we will get the model that will produce word embedding instead of word occurrence probabilities[4].

3.1.4. Word2Vec training

Let's now look at how we train the model to predict neighboring words.

Firstly, we initialize model weights with random numbers. Then, we pick the first feature from the first dataset sample and give it to the model.

The weights of the hidden layer are the word embeddings we are trying to get as a result of model training. These weights can be represented as a matrix, which in turn will be the matrix of word embeddings. The shape of this matrix will be $N \times M$, where N – is the number of words in the vocabulary, and M – is the number of neurons in the hidden layer[6].

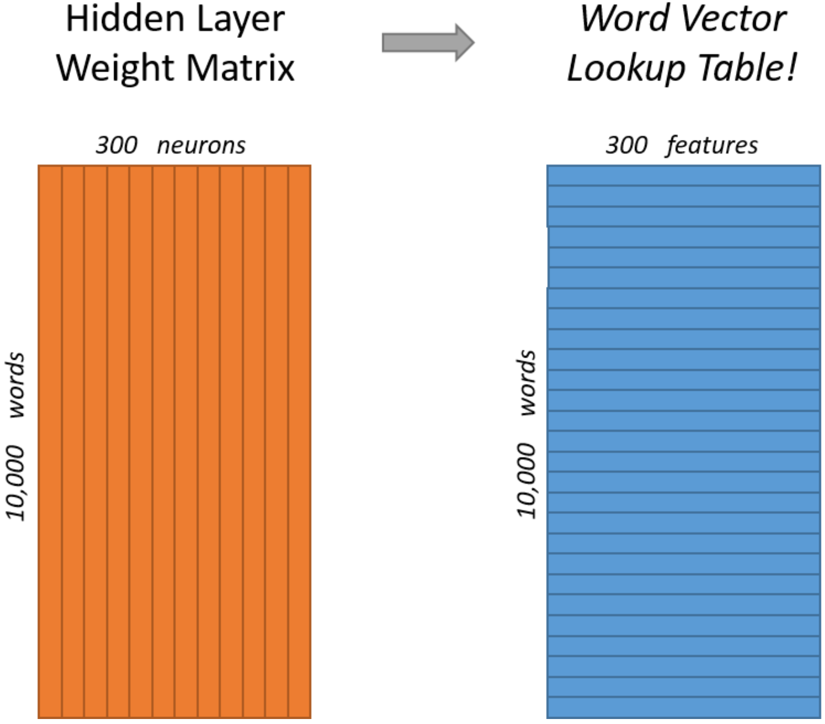


Figure 6. Word2Vec hidden layer weights [6]

So we multiply the encoded input word by the weight matrix, thus getting the corresponding word vector[6]. It will look like the following:

$$(0 \quad 0 \quad 1) \times \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = (7 \quad 8 \quad 9)$$

The resulting vector is multiplied by the weights of the output layer, which can be represented as a matrix with a shape of $M \times N$. Therefore, the input weights are represented by an $N \times M$ matrix, and the output weights are represented by an $M \times N$ matrix.

As was mentioned above, the output vector should contain the probabilities for each word in the vocabulary to appear near the input word. So, the numbers in the output vector should sum up to 1. To get this result, we apply a *softmax activation function* to each value of the output vector[6].

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

In the end, we subtract the vector of probabilities from the input vector to calculate the errors and adjust the weights using backpropagation[6].

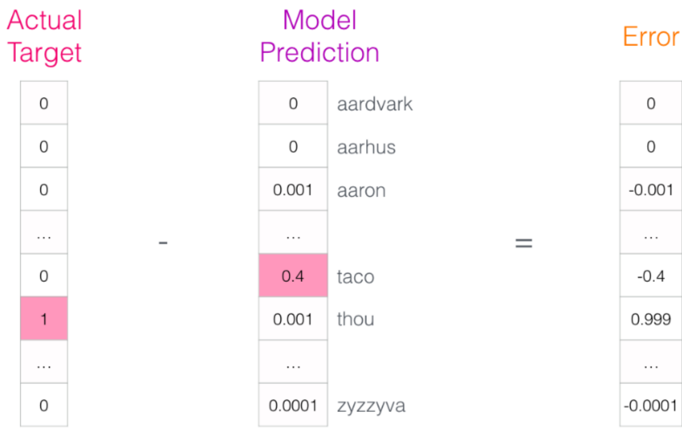


Figure 7. Word2Vec prediction error calculation [5]

3.1.5. Negative sampling

The last step of the Word2Vec training can be computationally expensive, especially if the size of the vocabulary is large. So, to improve the model performance, we may switch the task from predicting the probabilities of neighboring words to predicting the probability for two words to be neighbors, where 1 will mark that words are neighbors, and 0 otherwise[5].

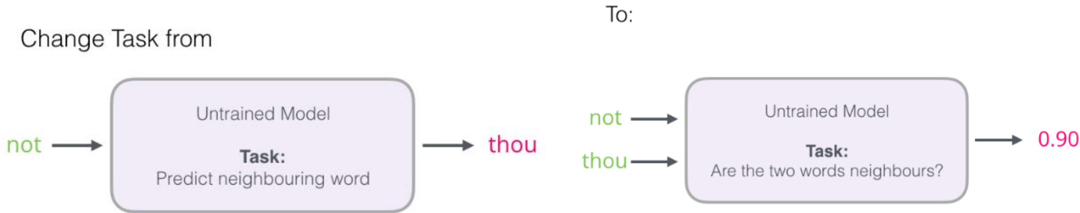


Figure 8. Word2Vec negative sampling [5]

This will turn our model from a neural network into a logistic regression, allowing us to perform computations at a much greater speed[5].

To perform this switch, we should also change the structure of the dataset. There will be no input and the target word, but input word, output word, and a target that will hold values of 0 or 1[5].

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine

input word	output word	target
not	thou	1
not	shalt	1
not	make	1
not	a	1
make	shalt	1
make	not	1
make	a	1
make	machine	1

Figure 9. Word2Vec negative sampling [5]

There's also a tiny problem left. Since the target is always 1 we may get a model that will always output 1 achieving 100% accuracy but learning nothing along the way. To combat this, we will use *negative samples* in our dataset. Negative samples - are pairs of words that are not neighbors (so the target for them is 0)[5].

input word	output word	target
not	thou	1
not		0
not		0
not	shalt	1
not	make	1

Figure 10. Word2Vec negative samples [5]

To fill the missing output words, we randomly sample words from the vocabulary. The probability of selecting a word as a negative sample depends on the frequency (word count) of this word. Specifically, each word is given a weight equal to its frequency raised to the power of 3/4. The probability for selecting this word will be this weight divided by the sum of the weight of all other words[4].

The idea of adding a negative sample is inspired by the concept of Noise-contrastive estimation. We contrast actual signals (being pairs of words that are neighbors) with noise (which are negative samples)[5].

3.1.6. Word2Vec training with negative sampling

At the start of the training, two matrices are initialized. The first is called *embedding matrix*, and the second one - *context matrix*. These two matrices store embeddings for our words from the vocabulary and have equal dimensions[5]. The number of rows corresponds to the number of words in the vocabulary and the number of columns corresponds to the desired length of word embedding vectors (300 hundred is a common value. It was used by Google in the Word2Vec model which was trained on the Google News dataset[7])

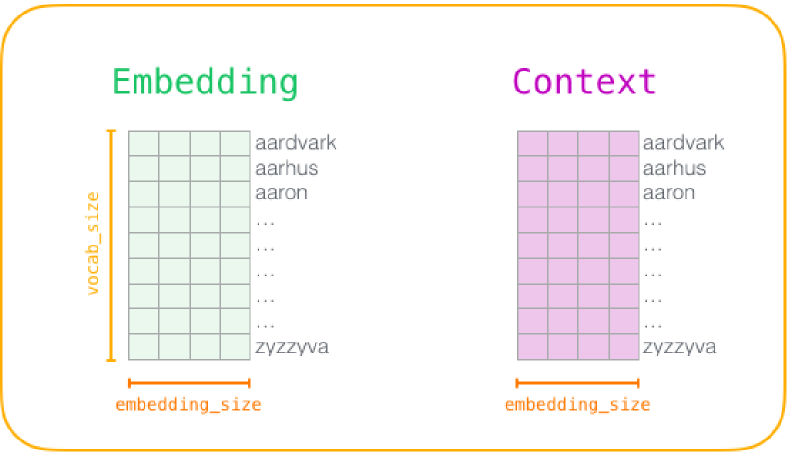


Figure 11. Word2Vec embedding and context matrix [5]

We initialize these matrices with random values and take one positive sample and its connected negative samples from the dataset[5].

dataset

input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	mango	0
not	finglonger	0
not	make	1
not	plumbus	0
...	...	...

model

Figure 12. Word2Vec dataset and model [5]

As a result, we will have one input word (not, in this case) and three output or *context* words: thou, aaron, and taco (the last two being negative samples). We then one-hot encode these words and multiply the vector of the input word by the embedding matrix and the vectors of the output words by the context matrix. In this way, we will get word embeddings of all the words we need[5].

We then take the dot product of the input embedding with each of the output embeddings. The resulting number will indicate the similarity between the input and the output embeddings[5].

input word	output word	target	input • output
not	thou	1	0.2
not	aaron	0	-1.11
not	taco	0	0.74

Figure 13. Word2Vec similarity calculation [5]

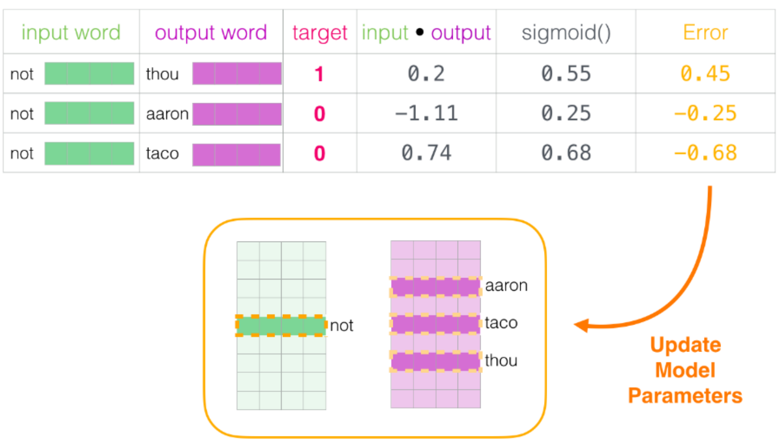
Since the output vector should contain probabilities (positive numbers between 0 and 1) we apply *sigmoid* to them[5].

$$S(x) = \frac{1}{1 + e^{-x}}$$

input word	output word	target	input • output	sigmoid()
not	thou	1	0.2	0.55
not	aaron	0	-1.11	0.25
not	taco	0	0.74	0.68

Figure 14. Word2Vec similarity calculation [5]

We can then subtract these values from the target values, calculate the error, and update the model weights (which are "stored" in the context and embedding matrices)[5].



input word	output word	target	input • output	sigmoid()	Error
not	thou	1	0.2	0.55	0.45
not	aaron	0	-1.11	0.25	-0.25
not	taco	0	0.74	0.68	-0.68

Figure 15. Word2Vec similarity calculation [5]

3.1.7. Limitations of Word2Vec

Word2Vec may give decent results nevertheless, sometimes it might not be accurate enough[8]. Word embeddings don't take into consideration the order of words in which they appear. This may lead to the loss of some syntactic and semantic information of the sentence.

For example[8], the sentences "You are going there to teach not play." and "You are going there to play not teach." will have similar representation in the vector space, however, their meaning is different.

Also, depending on the context, words may bear different meanings. For instance[8] the word "bucket" in the sentences "I have scuba diving in my bucket list." and "There is a bucket filled with drinking water." has a different meaning.

So we need a model that will preserve the contextual information relating to the words in a sentence. One of such models is *BERT*.

Before we look at BERT, we should get familiar with *transformers* which are heavily utilized in BERT.

3.2. Transformers

3.2.1. What is a transformer?

A *transformer* is a component used in neural networks to process sequential data, like text or time-series data. Quite often transformers are used in the area of NLP[11].

The transformer takes input text in the form of a sequence of vectors and converts it into a vector called *encoding*, and then decodes it back into another sequence[11].

Transformers also use an *attention mechanism*. The attention mechanism helps the model to "remember" relationships between input tokens. For example, this can be used in machine translation of sentences. The attention mechanism will allow the model to translate words like "it" into the word of correct gender in Spanish or French by paying attention to all neighboring words in the original sentence[11].

The structure of a transformer can be seen on the Figure 16.

Each encoder consists of two layers: *self-attention* and *feed-forward*. The self-attention layer is where the attention mechanism is used to produce better word encodings. The output of the self-attention layer is then fed to the feed-forward neural network[12]. The output of a feed-forward network is then fed to the next encoder.

A decoder has the same two layers, plus an *encoder-decoder attention* layer between them[12] (Fig. 17).

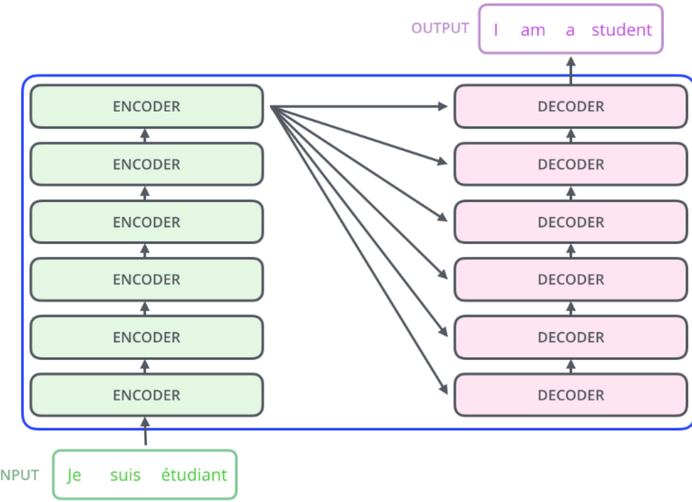


Figure 16. Transformer structure [12]

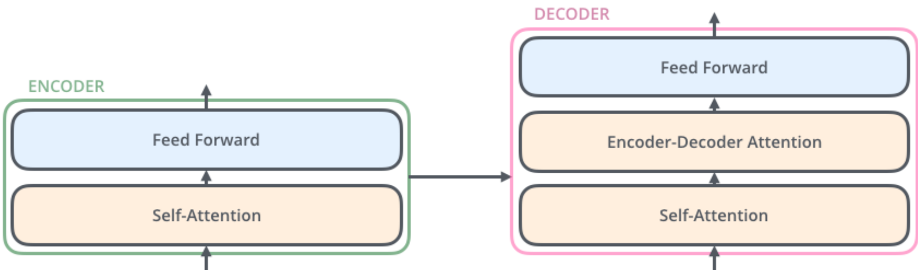


Figure 17. Transformer encoder and decoder [12]

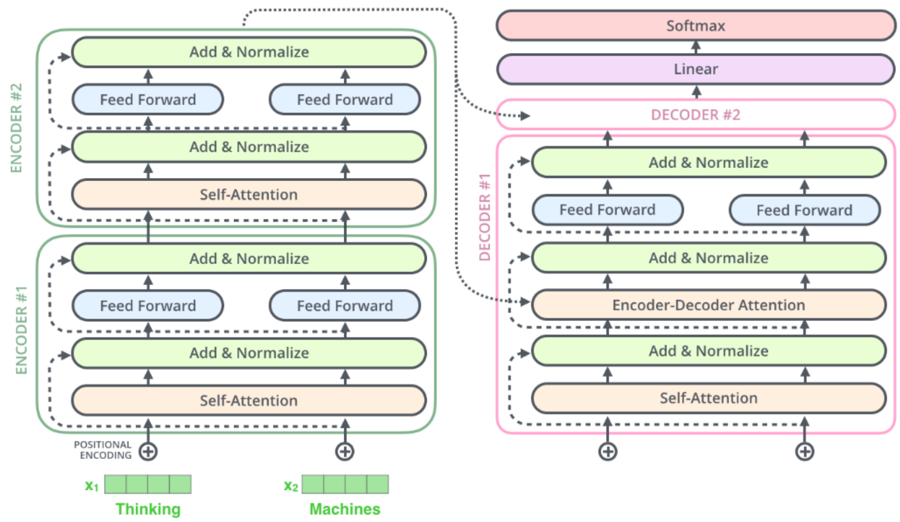


Figure 18. Encoder and decoder structure [12]

Attention layers of encoders and decoders operate similarly, yet they have a few small but significant differences[13]:

a) The *encoder self-attention layer* computes the relevance of each word in the input sentence to each other word in the same input sentence;

b) The *decoder self-attention layer* computes the relevance of each word in the output sentence to each other word in the same output sentence. Also, it is

allowed to take into consideration only the previous positions in the input sequence. The future positions are masked;

c) The *decoder encoder-decoder attention layer* computes the relevance of each word in the output to each other word in the input sentence.

Each layer of the encoder (decoder) has a residual connection around it and is followed by a layer-normalization step[12] (Fig. 18).

3.2.2. Embedding and position encoding

Before we pass any text to encoders, we should first preprocess it a bit. This procedure consists of three steps[14]:

a) Generate word embeddings for each word in the input text;

b) Compute the *position encodings* for each word in the input text;

c) Combine these encodings by summing them.

Transformer encoders process words from input text in parallel, with each word following its separate "path". Thus, information about the position of the words is usually lost. *Position encoding* is used to remember the position of the words in the input sentence[14].

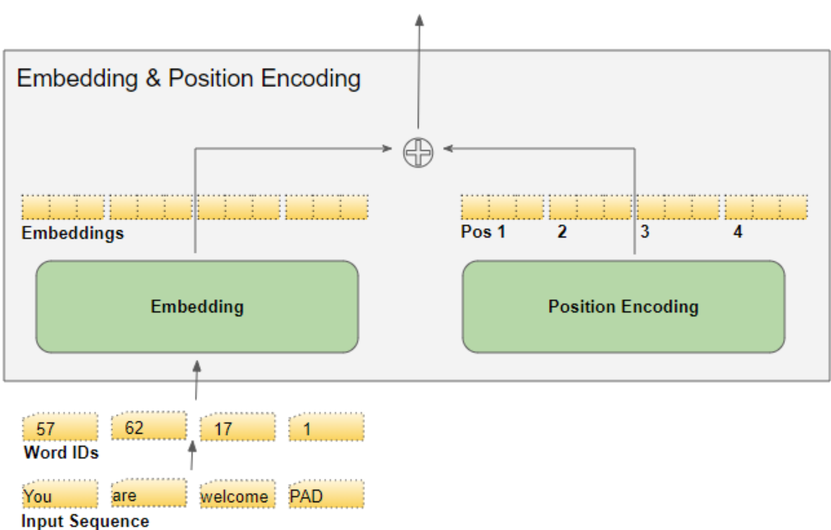


Figure 19. Transformer embedding and position encoding [14]

It is a fixed value that depends only on the length of the word embedding vector (which is usually equal to 512). To calculate it we use the following formula[14]:

$$P_{(pos,2i)} = \sin\left(pos/1000^{2i/d_{model}}\right)$$
$$P_{(pos,2i+1)} = \cos\left(pos/1000^{2i/d_{model}}\right)$$

pos – index of the word in the sentence; i – index of the value from the word embedding vector; d_{model} – length of the word embedding vector.

3.2.3. Self-attention

The self-attention layer works with matrices, where the number of columns corresponds to the length of the word embeddings, and the number of rows is a hyperparameter we can set (usually it is equal to the length of the longest sentence).

To simplify the explanation of how self-attention works we will use vectors in the examples below. Then, to show how it is implemented, we will give several examples with matrices.

Suppose we have the sentence "Thinking Machines". We generate embeddings for each word in this sentence and pass the resulting vectors to the encoder.

The first step in the self-attention layer would be to create three vectors from the encoder input vectors. These vectors are called Queue, Key, and Value. To get them we multiply the input vector by the three weight matrices that were learned during the model training[12].

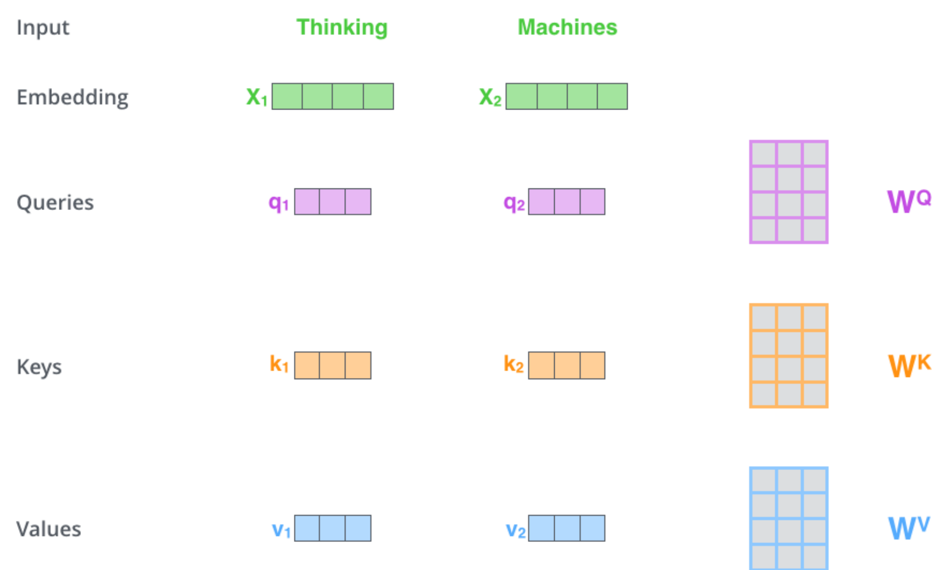


Figure 20. Transformer self-attention layer - Step 1 [12]

On the image, we multiply the word embeddings for "Thinking" and "Machines" by the matrices W^Q, W^K, and W^V producing Queries, Keys, and Values vectors respectively.

The resulting vectors will have smaller dimensionality of 64 (while the input vector dimensionality is usually 512). These vectors do not have to be smaller, but it is an architecture choice to make the computation of *multiheaded attention* (which will be discussed later) constant[12].

In the second step, we calculate a score. The score will determine how much focus we should place on other vectors (words) when we encode the current vector (word).

For example, when calculating self-attention for the input vector, that corresponds to the word “Thinking”, we will take the dot product of the query vector q_1 and the key vector $k_i, i = 1, \dots, n$ where n – is the number of input vectors[12].

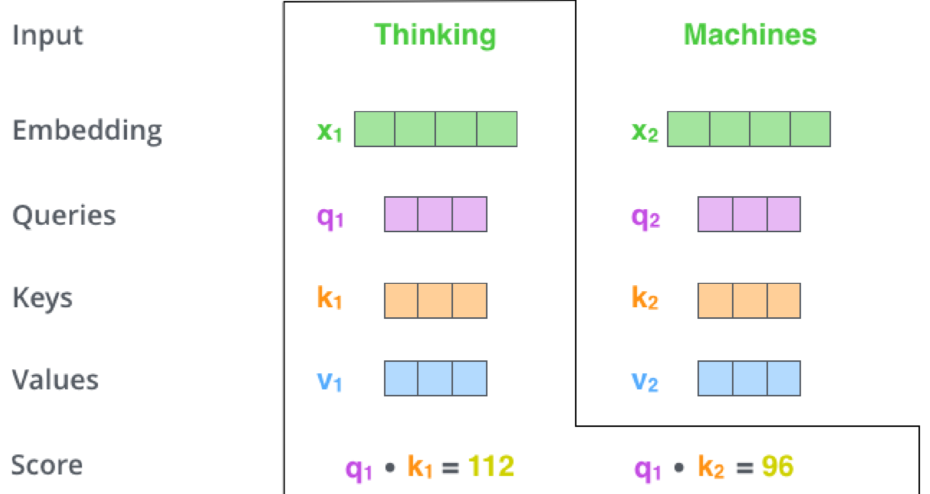


Figure 21. Transformer self-attention layer - Step 2 [12]

In the third and fourth steps, we will divide scores by the square root of 64 (which is the dimension of Query, Key, and Value vectors) and pass the resulting values to the softmax function to make them positive and add up to one[12].

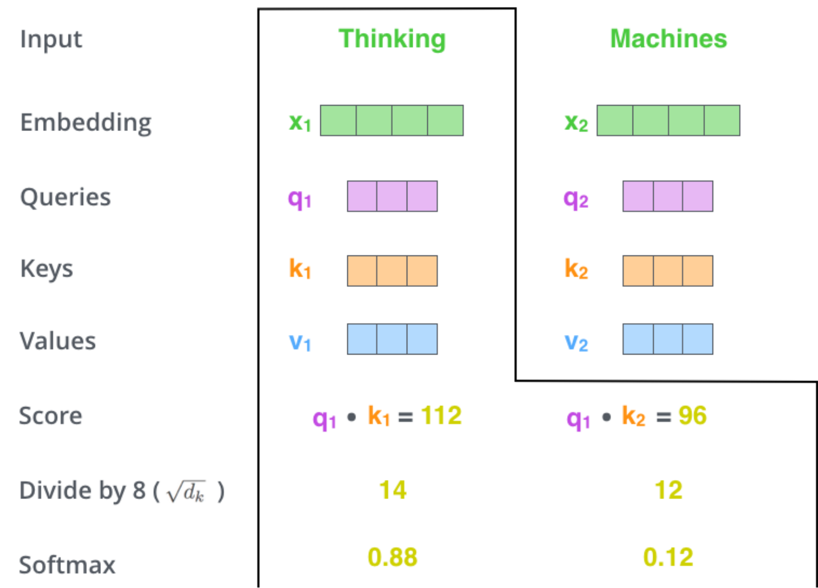


Figure 22. Transformer self-attention layer - Step 3, 4 [12]

In the fifth step, we multiply the value vector of each input vector by the output of the softmax function. This will help us to discard irrelevant words by multiplying them by values like 0.00001.

In the sixth step, we sum up all the vectors, produced in the previous step, to produce an output of the self-attention layer for the first input vector, that corresponds to the word "Thinking"[12].

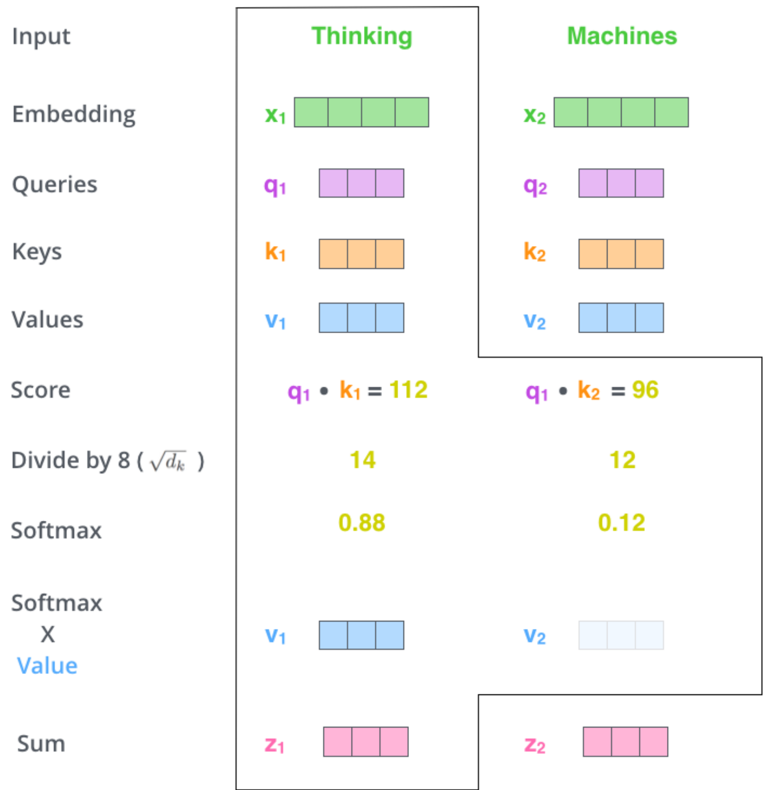


Figure 23. Transformer self-attention layer - Step 5, 6 [12]

If we arrange the input embeddings into the matrix, then, to get the self-attention output, we would use the following formula[12]:

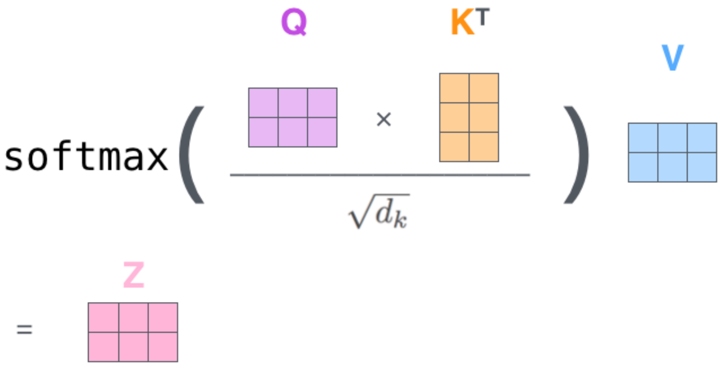


Figure 24. Transformer self-attention output [12]

Matrices Q, K, and V are the Query, Key, and Values matrices which we got by multiplying the input matrix by each of the three weight matrices W^Q, W^K, and W^V.

3.2.4. Multi-headed attention

In the Transformer, the self-attention layer repeats its computation multiple times in parallel. Each of these is called an *attention head*. Every attention head has a separate set of W^Q, W^K, and W^V matrices.

Suppose we have eight attention heads. We will compute eight different "attention outputs" and then combine them to produce a final attention score.

To combine the outputs, we concatenate them and multiply them by an additional weight matrix W^O[12].

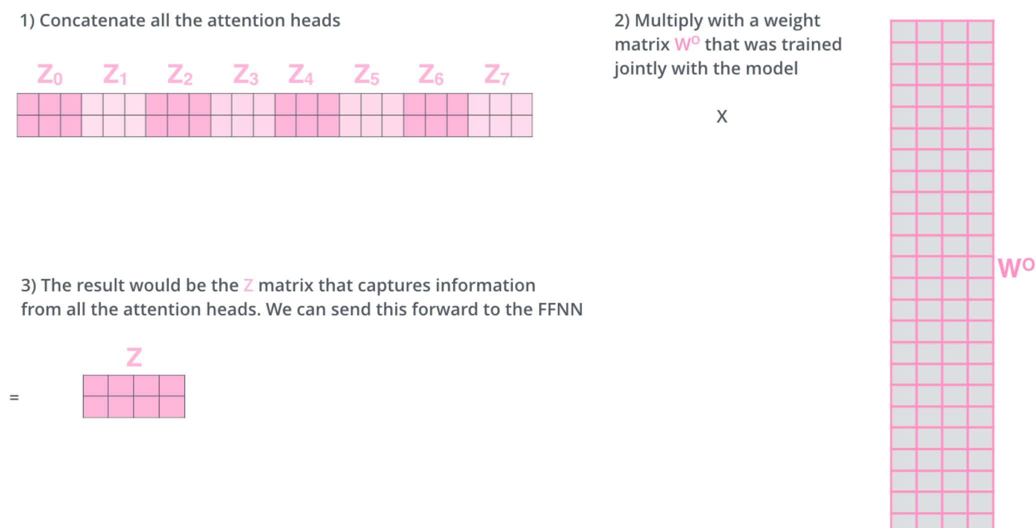


Figure 25. Transformer multiple attention heads [12]

Multiple attention heads allow the Transformer to learn different aspects of the meaning of each word. For instance, one attention head may capture the gender of a word, while the other can capture its cardinality[15].

3.2.5. Decoders

After the input sequence went all the way through the encoder stack, the output of the top encoder is passed to all the decoders to be used in the encoder-decoder attention layer. This layer is very similar to the encoder's attention layer but instead of the matrices W^K and W^Q an encoder stack output is used[14] (Fig. 26).

The self-attention layer of the decoder is also like that of the encoder except for the fact that when calculating the attention, it is allowed to take into consideration only the previous positions in the input sequence. The future positions are masked (set to `-inf`).

The decoder output is generated in a loop. The entire output of each loop is re-fed to the first decoder until we reach an end-of-sentence token.

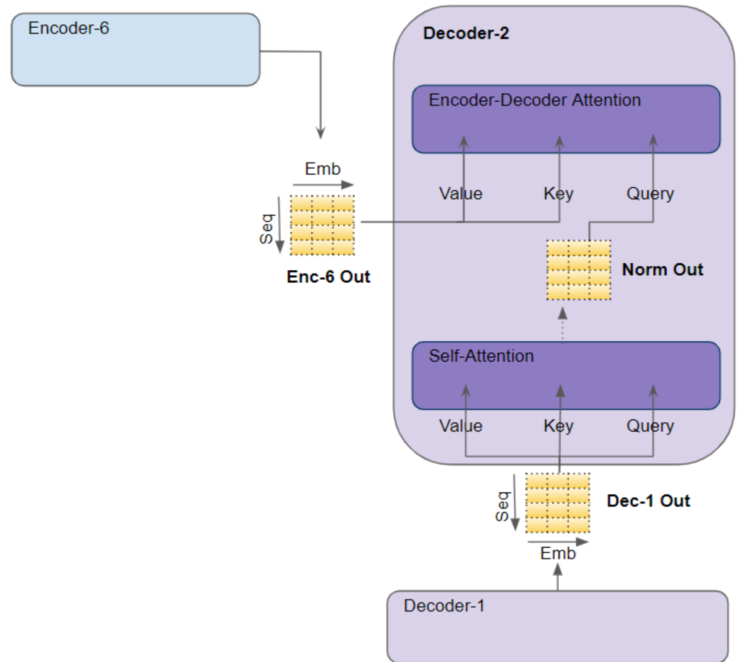


Figure 26. Transformer decoder structure [14]

Like with the encoders, we convert the input sequence into the embedding (with positional encoding) before passing it to the first decoder[16].

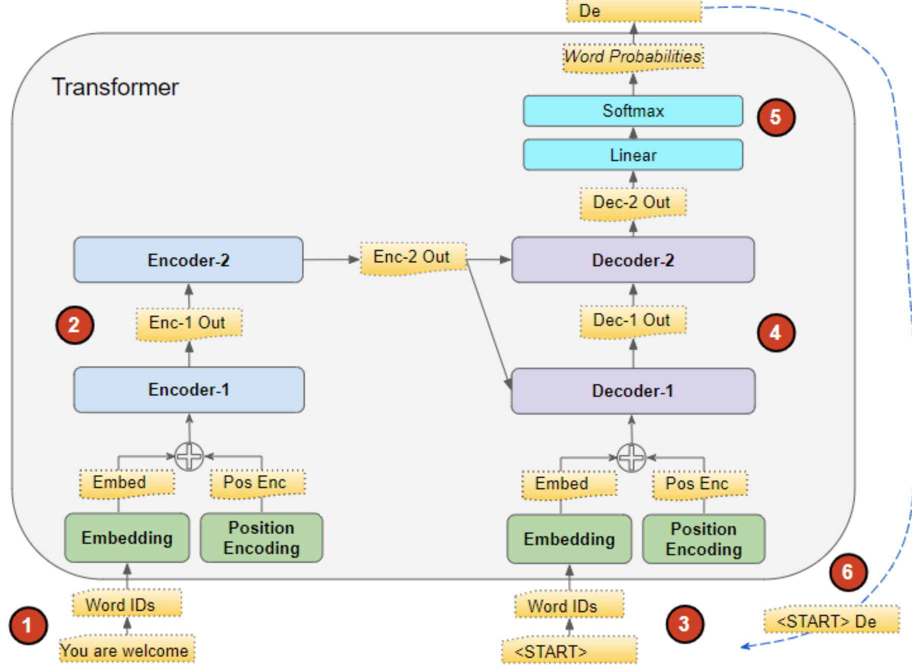


Figure 27. Transformer structure [16]

The output of the decoder stack is passed to the *linear layer*, followed by a *softmax layer*.

The linear layer projects the vector produced by the decoder stack into a much larger vector called a *logits vector*. The logits vector will contain scores for each unique word in the vocabulary (which is derived from the training dataset)[12].

The softmax layer will turn these scores into probabilities (that are positive and add up to one). The word that corresponds to the highest probability will then be outputted by the model.

3.3. BERT

3.3.1. What is BERT?

BERT or Bidirectional Encoder Representations from Transformers is a paper published by researchers at Google AI Language in 2018[9]. The release of BERT marked the beginning of a new era in the NLP, as it showed state-of-the-art results in such tasks as Question Answering, Natural Language Inference, and others[9].

3.3.2. BERT use cases

The first way to use BERT is for sentence classification[10].

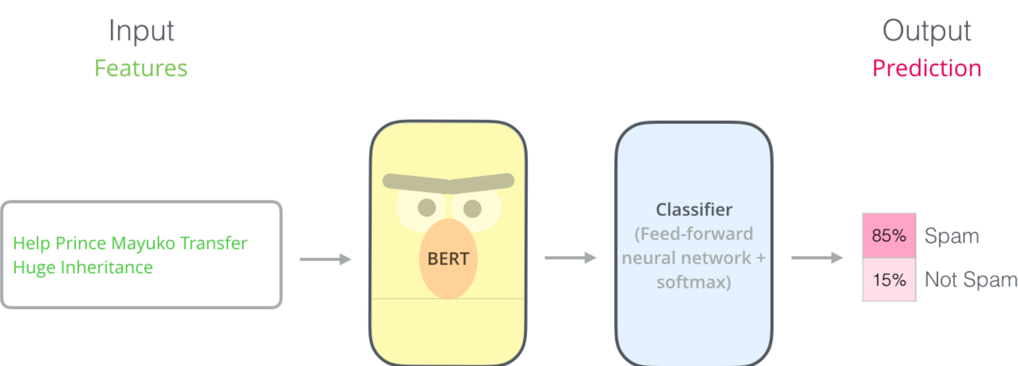


Figure 28. BERT sentence classification [10]

Training such a classifier requires almost no changes to BERT. We need to train only the classifier.

Some use cases for BERT include *sentiment analysis* (e.g. given a review, tell whether it is positive or negative) and *fact-checking* (e.g. given a sentence, tell whether it is a claim or not).

3.3.3. BERT origins

BERT has been built upon several ideas that include *Transformers*, *ELMo*, and *OpenAI Transformer*. In this section, we will briefly overview these technologies and see how BERT incorporates them.

ELMo is a model that is used to produce word embeddings for words. Models with a similar goal, like Word2Vec or GloVe, generate embeddings no matter what the context of the word is. Thus, the word “like” used in the sentences “I like apples” and “You look like Matt Damon” would have the same embedding vector[10].

ELMo introduces a concept of *contextualized word embeddings*. Therefore, before assigning a word an embedding, it looks at the whole sentence where this word is located[10].

To do this, ELMo uses a bi-directional LSTM trained to predict the next word in a sentence (a task called *Language Modelling*). Bi-directional LSTM consists of a Forward Language Model (contains information about a current word and other words *before* it) and a Backward Language Model (which contains information about the current word and other words *after* it)[10].

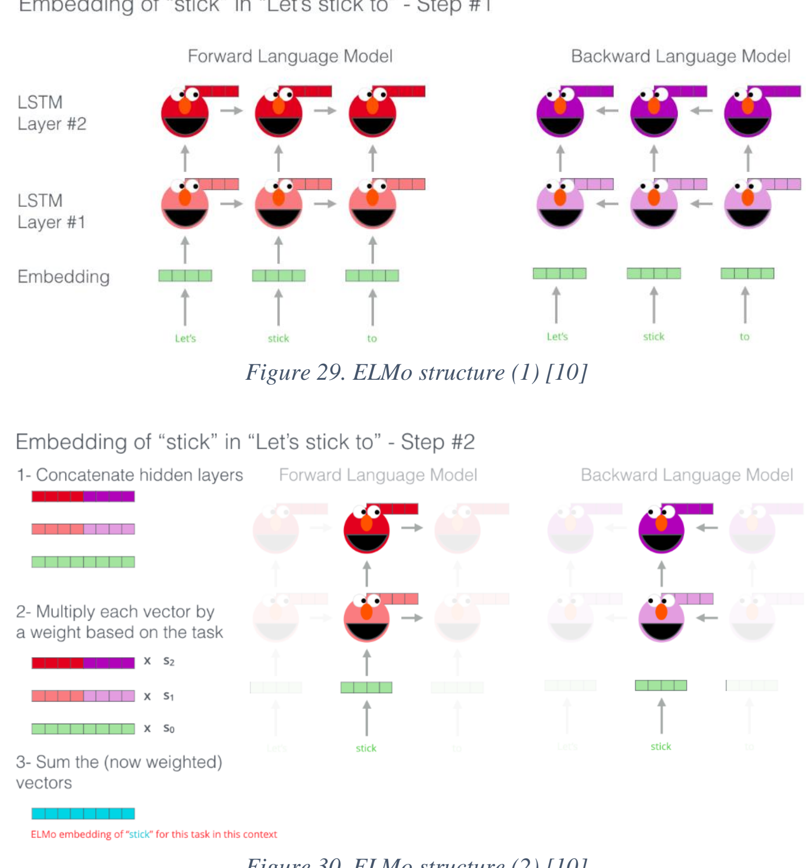


Figure 29. ELMo structure (1) [10]

Figure 30. ELMo structure (2) [10]

To produce a contextualized word-embedding for a single word, vectors produced by each LSTM layer are concatenated, multiplied by a weight, and summed together[10].

Later, a better alternative to LSTM appeared, called Transformers.

Transformers, as we have seen in previous sections, with their Encoder-Decoder structure are a perfect suit for machine translation. Thus, it could be a nice idea to try to adapt Transformers for other tasks, like sentence classification[10].

OpenAI Transformer does exactly this - tries to utilize the concepts offered by Transformers to produce a fine-tunable language model for NLP tasks.

OpenAI Transformer stacks twelve Transformer Decoders together. Decoders were picked for the model because they are a good fit for language modeling as they mask the future tokens while processing the input sequence[10].

These decoders do not have an encoder-decoder attention layer (as there are no Encoders in the model), however, a self-attention layer is still present.

With this structure, we can train the model on the language modeling task as it was in ELMo's case.

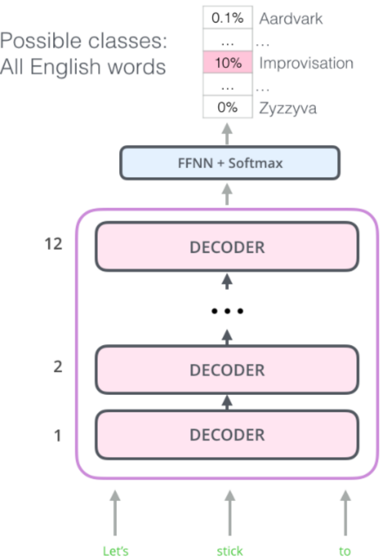


Figure 31. Open AI Transformer [10]

Having trained the model, we can use it for tasks other than language modeling. The paper, where the OpenAI Transformer was presented, shows necessary input transformations and model structures[10].

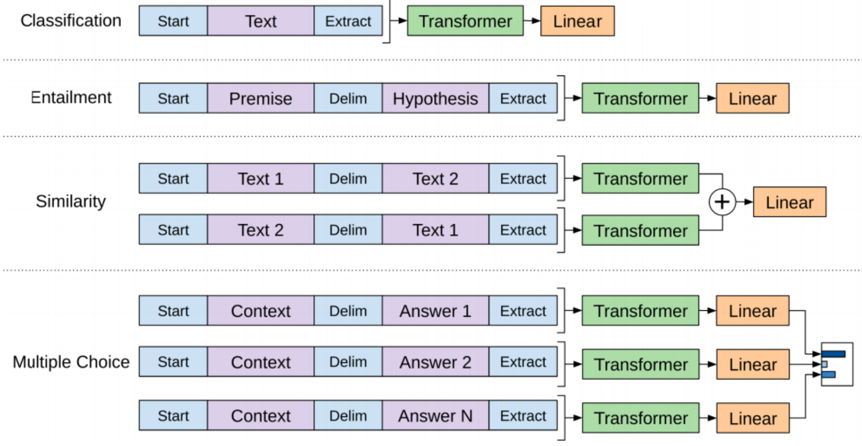


Figure 32. OpenAI Transformer modifications [10]

Thus, ELMo introduced a way to use bi-directional LSTM for generating contextualized word embeddings, and later, as a better alternative to the LSTM, OpenAI Transformer was introduced. BERT builds upon these concepts and offers a transformer-based model that uses both subsequent and previous words to produce embeddings.

To do this, BERT masks 15% of inputs tokens and tries to predict them by using the information derived from the other words in the sentence[10].

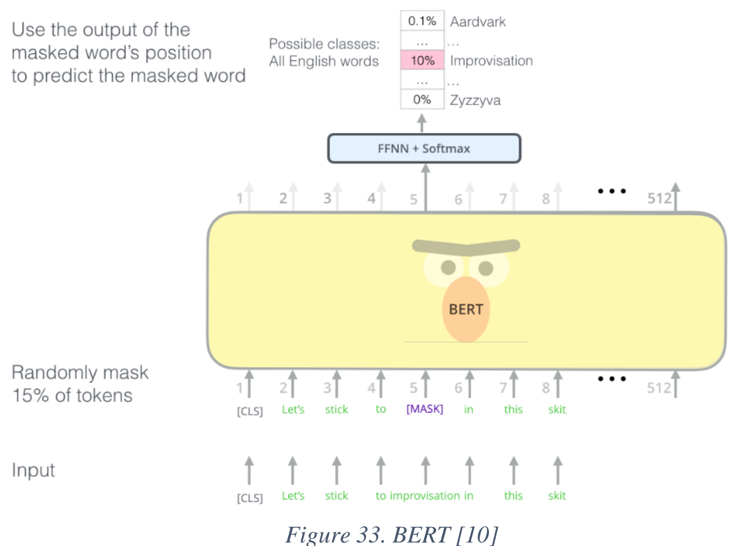


Figure 33. BERT [10]

3.3.4. BERT architecture

The paper, where BERT was presented, defines two versions of BERT[10]:

a) *BERT BASE* – similar in size to OpenAI Transformer;

b) *BERT LARGE* – huge model which achieved state-of-the-art results presented in the paper.

BERT is a trained Transformer Encoder stack. Both versions of the model have several layers (or Transformer Blocks, as it is stated in the paper) - the BASE version has twelve, and LARGE has twenty-four.

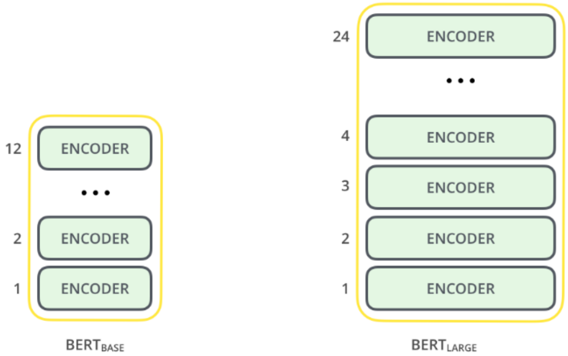


Figure 34. BERT BASE and BERT LARGE [10]

Apart from the usual Transformers, BERT BASE and BERT LARGE have larger feed-forward networks (768 and 1024 hidden units respectively), and more attention heads (12 and 16 respectively)[10].

Like the usual Encoder BERT receives a sequence of words as input with the first input token being a token called *[CLS]*, where CLS stands for Classification (Fig. 33).

To understand the purpose of this token, we should mention that BERT is trained to perform two tasks:

a) *Masked language modeling*, is when the model tries to predict the words masked with a [MASK] token during training;

b) *Next sentence prediction*, when given two sentences, the model learns to predict whether the second sentence is likely to be the sentence that might follow the first one. For this purpose, we use the [CLS] token. Its output will tell us how likely it is that the second sentence follows the first sentence.

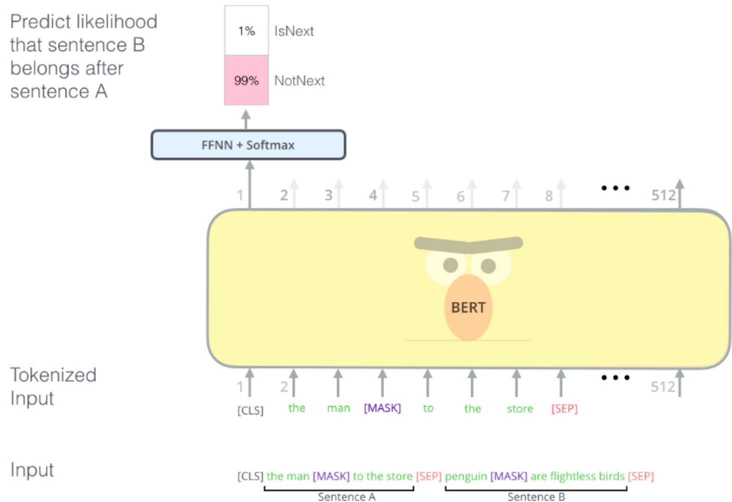


Figure 35. BERT next sentence prediction

BERT can be also adapted to solve other tasks, some of which are illustrated in the BERT paper[10].

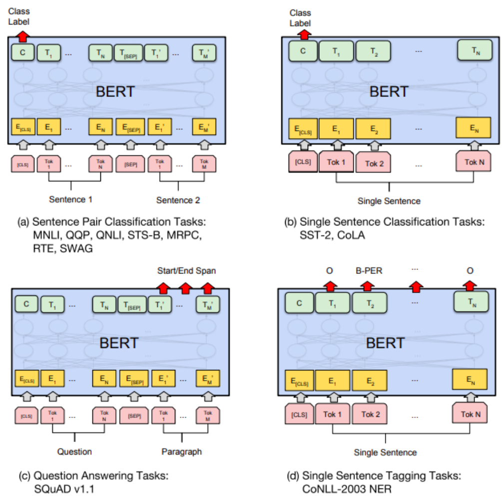


Figure 36. BERT modifications

4. Using Word2Vec and BERT for plagiarism detection

In this paper, we will look at external plagiarism detection in Ukrainian texts. External plagiarism detection involves comparing an input document with documents in the collection. We will use word embeddings to compare the documents, as texts with similar meaning or context produce similar word embeddings.

Using Word2Vec and BERT, we transform each document into a series of word embeddings. Calculating the similarity measure for these embeddings will help us determine whether the documents are similar or not.

Some of the most common ways to calculate vector similarity are *Euclidean distance* and *Cosine similarity*.

Cosine similarity calculates the cos of an angle between two vectors (Fig. 37) using the following formula:

$$\text{cosine similarity} = S_C(A,B) := \cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2}\sqrt{\sum\limits_{i=1}^{n} B_i^2}},$$

where A_i and B_i are the components of vectors A and B.

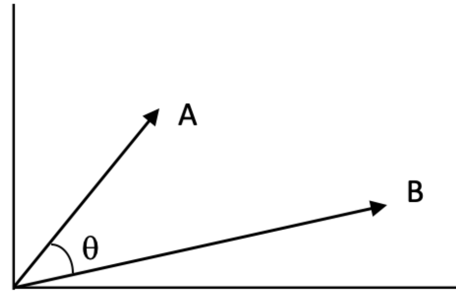


Figure 37. Cosine similarity

Cosine similarity belongs to the interval $[-1, 1]$. The more vectors are similar, the closer the value of cos (θ) will be to 1.

Euclidean distance calculates the distance between two vectors using the following formula:

$$d(\boldsymbol{a},\boldsymbol{b}) = d(\boldsymbol{b},\boldsymbol{a}) = \sqrt{\sum_{i=1}^{n}(b_i - a_i)^2}$$

Euclidean distance between two vectors can be illustrated in the following way:

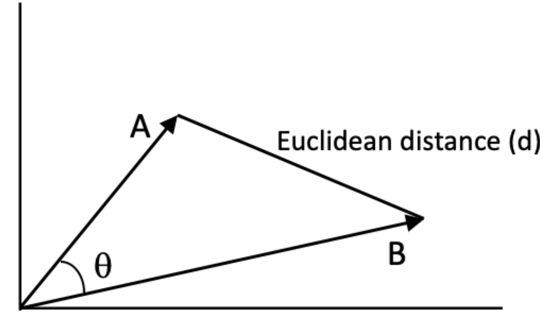


Figure 38. Euclidean distance

Thus, similar vectors will have a smaller distance between them.

5. Plagiarism detection in Ukrainian texts

5.1.Overview of BERT and Word2Vec implementations

Word2Vec and BERT can be trained to create embeddings for texts in an arbitrary language. However, the training of these models requires a large enough corpus of texts written in this language and a significant number of computational resources. Therefore, in this work, we will be using pre-trained Word2Vec and BERT models.

Word2Vec is provided by lang-uk, an open community of specialists in the field of computer word processing[17].

Word2Vec is available in three variants, each trained on a separate text corpus: fiction, news, or “ubercorpus”. Ubercorpus is a large 6GB corpus of texts from Ukrainian periodicals. The sizes and number of tokens in each corpus are provided in the table below[17].

Name	Number of tokens	Size (compressed)
Fiction	18323509	41 MB
News	461451019	1.1 GB
Ubercorpus	665419885	1.6 GB

For each text corpus, several pre-processing options are available:

a) Tokenized;

b) Tokenized and lowercased;

c) Tokenized and lemmatized;

d) Tokenized, lowercased, and lemmatized.

Every model produces a 300-dimensional embedding vector. In this work we will be using tokenized, lowercased, and lemmatized version of the Word2Vec trained on the Ubercorpus.

BERT is provided by the SentenceTransformers framework that features various BERT models for producing sentence, text, and image embeddings[18]. In this work we will be using the "paraphrase-multilingual-mpnet-base-v2" which is a multilingual model trained to generate sentence embeddings.

5.2. Application for plagiarism detection

Using the previously mentioned implementations of BERT and Word2Vec models, we can develop a program to detect plagiarism in Ukrainian texts. The UI of the completed web application can be seen in Figure 39.

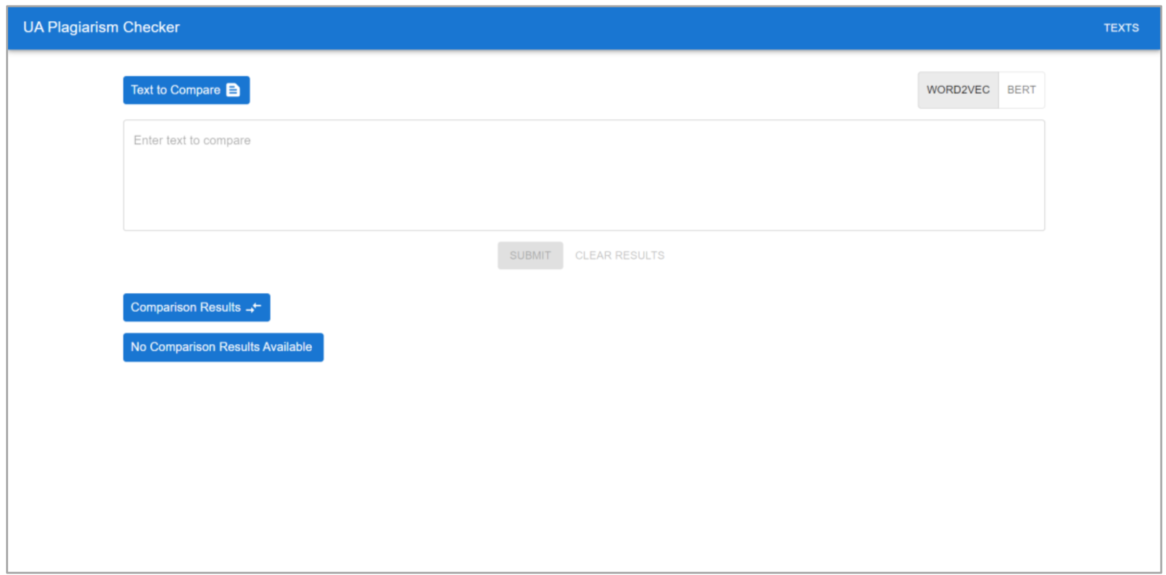


Figure 39. UA Plagiarism Checker

The application UI has been developed using *React*, a JavaScript library for building user interfaces[19]. React manages user interaction with the page, which are mostly button clicks and text input, and renders page *components*. Component – is an independent and reusable piece of code. Components are very similar to JavaScript functions: they receive arbitrary inputs and return React elements that describe what should be displayed on the page.

The web application uses a library of components called *Material UI*. Material UI offers many ready-made components for displaying inputs, buttons, text paragraphs, and other essential elements on the web page[20].

On the main page of the application, user can input a text that will be compared with the texts from the application's database for similarity. The comparison will be performed by generating word embeddings from the texts. Similarity of the embeddings is calculated using either cosine similarity or Euclidean distance.

Users can select the model to be used to generate word embeddings using the buttons in the upper right corner of the text input.

An example of text comparison can be seen below.

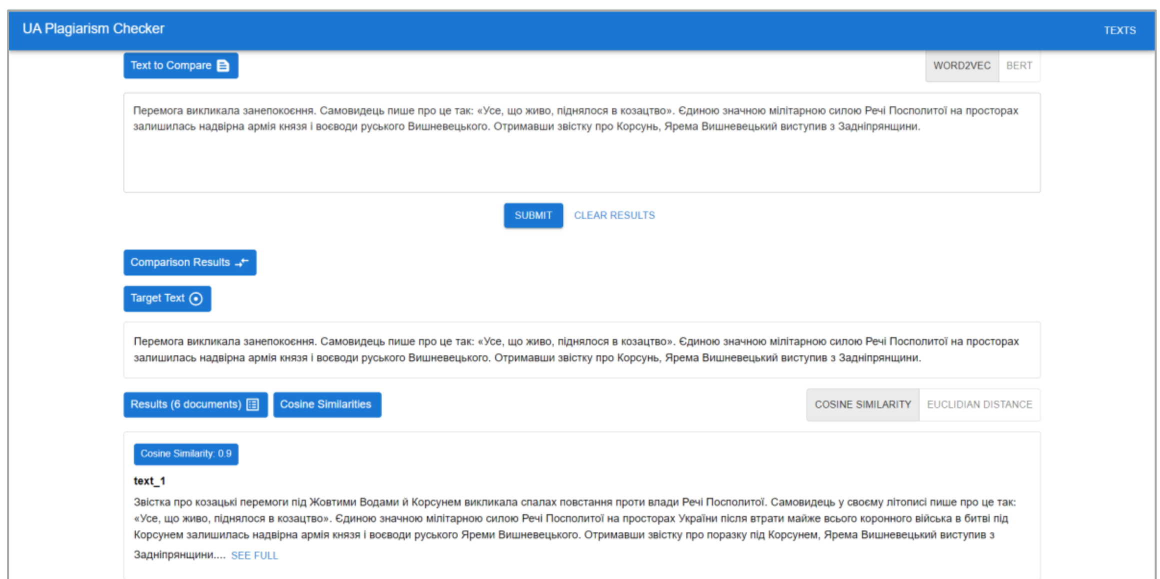


Figure 40. Text comparison results

The comparison metric can be changed using two buttons next to the “Results” label.

Texts database of the application can be modified by clicking on the “Texts” button in the top-right corner of the page. Clicking this button will open a new page with the ability to view existing texts, delete existing text or add a new text to the database (Fig. 41).

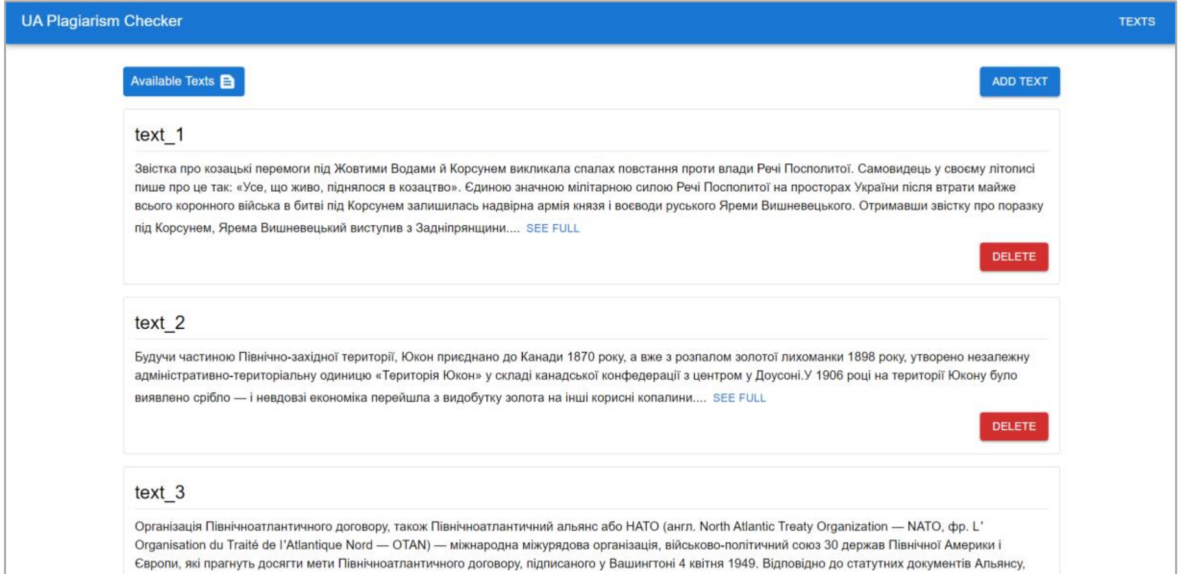


Figure 41. List of texts

The calculation of word embeddings and their comparison is performed on the application’s backend.

The backend is written using the Python programming language and *Flask*, a lightweight framework for developing APIs and web applications in general[21].

An endpoint called `compare_texts` is responsible for generating and comparing word embeddings.

```
@app.route('/compare/', methods=["POST"])
def compare_texts():
    # Extract data from the request
    raw_data = request.get_json()
    query_text = str(raw_data['text'])
    model = str(raw_data['model'])

    # Extract all documents from the DB
    cursor = ua_texts_collection.find({}).sort("title")
    documents = [UaText(**doc) for doc in cursor]

    # List of texts
    texts = [query_text] + [doc.text for doc in documents]
    # List of slugs
    slugs = [query_text] + [doc.slug for doc in documents]

    document_embeddings = []

    if model == 'word2vec':
        processed_texts = process_texts(texts)

        tfidfvectoriser = get_tf_idf_model(processed_texts, my_tokenizer)
        tokenizer = get_tokenizer(processed_texts)

        document_embeddings = get_document_embeddings(
            tfidfvectoriser, tokenizer, w2v_model, processed_texts)
    elif model == 'bert':
        unnorm_processed_texts = process_texts(texts, False)

        document_embeddings = sbert_model.encode(unnorm_processed_texts)
    else:
        return custom_error({'message': 'Model not found'}, 404)

    scores = get_model_scores(document_embeddings, slugs)

    scores['cosine_similarities']['similar'] = list(
        map(lambda item: list(map(str, item)), scores['cosine_similarities']['similar']))
    scores['euclidian_distances']['similar'] = list(
        map(lambda item: list(map(str, item)), scores['euclidian_distances']['similar']))

    return jsonable_encoder(scores, exclude_none=True)
```

Figure 42. compare_texts endpoint

This endpoint receives a request with the text to compare against other texts and a preferred model for calculating embeddings. After extracting this data from the request, we fetch all documents from the *MongoDB* database and create a list of texts and a list of *slugs*. In the context of this application, a slug is a hash produced by applying SHA256 algorithm to the text contents concatenated with random string, produced by *uuid* – a Python library for producing unique identifiers. Slugs are URL-safe and are used to identify documents when performing requests.

After initializing the lists, texts undergo preprocessing. For this purpose, the `process_texts` function is used.

This process includes dividing the text into sentences and sentences into tokens (words), lowercasing tokens, removing punctuation marks, any special characters, and stop words – common words whose removal usually does not affect

the meaning of the text. In the case of Word2Vec tokens also undergo *lemmatization*. Lemmatization aims to reduce the inflectional form of a word back to its root using morphological analysis[22].

After preprocessing stage, we generate text embeddings. This process varies depending on the model picked by the user.

Since Word2Vec can only generate embeddings for individual words, we need a way to calculate an embedding for the entire document from the embeddings of its words. One of the ways to do this, would be to multiply the embedding of each word by its *tf-idf weights* and then sum the resulting vectors[23].

tf-idf or term frequency-inverse document frequency assigns a weight to every word in the document. This weight depends on the number of occurrences of a word in the current document (or *tf* for short) and the number of documents that contain the given word (*idf* for short)[24].

$$idf_t = \log\left(\frac{N}{df_t}\right)$$

where N – is the number of documents in the collection, df_t – is the number of documents that contain term t.

Having tf and idf we can calculate tf-idf of a term t from the document d.

$$tf\text{-}idf_{t,d} = tf_{t,d} \times idf_t$$

The `get_document_embeddings` function calculates document embeddings for the Word2Vec using the process described above.

```
def get_document_embeddings(tfidfvectoriser, tokenizer, w2v_model, texts):
    tfidf_vectors = tfidfvectoriser.transform(texts)
    tfidf_vectors = tfidf_vectors.toarray()

    tokenized_documents = tokenizer.texts_to_sequences(texts)

    tokenized_paded_documents = tf.keras.utils.pad_sequences(
        tokenized_documents, padding='post')

    vocab_size = len(tokenizer.word_index) + 1
    embedding_matrix = np.zeros((vocab_size, 300))

    for word, i in tokenizer.word_index.items():
        if word in w2v_model:
            embedding_matrix[i] = w2v_model[word]

    document_word_embeddings = np.zeros(
        (tokenized_paded_documents.shape[0], tokenized_paded_documents.shape[1], 300))

    for i in range(tokenized_paded_documents.shape[0]):
        for j in range(tokenized_paded_documents.shape[1]):
            document_word_embeddings[i][j] = embedding_matrix[tokenized_paded_documents[i][j]]

    document_embeddings = np.zeros((tokenized_paded_documents.shape[0], 300))
    words = tfidfvectoriser.get_feature_names_out()

    for i in range(len(document_word_embeddings)):
        for j in range(len(words)):
            document_embeddings[i] += embedding_matrix[tokenizer.word_index[words[j]]
                                                       ] * tfidf_vectors[i][j]

    return document_embeddings
```

Figure 43. get_document_embeddings function

The `get_document_embeddings` function also uses a *tokenizer* to turn a document into a sequence of numbers, by assigning a unique identifier to each word in the document. This is done by using the `texts_to_sequences` function.

Number sequences are also padded to have equal lengths. This way, even if the input texts have different lengths, the resulting embedding vectors would have equal sizes.

Unlike Word2Vec, BERT can generate embedding for the whole document, so we only need to call the `encode` function.

After we have calculated embeddings, a `get_model_score` function is used to compare embeddings using cosine similarity and Euclidean distance.

```
def most_similar(text_id, slugs, similarity_matrix, matrix):
    similar = []
    target = slugs[text_id]

    if matrix == 'cosine':
        similar_ix = np.argsort(similarity_matrix[text_id])[::-1]
    elif matrix == 'euclidean':
        similar_ix = np.argsort(similarity_matrix[text_id])

    for ix in similar_ix:
        if ix == text_id:
            continue
        similar.append([slugs[ix], similarity_matrix[text_id][ix]])

    return {"target": str(target), "similar": list(similar)}


def compare_documents(document_embeddings):
    pairwise_similarities = cosine_similarity(document_embeddings)
    pairwise_differences = euclidean_distances(document_embeddings)
    return (pairwise_similarities, pairwise_differences)


def get_model_scores(document_embeddings, slugs):
    (pairwise_similarities, pairwise_differences) = compare_documents(
        document_embeddings)

    cosine_similarities = most_similar(
        0, slugs, pairwise_similarities, 'cosine')
    euclidian_distances = most_similar(
        0, slugs, pairwise_differences, 'euclidean')

    return {"cosine_similarities": cosine_similarities, "euclidian_distances": euclidian_distances}
```

Figure 44. get_model_score function

`get_model_score` function performs a pairwise comparison of embedding vectors using the `cosine_similarity` and `euclidean_distance` functions provided by the *scikit-learn* library.

The comparison results are used to sort the list of document slugs with respect to a specific document. The `most_similar` function is used for this purpose. Sorted slugs are then sent to the frontend and displayed to a user.

CONCLUSION

In this article we discussed the concept of plagiarism and listed its types. Two machine learning models have been proposed for plagiarism detection: Word2Vec and BERT. We also provided an overview of both models and described how they could be used in the problem of plagiarism detection.

A web application for plagiarism detection has been developed. This application features React, a JavaScript framework, on the frontend and Python on the backend. To store application data, MongoDB is used.

This application allows a user to input a text that will be compared with the texts from the application database using cosine similarity or Euclidean distance as metrics. Comparison is performed using word embeddings, calculated by pre-trained BERT or Word2Vec model. A user can choose the model and similarity metric using the application's UI.

The application can be further improved to not only output similarity metric but also highlight the similar sentences in the texts. The ability to save comparison results can also be added.

REFERENCES

1. A Deep Learning Approach to Persian Plagiarism Detection / Erfaneh Gharavi, Kayvan Bijari et Kiarash Zahirnia – 2011 – Journal of Machine Learning Research.
2. 7 common types of plagiarism explained / Jessica Malnik [Electronic resource] – https://writer.com/blog/types-of-plagiarism/
3. A deep learning-based technique for plagiarism detection: a comparative study / El Mostafa Hambi, Faouzia Benabbou – 2020 – International Journal of Artificial Intelligence, Vol. 9, No. 1.
4. Word2Vec Explained / Julian Gilyadov [Electronic resource] – https://israelg99.github.io/2017-03-23-Word2Vec-Explained/
5. The Illustrated Word2vec / Jay Alammar [Electronic resource] – https://jalammar.github.io/illustrated-word2vec/
6. A simple Word2vec tutorial / Zafar Ali [Electronic resource] – https://medium.com/@zafaralibagh6/a-simple-word2vec-tutorial-61e64e38a6a1
7. word2vec [Electronic resource] – https://code.google.com/archive/p/word2vec/
8. All about Embeddings / Kashyap Kathrani [Electronic resource] – https://medium.com/@kashyapkathrani/all-about-embeddings-829c8ff0bf5b
9. BERT Explained: State of the art language model for NLP / Rani Horev [Electronic resource] – https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270
10. The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning) / Jay Alammar [Electronic resource] – https://jalammar.github.io/illustrated-bert/
11. Transformer Neural Network / Thomas Wood [Electronic resource] – https://deepai.org/machine-learning-glossary-and-terms/transformer-neural-network
12. The Illustrated Transformer / Jay Alammar [Electronic resource] – https://jalammar.github.io/illustrated-transformer/
13. Transformers Explained Visually — Not Just How, but Why They Work So Well / Ketan Doshi [Electronic resource] – https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3
14. Transformers Explained Visually (Part 2): How it works, step-by-step / Ketan Doshi [Electronic resource] – https://towardsdatascience.com/transformers-explained-visually-part-2-how-it-works-step-by-step-b49fa4a64f34

15. Transformers Explained Visually (Part 3): Multi-head Attention, deep-dive / Ketan Doshi [Electronic resource] – https://towardsdatascience.com/transformers-explained-visually-part-3-multi-head-attention-deep-dive-1c1ff1024853
16. Transformers Explained Visually (Part 1): Overview of Functionality / Ketan Doshi [Electronic resource] – https://towardsdatascience.com/transformers-explained-visually-part-1-overview-of-functionality-95a6dd460452
17. lang-uk: Models [Electronic resource] – https://lang.org.ua/uk/models/
18. SentenceTransformers [Electronic resource] – https://www.sbert.net/
19. React [Electronic resource] – https://reactjs.org/
20. MUI [Electronic resource] – https://mui.com/
21. Flask [Electronic resource] – https://palletsprojects.com/p/flask/
22. What is the difference between stemming and lemmatization? [Electronic resource] – https://blog.bitext.com/what-is-the-difference-between-stemming-and-lemmatization/
23. Calculating Document Similarities using BERT, word2vec, and other models / Varun [Electronic resource] – https://towardsdatascience.com/calculating-document-similarities-using-bert-and-other-models-b2c1a29c9630
24. The quantitative value of text, tf-idf and more… / Varun [Electronic resource] – https://medium.com/analytics-vidhya/the-quantitative-value-of-text-tf-idf-and-more-e3c7883f1df3