DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2021

Evaluating semantic similarity using sentence embeddings

JACOB MALMBERG

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Evaluating semantic similarity using sentence embeddings

JACOB MALMBERG

Master in Computer Science
Date: February 12, 2021
Supervisor: Johan Boye
Examiner: Joakim Gustafson
School of Electrical Engineering and Computer Science
Host company: Peltarion AB
Swedish title: En utvärdering av metoder som använder vektorrepresentationer för att avgöra semantisk likhet mellan meningar

Abstract

Semantic similarity search is the task of searching for documents or sentences which contain semantically similar content to a user-submitted search term. This task is often carried out, for instance when searching for information on the internet.

To facilitate this, vector representations referred to as embeddings of both the documents to be searched as well as the search term must be created. Traditional approaches to create embeddings include the term frequency - inverse document frequency algorithm (TF-IDF). Modern approaches include neural networks, which have seen a large rise in popularity over the last few years. The BERT network released in 2018 is a highly regarded neural network which can be used to create embeddings. Multiple variations of the BERT network have been created since its release, such as the Sentence-BERT network which is explicitly designed to create sentence embeddings.

This master thesis is concerned with evaluating semantic similarity search using sentence embeddings produced by both traditional and modern approaches. Different experiments were carried out to contrast the different approaches used to create sentence embeddings. Since datasets designed explicitly for the types of experiments performed could not be located, commonly used datasets were modified.

The results showed that the TF-IDF algorithm outperformed the neural network based approaches in almost all experiments. Among the neural networks evaluated, the Sentence-BERT network performed proved to be better than the BERT network. To create more generalizable results, datasets explicitly designed for the task are needed.

Sammanfattning

Semantisk likhets-sökning är en typ av sökning som syftar till att hitta dokument eller meningar som är semantiskt lika en användarspecifierad sökterm. Denna typ av sökning utförs ofta, exempelvis när användaren söker efter information på internet.

För att möjliggöra detta måste vektorrepresentationer av både dokumenten som ska genomsökas såväl som söktermen skapas. Ett vanligt sätt att skapa dessa representationer har varit term frequency - inverse document frequency-algoritmen (TF-IDF). Moderna metoder använder neurala nätverk som har blivit mycket populära under de senaste åren. BERT-nätverket som släpptes 2018 är ett väl ansett nätverk som kan användas för att skapa vektorrepresentationer. Många varianter av BERT-nätverket har skapats, exempelvis nätverket Sentence-BERT som är uttryckligen skapad för att skapa vektorrepresentationer av meningar.

Denna avhandling ämnar att utvärdera semantisk likhets-sökning som bygger på vektorrepresentationer av meningar producerade av både traditionella och moderna approacher. Olika experiment utfördes för att kontrastera de olika approacherna. Eftersom dataset uttryckligen skapade för denna typ av experiment inte kunde lokaliseras modifierades dataset som vanligen används.

Resultaten visade att algoritmen TF-IDF överträffade approacherna som var baserade på neurala nätverk i nästintill alla experiment. Av de neurala nätverk som utvärderades var Sentence-BERT bättre än BERT-nätverket. För att skapa mer generaliserbara resultat krävs dataset uttryckligen designade för semantisk likhets-sökning.

Contents

Chapter 1

Introduction

A substantial amount of data is produced daily in different organizations. For instance, more than 2.5 billion pieces of content and 500 terabytes of data are processed every day at Facebook [1]. To sift through this data efficiently, artificial intelligence and machine learning is needed [2].

In the last decade, machine learning has seen a large rise in popularity. In particular, neural networks have been successfully been applied to a number of domains such as computer vision and language [3].

Particularly in the area of language and text, the state-of-the-art advancement has been large in recent years. Devlin et al. published BERT [4] in 2018, a large neural network engineered specifically for understanding text in a contextualized setting. By 2019, Google was making use of BERT in about 10% of Google searches. In 2020, BERT was used for almost all English based searches [5].

1.1 Background

As experienced by Peltarion, the textual format is one of the most common types of data formats in real-world industrial settings. Peltarion applies many different AI prediction tasks to text, such as sentiment analysis, translation, summarization, and question answering. However, according to Peltarion, one of the most useful prediction tasks is semantic similarity search. The task of semantic similarity involves searching for documents that are semantically similar to a user-submitted sentence or query. For example, when a user searches for documents using the search term 'sports' the user may want to be presented with documents that contain the search term but also documents that do not contain the search term but instead contain terms that share some semantic

similarity to the search term, such as the word 'soccer'.

To facilitate semantic similarity, documents and sentences must be represented in a suitable way. There are many algorithms to produce document representations, and one of the most popular is TF-IDF, i.e., term frequency / inverse document frequency. This algorithm calculates how a weight representing the importance of each word compared to a collection of documents. The output is a vector for each word, which can be aggregated to create documents vectors, representing a set of words [6].

Further, neural networks, such as BERT, can be used to produce document vectors. BERT takes words for input and produces a vector for each word. To create document vectors, the word vectors belonging to the document are aggregated.

Peltarion has experimented with numerous techniques to provide semantic search solutions in various projects. However, there is a clear need to scientifically evaluate these techniques and provide recommendations on applying semantic similarity for different datasets and applications. Therefore, this study will compare the algorithms mentioned above regarding their ability to produce representations that capture semantic similarity, specifically in the context of ranking semantic similarity predictions. Previous work has evaluated semantic similarity for several classification and regression tasks, such as in the STS-B and MRPC datasets. However, little research has been performed in a ranking context where a set of sentences are compared to a single query and ranked according to their semantic similarity.

1.2 Research Question

Given the recent breakthroughs in NLP with the advent of BERT like models, this study is guided by the hypothesis that such models produce representations that are better suited for text search applications than TF-IDF models. Specifically, the research question explored in this thesis is:

- Can BERT-like models produce embeddings that are better suited for semantic similarity applications than TF-IDF models?

1.3 Ethical and societal aspects

The results of this master thesis may lead to companies choosing models which are better suited for semantic similarity applications than they were previously

using. This should lead to users receiving search results more semantically similar to their search term. The societal aspects of this are that information is spread more efficiently throughout society. However, one ethical downside to this is that it includes information that can be used to harm other human beings. Thus, while algorithms better suited for semantic similarity applications should produce better search results they also improve users' ability to find information which can be used for harmful purposes.

1.4 Disposition

Chapter 2 covers the theoretical background and gives an in-depth explanation of the algorithms used. Chapter 3 explains the methodology and tools used in this thesis. The results are presented in chapter 4 followed by a discussion of the results in chapter 5. Lastly, the thesis is concluded in chapter 6.

Chapter 2

Background

2.1 Natural language processing

Natural language processing (NLP) is a discipline within artificial intelligence, linguistics, and computer science. The field started in the 1950s, and early approaches relied on hand-crafted rules to parse text. One of the problems attacked include Russian-to-English machine translation [7].

Since the 1980s, NLP has been dominated by a statistical and probabilistic approach. Statistical models leverage the large collections of textual data, with the models being more accurate when fed more data [7]. Datasets containing large collections of documents, so-called corpora, are therefore central to the field. Common corpora include the Stanford Question Answering Dataset [8], consisting of Wikipedia articles with questions attached to each article, as well as the Reuters-21578 corpus [9], consisting of news articles from Reuters news wire.

There are a variety of tasks solved within modern NLP. These tasks include named entity recognition (NER), which involves identifying and classifying named entities, such as persons and organizations, as well as question answering, which involves finding an answer to a question within a span of text [7].

To build statistical models, algorithms are needed. Algorithms often parse large amounts of data during a training phase to build a model. The model is then used to make predictions about unseen data. Models historically used within NLP include Hidden Markov Models (HMM) and Support Vector Machines (SVM) [7]. Recently, neural algorithms such as the BERT network has been used for a variety of tasks [4].

To solve tasks within NLP, vector space models are often used. A docu-

ment vector can be thought of as a vector representation of a document, where the term 'document' can refer to any set of words. Document vectors are sometimes referred to as document embeddings. This vector does not contain any information of word order, and can therefore be thought of as a bag-of-words [10]. In this thesis, both modern neural network approaches as well as more traditional approaches will be used to create document vectors.

2.2 Term Frequency - Inverse Document Frequency

Term frequency - inverse document frequency (TF-IDF) is commonly used to create document vectors [6]. The algorithm does not take word order into account. Instead, a bag-of-words approach is used where each term receives a weight corresponding to its frequency in a document compared to the inverse frequency in the corpus. This assigns large weights to terms appearing infrequently in the corpus but often in a certain document, which potentially are more representative of the document than more common words [11].

To create TF-IDF vectors, first the term frequencies of the terms in the corpus need to be calculated. Let D be the documents in the corpus and V be the vocabulary of the words appearing in the corpus. The term frequency can then be determined as

$$tf(t, d) = f_{t,d} \tag{2.1}$$

where $t \in V$ is the term, $d \in D$ is the document, and $f_{t,d}$ is the number of times t appears in d.

The inverse document frequency of the term can then be calculated as

$$idf(t, N) = log\left(\frac{N}{1 + n_t}\right) + 1 \tag{2.2}$$

where N is the number of documents in the corpus, and n_t is the number of documents in the corpus containing the term.

The TF-IDF can then be calculated as

$$tfidf(t, d) = tf(t, d) * idf(t, D) \tag{2.3}$$

for each term and document in the corpus.

The above equations are repeated for each document and term in the corpus, creating document vectors v. The vectors have a fixed dimensionality and are normalized according to equation 2.4 [12]:

$$v_{norm} = \frac{v}{||v||_2}. \tag{2.4}$$

2.3 Cosine similarity

Cosine similarity is defined as the inner product of two vectors divided by the product of their length. Cosine similarity is defined as

$$cosine\_sim(a, b) = \frac{\sum_{i=1}^{N} a_i b_i}{\sqrt{\sum_{i=1}^{N} a_i^2}\sqrt{\sum_{i=1}^{N} b_i^2}} \tag{2.5}$$

where vectors a and b have the same number of dimensions N [13]. Cosine similarity can be used to compare similarity between document vectors [14].

2.4 Neural networks and deep learning

While neural networks have existed since the 1960s, significant progress has been made over the last decade. Reasons for this progress include more and better designed datasets, advances in modern computing hardware, and better algorithms [3].

Neural networks are used in a variety of applications in different domains today. In computer vision, neural networks have been successfully used to detect objects. Translating tasks are solved within the field of natural language processing [3].

A neural network consists of multiple neurons which receive inputs and compute an activation value. Neurons which receive input from the environment are referred to as input neurons. Non-input neurons in the network receive their through weighted connections from other neurons in the network. Finally, output neurons produce a final output often referred to as a prediction [3].

There are different architectural styles of neural networks, which contain different mechanisms and arrange their neurons in different ways. One simple neural network is the fully connected feed-forward neural network (FFNN), which arranges its neurons in layers, displayed in figure 2.1 [16]. More complicated architectures include the Transformer architecture, which contains 'encoder' modules that processes information about how different words in a sentence relate to each other. An 'encoder' module further contains different sub-modules, such as an FFNN [17]. In neural network literature [17, 4],

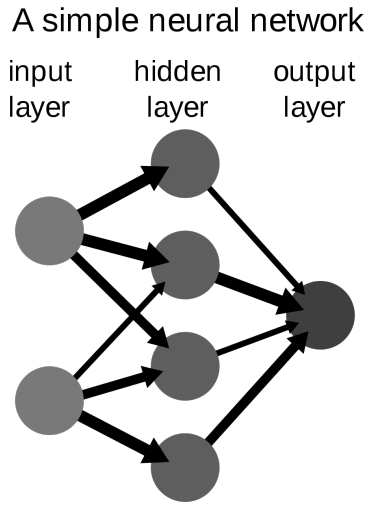

Figure 2.1: A simple neural network [15].

such modules and sub-modules are often referred to as layers although they are not equivalent to a layer in a FFNN.

Neural networks can be tasked to solve specific problems, and different architectures fit different tasks. Common tasks to be solved include image classification, i.e. discerning cats from dogs, and translating text between languages. A FFNN may be able to solve a simple image classification task, but may not be able to efficiently translate text [3]. For text and language related tasks, a Transformer-esque architecture is often a good fit because of its encoder layers [17].

A neural network needs to be trained for the task it is tasked to solve. To train a network data is needed as well as an algorithm which can calibrate the parameters, such as the neurons or the encoders, of the network. The data is fed into the network to produce predictions which are used to calibrate the network [16].

Deep learning refers to a neural network with a large number of layers. The network depicted in figure 2.1 would not qualify as a deep neural network since it does not have a large number of layers and would instead be referred to as a shallow neural network. Complex tasks, such as translating text, typically require a larger number of layers to be successfully carried out [16].

2.4.1 Feedforward neural networks

Feedforward neural networks approximate a function $f^*(x)$. This is commonly done by constructing a parameterized function

$$y = f(x; \theta) \tag{2.6}$$

where x is an input vector and θ are the network parameters. θ can be randomly initialized. The network then learns the parameters θ that result in the best approximation of the function $f^*(x)$. x is accompanied by a label vector y which is used to evaluate the predictions of the network during training. This type of training is referred to as supervised learning [16].

A network is composed of multiple different function as

$$f(x) = f^{(1)} \circ f^{(2)} \circ f^{(3)} \tag{2.7}$$

where $f^{(1)}$ is commonly referred to as the first layer, $f^{(2)}$ is referred to as the second layer, and so forth. While the first layer is referred to as the input layer and the last layer is referred to as the output layer, the intermediate layers are referred to as hidden layers. In equation 2.7, $f^{(2)}$ is a hidden layer. The depth of the network is given by the number of layers [16]. Figure 2.1 displays a simple neural network.

In a fully connected neural network, each function $f^{(i)}$in equation 2.7 consists of a linear transformation followed by a non-linear transformation. The linear function for layer i can be written as

$$s_i(x) = W_i x + b_i \tag{2.8}$$

where W_i is a weight matrix and b_i is an intercept. W_i and b_i are commonly referred to as θ_i and are the learnable parameters in layer i. s_i can be referred to as the activations vector of layer i. A non-linear tranformation, a so-called activation function, is then applied component wise to $s_i(x)$. Commonly used activation functions include ReLU, $z(x) = max(0, x)$, and sigmoid, $z(x) = \frac{1}{1+e-x}$. $z(x)$ is then used as input to $f^{(i+1)}$. The output of the activation function for layer i can be referred to as the output vector of layer i [16].

θ is the set of all θ_i, where i is the number of layers in the network. To find the optimal values of the parameters θ, the network is trained by feeding data through it and updating the parameters so that it better fits the desired output [16].

The network can have a probabilistic output. This can be achieved by using the softmax function

$$softmax(z)_i = \frac{exp(z_i)}{\sum_j exp(z_j)} \tag{2.9}$$

where z_i is each element in the last layer of the network. The softmax function returns the probabilities of the data point belonging to each class [16]. For instance, this can be useful when the network is constructed to perform a classifying task.

2.4.2 Training

Data

As stated above, the parameters of the network are randomly initialized. Training the network is therefore needed to find optimal parameter values. Supervised learning requires a labelled dataset $D = \{x_i, y_i\}_i^n$, where D is the dataset consisting of n data points (x_i, y_i) commonly referred to as input vectors. x_i is the vector inputted into the neural network and y_i is the accompanying label. The label is a scalar which contains information about which class the data point belongs to. Labels are needed to evaluate the quality of the predictions the network produces, and to improve the network. The dataset is commonly split into training, validation, and test tranches. The training dataset is used to train the network, the validation dataset is used to validate the training and to ascertain that the network has not been over-fitted to the training set, and the test set is used to achieve a final score. Over-fitting and validation are explained further in section 2.4.2 [16].

Loss function

Predictions are produced by inputting input vectors to the network which are propagated forward through the network to create output vectors. This is sometimes referred to as a forward pass. To evaluate the predictions produced, a loss function is used. Cross-entropy loss is commonly used and is defined as

$$l_{cross}(\hat{y}, y) = -\sum_{i=1}^{C} t_i log(\hat{y}_i) \tag{2.10}$$

where C is the number of classes in the dataset, $\hat{y}$ is a vector which contains the predicted probabilities over the classes for the data point, y is the label for

the data point, t_i is a binary indicator which indicates if i is the correct class for the data point, and $\hat{y}_i$ is the predicted probability of the data point belonging to class i. During training, the goal is to find the parameter values that optimizes this loss function [16]. This can be formulated as:

$$\arg\min_{\theta} \frac{1}{|D|} \sum_{(x,y)\in D} l(\hat{y}, y) \tag{2.11}$$

Over-fitting and regularization

While the network may achieve a low loss score during training, its performance on data not in the training dataset may be worse. If the network's performance is considerably better on the training dataset than on unseen data, the network has been over-fitted to the training data. An over-fitted network is said to generalize poorly if it cannot predict well on data it has not encountered during training. [16].

The validation set can be used to verify if the network has been over-fitted to the training data. The network is first trained on the training data, and its generalization performance is evaluated on the validation data. If the generalization performance of the network is poor, the network can be modified and re-evaluated on the validation data [16].

To remedy over-fitting, the network can be regularized. Regularization is a modification to the network, or other learning algorithm, that is intended to improve its ability to generalize while not decreasing its performance on training data. One such modification is dropout, which is explained in section 2.4.3 [16].

Gradient descent and back-propagation

Optimization of the network can be done using a technique called gradient descent. Essentially, the technique involves calculating the gradient of the loss function with respect to θ and then moving, i.e. changing the parameter values of θ, in small steps in the direction of the negative gradient direction. This step size is normally referred to as the network's learning rate and is commonly denoted as η. This process is repeated for every data point in the training dataset. A more efficient variant of gradient descent is mini-batch gradient descent, which involves randomly splitting the data points into batches, and then performing one model parameter update per batch [16].

Gradient descent requires the calculation of the gradient of the loss function with respect to the weights of the network. The algorithm commonly used

for calculating the gradient is called back-propagation, since it allows for the information from the loss function to propagate through the network to compute the gradient [16].

Recall from equation 2.7 that a neural network is a composition of functions. Back-propagation is an algorithm that computes the differentiation chain rule for a composition of an arbitrary number of functions. Algorithm 1 displays the back-propagation algorithm for a neural network with an arbitrary number of layers [16].

Algorithm 1: Back propagation

Input: Neural network θ with k layers, with $k > 0$; output vectors x_k for each layer k with x_0 being the input vector to the network; label vector y; vector $\hat{y}$, the network's predicted probabilities over the classes for input vector x_0; activation vectors a_k for each layer k; activation function f; loss function $l(\hat{y}, y)$.

compute gradient g on the output layer:

$g \leftarrow \nabla_{\hat{y}} l(\hat{y}, y)$

for *i = k, k-1,...,1* **do**

$g \leftarrow \nabla_{a_i} l = g \odot f'(a_i)$, where $\odot$ is element-wise multiplication.

Compute the gradient of the loss function with respect to the layer's bias vector b_i:

$\nabla_{b_i} l = g$

Compute the gradient of the loss function with respect to the layer's weight matrix W_i:

$\nabla_{W_i} l = g^T x_{(i-1)}^T$

if *i > 1* **then**

Propagate the gradient vector g to the previous layer:

$g \leftarrow \nabla_{x_{i-1}} l = W_i^T g$

else

end

end

The weights and biases are then updated by taking a step proportional to the negative gradient direction. To update layer k's biases equation

$$b_k = b_k - \eta * \nabla_{b^k} L \tag{2.12}$$

is used and to update layer k's weight matrix equation

$$W_k = W_k - \eta * \nabla_{W^k} L \tag{2.13}$$

is used [16].

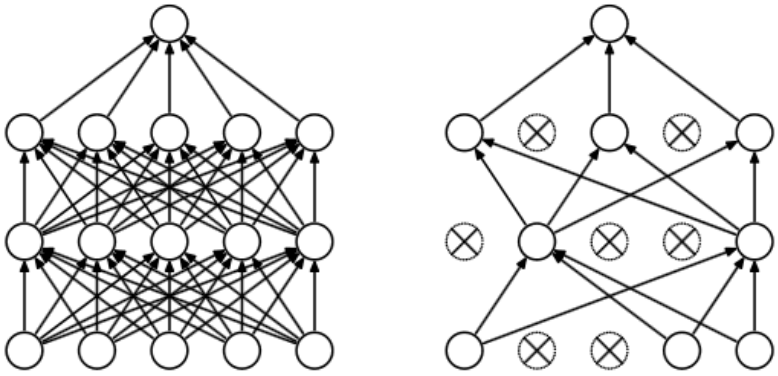

Figure 2.2: Dropout [18].

The training process is iterated until some criteria have been met. Such criteria include a stagnating loss value, as well as the loss value of the training dataset and the loss value of the validation dataset diverging [16].

2.4.3 Dropout

Dropout is a technique which provides a computationally inexpensive way of regularizing a neural network. The technique involves multiplying the output of random neurons with zero, disabling them and "dropping" them out during the training phase. Effectively, dropout randomly samples from a number of different pruned networks. This prevents the network from over-fitting to the training data. After training and during inference, that is when the network is tasked with predicting output from input vectors not seen during training, these pruned networks are approximated by using a single unpruned network with smaller weights. A conceptual figure of dropout is displayed in 2.2 [18].

2.5 Recurrent neural networks

Recurrent neural networks (RNN) are a family of neural networks specialized for processing a sequence of values. The sequences can be of variable length. Instead of having hidden states as in the case of vanilla neural networks, an RNN contains state vectors which can be described as a summary of the past sequence of inputs. RNNs have been successfully used in applications involving images as well as text [16].

RNNs builds on the idea of parameter-sharing from the 1980s. Instead of

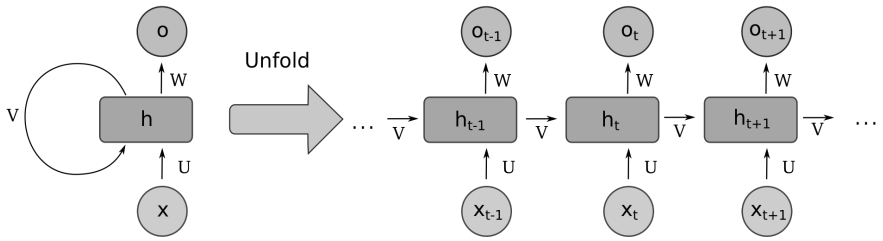

Figure 2.3: (Left) Circuit diagram of an RNN. (Right) The same RNN depicted as an unfolded graph [19].

applying different parameters to each position in the sequence, it applies the same parameters to each position. Consider the examples "I started school in 2015" and "In 2015 I started school". If a fully-connected network was tasked with extracting the year I started school, it would have to learn the rules of language at each position separately since the network has different parameter values at each position. An RNN instead shares parameter values across all positions, or time-steps, which allows it to generalize better [16].

RNNs typically consist of an input vector x_t for each timestep in the sequence, a state vector h_t for each timestep, an initial state vector h_0, an activations vector a_t for each timestep, an activation function f, an output vector o_t for each timestep, as well as a probability vector $\hat{y}_t$ for each timestep. The network further has a vocabulary of output tokens, which for instance can be characters or words. These tokens can be thought of as classes and $\hat{y}_t$ contains probabilities of the output of timestep t being one of these tokens. For each timestep $t = 1, 2, ..., \tau$ in the sequence these vectors are calculated as

$$a_t = b + Vh_{t-1} + Ux_t \tag{2.14}$$

$$h_t = f(a_t) \tag{2.15}$$

$$o_t = Wh_t + c \tag{2.16}$$

$$\hat{y}_t = softmax(o_t) \tag{2.17}$$

where U, V, and W are learnable weight matrices and b and c are learnable bias vectors. The optimal values of the learnable parameters can be found through gradient descent and back-propagation [16].

To visualize an RNN, two viewpoints are often offered. The first viewpoint is the network as a circuit diagram which clearly displays the circular nature

of an RNN. The second viewpoint is referred to as an unfolded graph where each node is associated with one time step. Both viewpoints are displayed in figure 2.3.

While regular RNNs can model sequences, they are not well suited for problems involving long-term dependencies. The difficulty with long-term dependencies stems from the phenomenon of vanishing or exploding gradients. The basic idea is that gradients propagated through the network often vanish, shrink to zero, or explode, growing very large. This can lead to the weights in the network not being updated, or that the weights will have large updates, leading to inconsistent performance [16].

2.6 The Transformer

The Transformer neural network was introduced in 2017 as an attempt to alleviate some of the problems associated with RNNs. The input to the current step in an RNN includes the hidden state from the previous timestep, which creates a sequential nature that hinders parallelization of the calculations. To remove this hindrance, the Transformer does not contain recurrent connections. The original Transformer network was trained using a translation task. Sentences were translated to French from English, and to German from English [17].

The Transformer takes a sequence of tokens, which can be words but can also be parts of words, as input. In the original setting the Transformer produces a translation of those tokens. Turning words into tokens included in the model vocabulary helps the model deal with rare words at inference time that it may not encounter during training and thus is not part of the vocabulary.

To split words into tokens, the Transformer uses an algorithm called byte-pair encoding (BPE) [17]. First, the frequencies of all the words in the corpus are counted. Second, the words are turned into sequences of characters and a special "</w>" token is appended to each sequence to create an initial vocabulary. Third, the frequency of all character pairs in the vocabulary are counted. Fourth, every occurrence of the most frequent pair is merged into a new token and added to the vocabulary. This procedure is then repeated to create a vocabulary of desired size [20]. For example, using byte-pair encoding trained on the WikiText Long Term Dependency Language Modeling Dataset [21] with a vocabulary of 32000 tokens, the sentence "i am the walrus" turns into "i</w> am</w> the</w> wal rus</w>", with "</w>" marking the end of each word for easy decoding. The words "i", "am, "the", are turned into tokens while the word "walrus" is turned into the tokens "wal" and "rus".

The token embeddings are then initialized to create input embeddings with 512 dimensions. However, these input embeddings do not contain any information about the order of tokens in the sequence. The Transformer therefore adds positional encodings to each input embedding [17].

The Transformer network consists of an encoding component as well as a decoding component. In this thesis, only the encoding component is used and thus this theory section will focus on the encoder. The encoder stack maps a sequence of input tokens to a sequence of embeddings. The embeddings are then decoded by the decoder stack to generate a sequence of output tokens [17].

Both the encoding component and the decoding component consists of 6 stacked encoder/decoder layers. The stacked encoders have the same structure, but do not share weights. The inputs to the first encoder in the stack are the input embeddings, and these input embeddings are transformed into output embeddings which are inputted into the next encoder layer. The inner workings of an encoder layer are explained in detail below. The output embeddings of the last encoder layer are then inputted into the decoder stack [17]. The Transformer stack is displayed in figure 2.4.

An encoder layer is composed of two sub-layers: the first sub-layer consists of multiple self-attention units, referred to as self-attention heads, and the second sub-layer is a feedforward network. These two sub-layers have residual connections around them, which are detailed in section 2.6.3, and each sub-layer is followed by layer normalization, which is explained in 2.6.4. An illustration of an encoder layer is displayed in figure 2.5. The components are explained in detail below.

2.6.1 Self-attention

Self-attention is a mechanism which allows the model to relate different positions in the input sequence to each other. For example, this enables the network to understand what the word "it" refers to in the tokenized sequence "the</w> dog</w> didn ’ t</w> eat</w> my</w> hom ework</w> because</w> it</w> was</w> happy</w>". As the network processes each position in the input sequence, the self-attention mechanism allows it to relate the position to each other position in the sequence which creates better encoding. In the Transformer the specific variant of attention is called scaled dot-product attention since it involves calculating dot-products as well as scaling the dot-products (dividing by a factor). [17].

To calculate self-attention, a query and a set of key-value pairs are mapped

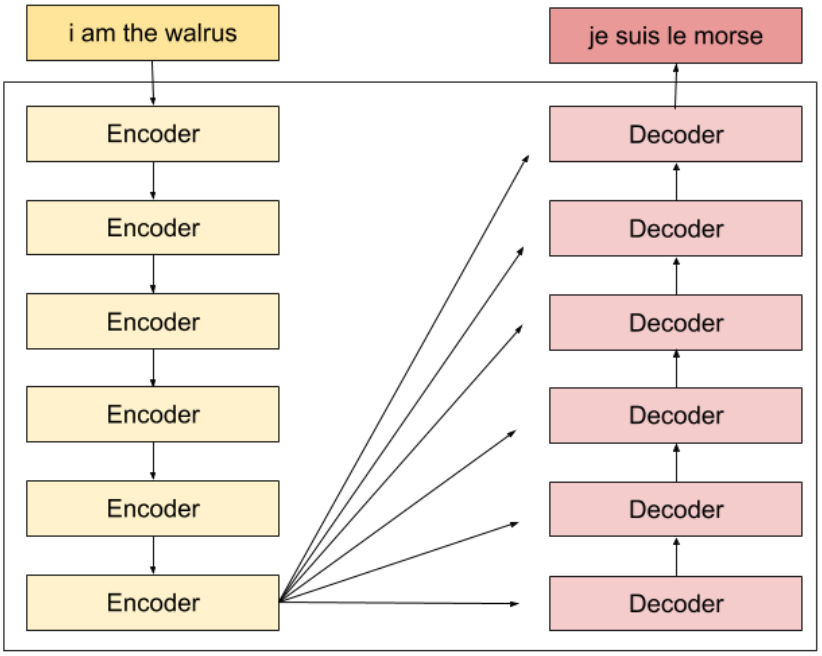

Figure 2.4: The Transformer with stacked encoders and decoders.

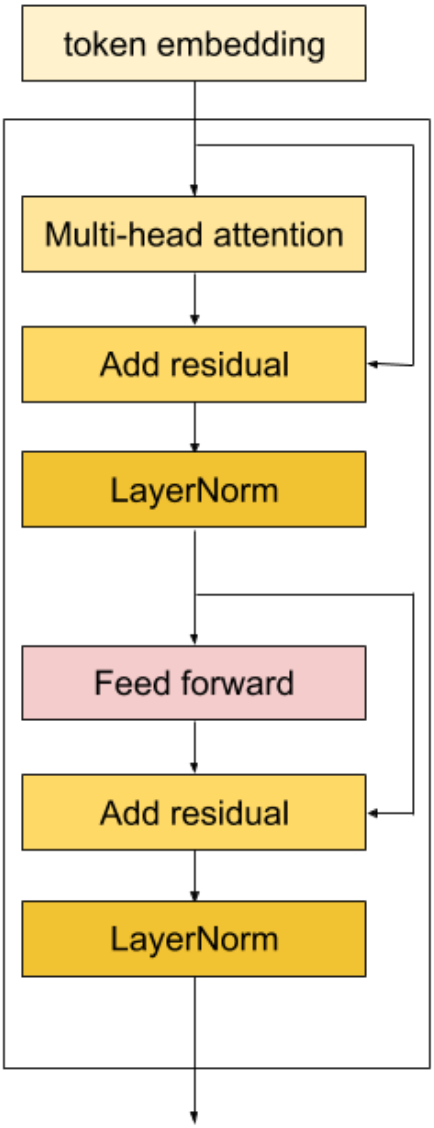

Figure 2.5: High-level encoder outline.

to an output. The query, keys, value, as well as the output are vectors. For each position in the input sequence, these vectors are created. This is done by creating a matrix X which contains the input embeddings, and multiplying this matrix with matrices W^Q, W^K, W^V respectively to create matrices Q, K, and V which contain the queries, keys, and values:

$$Q = W^Q X \tag{2.18}$$

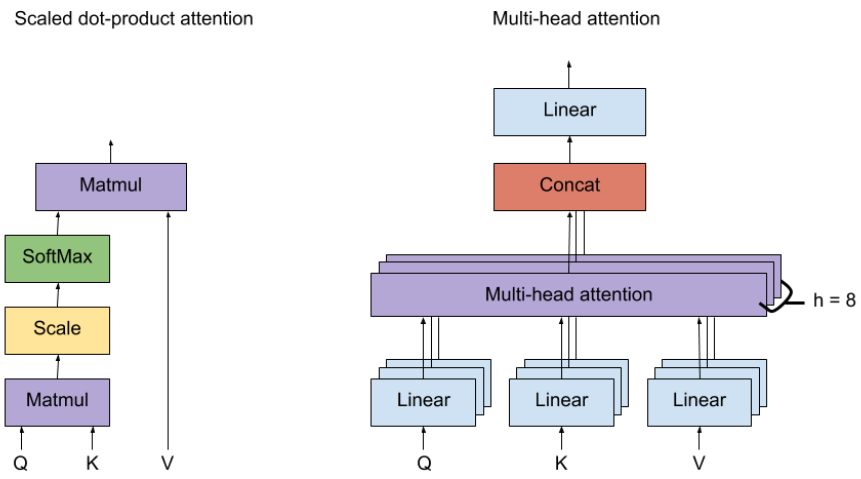

Figure 2.6: Attention mechanism. (Left) Illustration of scaled dot-product attention. (Right) Multi-headed attention consists of multiple attention layers.

$$K = W^K X \tag{2.19}$$

$$V = W^V X \tag{2.20}$$

where the queries and the keys have the dimension d_k and the values have the dimension d_v. During training, the matrices W^Q, W^K and W^V are learned [17].

The output of the self-attention mechanism is then calculated as a weighted sum of the values, where the weight is a function of how similar the query is to the key vectors corresponding to the tokens in the input sequence. While the actual calculations are carried out in matrix form, intuitively the first step toward creating this output is to calculate how much focus should be placed on other parts of the input sequence when encoding a token at a certain position. That is, when encoding the token 'it' in the example above, where should the network put its focus? This is done by calculating the dot-product between the query vector of the token and the key vector of each token in the input sequence. Next, the dot-products are scaled by $\frac{1}{\sqrt{d_k}}$ to prevent the gradients from becoming extremely small. To obtain weights, the scaled dot-products are in-

put to the softmax function. To calculate the final output, the value vectors are summed up according to the weights produced by the softmax function [17]. This is displayed to the left in figure 2.6. In matrix form this is computed as

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V. \tag{2.21}$$

The Transformer contains multiple attention heads in one sub-layer to improve performance. Multi-head self-attention is displayed to the right in figure 2.6. In the original setting the Transformer network contains eight such heads. The input to the different heads are not the same set of query, key, and value vectors. Instead, the vectors are linearly projected to yield different versions of them before being input to the different heads. The output of the different heads as calculated by equation 2.21 is then concatenated and multiplied with a weight matrix W^O to achieve a final attention score [17]:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O. \tag{2.22}$$

2.6.2 Fully connected sub-layer

The output of the attention sub-layer is fed into a fully connected feed-forward network. This network is applied to each position in the token sequence separately. It consists of two transformations with a ReLU activation function applied in between:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{2.23}$$

where x is the output vector for one token produced by the multi-headed attention sub-layer, and W_1, W_2, and b_2 are learnable parameters. The weights for these fully connected sub-layers are not shared between different encoder layers [17].

2.6.3 Residual connections

Neural networks can suffer from degrading accuracy as the depth of the network increases. This degradation is not caused by over-fitting. Neural networks employing residual connections are easier to optimize, and the connections allow deeper networks without lower accuracy [22].

Neural networks with residual connections approximate residual functions. Let $\mathcal{H}(x)$ be the underlying mapping to be fit by a few stacked layers in the network, where x is the input vector to the first of these layer. Instead of expecting these layers to approximate $\mathcal{H}(x)$, residual connections explicitly allows

the layers to approximate a residual function $\mathcal{F}(x) = \mathcal{H}(x) - x$. The original mapping becomes $\mathcal{F}(x) + x$. By reformulating the mapping, degradation of accuracy in deep neural networks can be repelled by making the network easier to train [22].

A residual connection is employed around both the multi-headed attention sub-layer as well as the fully connected feed-forward sub-layer. The residual connection copies the input x to the sub-layer and then adds it to the output of the sub-layer [22]. This addition is done component wise. The output of the sub-layer after the residual connection is

$$output_{res} = x + sublayer(x) \tag{2.24}$$

where the sub-layer function is either the attention function or the fully connected network [17].

2.6.4 Layer normalization

Layer normalization is applied to the output of the residual connections. Layer normalization is a normalization technique which improves the training speed of the network [23]. In the Transformer network, layer normalization is computed as:

$$LayerNorm(output_{res}) = \gamma \frac{output_{res} - \mu}{\sigma} + \beta \tag{2.25}$$

where μ and σ are the mean and standard deviation of the elements in $output_{res}$ and γ and β are learnable parameters [24].

2.6.5 Decoder

The decoder consists of the same number of layers as the encoder, but adds a multi-attention layer that performs multi-head attention over the output of the encoder stack. Further, the self-attention layer is modified by masking subsequent tokens to the token that is being decoded [17].

In the originally proposed Transformer, the output of the decoder stack is inputted into a linear layer and a softmax layer over the output token vocabulary. The model was used to translate Engligh-German sentence pairs as well as English-French sentence pairs [17].

2.7 BERT

Bidirectional Encoder Representations from Transformers (BERT) is a network architecture released in 2018 by Devlin et al. It consists of a stack of

Transformer encoders, eschewing the decoder stack. While the original Transformer network has 6 encoders stacked with 8 attention heads per layer, BERT comes in two sizes: $BERT_{base}$, with 12 encoder layers and 12 attention heads per layer, outputting embeddings with 768 dimensions, and $BERT_{large}$, with 24 encoders and 16 attention heads per layer, outputting embeddings of size 1024 [4].

The training scheme for BERT is divided into two phases. The first is referred to as 'pre-training' and involves training the model on unlabeled data. The second step is referred to as 'fine-tuning' and involves training the model on labeled data performing downstream tasks. Such down-stream tasks involve question answering and named entity recognition. Question answering involves giving the model a question and a context that may or may not contain the answer to a question, and then asking the model to supply an answer to the question. During fine-tuning, all of the model's parameters are trained. Pre-trained models were made public by the architects of BERT [4].

BERT is designed to take input sequences that can consist of single sentences as well as a pair of sentences, such as a question and an answer. A sentence is defined as an arbitrary amount of words and may not correspond to a sentence in the linguistic sense. The first token of every sequence is always the special [CLS] token. This token can be thought as an aggregate representation of the sequence. As sentence pairs are packed into one sequence, there is a need for a special token to separate them. BERT therefore uses the [SEP] token to separate sentences in sentence pairs. By having both sentences in one token sequence, the model can perform attention between tokens belonging to different sentences [4].

2.7.1 Input embeddings

The input to to the lowest layer in the BERT network are a sequence of tokens, which are not necessarily words. BERT uses WordPiece tokenization to create tokens that help facilitate this. Essentially, WordPiece tokens are sub-word units that include single characters. The BERT vocabulary contains 30,000 tokens in the original setting for the English language. The max sequence length is 512 tokens. At inference time, rare words that are not in the vocabulary are turned into sub-words [4].

To break words into sub-words, the WordPiece tokenizer is first trained on a fixed vocabulary. After training, words that are not included in the trained WordPiece sub-word vocabulary are broken down into sub-words using the tokenizer. For instance, the sentence "once only vegetables and chocolate were

left, the dog was in zugzwang" becomes "once only vegetables and chocolate were left , the dog was in zu ##g ##z ##wang" once tokenized. Each word was included in the sub-word vocabulary except zugzwang, which has been turned into the tokens "zu", "g", "z", "wang", with ## marking where the word was split [25].

Similar to the Transformer, BERT adds positional embeddings to the token embeddings to inform the model of where in the sequence the token belongs. Further, BERT adds segment embeddings to inform the model of which sentence in a sentence pair the token belongs to. To summarize, each input embedding consists of the summation of a token embedding, a positional embedding, and a segment embedding [4].

2.7.2 Pre-training regimes

The BERT network is trained using two different pre-training tasks to enable a bidirectional representation of each token as well as enabling the model to understand the relationship between sentences [4].

Masked language model

While the Transformer is trained using conditioning on the previous tokens in the sequence, BERT uses a masked language procedure to enable bidirectional representations. This allows the model to condition on both the forward as well as the backward context. This is important for downstream tasks such as question answering, where crucial context may be in the forward as well as the backward context. To perform this masked language procedure, 15% of the input tokens are masked at random using a special [MASK] token, and the network is then tasked to predict those tokens [4]. This task is sometimes referred to as a Cloze task [26]. BERT does this by feeding the masked token into a fully connected neural network and a softmax function over the output vocabulary. The token corresponding to the argument which maximizes the softmax function is then taken as BERT's guess of which token hides behind the [MASK]. Cross-entropy is used as the loss function [4].

A downside to this procedure is that the tokens are not masked during fine-tuning, creating an inconsistency between pre-training and fine-tuning. To mitigate this, the tokens randomly chosen to be masked are not always masked. 80% of the time, the randomly chosen token is replaced by the [MASK] token. 10% of the time, the token is instead replaced by a randomly chosen token in the vocabulary. 10% of the time, the token remains unchanged to bias the representation towards the observed word. Using this procedure the encoder

does not know which token it will have to predict, forcing the model to keep a representation of each token which depends on the surrounding context [4].

Next sentence prediction

In order to train a neural model that understands the relationship between sentences, BERT is tasked to predict the next sentence during pre-training. During training two sentences A and B are chosen, with B being the sentence that actually follows sentence A in the corpus 50% of the time and 50% sentence B will instead be chosen randomly from the corpus. Next sentence prediction is important to tasks that are based on understanding the relationship between two sentences, such as question answering [4].

2.7.3 Sentence embeddings

While the [CLS] token can be used to as a representation for the entire sentence, the token embeddings outputted by BERT can be used to create a sentence embedding as well. To do so, the individual token embeddings excluding the [CLS] and [SEP] token are averaged. This process is referred to as 'mean-pooling'. Other procedures such as 'max-pooling, which takes the maximum of the token embeddings, exist. Since BERT consists of multiple stacked Transformer encoders, the token embeddings can be excluded from any layer [27].

2.7.4 Fine-tuning

BERT can be fine-tuned for a multitude of tasks. This section contains information about the fine-tuning tasks BERT performed in this thesis.

Fine-tuning for STS-b

The STS benchmark (STS-B) dataset is a benchmark for comparing semantically similar sentences. It contains 8626 sentence pairs. For each sentence pair, there is a floating point between 0 and 5 specifying how semantically similar the sentences are, with 5 being the most similar [28]. To fine-tune BERT on the STS-B dataset, the sentences in the sentence pair are separated by the [SEP] token. During fine-tuning, the [CLS] token is fed into a fully connected layer which outputs a continuous value between 0 and 5. The mean-squared

error loss is used as loss function

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=0}^{N} (y - \hat{y}_i)^2 \tag{2.26}$$

where y is the ground truth similarity score and $\hat{y}$ is the predicted score [4].

Fine-tuning for MRPC

The Microsoft Research Paraphrase Corpus (MRPC) consists of 5800 sentence pairs extracted from news sources. The pairs have been annotated by humans to indicate whether they are semantically equivalent or not [29].

To fine-tune BERT on the MRPC dataset, the sentences in the sentence pair are separated by the [SEP] token. During fine-tuning, the [CLS] token is fed into a fully connected layer to classify the questions as being duplicate or not. Cross-entropy loss is used as loss function [4].

Fine-tuning for QQP

The Quora Question Pairs (QQP) consists of over 400,000 question pairs based on real Quora data. The question pairs are marked as being duplicate or not. The dataset can be used to train models to recognize semantic equivalence [30].

To fine-tune BERT on the QQP dataset, the questions in the question pair are separated by the [SEP] token. During fine-tuning, the [CLS] token is fed into a fully connected classification head to classify the questions as being duplicate or not. Cross-entropy loss is used as loss function [4].

2.8 Sentence-BERT

The Sentence-BERT (SBERT) network is a modification of a pre-trained BERT network that uses Siamese and triplet network structures to output semantically meaningful sentence embeddings [14]. A Siamese neural network is a network that consists of two neural networks that are identical, but share weights. Both networks work in tandem and their end outputs are compared, usually with a distance metric such as cosine distance. A Siamese network is well-suited for similarity problems [31]. The weight-sharing property of Siamese networks guarantee consistent predictions, since each network computes the same function [32].

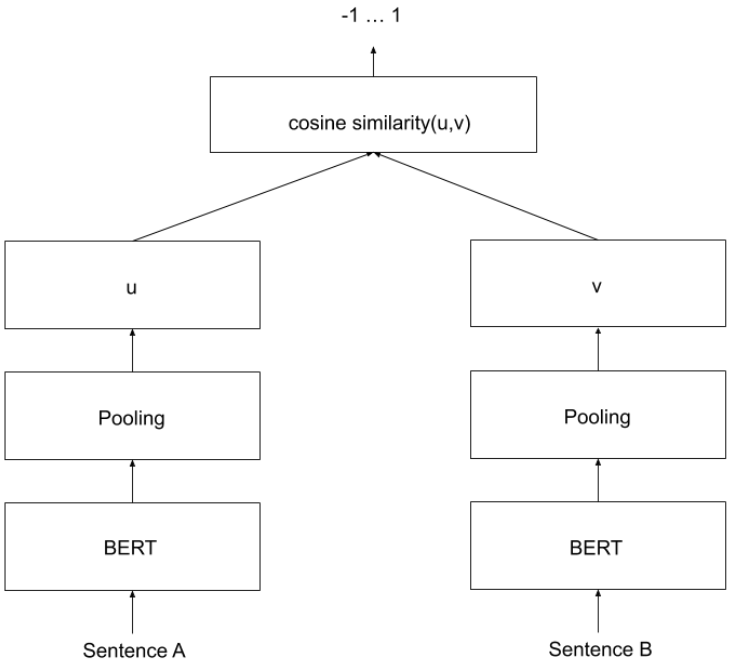

Figure 2.7: The SBERT architecture used in this thesis.

Instead of outputting embeddings for each token in a sentence as BERT does, SBERT outputs one embedding for the entire sentence. The authors of SBERT argue that the embeddings produced by SBERT are superior to the sentence representations that can be derived from a vanilla BERT network [14].

2.8.1 Fine-tuning

The SBERT network architecture depends on what dataset the network is fine-tuned on. In this thesis SBERT is fine-tuned on the STS-B dataset, which is further explained in section 3.2.1. The architecture is constructed around two BERT networks with shared weights, which is displayed in figure 2.7. First, each sentence in a sentence pair is inputted to the SBERT network. The output embeddings are then pooled into sentence embeddings u and v. Pooling is done by computing the mean of all output embeddings of the BERT network, excluding [CLS] and [SEP] tokens. Cosine similarity is then computed between these two networks. The loss is computed using the mean-squared error loss function [14].

2.9 Related work

2.9.1 Perturbing sentences

In BERT-attack [33], Li et. al uses BERT to perturb and create sentences in order to mislead target models and make them predict incorrectly. For each sentence in the dataset, one or more sub-words was masked and replaced by using BERT to guess what those masked sub-words were to create adversarial samples. The sentence and its adversarial counterpart was then fed into a model which was tasked with either a classification task or a natural language inference (NLI) task. Such NLI tasks include the Stanford language inference task, where the input to the model is a premise and a hypothesis. The model is then tasked with predicting if the hypothesis is entailed in the premise, if the premise contradicts the hypothesis, or if the relationship between the two are neutral. For instance, the sentence "Some rooms have balconies" was accompanied by the hypothesis "All of the rooms have balconies off of them" with the label being 'contradictory'. The sentence was then perturbed to become "Many rooms have balconies". In the study, the target model classified the perturbed sentence as 'neutral'. Since this is different from the label, the authors had successfully mislead the target model. The authors note that they had a high success rate.

2.9.2 Sentence embeddings

BERT is tasked to create sentence embeddings in multiple papers [14, 34, 35]. Different modes of creating sentence embeddings are used, with Reimers et al. [14] using both 'mean-pooling' and the [CLS] token, while Yankovskaya et al. [35] uses 'mean-pooling'.

2.9.3 Semantic similarity

Semantic similarity between short texts is investigated in [36]. To do this, first word2vec embeddings and inverse document frequency (idf) factors were trained on a corpus consisting of short passages extracted from Wikipedia articles. The passages in the dataset were randomly split into pairs and tagged as being semantically similar if the two sentences were extracted from the same passage. Otherwise, the sentences were tagged as being semantically dissimilar. The word2vec embeddings belonging to each word in each sentence were then sorted according to the idf factor corresponding to that word, creating

new sentences where the words were arranged in declining order of idf factor.

Next the embeddings were multiplied by learned importance factors, with each word embedding having its own importance factor. The importance factors were learned by optimizing an objective function which involves a loss function that minimizes the distance between the embeddings belonging to a semantically similar pair and maximizes the distance between the embeddings of a semantically dissimilar pair [36].

Lastly, the mean of the ranked word2vec embeddings in each sentence were multiplied by the corresponding importance factor to create a final sentence embedding. To compare semantic similarity between two sentence embeddings, different distance metrics were used including Euclidean distance and cosine distance [36].

The authors note that this approach is restricted to texts of a fixed length. The approach is compared to using only TF-IDF, with the importance factor approach outlined in the paper outperforming TF-IDF on sentences of 10 words and being on par with TF-IDF on sentences of 30 words [36].

Chapter 3

Method

3.1 Method outline

To study semantic text similarity in an efficient manner, a multitude of experiments were carried out. The common denominator of these experiments were that they involved a set of sentences, labeled as context, as well as a sentence labeled as query. One of the sentences belonging to the context was labeled as being semantically similar to the query. The query was then compared to the sentences belonging to the context. The query and the sentence labeled as being semantically similar should then have a shorter cosine distance between them than any other query-context pair.

For example, consider the context consisting of the following sentences [37]:

1. Construction is the process of constructing a building or infrastructure.
2. Construction differs from manufacturing in that manufacturing typically involves mass production of similar items without a designated purchaser, while construction typically takes place on location for a known client.
3. Construction as an industry comprises six to nine percent of the gross domestic product of developed countries.

If the user queries the model with "What percentile of gross domestic product is construction comprised of?", the sentence that contains the answer to that question is the third sentence in the context. It is therefore tagged as being semantically similar to the query. If the distance between the embeddings

representing the query and the third sentence is smaller than between the embeddings representing the query and any other sentence in the context, the model would have been accurate.

In order to perform such experiments, the context sentences as well as the query sentence were turned into embeddings to evaluate their semantic similarities. The embeddings were produced by either a BERT network, a Sentence-BERT network, or the TF-IDF algorithm. This enabled a comparison of embeddings produced by different algorithms.

The semantic similarity of the sentences in the context and the query was evaluated by comparing the cosine distance between them and ranking them using the distance as metric. If the question labeled as being semantically similar to the query has the shortest distance to the query and thus has the highest ranking, the query is considered as being accurate.

The stages described above are described in further detail in the following sections.

3.2 Creating a context

3.2.1 Datasets used for creating a context

To study semantic text similarity, datasets are needed. Preferably, it should contain ranked contexts such that it can be evaluated quantitatively. However, no such dataset was located. Instead, the datasets SQuAD 1.0 and STS-B were modified and used. Two datasets were used to avoid possible biases in the data.

SQuAD 1.0 dataset

SQuAD 1.0 is a dataset designed for reading comprehension. It contains a set of Wikipedia articles with questions posed by crowd workers. For every question, there is a corresponding answer consisting of a span of text from the matching passage of the article. An example context with two questions is displayed in figure 3.1. An article may have multiple questions attached to it [8].

The SQuAD 1.0 dataset was modified by splitting each article into a set of sentences, and marking the sentences containing the answer to the questions as being semantically similar. As an example, in figure 3.1 the article was split into 4 sentences, and the second sentence was labeled as being semantically similar to the query 'How far from Warsaw does the Vistula river's environment change noticeably?' since it contains the answer. The NLTK toolkit [39]

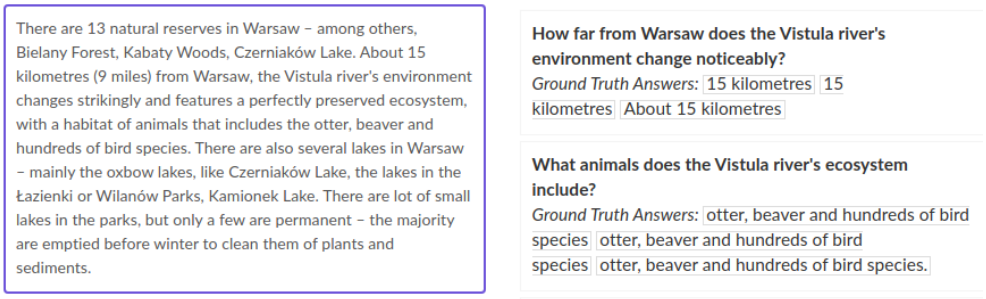

Figure 3.1: SQuAD example context and corresponding questions [38].

was used to split each article into sentences. The end product was a set of contexts consisting of sentences, with the sentence containing the answer to the query being marked as semantically similar, as well as corresponding queries. This allowed for the ranking of semantic similarity between the question and the sentences belonging to the context, with the supposition that the query and the sentence containing the answer will have a larger semantic similarity than the question will have with the other sentences belonging to the context.

SQuAD 1.0 was chosen and modified this way to enable comparison with results from other papers. In the SQuAD 1.0 paper, logistic regression was used to identify the sentence containing the answer to the question with an accuracy of 79% [8]. This was used as a baseline.

STS benchmark dataset

The STS benchmark (STS-B) dataset is a benchmark for comparing semantically similar sentences. It contains 8626 sentence pairs. For each sentence pair, there is a floating point between 0 and 5 specifying how semantically similar the sentences are, with 5 being the most similar [28].

Since each data point in the STS-B dataset only contains two sentences, the dataset was modified to contain a set of contexts similar to the modified SQuAD 1.0 dataset. For each sentence pair, one random word belonging to one of the sentences was masked. The masked sentence was then fed into a BERT large network. As described in section 2.7.2, the argmax of the softmax layer is taken as the network's best guess of what word is hidden behind the masked word during pretraining. To create a synthetic sentence similar to the masked sentence, the output was instead chosen to be the 100th best argument of the softmax layer for the masked token. The rationale for choosing the 100th argument was that the synthetic sentences should be somewhat similar to the original sentence, but not too similar as it could make the experiments too

hard.

The output of the network was therefore a sentence similar to the original network, but with one word that the network has replaced with a word that should be somewhat similar. This procedure is repeated 5 times to create a context consisting of 5 sentences where one token has been replaced in each sentence. To make the synthetic sentences less similar to the original sentence, the synthetic sentences can be masked and fed to the BERT network to change yet another token. The modified version of STS-B used thus contains a set of contexts which in turn consists of the original sentence pair, as well as 5 synthetically created sentences.

As an example of a synthetic context, consider the following original sentence pair:

- A man ate a piece of fruit.
- The man chewed a banana and enjoyed the experience.
- Similarity score: 4/5.

Now, the second sentence is chosen to be perturbed. Following the outlined procedure, we perturb the second sentence 3 times to produce a synthetic context:

- The man chewed a banana and enjoyed the experience.
- The **woman** chewed a banana and enjoyed the experience.
- The man chewed a **lamb** and enjoyed the experience.
- The man **threw** a banana and enjoyed the experience.

The procedure was repeated 3 times instead of 5 for the sake of brevity in this example. The synthetic sentences could undergo another iteration of the outlined procedure to perturb the sentences further.

The supposition of this experiment was that the original sentence pair share a larger semantic similarity with each other rather than with any of the synthetically created sentences.

3.3 Sentence embeddings

To compare semantic similarity between sentences, embeddings were created. The method used for creating embeddings using BERT [4], SBERT [14], and TF-IDF [12] have some similarities but also some differences. As BERT does not output sentence embeddings directly but rather token embeddings, these embeddings have to be transformed to create sentence embeddings. SBERT on the other hand directly outputs sentence embeddings. BERT and SBERT embeddings both had 1024 dimensions, while TF-IDF produced embeddings of varying dimensions.

3.3.1 BERT

As described in section 2.7, BERT outputs an embedding for each token in the sentence as well as an embedding for the [CLS] token and the [SEP] token respectively. However, to be able to compare entire sentences, it is necessary to find a representation for sentences.

To create a representation for an entire sentence, two strategies were employed. The first strategy involved using the [CLS] token as a representation for the sentence. This strategy was employed in [14]. The second strategy involves 'pooling' the token embeddings. This was done by averaging the sentence's token embeddings excluding [CLS] and [SEP] embeddings, so-called 'mean-pooling', following the approach used in [27]. Both strategies were employed to create different embeddings to allow for a comparison between the techniques.

Due to the stacked encoder network layout of BERT, the word embeddings used to produce the sentence embeddings can be extracted from any layer in the encoding stack. In this thesis, the embeddings were extracted from the second last layer following the procedure used in [27].

3.3.2 SBERT

SBERT uses pooling to create sentence embeddings [14]. The default mode of 'mean-pooling' was used in this thesis.

3.3.3 TF-IDF

TF-IDF embeddings were created using different techniques. Lemmatization was used to create the same embedding for a word regardless of the word's

inflected form. As an example, the word "walk" and "walking" receive the same embedding following lemmatization. The dimensionality of TF-IDF embeddings is dependent on the size of the vocabulary and in this paper the vocabulary was built on the train set of the dataset. The size of this vocabulary was 30277, which is larger than the dimensionality of SBERT and BERT embeddings. Therefore, embeddings were also produced by restricting the features used by the algorithm to the top 1024 features ordered by term frequency across the corpus. This enabled a comparison of TF-IDF embeddings of different dimensionality, as well as comparing TF-IDF embeddings and BERT/SBERT embeddings of equal size.

3.4 Experiments

Different modes of fine-tuning and different values of dropout was tested to quantify the impact of these techniques on the semantic properties of sentence embeddings.

3.4.1 Fine-tuning

To enable an equitable comparison between BERT and SBERT, both networks were fine-tuned using the same parameters. These shared parameters were a batch size of 16, a learning rate of 2e-5, using 10% of the data as warm-up, and fine-tuning for 4 epochs. In table 3.1 and in table 3.2 these models are commented as having 'SBERT params'.

Further, BERT was fine-tuned using the parameters used in the original BERT paper [4]. The parameters were a batch size of 32, a learning rate of 5e-5, using 5% of the data as warm-up, and fine-tuning for 3 epochs. In table 3.1 and in table 3.2 this model are marked as having 'Devlin params'.

Datasets used for fine-tuning

The Quora Question Pairs, Microsoft Research Paraphrase Corpus, and STS-B datasets were used for fine-tuning. The STS-B dataset is described in section 3.2.1, the QQP dataset and the MRPC dataset are described in section 2.7.4.

3.4.2 Dropout

Dropout was varied to test its impact on sentence embeddings. In the original paper, fine-tuning using a dropout value of 0.1 [4]. In this paper, dropout

Table 3.1: Models used in SQuAD experiments

Model	Pooling	Fine-tuned	Dropout	Comment
BERT	Mean	Pre-trained	0.1	-
BERT	Mean	STS-B	0.1	Devlin params
BERT	Mean	QQP	0.1	SBERT Params
BERT	Mean	STS-B	0.1	SBERT params
BERT	Mean	STS-B	0.3	SBERT params
BERT	Mean	MRPC	0.1	SBERT params
BERT	Mean	MRPC	0.3	SBERT params
SBERT	Mean	STS-B	0.1	SBERT params
SBERT	Mean	STS-B	0.3	SBERT params
BERT	CLS	Pre-trained	0.1	-
BERT	CLS	STS-B	0.1	Devlin params
BERT	CLS	QQP	0.1	SBERT Params
BERT	CLS	STS-B	0.1	SBERT params
BERT	CLS	STS-B	0.3	SBERT params
BERT	CLS	MRPC	0.1	SBERT params
BERT	CLS	MRPC	0.3	SBERT params
TF-IDF	-	-	-	Lemma'd, max size vocab
TF-IDF	-	-	-	Lemma'd, vocab size 1024

values of 0.1 and 0.3 were tested.

3.4.3 SQuAD experiments

A dataset consisting of a set of contexts and queries were created using the SQuAD 1.0 dataset and the procedure outlined in section 3.2.1. The models used in these experiments are summarized in table 3.1.

3.4.4 STS-B experiments

The STS-B dataset was split into 3 basic datasets: one dataset containing the entire STS-B dataset, one dataset containing sentences with a semantic similarity greater or equal to 3, and one dataset containing sentences with a semantic similarity less or equal to 3. This was done to test if the models are equally well suited for sentences that are graded as being similar or dissimilar. Further, the 3 datasets underwent the procedure outlined in section 3.2.1 to

Table 3.2: Models used in STS-B experiments

Model	Pooling	Fine-tuned	Dropout	Comment
SBERT	Mean	STS-B	0.1	SBERT params
SBERT	Mean	STS-B	0.3	SBERT params
TF-IDF	-	-	-	Lemma'd, max size vocab
TF-IDF	-	-	-	Lemma'd, vocab size 1024

create synthetic datasets. The procedure was conducted to create 5 different synthetic datasets for each of the 3 basic datasets, with tokens changed per sentence ranging from 1 to 5. The end product was thus 15 different datasets.

The models used in the experiments on these datasets are summarized in table 3.2.

3.5 Software

To split paragraphs into sentences and lemmatize words NLTK 3.5 [39] was used. HuggingFace's Transformers 2.3.0 [40] was used to efficiently fine-tune the neural models . To calculate cosine distance, scipy 1.4.1 [41] was used. To use SBERT, sentence-transformers 0.3.2 [14] was used.

Chapter 4

Results

4.1 SQuAD Experiments

The results from the SQuAD experiment are summarized in table 4.1.

4.2 STS-B experiments

The results from using all of the sentence pairs in the dataset is displayed in figure 4.1. The results from only using sentence pairs with a semantic similarity greater or equal to 3 is displayed in 4.2. The results from only using sentence pairs with a semantic similarity less or equal to 3 is displayed in 4.3.

Table 4.1: SQuAD experiments

Model	Pooling	Fine-tuned	Dropout	Comment	Accuracy
BERT	Mean	Pre-trained	0.1	-	70.1%
BERT	Mean	STS-B	0.1	Devlin params	54.5%
BERT	Mean	QQP	0.1	SBERT Params	50.2%
BERT	Mean	STS-B	0.1	SBERT params	52.2%
BERT	Mean	STS-B	0.3	SBERT params	72.2%
BERT	Mean	MRPC	0.1	SBERT params	70.8%
BERT	Mean	MRPC	0.3	SBERT params	71.8%
SBERT	Mean	STS-B	0.1	SBERT params	72.9%
SBERT	Mean	STS-B	0.3	SBERT params	71.0%
BERT	CLS	Pre-trained	0.1	-	37.0%
BERT	CLS	STS-B	0.1	Devlin params	32.0%
BERT	CLS	QQP	0.1	SBERT Params	50.0%
BERT	CLS	STS-B	0.1	SBERT params	30.9%
BERT	CLS	STS-B	0.3	SBERT params	44.8%
BERT	CLS	MRPC	0.1	SBERT params	43.2%
BERT	CLS	MRPC	0.3	SBERT params	42.5%
TF-IDF	-	-	-	Lemma'd, max size vocab	77.2%
TF-IDF	-	-	-	Lemma'd, vocab size 1024	70.5%
Log. reg.	-	-	-	From SQuAD paper [8]	79.0%

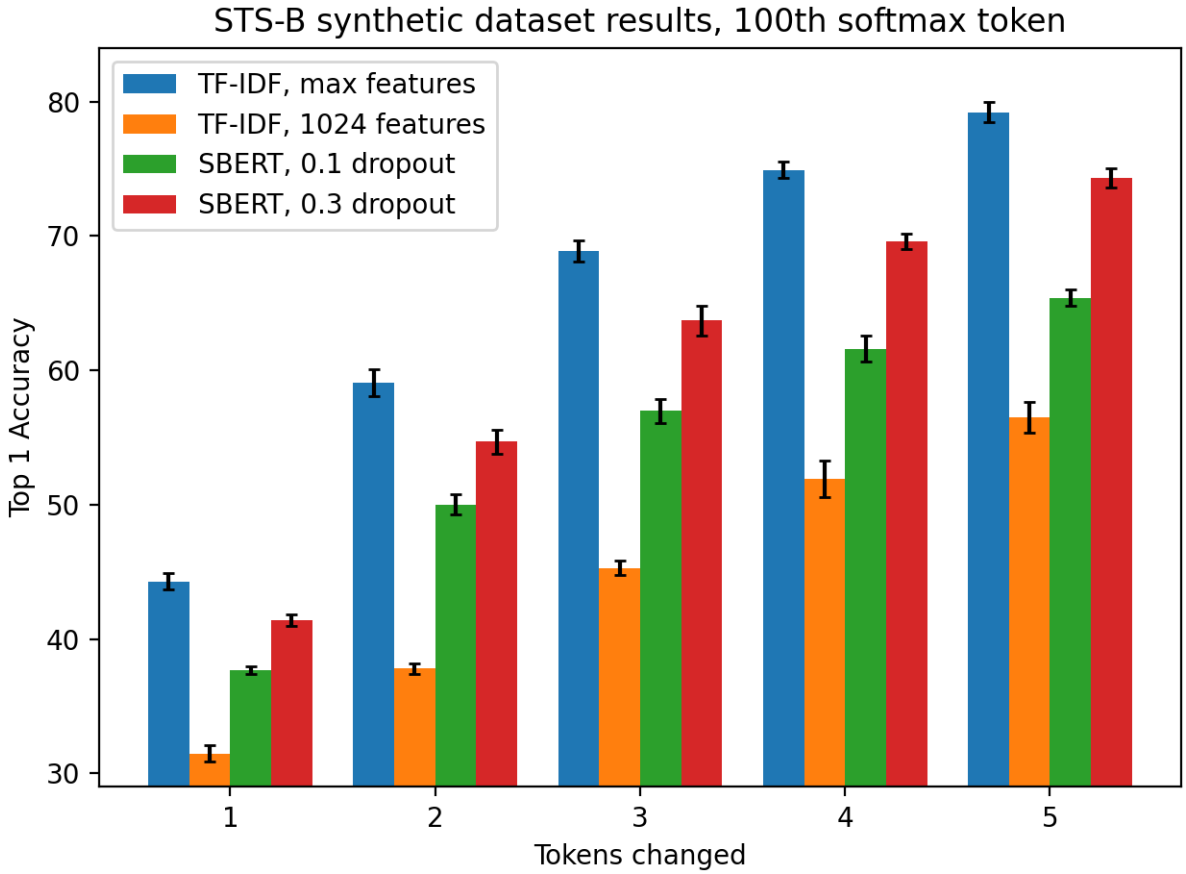

Figure 4.1: Results from replacing a masked token with the 100th ranked softmax token using BERT using all sentence pairs in the dataset.

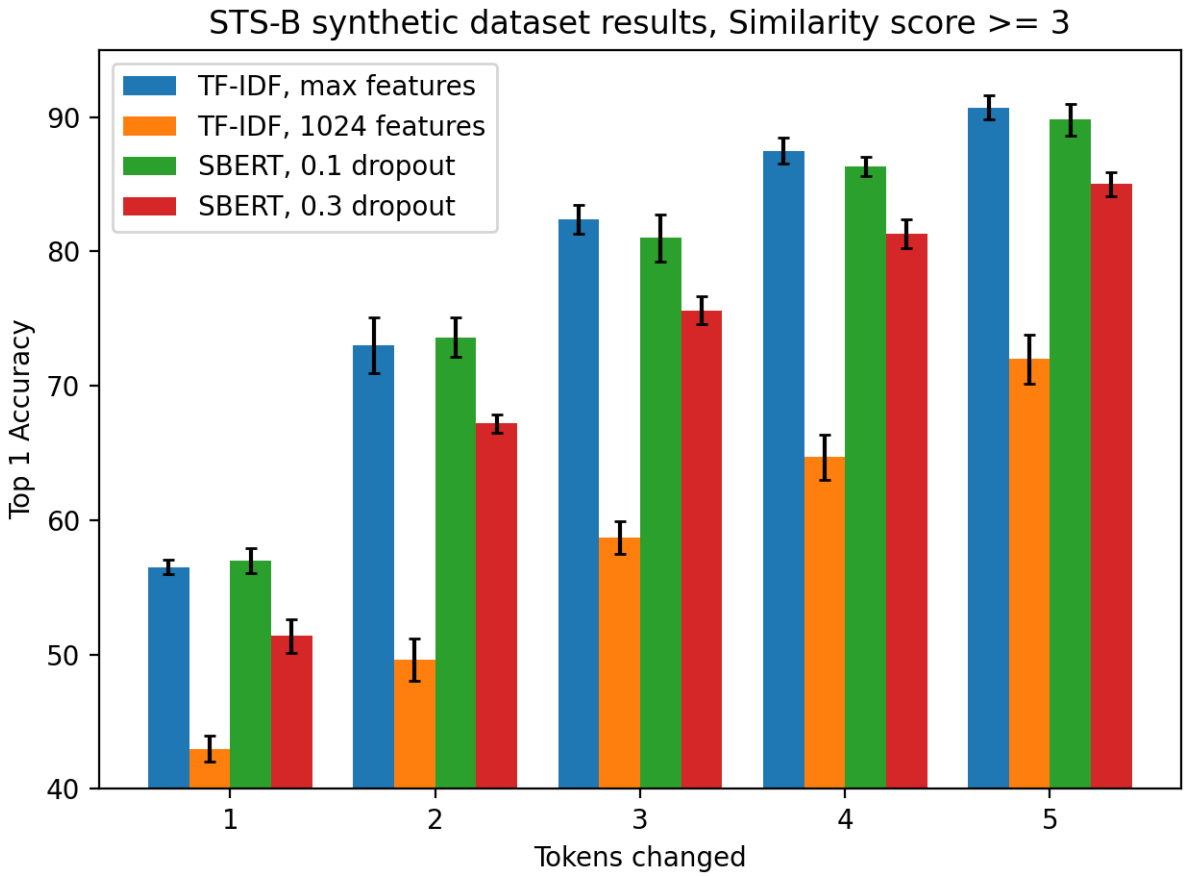

Figure 4.2: Results from replacing a masked token with the 100th ranked softmax token using BERT using sentence pairs with similarity greater than 3.

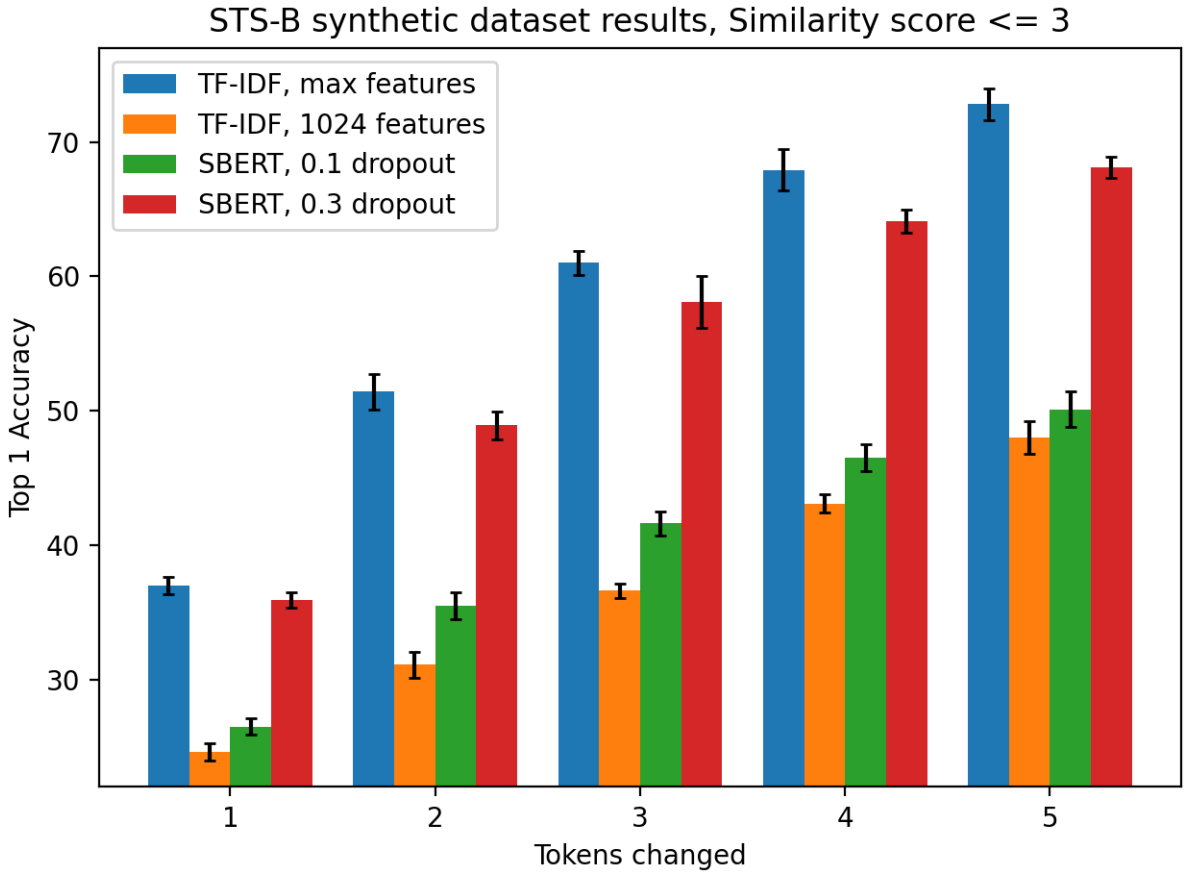

Figure 4.3: Results from replacing a masked token with the 100th ranked softmax token using BERT using sentence pairs with similarity less than 3.

Chapter 5

Discussion

5.1 SQuAD experiment discussion

The baseline logistic regression model used in [8] could not be beaten by any model. In the article, the authors note that "the model is able to select the sentence containing the answer correctly with 79.3% accuracy" [8]. However, upon reviewing the code published with the paper, it appears that the authors seem to have tasked the model with selecting a span of text that contains the answer to the question rather than selecting a sentence. As the SQuAD experiment operates on a sentence level, the comparison between the logistic regression model and the models in this paper may not be entirely fair.

TF-IDF performs competitively. Using an unlimited vocabulary, TF-IDF outperforms both SBERT and BERT models. However, constricting TF-IDF to produce embeddings with the same number of dimensions as BERT and SBERT embeddings lowers performance significantly. Further, TF-IDF embeddings are less resource intensive to produce and do not require a GPU. One reason for TF-IDF performing so well may be that the questions and the sentences containing the answers may have large lexical similarity, i.e., they may contain the same words to a large degree.

Embeddings produced by SBERT outperforms embeddings produced by BERT, regardless of which dataset the BERT models were fine-tuned on. This is consistent with the findings in [14].

The [CLS] token performs poorly as a sentence embedding compared with using mean-pooling. Using the [CLS] token outputted by the best model is worse than using the worst performing mean-pooling model. This is consistent with the findings in [14].

The impact of using 'SBERT params' instead of 'Devlin params' is not

large. Batch size, learning rate, and epochs trained do thus not appear to be important.

Which task and dataset to fine-tune BERT on is important accuracy wise. Starting with the mean-pooled variants, models trained on the MRPC dataset have the highest accuracy regardless of dropout. Compared with the model fine-tuned on the QQP dataset there is a stark contrast in accuracy score. Both datasets contain text pairs which either are semantically equivalent or not, but QQP contains question pairs while MRPC contains sentence pairs. Thus, using a dataset or fine-tuning task which involves a binary task is not a guarantee for either low or high accuracy. When fine-tuning BERT on STS-B, dropout has a large impact. Comparing with the BERT model fine-tuned on MRPC, dropout has a much larger impact on the model fine-tuned on STS-B regarding accuracy.

The fine-tuning results of the [CLS] models display a different pattern. The model trained on QQP outperforms the other models. Dropout of 0.3 raises the accuracy of the model fine-tuned on STS-B while lowering the accuracy of the model fine-tuned on MRPC.

The pre-trained only models achieve higher accuracy than some fine-tuned models and lower accuracy than other models. For mean-pooling, embeddings produced by the pre-trained BERT network have slightly lower accuracy than the best performing fine-tuned BERT model. When fine-tuning the models used to produce the [CLS] token, the pre-trained model trails the model fine-tuned on the QQP dataset with 13 percentage points.

Fine-tuning lowers the accuracy when carried out on some datasets. Given this and the above discussion on how the MPRC dataset and the QQP dataset contain similar data and involves a similar task, it is not clear which dataset and fine-tuning task to choose to produce sentence embeddings with suitable semantic properties. An acceptable solution when using the BERT network may thus be to use a pre-trained network and mean-pool the token embeddings.

While the experiment rests on the supposition that the question and the sentence that contains the answer are semantically similar, ultimately the SQuAD dataset is designed for a question answering task and not for semantic similarity tasks. Consider the question "An illustrated, paraphrased version of this appeared when?" with the corresponding answer "1487" from the SQuAD dataset. While the ground truth answer of "1487" perfectly answers the question, the semantic similarity between the sentence containing the answer, "A two-volume illustrated folio paraphrase version based on his manuscript, by Jean de Rély, was printed in Paris in 1487.", and the question is questionable. The results therefore have to be discounted to some degree. To achieve more

robust results, a dataset more suited to the experiment would be needed.

5.2 STS-B experiments

The accuracy increases when more tokens are changed. As more tokens are changed, the perturbed sentence share less tokens with the original sentence, which intuitively should make the task easier. This partially confirms the validity of the experiment, in that it strengthens the supposition that the original sentence pair share a larger semantic similarity with each other rather than with any of the synthetically created sentences.

As displayed in figure 4.1, TF-IDF with a full-size vector outperforms both Sentence-BERT models when using all the sentences in the dataset. Using a two-sided t-test on the 5% significance level with the null hypothesis that the TF-IDF model and the best performing Sentence-BERT model have the same accuracy, the null hypothesis can be rejected. This applies regardless of the number of tokens changed. Constricting TF-IDF to use a feature vector of equal size to the Sentence-BERT vectors severely lowers performance.

The Sentence-BERT model with 0.1 dropout is on par with the large TF-IDF model when only perturbing sentences with similarity score greater or equal to 3 as displayed in 4.2. Using a two-sided t-test on then 5% significance level, the null hypothesis that both models have the same accuracy cannot be rejected. While both models improve over the accuracy achieved on the dataset containing all sentence pairs, the Sentence-BERT model improves the most.

The large TF-IDF model outperforms the other models when only perturbing the sentence pairs with similarity scores less or equal to 3 as displayed in figure 4.3. The null hypothesis of the large TF-IDF model and the Sentence-BERT model with 0.3 dropout having the same accuracy can be rejected on a 5% significance level.

The level of dropout has a large impact on the Sentence-BERT models. A dropout of 0.3 achieves the highest overall accuracy, while a dropout of 0.1 boost accuracy on sentences with large similarity with the cost of lower accuracy on sentences with lower similarity score.

5.3 Future work

The models used in the experiments in this thesis were trained once. Reimers & Gurevych note that due to the effect on randomness the conclusions drawn from such experiments are weak, and recommend that experiments are re-

peated multiple times with models trained with different seeds [42]. Thus, to draw stronger conclusions the experiments should be repeated to create stronger conclusions.

A dataset designed for semantic similarity should be procured. As noted in section 5.1, the SQuAD dataset is not designed for semantic similarity. While a more suitable dataset could not be located, one such dataset may be constructed by researchers.

Other neural network models should be tested to create broader results. In this thesis, the only neural models used were BERT and Sentence-BERT. While these models are widely used, the state-of-the-art is always changing and newer models should be tested. One such model which could be used to produce embeddings is the ELECTRA model [43].

Chapter 6

Conclusions

Numerous experiments were performed to answer the research question. A TF-IDF model with a large embedding vector outperformed BERT & Sentence-BERT in every experiment except one. Using an equal sized embedding vector, the best Sentence-BERT and BERT models outperformed TF-IDF. To conclude, the experiments indicate that BERT-like models are better than TF-IDF in some settings.

Sentence-BERT produce sentence embeddings that are better suited for semantic similarity than BERT models do. Further, the use of mean-pooling instead of the [CLS] token produces the best results.

The experiments conducted are flawed, which make conclusions drawn weaker. Since the datasets were not explicitly designed for semantic similarity experiments, they had to be modified. Further, the models were trained only once which make the results susceptible to randomness.

While the experiments conducted in this thesis are early attempts to evaluate sentence embeddings produced by BERT-like algorithms, the results indicate a way forward. Future work includes training the models multiple time to decrease the role of randomness in the results, as well as procuring datasets explicitly designed for semantic similarity.

Bibliography

[1] Kayla Matthews. *Here's How Much Big Data Companies Make On The Internet*. https://bigdatashowcase.com/how-much-big-data-companies-make-on-internet/. 2018.

[2] F. Wang. "A Big-Data Perspective on AI: Newton, Merton, and Analytics Intelligence". In: *IEEE Intelligent Systems* 27.5 (2012), pp. 2–4. DOI: 10.1109/MIS.2012.91.

[3] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural networks* 61 (2015), pp. 85–117.

[4] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[5] Barry Schwartz. *Google: BERT now used on almost every English query*. https://searchengineland.com/google-bert-used-on-almost-every-english-query-342193.

[6] Akiko Aizawa. "An information-theoretic perspective of tf–idf measures". In: *Information Processing & Management* 39.1 (2003), pp. 45–65.

[7] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. "Natural language processing: an introduction". In: *Journal of the American Medical Informatics Association* 18.5 (2011), pp. 544–551.

[8] Pranav Rajpurkar et al. "Squad: 100,000+ questions for machine comprehension of text". In: *arXiv preprint arXiv:1606.05250* (2016).

[9] Chien-Hsing Chen. "Improved TFIDF in big news retrieval: An empirical study". In: *Pattern Recognition Letters* 93 (2017), pp. 113–122.

[10] Dik L Lee, Huei Chuang, and Kent Seamons. "Document ranking and the vector-space model". In: *IEEE software* 14.2 (1997), pp. 67–75.

[11] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. "Introduction to information retrieval". In: *Natural Language Engineering* 16.1 (2010), pp. 100–103.

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

[13] Jun Ye. "Cosine similarity measures for intuitionistic fuzzy sets and their applications". In: *Mathematical and computer modelling* 53.1-2 (2011), pp. 91–97.

[14] Nils Reimers and Iryna Gurevych. "Sentence-bert: Sentence embeddings using siamese bert-networks". In: *arXiv preprint arXiv:1908.10084* (2019).

[15] Wiso. *Unfolded basic recurrent neural network*. `https://upload.wikimedia.org/wikipedia/commons/thumb/9/99/Neural_network_example.svg/768px-Neural_network_example.svg.png`. The image in the thesis is a black and white version of the original. Original image is in the public domain, 2018.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[17] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[18] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[19] fdeloche. *Unfolded basic recurrent neural network*. `https://upload.wikimedia.org/wikipedia/commons/b/b5/Recurrent_neural_network_unfold.svg`. The image in the thesis is a black and white version of the original. Licensed under CC BY-SA 4.0 `http://creativecommons.org/licenses/by/4.0/`. 2017.

[20] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural machine translation of rare words with subword units". In: *arXiv preprint arXiv:1508.07909* (2015).

[21] Stephen Merity et al. "Pointer sentinel mixture models". In: *arXiv preprint arXiv:1609.07843* (2016).

[22] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[23] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016).

[24] Ruibin Xiong et al. “On layer normalization in the transformer architecture”. In: *arXiv preprint arXiv:2002.04745* (2020).

[25] Yonghui Wu et al. “Google’s neural machine translation system: Bridging the gap between human and machine translation”. In: *arXiv preprint arXiv:1609.08144* (2016).

[26] Wilson L Taylor. ““Cloze procedure”: A new tool for measuring readability”. In: *Journalism quarterly* 30.4 (1953), pp. 415–433.

[27] Han Xiao. *bert-as-service*. `https://github.com/hanxiao/bert-as-service`. 2018.

[28] Daniel Cer et al. “Semeval-2017 task 1: Semantic textual similarity-multilingual and cross-lingual focused evaluation”. In: *arXiv preprint arXiv:1708.00055* (2017).

[29] Microsoft. *Microsoft Research Paraphrase Corpus*. `https://www.microsoft.com/en-us/download/confirmation.aspx?id=52398`. 2005.

[30] Shankar Iyer, Nikhil Dandekar, and Kornél Csernai. *Quora Question Pair dataset*. `http://qim.fs.quoracdn.net/quora_duplicate_questions.tsv`. 2017.

[31] Davide Chicco. “Siamese Neural Networks: An Overview”. In: *Artificial Neural Networks*. Springer, pp. 73–94.

[32] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. “Siamese neural networks for one-shot image recognition”. In: *ICML deep learning workshop*. Vol. 2. Lille. 2015.

[33] Linyang Li et al. “Bert-attack: Adversarial attack against bert using bert”. In: *arXiv preprint arXiv:2004.09984* (2020).

[34] Hebatallah A Mohamed Hassan et al. “BERT, ELMo, USE and InferSent Sentence Encoders: The Panacea for Research-Paper Recommendation?” In: *RecSys (Late-Breaking Results)*. 2019, pp. 6–10.

[35] Elizaveta Yankovskaya, Andre Tättar, and Mark Fishel. "Quality estimation and translation metrics via pre-trained word and sentence embeddings". In: *Proceedings of the Fourth Conference on Machine Translation (Volume 3: Shared Task Papers, Day 2)*. 2019, pp. 101–105.

[36] Cedric De Boom et al. "Learning semantic similarity for very short texts". In: *2015 ieee international conference on data mining workshop (icdmw)*. IEEE. 2015, pp. 1229–1234.

[37] Pranav Rajpurkar et al. *SQuAD 2.0 Explore*. https://rajpurkar.github.io/SQuAD-explorer/explore/v2.0/dev/Construction.html. 2020.

[38] Pranav Rajpurkar et al. *SQuAD 1.1 Explore Warsaw*. https://rajpurkar.github.io/SQuAD-explorer/explore/1.1/dev/Warsaw.html. 2020.

[39] Edward Loper and Steven Bird. "NLTK: the natural language toolkit". In: *arXiv preprint cs/0205028* (2002).

[40] Thomas Wolf et al. "HuggingFace's Transformers: State-of-the-art Natural Language Processing". In: *ArXiv* abs/1910.03771 (2019).

[41] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: https://doi.org/10.1038/s41592-019-0686-2.

[42] Nils Reimers and Iryna Gurevych. "Why comparing single performance scores does not allow to draw conclusions about machine learning approaches". In: *arXiv preprint arXiv:1803.09578* (2018).

[43] Kevin Clark et al. "Electra: Pre-training text encoders as discriminators rather than generators". In: *arXiv preprint arXiv:2003.10555* (2020).

TRITA-EECS-EX-2021:48