BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

ANSWER CORRECTNESS ESTIMATION ON A QUESTION

ODHAD SPRÁVNOSTI ODPOVĚDÍ NA OTÁZKU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MARIÁN LIGOCKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. IGOR SZŐKE, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

148176

Institut:	Department of Computer Graphics and Multimedia (UPGM)
Student:	**Ligocký Marián**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Answer Correctness Estimation on a Question**
Category:	Speech and Natural Language Processing
Academic year:	2022/23

Assignment:

1. Get familiar with the basics of natural language processing and machine learning.
2. Study methods of sentence similarity estimation (sentence embeddings) using machine learning algorithms. Find suitable datasets (some of the data will be provided to you). Divide the data into training and testing subsets.
3. Design and implement selected methods of the sentence similarity estimation in the domain of language learning. Evaluate achieved results.
4. Improve the proposed method.
5. Draw a conclusion and propose further approaches.
6. Create A2 poster or ~30 seconds long video presenting your work.

Literature:

- Perone C., et al. Evaluation of sentence embeddings in downstream and linguistic probing tasks, arxiv:1806.06259
- According to supervisor's recommendation.

Requirements for the semestral defence:
Steps 1 - 3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Szőke Igor, Ing., Ph.D.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	9.5.2023

Abstract

When it comes to language learning apps that allow sentence-like answers, accurately estimating the correctness score is crucial. A possible approach is to compute the semantic similarity of input sentences and predefined correct answers. The similarity score of the student and the correct answer can be computed by a deep language model based on transformer architecture. We examined different models for Semantic Textual Similarity. The best model stsb-TinyBERT-L-4 (cross-encoder), improves the old model by 27.8% in MSE on a human-annotated dataset and calibrated by linear regression. While incorporating Natural Language Inference labels may enhance performance, further research is needed.

Abstrakt

Odhad správnosti odpovede na otázku je kritický pre aplikácie na výučbu jazyka, kde sa očakáva odpoveď v podobe vety. Možný prístup je spočítať sémantickú podobnosť vstupnej vety a vopred definovných správnych odpovedí. Skóre podobnosti študentovej a správnej odpovede môže byť vypočítané pomocou hlbokých jazykových modelov, založených na architektúre transformerov. Pre získanie podobnosti viet (STS) sme preskúmali rôzne modely. Najlepší model stsb-TinyBERT-L-4 (cross-encoder) vylepšuje pôvodný model o 27.8% (stredná kvadratická chyba) na reálnych odpovediach študentov, oskórovaných učiteľom a kalibrovaných lineárnou regresiou. Označenia NLI môžu zlepšiť výsledky, ale je nutný ďalší výskum.

Keywords

sentence embeddings, language learning application, semantic textual similarity

Klíčová slova

kódování vět vektory, aplikace pro výuku jazyka, sémantická podobnost vět

Reference

LIGOCKÝ, Marián. *Answer Correctness Estimation on a Question.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Igor Szőke, Ph.D.

Rozšířený abstrakt

Motivácia. Učenie jazykov bez ľudského učiteľa sa dlho obmedzovalo len na odpovede, ktoré sa dajú vyhodnotiť doslovným porovnaním. Celovetné odpovede neboli možné, pretože majú príliš veľa možných správnych odpovedí. Nedávne vylepšenia v spracovaní prirodzeného jazyka umožnili vytvárať modely, ktoré možno použiť na vyhodnotenie správnosti dvoch viet.

Definícia problému. Majme súbor preddefinovaných správnych odpovedí na otázku. Po odpovedi študenta sa jeho odpoveď porovná s každou správnou odpoveďou. Model určí, či je odpoveď študenta správna a vráti skóre podobnosti. Určovanie sémantickej podobnosti viet (odpovedí) je známe pod skratkou STS (Semantic Textual Similarity). Podobnosť dvoch viet sa dá vypočítať dvoma spôsobmi. Prvý z nich je použitie tzv. architektúry **bi-encoder**, kde model vypočíta vektorovú reprezentáciu pre každú vetu zvlášť a následne sa skóre podobnosti určí ako kosínová vzdialenosť vektorov (Obrázok 1). Tieto reprezentácie sa vypočítajú pre každú vetu iba raz a môžu byť uchované. To urýchľuje najmä porovnávanie veľkého množstva viet naraz. Napríklad ak chceme nájsť dvojicu najpodobnejších viet spomedzi 1000 viet, model sa spustí iba 1000 krát. Uplatnenie nachádza pri úlohách ako sémantické vyhľadávanie, odpovedanie na otázky a kategorizácia. Použitie druhého spôsobu, tzv. **cross-encoder** (Obrázok 2) by nebol použiteľný pre tento druh úloh, pretože vyžaduje ako vstup obe vety naraz, takže pre nájdenie najpodobnejšieho páru viet spomedzi 1000 viet by bolo nutné pustiť model 499 500 krát (počet kombinácií). Cross-encoder dokáže zachytiť zložitejšie vzťahy medzi vetami a lepšie si počínať v úlohách, ktoré si vyžadujú pochopenie vzťahov medzi dvoma vetami. Pre zlepšenie odhalenia negácií a odlišného významu využívame aj NLI (Natural Language Inference). Jedna veta je predpoklad, druhá hypotéza. NLI model vracia tri označenia, podľa toho či sa jedná o dôsledok, rozpor, alebo sa nedá určiť (neutrálne). Pre úlohu NLI sa používajú modely založené iba na architektúre cross-encodera.

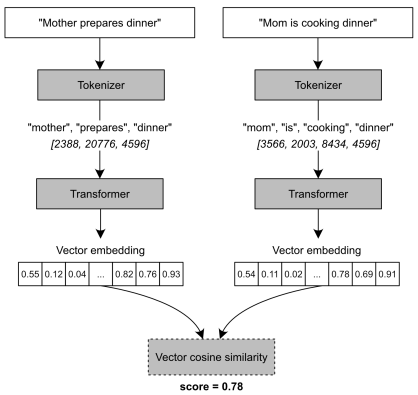

Figure 1: Bi-encoder počíta vektorovú reprezentáciu viet individuálne a porovnáva vektory matematickou operáciou kosínovej podobnosti. Vektory môžu byť uložené v databáze/cache na urýchlenie porovnávania.

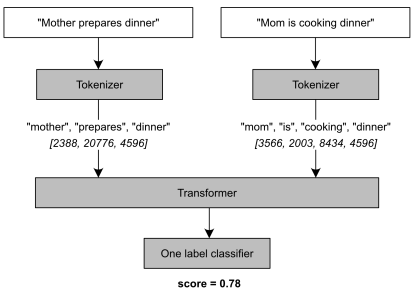

Figure 2: Cross-encoder berie obe vety ako vstup, následne klasifikátor vracia skóre podobnosti. Tento spôsob vie lepšie zachytiť zložitejšie vzťahy vo vetách.

Existujúce riešenia. Súčasné najmodernejšie modely na spracovanie jazyka sú založené na architektúre tzv. transformátorov (BERT, RoBERTA, DeBERTA a iné). Tieto modely možno použiť pre rôzne úlohy NLP, vrátane STS (benchmark STSB) a NLI (benchmarky MNLI a SNLI). Existujúce predtrénované modely sú dostupné na úložiskách (GitHub, Hugging Face), čo uľahčuje ich použitie a možné predtrénovanie na vlastných dátach.

Dostupné dáta. Jednotlivé modely sme hodnotili pomocou zaužívaných hodnotiacich sád. Pre STS to bol dataset STSB, ktorý obsahuje dvojicu viet a ich skóre podobnosti. Pre NLI sme použili kombináciu SNLI, ktoré obsahujú predpoklad, hypotézu a označenie (dôsledok, rozpor, neutrál). Modely sme validovali aj na logoch z aplikácie na výučbu jazyka, ktoré obsahovali otázku, správne odpovede a odpovede študentov so skóre, ktoré získali pomocou starého modelu podobnosti viet a obodovania učiteľom angličtiny.

Naše riešenie. Skúmali sme, či je možné použiť jeden model pre STS a zároveň NLI, prípadne či sú vektorové reprezentácie po pretrénovaní na NLI presnejšie na detekciu negácie. Zistili sme však, že nie je možné použiť jeden model pre STS aj NLI. Po pretrénovaní na NLI sa znížilo skóre v úlohe sémantickej podobnosti (STS) veľmi výrazne. Pretrénovaný model šlo použiť na NLI, ale už nie na STS. Lepší prístup je použiť dva samostatné modely. Nami vybraný model pre sémantickú podobnosť viet sme nakalibrovali na dodaných dátach pomocou lineárnej regresie. Pôvodné skóre sme vylepšili o $27,8\%$ (stredná štvorcová chyba) pri použití nového modelu založeného na architektúre cross-encoder, *stsb-TinyBERT-L-4*. Pokiaľ je rýchlosť kľúčová, odporúčame zachovať pôvodný model, ktorý sa ukázal ako najlepší spomedzi bi-encoderov. Kalibráciou sme zlepšili systém o 18.5% (stredná štvorcová chyba).

Možné rozšírenia. V práci sme rozoberali potenciál využitia modelov pre NLI označenia viet. Zistili sme, že prínos závisí od konkrétne použitého STS modelu. Táto oblasť si ale vyžaduje ďalšie skúmanie, kedže z dôvodu nedostatku dát sme nemohli vierohodne určiť v akej podobe sa má výsledné označenie reflektovať do konečného hodnotenia podobnosti. Všetky STS modely mali veľký problém detekovať nesprávnu gramatiku. Navrhujeme preto pridať aj gramatický model, ktorý by hľadal gramatické chyby, klasifikoval ich a upravoval hodnotenie, aby reflektovalo aj nesprávnu gramatiku, nie len sématickú správnosť.

Answer Correctness Estimation on a Question

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Igor Szőke, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Marián Ligocký
May 9, 2023

Acknowledgements

I would like to express my gratitude to the supervisor of this thesis, Ing. Igor Szőke, Ph.D. for his patience and guidance and all knowledge he was willing to share. Special thanks to my dear Natálka and my parents for their support and encouragement. I also thank the Faculty of Information Technology for the education in the field of computer science.

Contents

Chapter 1

Introduction

Goal of thesis. This thesis aims to develop a model that evaluates the correctness of a student's sentence-alike answer to a question in a language learning application GoLearn[1]. Correctness should be evaluated by comparing the answer with predefined correct answers. Our results are compared with the app's previously used sentence similarity scoring system.

Exercise example. Imagine an exercise where user has to say hello to his neighbour. This is a speaking task, so student uses microphone to answer: „Hello, I am your neighbour."

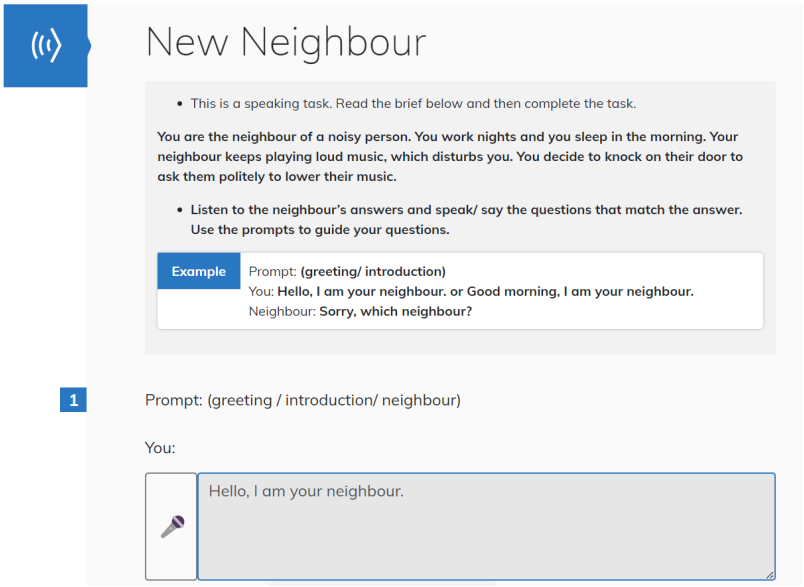

Figure 1.1: A sample question from application GoLearn. Description of situation and first question for student. Because this is a speaking exercise, student uses microphone.

Problem definition. There are many correct answers possible, but in the app, there are only three predefined correct answers and none of them literally matches the student's answer: „Good morning, I'm your neighbour", „Hello there I am your neighbour", „Hello, I live next door".

[1] https://diagnostics.golearn.guru/tasks

Usually, this answer would be rejected because it does not match any correct answer literally. However, the sentence similarity system understands that the answer from the user is very similar to one of the given correct answers. Therefore the answer might be accepted as correct.

To find the sentence similarity, the most trivial solution is to compare individual words syntactically, for example, to find a number of common words. This approach does not understand the true meaning of a sentence since it focuses only on its word structure. It would fail to understand synonyms, negation and semantic nuances. The better solution for finding sentence similarity is to compare their semantics.

Word embeddings. It is possible to construct word embeddings and compare two sentences by embeddings of individual words in the sentence. In Chapter 2 we talk about methods of word vector embeddings.

Sentence embeddings. Vectors of words in a sentence can be combined together to form a sentence embedding [12]. However, the-state of art approach is to use Transformer architecture [24]. It is a deep neural network which is more powerful and can capture more complex relationships between words and sentences using self-attention. It might be used as a bi-encoder, where a score of similarity is calculated as a cosine similarity of sentence vectors, or as a cross-encoder, where both sentences are put into the model at the same time and score is outputted from classification layer. Chapter 3 talks about sentence embeddings, transformer architecture, individual models (BERT, RoBERTa), bi-encoder and cross-encoder architectures.

NLP tasks. In Chapter 4, we describe two different NLP tasks that can be used to evaluate the similarity of sentences. In the beginning, it seemed that only Semantic Textual Similarity is needed. However, experiments showed that some STS models have problems with negations, for example, „Mom is cooking a dinner.“ and „Mom is not cooking a dinner“. To find whether two sentences are entailment or contradiction, a different task Natural Language Inference is used, which returns one of three labels (entailment, contradiction, neutral) for a pair of sentences. We examine whether using NLI labelling might improve the performance of models.

Fine-tuning. We decided to fine-tune a model, originally pretrained on STS to NLI task, to see whether the fine-tuning solves problems with detecting contradictions. The process of fine-tuning with results is described in Chapter 5.

Chapter 2

Vector representation of words

When working with the meaning of words in the Natural Language Processing, it is necessary to represent them in a data structure that can contain information about their meaning. As described in this chapter, words can be represented in many ways. However, all methods share the same data structure, a **vector**. In this chapter, we discuss different encoding methods for the representation of words in vectors. At the end of the chapter, we talk about tokenization and different tokenizers which split words into smaller parts, tokens.

2.1 One-hot encoding

The most straightforward idea is to encode each word of a vocabulary into a vector of zeroes vector with a length equal to the vocabulary and place a one in the index corresponding to the word. Then, it is possible to concatenate the vectors to create a representation of the sentence.

This approach is used sparingly due to its drawbacks. It is inefficient because most numbers are zeros, so valuable data are sparse. This might be solved by only saving the index of the word in a vocabulary, but this type of integer encoding is challenging for certain models, such as linear classifiers.

The second problem is that extracting the meaning from the words is impossible. The distance among all vectors is the same.

2.2 Word features

Another idea is to have information about certain features of the words. As shown in Table 2.1, let's have words **king**, **queen**, **man**, **woman**, **girl**, **horse**. For these words, we might have features:

- **gender** - male is positive one, female is negative one,
- **authority** - zero means no authority, one means full authority,
- **richness** - similar to authority,
- **animal** - zero means it is not an animal, and one means it is

Words	Gender	Authority	Richness	Animal
King	1	0,8	0,9	0
Queen	-1	0,8	0,8	0
Man	1	0,2	0,3	0
Woman	-1	0,2	0,3	0
Girl	-1	0,1	0,1	0
Cat	0	0	0	1

Table 2.1: Words are represented as vectors with four features (dimensions)

Now it is possible to perform arithmetic operations with the vectors, such as sum and subtraction. If we take the vector for the word **king** minus **man** plus **woman**, the result is very similar to the vector representation of the word **queen**.

	Gender	Authority	Richness	Animal
King - Man + Woman	-1	0,8	0,9	0
Queen	-1	0,8	0,8	0

Table 2.2: Vector arithmetic is possible for word vectors that include features.

This approach is better than one-hot encoding because it enables semantic information about the words (their meaning). Nevertheless, the question is how to choose meaningful features and find the correct value for a feature in a word. In practice, finding the features and values manually is impossible, and an algorithm must be used. It is possible to learn word embedding by solving a problem (for example, filling a missing word into a sentence). By solving this problem via a neural network, we can get word embedding as a side effect (weights of neurons).

2.3 Word2Vec

The breakthrough in the computation of word embeddings came with the tool **word2vec**[1] developed by the team led by Tomas Mikolov at Google [15]. The main idea is that surrounding words can infer the word's meaning. It uses a neural network to learn the word associations from a large corpus of text. A fixed-size vector then represents each word. The semantic information is present in this vector, so two words with similar meanings are closer in the vector space. Also, it is possible to ask a question: „What is the word that is similar to *small* in the same sense as *biggest* is similar to big?“ and get an answer by simple vector arithmetic [15].

Two different architectures are used to find word embeddings. Examples found in this section are from [8]. The first architecture **skip-gram** is based on predicting words around the current word (Figure 2.1). The number of words before and after is called *window size* m, forming a context window. So for $m = 1$, one word before and one word after is used. The context window is of size $2m + 1$ (including the target word w_i) and is in the form of $[w_{i-m}, \ldots\ w_i, w_{i+1}, \ldots, w_{i+m}]$. Then pairs are formed as a combination of all pairs of the current word and one word from the context window:

$$[\ldots, (w_i, w_{i-m}), \ldots\ (w_i, w_{i-1}), (w_i, w_{i+1}), \ldots, (w_i, w_{i+m}), \ldots]$$

[1] https://code.google.com/archive/p/word2vec/

For a sentence: „*The dog barked at the mailman.*“ and window size $m = 1$, all pairs in the form (*input*, *output*) are:

[(dog, The), (dog, barked), (barked, dog), (barked, at), ... (the, at), (the, mailman)]

When we have the pairs in the *(input, output)* format, the neural network may be used to learn the word embeddings. The matrix where the word embeddings are stored is of size $V \times D$, where V is the vocabulary size (usually in thousands) and D is the size of word embedding (how many numbers represent the word). This matrix is called *embedding space* or *embedding layer*. The size of D (word embedding) is a user-defined parameter, usually around 512 - 1024. The higher the embedding size, the more computation power is needed.

The second architecture that is used is called **The Continuous Bag-of-Words**. It is similar to skip-gram, but it does not predict the context words from the target word, but the target word from contextual words (Figure 2.1). So the *(input, output)* pairs for the previous sentence would look like this:

[([the, barked], dog), ([dog, at], barked), ...]

The result from the authors of *word2vec* shows that CBOW model works better on syntactic tasks, but skip-gram works much better on semantic tasks [15].

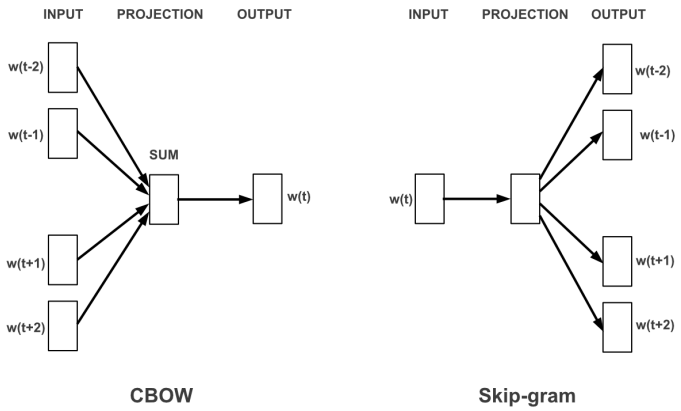

Figure 2.1: The CBOW architecture tries to predict the current word based on words around it. The skip-gram predicts words around the current word. (Source of image: [15])

2.4 Glove

GloVe is another unsupervised learning algorithm for creating word embeddings. The word embeddings are learnt on aggregated global word-word co-occurrence statistics from a corpus [17]. The previous tool word2vec utilized only methods based on **local context window** (skip-gram, CBOW). At the same time, GloVe also uses **global matrix** factorization-based method **Latent Semantic Analysis (LSA)** for the global context of words. Global matrix factorization-based methods can get information about the co-occurrence of words in the global scope but cannot work on word analogy tasks. On the other hand, context

window-based methods work well at word analogy tasks but are missing global scope information. Since GloVe uses both approaches, it maintains performance on word analogy tasks and contains global scope information [8].

Glove works with the ratios of word-word co-occurrence probabilities. Two words are more similar if they occur with higher probability next to each other than next to various probe words, k. The example from the authors of GloVe (Figure 2.3) uses probabilities of co-occurrence of target words *ice* and *steam* with other words k = [*solid*, *gas*, *water*, *fashion*]. The probability ratio in the third row of Table 2.3 is the primary indicator of whether two words are similar. If the resulting number is higher than one, the word k and the word *ice* are closer in meaning to each other (*ice-solid*). If the number is less than one, the word k and *steam* are close in meaning (*ice-gas*), and if the number is around one, the word k does not have a specific relationship with any of the target words [17].

Probability and Ratio	**k = *solid***	**k = *gas***	**k = *water***	**k = *fashion***
P(k|ice)	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
P(k|steam)	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-4}	1.8×10^{-5}
P(k|ice) / P(k|steam)	8.9	8.5×10^{-2}	1.36	0.96

Table 2.3: Probabilities of co-occurrence of words *ice* and *steam* with various words k. The word *ice* and the word k are closer in meaning if the ratio in the third row is higher than one. If it is smaller than one, the word k and *steam* are closer in meaning, and if the ratio is around one, there is no specific relation with any of the target words. (Source [17])

2.5 FastText

The previous methods ignored the morphology of words because every word of vocabulary had its vector. This is a limitation in the case of a language with an extensive vocabulary and many rare words. FastText proposes a new skip-gram model where words consist of smaller parts. The resulting vector for a word is the sum of its smaller parts' vectors [3]. Each word w is represented as a bag of character n-gram, and special characters $<$ and $>$ are used to distinguish prefixes and suffixes from other character sequences. For example the word „*where*“ and $n = 3$, the representation is *<wh, whe, her, ere, re>* and a special sequence *<where>* (the word w is also included in the set of n-grams).

It was observed that the optimal n is in the range 3-6. The resulting score in various benchmarks and languages shows that splitting a word into n-grams leads to better similarity scores. The training time is also faster, and the model can determine the meaning of new words in vocabulary if they consist of already known subparts [3].

2.6 Tokenization

Tokenization is a process of breaking down a text into smaller units called tokens. The most trivial way of tokenization is to break individual words by white space. This is problematic because words not included in the vocabulary are treated as unknown. Some languages, such as Turkish and German, make words as a combination of other words. Their vocabulary can be very large. Advanced algorithms break a text into sub-words, which are defined in the vocabulary. This vocabulary has a limit of words, for example, 5000, which means that only 5000 different tokens are saved. Frequently used words should not be split into smaller

subwords, but rare words should be decomposed into meaningful subwords. Let's take an example sentence:

„*The Transformer is an innovative NLP model!*“

it would split into:

['the', 'transform', 'er', 'is', 'an', 'innovative', 'n', 'l', 'p', 'model', '!']

Notice word *NLP*, which is separated into three individual tokens because it is a rarely used word. String tokens would be converted to unique numbers such as:

[1996, 4937, 7771, 2006, 1996, 6411, 1012, 2009, 200, 345, 877].

Cleaning

Input sentences might contain punctuation, contractions (do not = don't) and other language specific formulations. **Rule-based** tokenizers such as spaCy[2] are able to clean sentences.

WordPiece

WordPiece algorithm was used in Google translation services, where authors [27] tried to solve the problem of rare words in vocabulary by dividing these words into smaller parts *wordpieces*.

According to the Hugging Face tutorial[3], individual units are extracted by statistical analysis of the training data. The vocabulary ranges from 10 up to 100 thousands of subword units. First, a pre-tokenizer separates words in a sequence by a space. Next, the frequency of occurrence in the training corpus for each word is found. The first units placed into vocabulary are all base symbols in the training data (all letters, digits, punctuation). Later these base symbols are merged by specific rules based on their occurrence until desired vocabulary size is attained (for example, letters „*c*“ and „*a*“ are merged into „*ca*“, and later it is combined with the letter „*t*“ to form „*cat*“). The vocabulary size is a hyperparameter that can be changed.

To find which symbols to merge, it finds a symbol pair whose frequency, divided by the frequencies of individual symbols, is the greatest. The score for each pair of symbols is therefore calculated as:

$$score = \frac{f_{1,2}}{f_1 \cdot f_2}$$

Where f_1 is the frequency of the first symbol, f_2 is the frequency of the second symbol and $f_{1,2}$ is the frequency of the first and second symbols concatenated.

[2] https://spacy.io/api/tokenizer
[3] https://huggingface.co/learn/nlp-course/chapter6/6?fw=pt

2.7 Summary

Word embeddings are an important building block for sentence embeddings. The described methods of encoding words into vectors (Word2Vec, Glove, FastText) can be used directly to compute sentence embedding by averaging all word vectors, however this approach is not used nowadays (doc2vec). The state-of-art method of sentence similarity, Transformer, learns their own word/token representations. However it is important to understand how word/token embeddings work to understand methods for sentence embedding generation.

Chapter 3

Vector representation of sentences and documents

Sentence and document embeddings can combine embeddings of individual words into the final vector that describes the whole sentence or document, as done by Doc2Vec. This document embedding can be used to categorise or to find similar documents and sentences in the same way as it is done with words. The state-of-art methods based on Transformer architecture [24] uses a different approach based on self-attention mechanism.

3.1 Doc2Vec

Doc2vec is an unsupervised machine-learning algorithm that converts a document to a vector. Mikolov and Le presented this concept in their paper [12] and is heavily dependent upon word2vec. The input can be of different lengths, so it might be used to convert a sentence, paragraph or the whole document. The output is a dense vector of fixed-sized length called Paragraph Vector. The word vectors are trained using the *word2vec* algorithm [12].

The Doc2Vec algorithm consists of two main models: the **Distributed Memory** (DM) model and the **Distributed Bag of Words** (DBOW) model. The DM model extends the Word2Vec algorithm by adding an extra vector to represent the document, in addition to the word vectors. The DBOW model, on the other hand, ignores the context words and only trains the document vector to predict the next word in the document. Both models are trained using a stochastic gradient descent algorithm and a negative sampling technique to reduce the computational cost [12].

During training, the algorithm learns to assign similar vector representations to documents that have similar contexts or meanings. The resulting document vectors can be used for a variety of NLP tasks such as document classification, clustering, and similarity search.

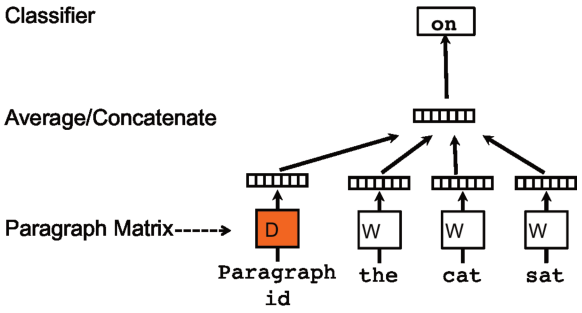

Figure 3.1: The architecture for learning paragraph vectors. The paragraph vector represents information about the current context. It works as a memory of the topic. (Source of image: [12])

This approach set new state-of-art performance for sentiment analysis tasks, and it was better than bag-of-words models by more than 30%. [12]

3.2 Transformer Architecture

The Transformer was a breakthrough in the research of NLP models. It uses an attention mechanism which achieves state-of-art performance for various NLP tasks and is much faster to train. Its architecture allows us to understand the relationship of words that are sequentially far from each other. It is easily parallelisable. Before the Transformer, recurrent models were used. They use sequential computation by processing the input words individually, preventing high-performance computing such as GPUs [21].

The original Transformer in the paper *Attention Is All You Need* [24] uses a stack of 6 layers of encoders and decoders and is used for text translations. The output of layer l is used as an input of layer $l + 1$ (Figure 3.2). The encoder's layers use all the tokens in the sequence to find which words are affected by which other words. On the other hand, the decoder's layers work sequentially and pay attention to previously generated words.

Encoder stack

The encoder stack consists of N layers that contain two sub-layers: a **multi-headed attention mechanism** and a fully connected position-wise **feedforward network**. To make sure that key information such as positional encoding is not lost on the way, there are two residual connections: 1) between the input of the multi-headed attention mechanism and input of the first normalisation layer, and 2) between the input of feedforward network and input of the second normalisation layer. The output of each normalisation layer is calculated for the input x as [24]:

$$output = LayerNormalization(x + Sublayer(x))$$

In the original paper, there are $N = 6$ layers. The first layer is responsible for word embeddings; other layers learn from the previous layer and explore different associations of the tokens in the sequence. The output of each layer is of fixed size d_{model}, which speeds up the computation of dot products [21]. In the original paper, $d_{model} = 512$ [24].

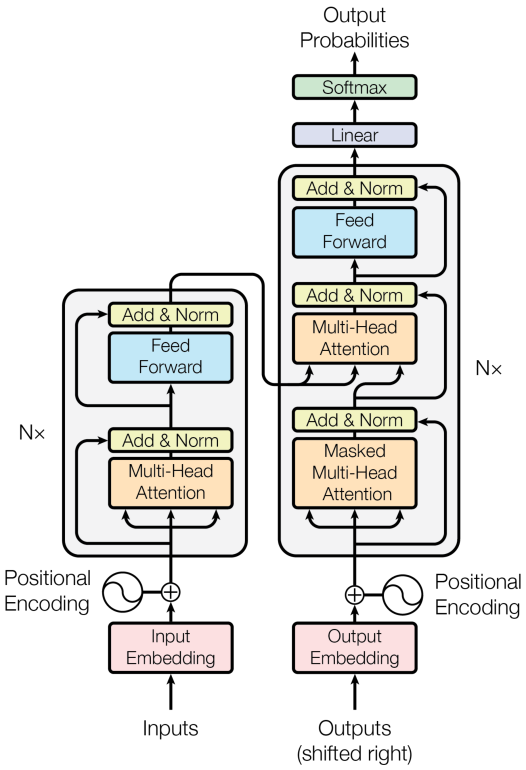

Figure 3.2: Transformer architecture proposed in the original paper Attention Is All You Need. On the left and right sides are encoder and decoder blocks with stacked sub-layers. (Source of image [24])

Input embedding

The **Input Embedding** (Figure 3.2) on the bottom of the encoder stack converts input tokens into vectors of size d_{model} using learnt embeddings. These embeddings might be created by various algorithms such as previously discussed *word2vec*, *Glove* and *Fast Text*. Input tokens are created by a **tokeniser**.

Later, these numbers are used as input for embedding algorithms such as *skip-gram* or *CBOW* described in Section 2.3. Vectors of size d_{model} are created for each token, in contrast to *word2vec*, where vectors are created for each word. Various tokenizer models are available from Hugging Face[1]

Positional encoding

Next, positional encoding is calculated for each token in the sequence. This has to be done because the model would need more information about the order of tokens in the sequence. The positional encoding and the embeddings can be summed because they have the same dimension d_{model}. There are many possibilities for positional encoding algorithms. The goal is to find a way to provide information about the position to every dimension i of embedding in range(0, $d_{model} = 512$). The authors used sine and cosine functions for positional encoding where *pos* is a word's position in a sequence, and i is the dimension. Because the sinusoid is different for each position, it enables encoding each position uniquely.

[1] https://huggingface.co/docs/transformers/main_classes/tokenizer

$$PE_{pos,2i} = sin(\frac{pos}{10000^{2i/d_{model}}})$$
$$PE_{pos,2i+1} = cos(\frac{pos}{10000^{2i/d_{model}}})$$

This function was chosen because the authors hypothesised it would allow the model to learn to attend to relative positions quickly. Also, learned positional embeddings were tried, but with similar results [24].

After both positional encoding and word embeddings are known, they are summed up to create the final output that is used as input for the multi-head attention layer (Figure 3.3). Before summing, token embeddings are multiplied by a constant $\sqrt{d_{model}}$ [24] to give more weight to token embedding and less weight to positional encoding [21].

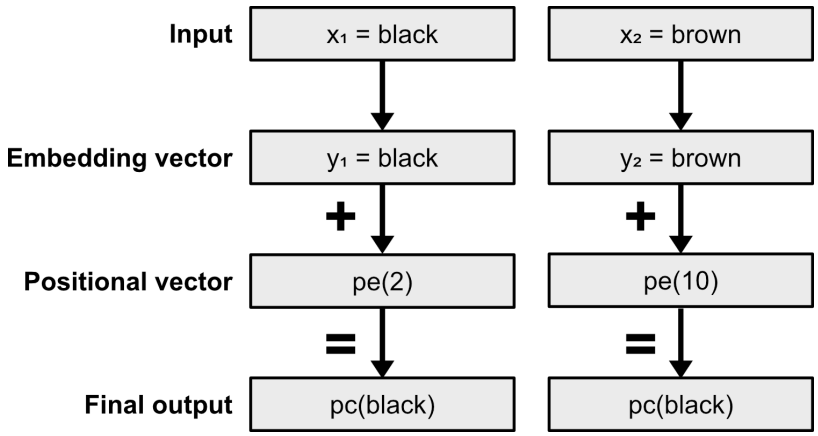

Figure 3.3: Embedding vector and positional vector are summed up to create final output for input words *black* and *brown* and their position in the sequence *2* and *10*.

Multi-head attention

Multi-Head Attention implements a self-attention mechanism. This mechanism maps each word to other words to see how they fit in a sequence. It calculated weights representing the relative importance of the tokens in the sentence [21]. Imagine a sequence:

„The cat sat on the rug, and it was dry-cleaned."

The self-attention should find out that word *it* is related to nouns *cat* or *rug*.

Authors advise not to train the model using $d_{model} = 512$ but to divide it into 8 $d_k = 64$ dimensions. The heads can run in parallel, and they calculate eight different representations of how words relate to each other. The output of a multi-attention head is defined as a concatenation of results of individual heads:

$$output = Concat(Z) = Concat(z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7)$$

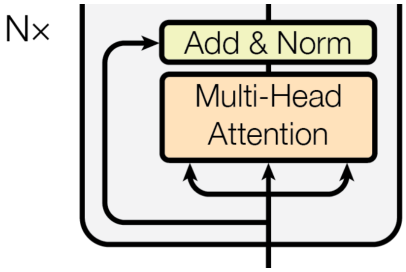

Figure 3.4: Each of the eight heads has input and output of length $d_k = 64$. Input is taken from the previous layer or, in the case of the first layer, input embeddings and positional encoding. (Source [24])

The first layer's multi-head input is a vector containing positional encoding and embedding for each token, as described in the previous section. The following layers do not start these operations over.

In each head h_n, there are three trained vectors for each word:

- query vector Q containing the current token vector from the input sequence, length of d_k
- key vector K containing all token vectors from the input sequence, length same as query vector d_k
- value vector V containing all token vectors from the input sequence, length of d_v

In practise, the attention function is calculated on matrices Q, K, V, instead of single vectors, simultaneously [24]:

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

This equation benefits from fast matrix multiplication operations.

Feedforward network

A Feedforward network is an artificial neural network where connections among nodes do not form a loop. It is one of the simplest types of neural networks where information moves only forward [18].

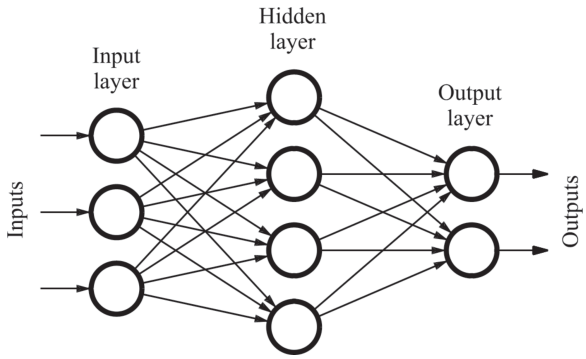

Figure 3.5: Feedforward neural network consists of input layer, n hidden layers and output layer. There are no loops and cycles, and information is going only forwards. (Source [18])

In Transformer original architecture, each feedforward network in the encoder and decoder is described as a fully connected and position-wise network. It contains two layers and uses a *Rectified Linear Unit (ReLU)* [16] activation function. The input and output are of length $d_{model} = 512$, but the inner hidden layer is $d_{ff} = 2048$. A linear transformation is the same across different positions, but each layer has different parameters for the weights W_1, W_2 and biases b_1, b_2 [24] [21].

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

The output of the FFN is normalised in the normalisation layer (Figure 3.2). It goes into the next layer of the encoder and multi-head attention layer of the decoder stack [21].

Decoder stack

In the original paper, the Transformer translates the English sequence into French and German. The encoder stack, which consists of $N = 6$ layers, each one with multi-head attention, feedforward network and two normalisation sub-layers, deals with encoding the English sequence into features that are later outputted as French or German sequence by **decoders** (Figure 3.6) [24].

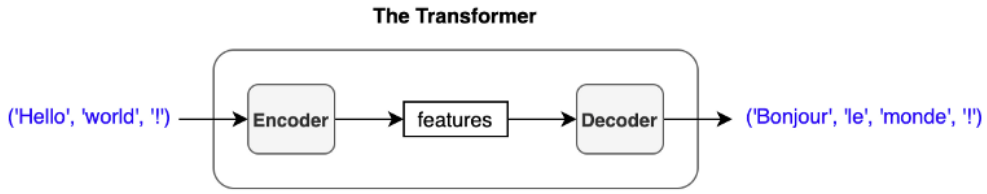

Figure 3.6: Encoders are used to transform English sequences into features decoders transform into French. (Source [22])

The stack of decoders contains the same number of layers as encoder $N = 6$. There are the same architectural blocks as in encoders: multi-head attention and a fully connected position-wise feedforward network. However, each decoder layer contains a third sub-layer **masked multi-head attention** (Figure 3.7). This sub-layer receives input with masks, preventing the attention mechanism from seeing masked positions. The decoder is fed the whole sequence when the model is trained. Hence, the masking ensures that the model cannot see future parts of the sequence during training [21]. So when generating the fifth sequence token, the decoder has access only to tokens on positions 1 to 4.

The output sequence is generated in a pattern called *auto-regressive*, commonly used for sequential outputting (one token at a time). It allows for generating an output of a different length than the input. The decoder output is used as additional input for generating the next token [21].

There are more methods for how the decoder knows which word to use as output. The simplest one is *greedy method*, which selects the token with the highest probability number. However, looking for the best combination of the tokens is better, not just a single token, so a heuristic search algorithm called *beam search* is used. It finds the best combination of tokens with the highest probability. This modified search algorithm [27] is used in the original Transformer with a beam size of 4 and length penalty $\alpha = 0.6$ [24].

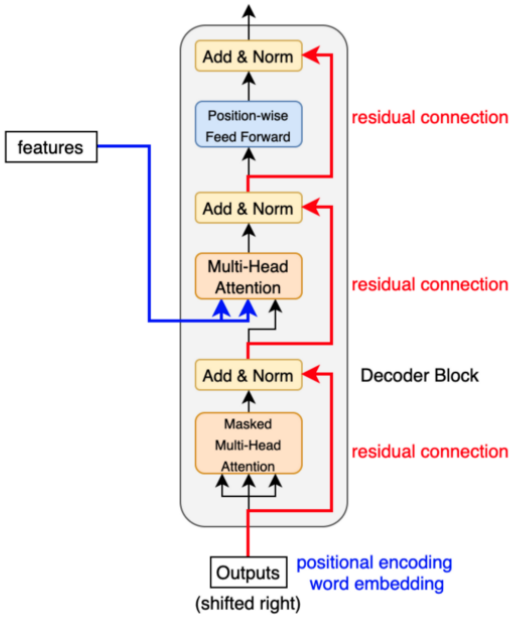

Figure 3.7: One layer of a decoder stack. Features from the encoder and previously masked outputs are input for multi-head attention. (Source [22])

Different architectures

While the original Transformer model was used for the translation of text and contained both an encoder and decoder, there are more architectures:

- Encoder-only transformers: for tasks where the crucial part is understanding of text such as text classification, named entity recognition (*Albert*, *Bert*, *Roberta*)
- Decoder-only transformers: for tasks where text is generated (*GPT-2*)
- Encoder-decoder (sequence-to-sequence): where some text is generated from the input text, used for translations or summarising (*BART*, *T5*)

3.3 BERT

Bidirectional Encoder Representations from Transformers is a model that uses encoder-only architecture from the Transformer model. It introduced bidirectional attention, meaning that an attention head attends to all the words from left to right and right to left. Previous methods used only the left-to-right approach [21].

This allows BERT multiple vector representations for the same word based on the context. Imagine a word *rock* in two sentences:

My friend likes to play *rock* music on guitar.
A geologist is studying a *rock* he has found in the ground.

The previous methods for word embeddings such as *word2vec* would give the same vector embedding for the both words *rock* making it **context-independent**. The BERT knows the different contexts of the sentences. Thus BERT embeddings are **context-dependent**.

The model is trained on two different tasks:

- Masked Language Modeling (MLM)
- Next Sentence Prediction (NSP)

The first task is **Masked Language Modeling**, where BERT uses a random mask on a word of a sentence, and then it tries to predict the masked word. To improve the results and prevent over-fitting, authors used masking [7]:

- in 80% of cases the random mask was used: *my dog is hairy -> my dog is [MASK]*
- in 10% of cases no mask: *my dog is hairy -> my dog is hairy*
- in 10% of cases the token was replaced with a random token *my dog is hairy -> my dog is apple*

The second task is **Next Sentence Prediction**. Two special tokens *[CLS]* and *[SEP]* are introduced. *[CLS]* is a binary classification token inserted at the beginning of a sequence. It predicts if the second sequence follows the first one. *[SEP]* is a token that separates two segments. The training goal for BERT is to determine whether the two sentences are consecutive, which means they follow each other. A positive sample is a pair of successive sentences from a dataset such as [21]:

[CLS] mother cooked a dinner [SEP] it was delicious.

A negative sample is a pair of sentences from different documents, such as:

[CLS] birds are signing [SEP] he fell down on ice.

The final embedding for an input is a sum of token embeddings, sentence embeddings and positional embeddings (Figure 3.8).

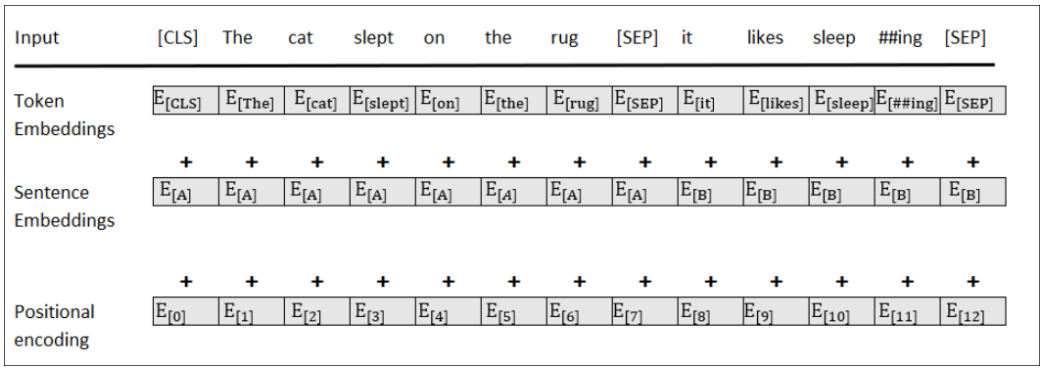

Figure 3.8: The input embeddings are the sum of token embeddings, sentence embeddings and positional encoding embeddings. (Source [21])

WordPiece tokeniser with a 30,000-token vocabulary was used to decompose words into tokens. In contrast to the sine and cosine method for positional encoding from the original Transformer, discussed in Section 3.2, positional encodings are learnt [7].

The original paper describes two models [7]:

- $BERT_{BASE}$ consists of 12 layers (Transformer blocks), 12 self-attention heads and length of model $d_{model} = 768$ (compared to $d_{model} = 512$ in the original Transformer).
- $BERT_{LARGE}$ that consists of 24 layers, 16 self-attention heads and $d_{model} = 1024$

There are many pre-trained models on HuggingFace for different languages [2]. These models can be fine-tuned to perform better at particular NLP tasks such as Question Answering, Recognising Textual Entailment and others [21].

Sentence Embeddings

To get sentence embedding from BERT for one sentence, it is necessary to use input with [CLS] token at the beginning and [SEP] token at the end (BERT expects two sentences on input, but the second sentence can be empty). For this task, BertTokenizer[3] is utilized. It adds [CLS], [SEP] to the sentence and converts the whole sentence to vocabulary ids. Then *BertModel*[4] is used to encode input into features (found in the hidden state of the model). In the last layer, there is an embedding of size $d_{model} = 768$ for each token. Sentence embedding is calculated as an average of all token embeddings in the sentence.

```
import torch
from transformers import BertModel, BertTokenizer

# load models from Hugging Face
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model_bert = BertModel.from_pretrained(
'bert-base-uncased', output_hidden_states=True)

# add [CLS], [SEP] and convert to ids
token_ids = tokeniser.encode("Birds are signing.", return_tensors="pt")

with torch.no_grad():
    out = model(input_ids=token_ids.unsqueeze(0))

# use mean to average last hidden layer
sentence_embedding = torch.mean(out.last_hidden_state, dim=1).squeeze()
```

For this code example, **Last Hidden** method is used to get embedding from the internal state of the model. Other options for getting sentence embeddings are:

The similarity of two sentences can be calculated as the cosine similarity of their separate embeddings, as shown in Section 4.1. However, this approach is often worse than averaging GloVe embeddings [20]. While it is possible to get embedding for one sentence, BERT is not adapted and does not perform well. BERT is very good on various sentence classification and sentence-pair regression tasks. It uses a cross-encoder where two sentences are passed as input, and the target value is outputted [20]. So for computing the similarity score of two sentences, the advised approach is to use both sentences as input for BERT, separated by special token [SEP] and then get the score from the last layer, or another approach in Table 3.1. Computing the similarity score for each pair of sentences in a collection of

[2] https://huggingface.co/models
[3] Hugging Face: BertTokenizer
[4] Hugging Face: Bert

Method	Score
Embeddings	91.0
Second-to-Last Hidden	95.6
Last Hidden	94.6
Weighted Sum Last Four Hidden	95.9
Concat Last Four Hidden	96.1
Weighted Sum All 12 Layers	95.5

Table 3.1: The authors of BERT [7] experimented with these methods for extraction of sentence embeddings from the internal state of $BERT_{BASE}$. Scores are calculated on *Named Entity Recognition* task and *CoNLL-2003 Named Entity Recognition* dataset.

10 000 sentences requires about 50 million computations (around 65 hours). Therefore, the architecture of BERT is unsuitable for sentence similarity search and clustering because it would take too long [20].

3.4 SBERT

Sentence-BERT (SBERT) is a Python framework for state-of-the-art sentence[5], text and image embeddings. It uses siamese and triplet network structures to compute fixed-sized vector embeddings for separate sequences [20]. These embeddings can be saved for each sentence and later compared by cosine similarity to derive a similarity score. This approach is called **Bi-Encoder** (Figure 3.9 left). SBERT also supports **Cross-Encoder** approach (Figure 3.9 right), which does not compute individual embedding, but takes two sentences as input and then utilises a binary classifier, where a score of similarity is a probability of the label. SBERT framework contains a class *Cross-Encoder*, a wrapper on AutoModelForSequenceClassification class from the library *transformers*.

Bi-Encoders are efficient where there is a given sentence S_1, and we want to compare it with a huge set of sentences A. A sentence embedding for each sentence in a set A is computed and then stored. When comparing a sentence S_1 with every sentence in A, an embedding for S_1 is computed and then using cosine similarity compared with every embedding in the set A. For different sentences $S_2, S_3, S_4, ...$, only their embeddings have to be calculated. The following comparison of newly generated embedding and already stored vectors for sentences A is a relatively cheap operation. To find a similarity score for each pair of sentences in a dataset containing 10 000 sentences, SBERT needs only 5 seconds compared to 65 hours with BERT (the same hardware setup) [20].

[5] https://www.sbert.net/

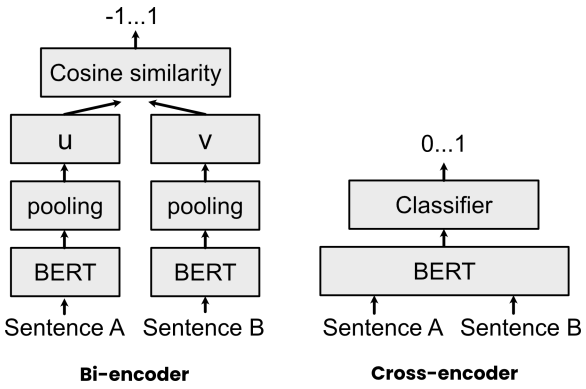

Figure 3.9: Bi-encoder creates two separate embeddings that are compared by cosine similarity. Cross-encoder uses both sentences as input to derive sentence similarity.

SBERT adds a new layer that performs a pooling operation to the output of BERT / RoBERTa / MPNet to derive a fixed-sized sentence embedding [20]. There are three polling strategies that the authors experimented with:

1. output of the *CLS* token,
2. mean of all output vectors (default configuration),
3. max-over-time of the output vectors.

The objective functions that were used [20]:

- Classification Objective Function which was used to train SBERT for *Natural Language Inference* task described in the Section 4.2. The sentence embeddings u and v are concatenated with element-wise difference $u - v$ and multiplied with the trainable weight W_t:
$$o = softmax(W_t(u, v, |u - v|))$$
- Regression Objective Function which was used to train SBERT for *Semantic Textual Similarity* task described in the Section 4.1 The cosine similarity of two embeddings u and v is computed:
$$o = cosineSimilarity(u, v)$$
- Triplet Objective Function. It tunes the network such that the distance between anchor sentence a and positive sentence p is smaller than the distance between a and negative sentence n. Example found in [20]: Anchor: „*Arnold joined the BBC Radio Drama Company in 1988.*“, positive: „*Arnold gained media attention in May 2012.*“, negative: „*Balding and Arnold are keen amateur golfers.*“

The authors of SBERT created a Python framework *SentenceTransformers*[6] for text and image embeddings. Even though it is called SBERT, it is not restricted only to BERT models, but the principle can be applied to other pre-trained models. There are more than 100 of them on Hugging Face repository[7].

[6] https://www.sbert.net/
[7] https://huggingface.co/sentence-transformers

To illustrate, finding cosine similarity for all corpus pairs is done by following the code snippet. Pre-trained model *all-MiniLM-L6-v2* maps sentences to 384-dimensional dense space (vector of size 384). It was chosen because it is five times faster than the best performing model *all-mpnet-base-v2*, but it still offers good quality. Information about models can be found in the documentation of SentenceTransformer[8] or Hugging Face repository.

```
from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer('all-MiniLM-L6-v2')

# Single list of sentences
sentences = ['The cat sits outside',
             'A man is playing guitar',
             'Do you like pizza?']

# Compute embeddings
embeddings = model.encode(sentences, convert_to_tensor=True)

# Compute cosine-similarities for each sentence with each other sentence
cosine_scores = util.cos_sim(embeddings, embeddings)
```

3.5 RoBERTa

The previously discussed BERT model has been shown to be undertrained by the authors of *RoBERTa* [13]. This new model *Robustly optimised BERT approach* increases performance by improving the mechanics of the pretraining process and finding the best values of hyperparameters.

The authors have found that the following aspects improved the previous BERT model and outperformed methods that were introduced after BERT [13]:

- training the model with bigger batches and longer
- removing the next prediction objective
- training on longer sequences
- changing the masking pattern of the training data dynamically
- Tokenizer *Byte-level Byte-Pair Encoding*

Training data

In addition to *BookCorpus* and English *Wikipedia* (16GB) that were used to train BERT, the authors of RoBERTa also used:

- *CC-News* with English news articles (76GB)
- *OpenWebText* web sites whose URLs were shared on Reddit and had at least three upvotes (38GB)

[8] https://www.sbert.net/docs/pretrained_models.html

- *Stories* a dataset containing a subset of CommonCrawl data that match the story-like style of Winograd schemas (31GB)

Dynamic masking

The original BERT relies on one static mask randomly created during data preprocessing. This creates a problem that the same mask is used for each training instance in every epoch. To solve this problem, training data are duplicated ten times so that there are ten different masks [13], creating a training overhead. On the other hand, RoBERTa uses *dynamic masking*, where the masking pattern is created every time a sequence is fed into the model. The performance of the *dynamic masking* is similar or better than *static masking*, but because there are no duplicated data, it is much faster [13].

Next Sentence Prediction and input format

The *Next Sentence Prediction* is one of two tasks the original BERT model utilises for training. The authors of BERT hypothesised that it is an important factor in training and that removing this task degrades the performance [7].

The authors of RoBERTA examined that when they used different input formats for the model and removed NSP loss, the results were similar or slightly better on downstream task performance. The input format they used is called *FULL-SENTENCES*, and it consists of complete sentences sampled from one or more documents with a total length of at most 512 tokens. The extra separator token is added between different documents [13].

Improvements over BERT

The performance of RoBERTA was compared with BERT on benchmarks *SQuAD* (Question Answering), *MNLI-m* (Natural Language Inference, see in Section 4.2) and *SST-2* (Sentiment analysis). The RoBERTa outperformed BERT in all of three benchmarks, even when the same data size was used for training (Table 3.2).

Model	**data**	**bsz**	**steps**	**SQuAD (v1.1/2.0)**	**MNLI-m**	**SST-2**
RoBERTa						
with *BOOKS* + *Wiki*	16GB	8K	100K	93.6/87.3	89.0	95.3
+ additional data	160GB	8K	100K	94.0/87.7	89.3	95.6
+ pretrain longer	160GB	8K	300K	94.4/88.7	90.0	96.1
+ pre-train even longer	160GB	8K	500K	94.6/89.4	90.2	96.4
BERT						
with *Books* + *Wiki*	16GB	256	1M	90.0/81.8	86.6	93.7

Table 3.2: Comparision of $BERT_{LARGE}$ and RoBERTa with different data sizes, batch sizes (bsz) and pretraining steps on benchmarks *SQuAD* (Question Answering), *MNLI-m* (Natural Language Inference) and *SST-2* (Sentiment analysis). The RoBERTa, with 500K steps and 160GB of data, outperformed BERT. Source [13].

3.6 MPNet

MPNet leverages the dependency among predicted tokens through permuted language modelling (vs MLM in BERT) and takes auxiliary position information as input to make the

model see a full sentence and thus reducing the position discrepancy [23]. Previous architectures like BERT neglect dependency among predicted tokens which is solved by architecture XLNet. It introduces permuted language modelling (PLM) for pretraining to address this problem. However, XLNet suffers from position discrepancy between pretraining and finetuning because it does use the full position information of a sentence [23] MPNet solves this.

3.7 Summary

In the following chapters, we use models based on previously described methods Bert, Roberta, MPNet and other transformers. The differences in the model architecture are not crucial and they might be perceived as „black-boxes“. The pretrained models based on discussed architectures are publicly available.

Chapter 4

Semantic comparison of sentences

This work aims to find a mechanism that will evaluate the correctness of an answer to a given question by comparing it with a set of predefined answers, which is basically a task of finding the most similar sentence among other sentences. In the NLP community, there is a name for this task - **Semantic Textual Similarity**.

However our experiments showed, that STS might not be enough. Some models have problems with detection of opposite meanings so we decided to enhance the numerical score with a **Natural Language Inference** (NLI) label.

4.1 Semantic Textual Similarity

Sentence similarity is a measure of how close in meaning two sentences are [9].

Semantic relatedness is less specific than sentence similarity. It includes any relation between two terms, including antonyms (good, bad) or meronymy (engine, car). It is not specific about the nature of the relationship. For example, „coffee“ and „mug“ might be related, but they are not semantically similar. In contrast, the words „coffee“ and „tea“ are semantically similar [6].

Understanding that similarity is estimated between mental representations, not real objects, is crucial. This means that the semantic similarity of natural language depends on our background as humans. A different score of similarity for words *smartphone* and *computer* would be given by a young adult, an expert in IT or an older person [9].

STS Benchmark

The STSB dataset [5] is a collection of sentence pairs with labels that indicate the amount of semantic similarity between the two sentences. Pairs of sentences that are judged by human judges and are assigned a real number ranging from 0 (no meaning overlap) to 5 (meaning equivalence) [5]. The selection of datasets include text from image captions, news headlines and user forums. The example of sentences with their scores:

(0) The two sentences are completely dissimilar.
The black dog is running through the snow.
A race car driver is driving his car through the mud.

(1) The two sentences are not equivalent but are on the same topic.
The woman is playing the violin.
The young lady enjoys listening to the guitar.

(2) The two sentences are not equivalent but are on the same topic.
They flew out of the nest in groups.
They flew into the nest together.

(3) The two sentences are roughly equivalent, but some important information differs/miss.
John said he is considered a witness but not a suspect.
„He is not a suspect anymore.“ John said.

(4) The two sentences are mostly equivalent, but some unimportant details differ.
Two boys on a couch are playing video games.
Two boys are playing a video game.

(5) The two sentences are completely equivalent, as they mean the same thing.
The bird is bathing in the sink.
Birdie is washing itself in the water basin.

Notice that two sentences that have opposite meanings, such as *Mother is cooking a dinner* and *Mother is not cooking a dinner* would be scored higher (2) than two random sentences with no common meaning (0), such as *Train is going fast* and *Cat is smaller than dog.*

The benchmark comprises 8628 sentence pairs divided into three splits:

- **train**: 5749 samples,
- **dev**: 1500 samples,
- **test**: 1379 samples.

Metrics

The score of similarity is a continuous value. For evaluation how well model works, the most popular metric is **Spearman correlation coefficient**. While there is also Pearson correlation coefficient, it was found to be badly suited for STS task according to [19].

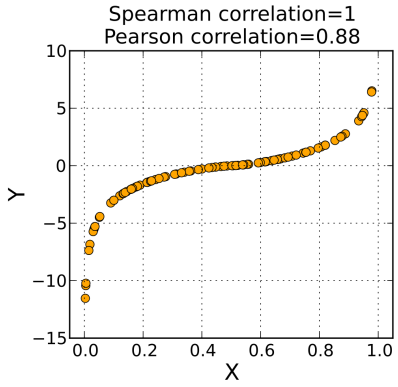

Figure 4.1: Even when relationship of two variables is not linear, but are monotonically related, Spearman correlation results to 1. Source [25].

While Pearson's correlation assesses linear relationships, Spearman's correlation assesses monotonic relationships (whether linear or not). If there are no repeated data values, a perfect Spearman correlation of +1 or -1 occurs when each of the variables is a perfect monotone function of the other [25], as shown in Figure 4.1.

There is also possibility to use Mean Square Error, however this is not commonly used in research papers about STS.

Pretrained models performance

We compared the STS performance of chosen models published on Hugging Face repository. The results were examined on STSB (described above) from Hugging Face repository using the bi-encoder and cross-encoder approach. More about their differences can be found in Section 3.4.

Bi-encoders were tested by adding a pooling layer on top of the model to compute individual embeddings for each sentence and then comparing pairs of embeddings by cosine similarity. Cross-encoders have a classification head on top of a model with only one label, where the final score of similarity of two sentences passed into the model is the probability of the label. It is possible to get embeddings from cross-encoder architecture (pooling layer for the internal state of model) and then compare it by cosine similarity. However, when tested, it was shown that their score in the STSB benchmark was inferior.

We included one model pre-trained on NLI task *cross-encoder/nli-deberta-v3-small* and one model pre-trained on question answering dataset msMarco[1]. They have a different architecture, but it is still possible to get the embedding by pooling hidden layers, but the quality of embeddings is relatively poor.

The results show that the best-performing models are based on a cross-encoder architecture and fine-tuned to the STS task. The speed of models differs significantly, where tinyBERT is 17x faster than RoBERTa large. If speed is essential, a compromise has to be made between accuracy and speed.

[1] https://huggingface.co/datasets/ms_marco

	Spearman ↑			
Bi-encoder model	**STSB**	**Time**	**Size**	**Architecture**
s-t/bert-base-nli-stsb-mean-tokens	**0.851**	3.053s	440MB	BERT
s-t/all-roberta-large-v1	0.835	9.797s	1.4 GB	Roberta
s-t/all-mpnet-base-v2	0.834	2.892s	440 MB	MPNet
s-t/all-MiniLM-L6-v2	0.820	**0.502s**	**90 MB**	MiniLM
bert-base-uncased	0.473	2.962s	440 MB	BERT
c-e/nli-deberta-v3-small	0.396	2.250s	560 MB	Deberta
c-e/stsb-roberta-base	0.298	5.89s	500 MB	Roberta
c-e/mmarco-mMiniLMv2-L12-H384-v1	0.223	1.065s	470 MB	MiniLM
Cross-encoder model				
c-e/stsb-roberta-large	**0.915**	11.321s	1.42 GB	Roberta
c-e/stsb-roberta-base	0.902	3.486s	500 MB	Roberta
c-e/stsb-distilroberta-base	0.879	1.808s	330 MB	Distil Roberta
c-e/stsb-TinyBERT-L-4	0.855	**0.663s**	**57 MB**	Tiny-BERT
c-e/quora-roberta-large	0.824	10.850s	1.4 GB	Roberta

Table 4.1: Comparison of models on STSB benchmark test split. The score is Spearman's rank correlation coefficient. Time is the number of seconds for evaluation whole split of the dataset (1,400 sentence pairs). Abbreviation *s-t* means *sentence-transformers* and *c-e* means *cross-encoder*. All models are available on Hugging Face repository.

4.2 Natural Language Inference

Textual entailment refers to the relationship between two pieces of text, where one piece of text can be inferred or implied from the other. In other words, if one fragment of the text, called a hypothesis, logically follows after another fragment of the text, called premise, then a textual entailment relationship exists between the two [11].

Labels that are used are [4]:

- **entailment**, for sentence pairs for which the second sentence entails the first sentence:
  At the other end of Pennsylvania Avenue, people began to line up for a White House tour.
  People formed a line at the end of Pennsylvania Avenue.

- **contradiction** for sentence pairs for which the second sentence contradicts the first, such as:
  Fun for adults and children.
  Fun for only children.

- **neutral** for sentence pairs where it is not possible to derive entailment nor contradiction, such as:
  And it is nice talking to you all righty
  I talk to you every day.

SNLI dataset

Stanford Natural Language Inference [4] is a dataset of 570k sentence pairs (premise, hypothesis) with one of the three label: entailment, contradiction or neutral. The SNLI dataset is based on image captioning. For the premises, the authors used captions from the

Flickr30k corpus, a collection of approximately 160k captions (corresponding to about 30k images).

Workers on Mechanical Turk[2] were given a caption for the photo, and their task was to write three captions [4]:

- Caption that is **definitely a true** description of the photo. Example: For the caption „Two dogs are running through a field.“ they could write „There are animals outdoors.“
- Caption that **might be true**. Example: For the caption „Two dogs are running through a field.“ they could write „Some puppies are running to catch a stick.“
- Caption that is **definitely a false**. Example: For the caption „Two dogs are running through a field.“ you could write „The pets are sitting on a couch.“

The dataset is divided into three splits:

- **train**: 550 152 samples
- **development**: 10 000 samples
- **test**: 10 000 samples

MNLI dataset

Multi-Genre Natural Language Inference [26] offers ten distinct genres (Face-to-face, Telephone, 9/11, Travel, Letters, Oxford University Press, Slate, Verbatim, Goverment and Fiction) of written and spoken English data. MNLI is modeled on the SNLI corpus, but differs in that covers a range of genres of spoken and written text, and supports a distinctive cross-genre generalization evaluation.

Workers read a line from non-fiction article (premise) and had to write three sentences that relate to it (hypothesis). Using only the premise, they had to:

- Write one sentence that is definitely correct about the situation or event in the line.
- Write one sentence that might be correct about the situation or event in the line.
- Write one sentence that is definitely incorrect about the situation or event in the line.

The dataset contains three splits:

- **train**: 392 702 samples,
- **development**: 20 000 samples,
- **test matched**: 10 000 samples, same genres than training and development sets,
- **test mismatched**: 10 000 samples, different genres than the training and development sets.

[2] A crowdsourcing platform outsourcing tasks to human workers

Pretrained models performance

Results in Table 4.2 show that the SNLI dataset (Flickr captions) is easier for models since a higher score is achieved. The difference among models is not so significant as in the case of MNLI (more-genres).

	Accuracy ↑				
Model	**SNLI**	**MNLI**	**Time**	**Size**	**Architecture**
c-e/nli-deberta-v3-base	**0.9238**	**0.9004**	68.19s	740MB	Deberta
c-e/nli-deberta-v3-xsmall	0.9164	0.8777	**23.2s**	**280MB**	Deberta
c-e/nli-roberta-base	0.9155	0.875	53.98s	500MB	Roberta
c-e/nli-MiniLM2-L6-H768	0.9137	0.8689	27.29s	330MB	MiniLM
c-e/nli-distilroberta-base	0.8996	0.8392	27.1s	330MB	DistilRoberta

Table 4.2: Comparison of models on STSB and MNLI benchmark. SNLI test split (10 000 sentence pairs) and MNLI mismatched (10 000 pairs) splits were used. Time is a number of seconds for evaluation whole dataset. Abbreviation *c-e* means *cross-encoder*. All models are available on Hugging Face repository.

4.3 Summary

In this chapter, we described two NLP tasks that helps to find similarity of two sentences. Semantic Textual Similarity finds a score of similarity and Natural Language Inference gives a label of textual entailment. We found the best models for each task based on publicly available datasets. We evaluated the chosen models on semantic similarity dataset STSB and textual entailmnemt SNLI and MNLI.

Chapter 5

Fine-tuning of PyTorch models

The pretrained models can be trained again on a different set of data to ensure better performance. This process of training is called fine-tuning. It is especially important when dealing with a particular problem domain (e.g. medical documents), training for a special task or after modification of the model's architecture (to train newly added pieces). The cost of fine-tuning is much smaller than full training.

5.1 Tools

Python3 was chosen as a programming language since it is the most popular language in this field and has neural network libraries such as PyTorch or TensorFlow. Hugging Face repository with many datasets and models was used because the repository contains almost all popular datasets, benchmarks and models, and manipulation with these is very simple due to Python libraries such as *datasets*, *metrics*, *transformers* and *evaluate*. As shown before, the authors of Hugging Face portal made it easy to start using pre-trained models such as BERT, Roberta, DistilBERT. The fine-tuning process is computationally expensive, and it is much faster on GPUs. Because we do not own one and training on CPU is too slow, we decided to use Google Colab[1] with GPU for computationally heavy operations. Google Colab supports Colab notebooks, similar to Jupyter Notebook[2].

PyTorch

PyTorch[3] is a machine learning framework currently developed as open-source software. It has Python and C++ interface, although Python is preferred. The framework provides many deep neural models and tools for building them.

It can accelerate computations via graphic processing units (GPUs) by using *Tensor*. Tensor is an n-dimensional array with predefined operations which can speed up computation by 50x or greater[4]. It can keep track of a computational graph and gradients which are essential in automatic differentiation used by optimisers. More about Automatic Differentiation can be found in Section 5.2).

[1] https://colab.research.google.com/
[2] https://jupyter.org/
[3] https://pytorch.org/docs/stable/index.html
[4] https://github.com/jcjohnson/cnn-benchmarks

Under the hood, PyTorch uses native code of methods for different platforms and for different target devices to ensure the best performance. This code is mainly written in C++ / Cuda.

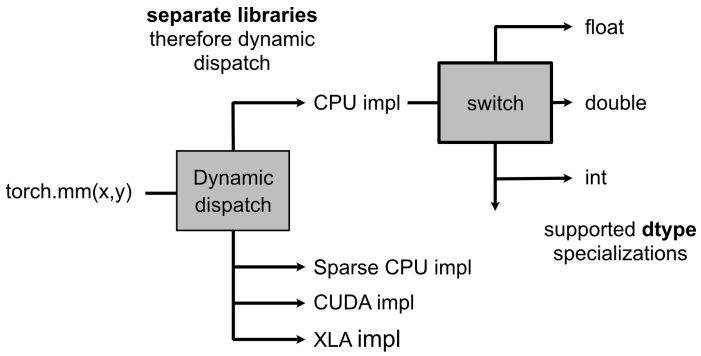

Figure 5.1: Implementation of Tensor operation *mm* is chosen dynamically according to the environment, device and data type.

Hugging Face datasets

The dataset in machine learning is a collection of data utilised for training, testing or validation of models. Public datasets can be found on Hugging Face repository or websites of individual projects. Hugging Face library *datasets* helps with downloading, sorting, slicing and manipulating of datasets. The following demonstration code downloads and loads 40% of a training slice from Natural Language Inference dataset *snli*[5].

```
from datasets import load_dataset
dataset = load_dataset("snli", split="train[40%]")
```

Hugging Face models

The database of mainly transformer models, including Bert, Roberta, DistilBert, T5 and Electra, can be found in repositories of Hugging Face. Their library *transformers* enables to use models and tokenisers for the models in just a few lines. The transformer library from Hugging Face utilises PyTorch on the background, allowing all models to be used in the PyTorch environment:

```
from transformers import AutoTokenizer, AutoModel
tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
model = RobertaModel.from_pretrained('roberta-base')
text = "Replace me by any text you'd like."
encoded_input = tokenizer(text, return_tensors='pt')
output = model(**encoded_input)
```

The model takes a tokenised sentence as input and returns an internal state. This state needs to be processed to get meaningful information. In the case of Sentence Similarity, we might compare the last hidden states of the model for different sentences to get information on their similarity. More sophisticated methods are described in Section 3.3.

[5] https://huggingface.co/datasets/snli

The models are usually pretrained, which mitigates the cost of training them from scratch, which is very expensive. To train the original $BERT_{BASE}$ model, 16 TPU chips were used, and the training took three days [7].

5.2 Training in PyTorch

This process can be summed up in the following steps:

1. Define hyperparameters (batch size, epoch size)
2. Construct the model (input, output size)
3. Construct **loss** and **optimizer**
4. Training loop
   (a) forward pass: compute prediction and loss
   (b) backward pass: gradients
   (c) update weights

Modification of model architecture

The architecture of the pretrained original model can be changed by adding an additional layer on top of the last hidden state of the original model. This is often the case for classification tasks, where one extra linear classification head is built on top. Later this layer has to be trained to be used as a classifier.

There are two ways how to modify the architecture: extend the PyTorch module, or use special classes from *transformers* library such as *AutoModelWithLMHead*, *AutoModelForSequenceClassification*, *AutoModelForQuestionAnswering*.

Extending PyTorch models is possible because all models from Hugging Face are PyTorch modules. The following code adds a dropout and linear layer to the original output. Layers with parameters are initialised in the *init* method. For forward computation in the model, the *forward* function is called.

```
class BertBaseClassifier(nn.Module):
    def __init__(self, dropout=0.5):
        super(BertBaseClassifier, self).__init__()
        self.bert = BertModel.from_pretrained('bert-base-cased')
        self.dropout = nn.Dropout(dropout)
        self.linear = nn.Linear(768, 5)

    def forward(self, input_ids, mask):
        _, pooled_output = self.bert(input_ids=input_ids,
        attention_mask=mask,return_dict=False)
        dropout_output = self.dropout(pooled_output)
        linear_output = self.linear(dropout_output))

        return linear_output
```

Loss function

During training, the performance of a model is evaluated by a loss function that measures the deviation of prediction values from reference values (correct values). The greater loss, the worse performance. According to the learning task, there are two categories of loss functions:

- Regression losses - when output is a continuous value (e.g. float value)
- Classification losses - when output is from a set of finite categorical values (e.g. Natural Language Inference)

Regression loss

One of the popular methods of **regression loss** is a method called *Mean Squared Error* (MSE). This method computes the mean of Squared Error Loss (L2) for each sample.

$$L2 = (y_{actual} - y_{predicted})^2$$

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(L2_i)$$

Here, N is the number of all samples, y_{actual} is a reference value, and $y_{predicted}$ is a value returned from our model. MSE is also called a quadratic loss because the penalty is squared, not proportional to the error. The behaviour of the quadratic function is illustrated in the following example:

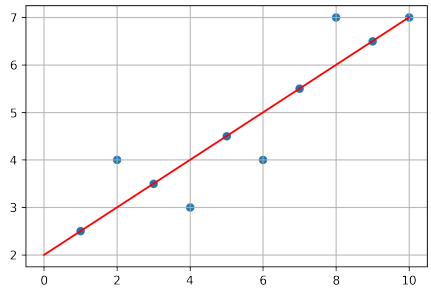

Figure 5.2: Lower loss, even though there are more points out of line, all of them have a loss of 1^2 so $L2 = 4 \cdot 1^2 = 4$ and $MSE = \frac{4}{10} = 0.4$

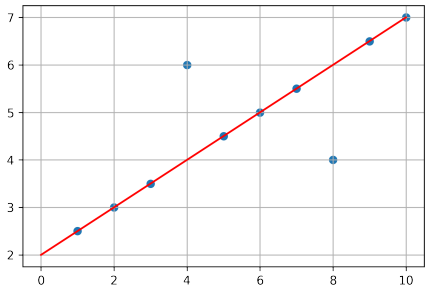

Figure 5.3: Greater loss, only two points are out of line, but both have a loss of 2^2 so $L2 = 2 \cdot 2^2 = 8$ and $MSE = \frac{8}{10} = 0.8$

Classification loss

Previously discussed regression loss can be used only for models with continuous predicted value (for example, scoring). For classification models, classification loss is used. In classification, the objective is to classify an input vector x into one of K distinct classes, denoted as C_k, where k ranges from 1 to K [2]. Typically, the classes are mutually exclusive, meaning each input is assigned to a single class, not multiple classes. The classification loss uses probability distributions of individual labels.

Binary classification is a task with only two labels/classes to predict. Target values are either labeled 0 or 1. Bernoulli distribution for the positive label can be used for classification tasks with only two labels. Therefore the only number that the model outputs is the probability of label 1, and the probability of label 0 is calculated as $p_0 = 1 - p_1$. The most common method for binary classification is *Binary Cross-Entropy Loss*. This method minimises the difference between the expected and predicted probability distributions for predicting label 1.

For classification tasks with multiple labels (more than 2), *Multi-Class Cross-Entropy Loss* is usually used, as described in [1]. The principle is similar to *Binary Cross-Entroy Loss*, but since there are more labels, each label has its probability distribution. The cross-entropy function calculates the difference between expected label probability p and predicted vector probability q for label x.

$$H(p, q) = -\sum_{x} p(x) log(q(x))$$

For example, let's have three labels from the NLI task *entailnment*, *contradiction* and *neutral*. The pair of sentences is an apparent contradiction, so expected label probabilities are $[0, 1, 0]$. The model outputs probabilities $[0.2, 0.3, 0.5]$ and wrongly predicts that pair of sentences has label *neutral*. Than, we can calculate cross-entropy loss:

$$H([0, 1, 0], [0.2, 0.3, 0.5] = -(0 \cdot log_2(0.2) + (1 \cdot log_2(0.7)) + (0 \cdot log_2(0.5))) = 0.5146$$

Optimizer and back propagation

Optimizer is crucial for learning in neural networks. The algorithm improves the model's performance by correcting its parameters (weights, biases). It works by minimising the loss function described in the previous section. The most common optimisation algorithm is called *Stochastic Gradient Descend*, and this algorithm has many improvements, such as momentum-based methods. SGD computes the parameters' gradient using a single or a few training examples (mini-batch). A gradient is a slope of change of the loss function with respect to model parameters. It tells us „how to move“ with parameters to minimise the loss function [1]. The learning rate is a parameter of SGD that determines how much can parameters update during each iteration:

$$\theta^{t+1} = \theta^t - \epsilon_k \cdot \nabla_\theta L$$

Where θ^{t+1} is a new model parameters at $t+1$, θ^t is old model parameters, ϵ_k is a **learning rate** at iteration k and $\nabla_\theta L$ is a gradient of loss function [1].

Momentum-based methods such as *Adam (Adaptive Moment Estimation)* [10] do not maintain the same learning rate but update the learning rate for each network weight individually. Adam is recommended as a default optimisation algorithm. We used slightly improved version AdamW [14].

Backpropagation works backwards (from output to input) to update the network weights based on the direction of gradients. Gradient is in practise calculated by *Automatic Differentiation* module of neural network library (PyTorch[6] / TensorFlow[7]). As described in TensorFlow documentation, it works by remembering operations during the forward pass

[6] https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html
[7] https://www.tensorflow.org/guide/autodiff

(when the model tries to predict the value during learning). Then it traverses the list of operations in reverse order to compute the gradient.

Hyperparameters

Hyperparameters are user-set parameters of the model, such as:

Learning rate

A higher learning rate can lead to faster convergence but may also result in instability and overshooting, while a lower learning rate may result in slower convergence.

Number of epochs

Epoch number is a constant that determines how many times the learning algorithm goes through the training samples. The larger number of epochs results in better accuracy. However, too large number may cause over-fitting (the model corresponds too closely to a particular set of data), and it takes more time to train the model [2].

Batch size

Batch size is the number of samples processed by the training algorithm before the back-propagation of new parameter values. Larger batch sizes can lead to faster convergence but may require more memory and may be less stable.

Regularization parameters

Regularisation tries to solve the over-fitting of a model by adding constraints on parameters which prevent weights or coefficients of the network from growing too big. Typical methods are *L1* (Lasso Regression) and *L2* (Ridge Regression) regularisation methods.

Both of them add special regularisation terms to the cost function, decreasing weight values. In the case of *L1*, the penalty is proportional to the absolute value of the weights of the model, making some of the weights to become exactly zero, resulting in a smaller model. Empirically is proven that *L2* increases performance. It encourages the network to use all of its input rather than only a part of it by diffusing the weight vectors more evenly. This is achieved by penalising the cost function with the squared magnitude of all weights in the neural network [1].

5.3 Fine-tuning of model for better contradiction recognition

Our goal was to find whether the bi-encoder sentence similarity model *all-MiniLM-L6-v2* can improve its ability to **recognise contradictions** and project it to its score because it has problems to detect contradictions and negations. The model was fine-tuned for the NLI task. Performance of sentence similarity was measured on the STSB dataset before and after fine-tuning by mean pooling of inner layers responsible for embeddings.

Hyperparameter optimization

To control the training loop and inject randomly selected parameters from configuration, a platform W&B (Weights & Biases)[8] was used with an initial set of parameters:

- **batch size** - 16, 32, 64 or 128,
- **optimizer** - AdamW or SGD with momentum,
- **learning rate** - from 10^{-5} to 10^{-4} with uniform distribution.

The optimisation process ran 54 times in three different runs (20 + 20 + 14 samples). After each run, parameters were bounded to find the best combination and leave values with bad performance aside. Each run consisted of 3 epochs and was trained on 5% of training split.

The optimisation results in Figure 5.4 showed that the most important parameter that affects performance was chosen optimiser. *AdamW* was much better than *SGD*. It is possible that *SGD* would perform better if more epochs or different parameters were set, but this was not investigated in detail.

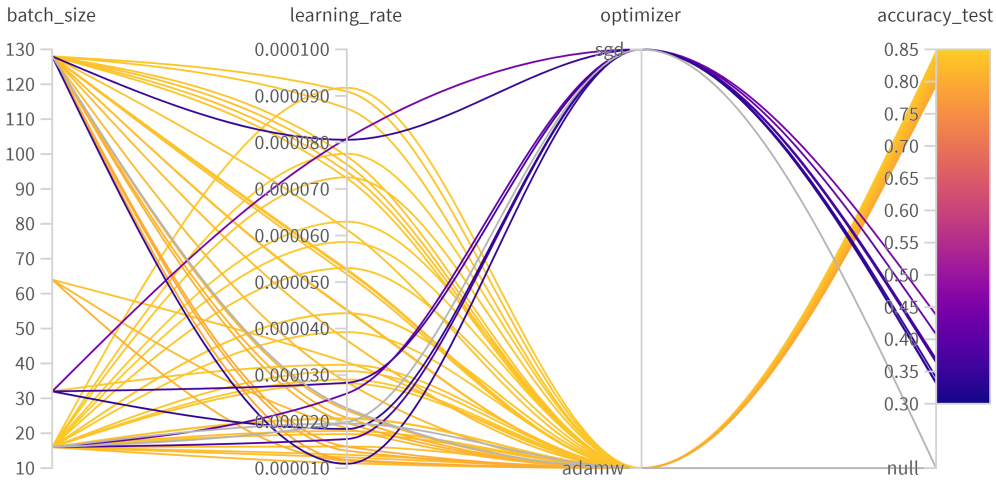

Figure 5.4: Results of hyperparameter optimisation in W&B platform for fine-tuning of NLI task. Each line is one run. Model *all-MiniLM-L6-v2*

After filtering runs with AdamW optimiser (46 runs), it was found that batch size 16 was better than 128 (all 10 best runs had a batch size of 16). This relationship is proved by importance-correlation Table 5.1 generated by W&B platform, where batch size negatively correlates with the metric. Learning rate had high importance and correlation in the table. For 10 best runs, learning rates are from $2.9 \cdot 10^{-5}$ up to $9.1 \cdot 10^{-5}$. The results for different learning rates are negligible (0.8457 vs 0.8408) and might be due to random seed setting.

[8] https://wandb.ai/site

parameter	importance	correlation
learning rate	0.667	0.636
batch size	0.333	-0.482

Table 5.1: Importance-correlation relationship of hyperparameters generated by W&B platform.

The resulting **optimal hyperparameters** were chosen to be:

- **batch size** - 16
- **optimizer** - AdamW
- **learning rate** - $5 \cdot 10^{-5}$

Adding a classification head

The model can be enhanced by an additional classification head on top of any transformer model. It is a linear layer on top of the model's pooled output and classifies a pair of sentences into one of three labels: entailment, contradiction, and neutral.

After an additional classification head is connected to the model's output, it is necessary to train this linear layer to learn how to classify individual pairs of sentences.

The classification head was added by *AutoModelForSequenceClassification* class from HuggingFace library *transformers*. This PyTorch class loads the original model and adds a sequence classification head on top of the pooled output of the original model. The number of labels the classification head outputs is set as a parameter to function. The following code shows how to set a classification head with the original BERT model for NLI tasks. Nevertheless, without fine-tuning the newly added classification head, the model would return completely non-sense predictions.

```
# Add classification head with three labels
model = AutoModelForSequenceClassification
       .from_pretrained("bert-base-cased", num_labels=3)
outputs = model(**encoded_input)
```

The fine-tuning process consisted of these steps:

1. **prepare** training, validation and test datasets (remove samples with no label, concatenate premise and hypothesis as one column)
2. **tokenise concatenated input** by Hugging Face tokeniser (truncate sentences longer than fixed length)
3. split training data into **batches** by batch size (a hyperparameter that defines the number of samples to work through before updating the internal model parameters)
4. define number of **epochs**
5. for each batch, go through all samples and then update the model by **computing loss** and **back propagate** the error into the model
6. repeat the previous step for each epoch

For fine-tuning the model, *snli* and *mnli* datasets were used. Both contains *premise*, *hypothesis* and *label*. The premise and hypothesis are concatenated into one string literal of two sentences (capital letters, dot between). Then **tokeniser** is used according to the default settings of individual models (AutoTokenizer class). However, only the first 128 tokens are used to speed up the training process. Only 0.298% of train split, 0.306% of test split and 0.366% of validation split inputs are truncated by this 128-token limit.

There are two ways how to update weights and parameters in the model with an extra classification head: allow updating embedding layers (called **non-frozen**) or do not (called **frozen**). It was found that the performance of the non-frozen model is much better than frozen, meaning that the ability of the model to update all of its parameters is better than just its last classification layer. On the other side, STS performance degraded significantly. There is possibility that different learning rate for inner model and classification head might help, but experiments were not performed.

The dataset was tested on validation and test splits of the dataset to find the variability of results. The variability was very low (under 1%), meaning the model has a balanced performance.

Model	**MNLI matched**	**Mismatched**	**STS before**	**After**
our/all-MiniLM-L6-v2	0.8094	0.8103	0.820	0.319
our/all-MiniLM-L6-v2 frozen	0.3847	0.3883	0.820	0.820

Table 5.2: Result of models after fine-tuning on SNLI data for NLI sentence classification. Non-frozen models kept the same score for Sentence Similarity (STS). However, it did not perform well for Natural Language Inference (NLI). The fine-tuning time for larger models extended up to 10 hours, so we experimented only with one small model (90MB).

Models are available publicly on Hugging Face repository *ligockym/all-MiniLM-L6-v2_snli_mnli*[9], *ligockym/all-MiniLM-L6-v2_snli_mnli_frozen*[10]. Non-frozen version is used in the thesis under the name *our/all-MiniLM-L6-v2_NLI*.

Results

After fine-tuning for NLI, performance was very good with the accuracy of 0.8103 on mismatch MNLI split. Larger model *cross-encoders/nli-distilroberta-base* (90MB vs 330MB) from authors of SBERT achieves an accuracy of 0.8392 (Table 4.1) which suggests that our smaller model was appropriately trained, but is small in size.

However, after fine-tuning for NLI, **performance for STS task degraded**. Spearman's correlation dropped from 0.9403 to 0.3190 in the case of non-freezed model. Freezed model (meaning that embedding sentence layers did not change during NLI fine-tuning, only classification head parameters) performed poorly for the MNLI task but kept its STS performance (because embedding layers did not change).

The degradation of non-frozen models for the STS task after NLI fine-tuning suggests that the model cannot perform well for both tasks simultaneously. It is understandable since both tasks are different and different features are important.

Better performance might be achieved if we train the model to STS and NLI tasks at the same time using multiple train objectives[11]. This approach was tried, but the performance

[9] https://huggingface.co/ligockym/all-MiniLM-L6-v2_snli_mnli

[10] https://huggingface.co/ligockym/all-MiniLM-L6-v2_snli_mnli_freezed

[11] GitHub SBERT, training multiple objectives

of both STS and NLI was not especially good. We decided not to investigate this in the detail because there were clues that the STSB dataset is not enough for training sentence similarity, and it would be out of the scope of this thesis.

Chapter 6

Evaluation on real data

During work on the thesis, the first students passed tests in the language learning application. The creators of this app provided us **private logs** with student answers and teacher-annotated scores for each answer. This enabled us to investigate individual models' performance in the language-learning application domain. However, data did not enable us to find the model performance specific to answers with the opposite meaning, wrong word ordering and different meaning. We decided to create our **own dataset** of 113 answers for three different topics: *negation*, *grammar* and *meaning*.

6.1 Human annotated dataset

Real student answers were collected from the app for learning of languages GoLearn and annotated by a English teacher. The structure of data is:

- **question prompt** e.g. „tell your neighbour to be quieter“
- set of **correct answers** (usually 3-6)
- set of real **students' answers**, each with
  - **reference score** that was given by currently implemented scoring system in the app (calibrated by equation 6.1),
  - **human score** that was annotated by the English teacher,

Dataset consists of 106 questions and 1000 students' answers. For example, a question prompt „*Tell them that you have just moved into no 26*“ has 7 correct answers. When a student answers to this question, their answer is compared to these 7 answers and the score of the most similar pair is outputted: „I am your new number 26 neighbour.“, „I just moved into number 26“, „I am your new neighbour from number 26“, „I have just moved into number 26“, „I've just moved in to number 26“, „I recently moved to no 26“, „I live next door at no. 26“.

Reference model

A model currently used in the app uses a sentence similarity model *bert-base-nli-stsb-mean-tokens*[1] which authors deprecate. The reasons of the deprecation are unknown but might be

[1] Hugging Face: Bert STSB NLI

due to worse results from other downstream tasks. However, in the STS task, it is superior among bi-encoders as shown in Table 4.1. NLI in the name suggests that it was also trained on the NLI task, so it detects contradictions easier than other bi-encoders.

The **reference score** is normalized (calibrated) by a linear equation 6.1. The parameters were chosen empirically by authors of the app. When sentence similarity score is 0.7, score that is given to student is 0.4 points. All sentence similarity score under 0.5 is 0.

$$score = 2 \cdot SimilarityScore - 1 \tag{6.1}$$

Chosen STS models compared with reference solution

A model returned a score of similarity (cross-encoders) or sentence embeddings compared by cosine similarity (bi-encoders). The calculated score of similarity between the student's answer and the most similar correct answer is taken as a score of the student's answer.

The performance of chosen sentence similarity models was compared in Table 6.1. Spearman and Pearson's coefficients were calculated to find a correlation. Mean Squared Errors were calculated before and after calibration.

Resulting scores were **calibrated** by Linear Regression using Least Squares Method (Equation 6.2), such that coefficients a and b are found. When the similarity model returns a score of 1.0 (totally correct answer), calibrated value has to be 1.0, too. Otherwise, a student would get an answer less than 100% points for a totally correct answer. Fulfilment of this condition is ensured by Equation 6.3.

$$y = ax + b \tag{6.2}$$

$$1 = a + b \tag{6.3}$$

The results of chosen method performance on the human-annotated dataset are in Table 6.1. Models performed much better after the calibration. The best model is *stsb-TinyBERT-L-4* with the improvement of 27.8% (Figure 6.4) over the previously used model (reference model) with old calibration (Figure 6.1). The reference model *sentence-transformers/bert-base-nli-stsb-mean-tokens* has very good performance (Figure 6.2), and after it was calibrated by linear regression, it achieved a performance increase of 18.5%. This is the best result among bi-encoders, so the app's authors used the proper model for their use case.

Model	**Spearman ↑**	**MSE ↓**	**Time ↓**	**a**	**b**	**%**
c-e/stsb-TinyBERT-L-4	0.827	**0.0504**	**3.6s**	2.127	-1.127	**27.8**
c-e/stsb-roberta-large	**0.839**	0.0525	59.6s	2.441	-1.441	24.8
Ref. model (new calib.)	0.821	0.0569	18.6s	2.926	-1.926	18.5
c-e/stsb-distilroberta-base	0.813	0.0597	10.3s	2.528	-1.528	14.5
s-t/all-mpnet-base-v2	0.788	0.0603	18.6s	2.118	-1.118	13.6
s-t/all-MiniLM-L6-v2	0.797	0.0605	5.7s	2.394	-1.394	13.3
s-t/quora-distilbert-base	0.776	0.0652	10.4s	2.766	-1.766	6.6
Ref. model (old calib.)	0.823	0.0698	18.6s	2	-1	0.0
our/all-MiniLM-L6-v2_NLI	0.475	0.1466	8.4s	8.925	-7.925	-52.4

Table 6.1: The performance of the chosen calibrated models on the validation dataset from the logs. Improvement (%) is calculated in percentages from the MSE of the reference model with old calibration. Parameters of linear regression a (slope) and b (intercept) were found for each model individually.

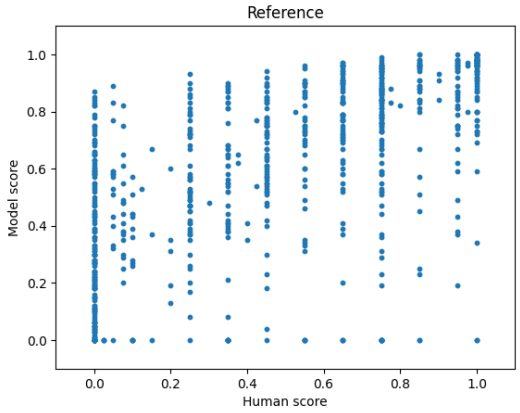

Figure 6.1: Reference model with old calibration.

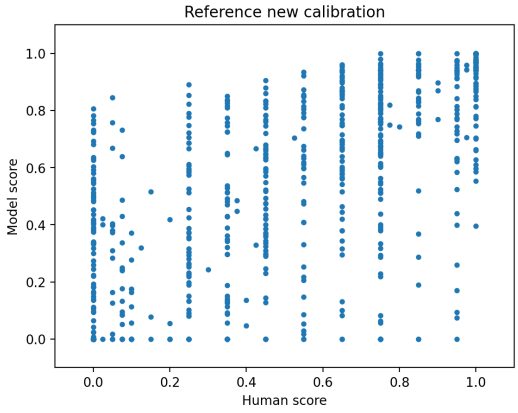

Figure 6.2: Reference model with new calibration.

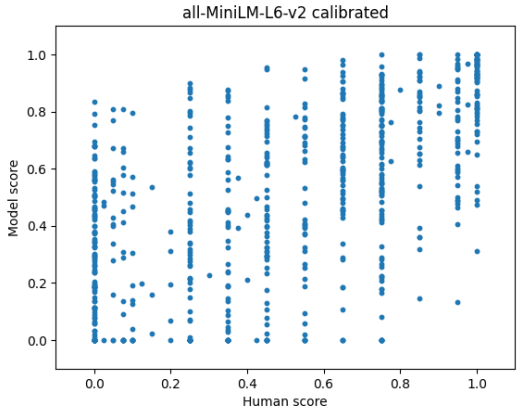

Figure 6.3: Model s-t/all-MiniLM-L6-v2

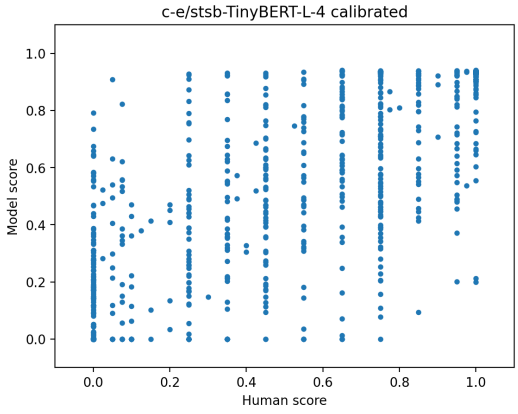

Figure 6.4: The best model c-e/stsb-TinyBERT-L-4 calibrated.

6.2 Enhanced grammar and meaning dataset

We created our dataset with one correct answer and set of evaluation answers to examine how models tackle wrong **grammar** (42 evaluation answers), **meaning** (63) and **negation** (8). Each discipline has its correct answer and set of evaluation answers. Each evaluation answer contains a human score given by us. Most answers in the negation and grammar split of the dataset have a human-annotated score of zero because of gross mistakes. Models were compared, and the results of their Mean Square Errors are shown in Table 6.2.

Results show that two bi-encoder models *all-mpnet-base-v2* and *all-MiniLM-L6-v2* have problems to detect negation. Reference model model (bi-encoder) does not have this deficiency. Almost all models have similar problems to detect wrong grammar (word ordering, third person „-s“, „is -ing“). Detection of correct meaning of sentences have similar performance among models.

Model	**Negation MSE ↓**	**Grammar MSE ↓**	**Meaning MSE ↓**
Old model (new calibration)	0.1272	0.4838	0.2451
Old model (old calibration)	0.2346	0.5764	0.231
s-t/all-mpnet-base-v2	0.4498	0.4576	0.2245
s-t/all-MiniLM-L6-v2	0.6683	0.5484	0.2934
s-t/quora-distilbert-base	0.1589	**0.3902**	0.2444
c-e/stsb-TinyBERT-L-4	**0.0388**	0.651	0.2602
c-e/stsb-roberta-large	0.1421	0.5759	**0.2101**
c-e/stsb-distilroberta-base	0.1633	0.5815	0.2495

Table 6.2: Comparison of STS models on sentences with negation, wrong grammar and meaning. The most distinct difference is among negation detection, where models *all-mpnet-base-v2* and *all-MiniLM-L6-v2* perform poorly.

We discovered that **NLI labels** might help further improve the score. The model to label sentence pairs is *cross-encoders/nli-deberta-v3-base*. The model was chosen because it is the best performing model on SNLI and MNLI datasets for textual entailment in the Table 4.2.

For **negation evaluation**, NLI labelling would be very useful since 7 out of 8 negation evaluation answers were marked as a contradiction by the NLI model, especially for the model *s-t/all-MiniLM-L6-v2* as shown in the Figure 6.5. The model *c-e/stsb-TinyBERT-L-4* has a very good ability to detect contradictions (Figure 6.6). Therefore NLI labelling would be less helpful.

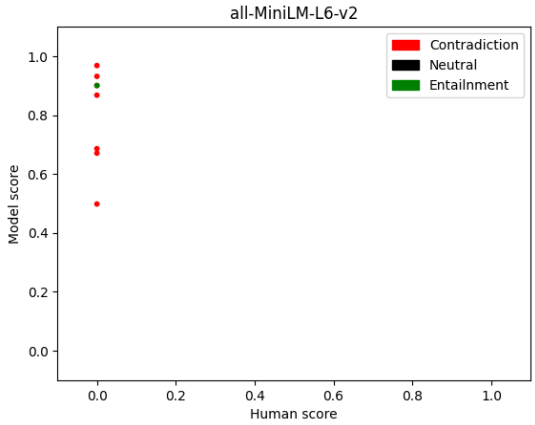

Figure 6.5: Negation detection. Model *s-t/all-Mini-LM-L6-v2* calibrated. Very high potential of NLI labelling.

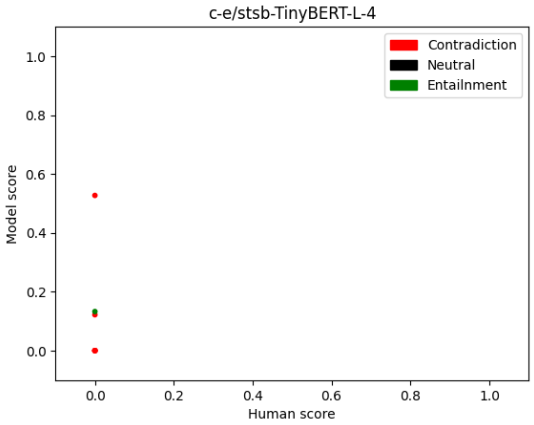

Figure 6.6: Negation detection. Model *c-e/stsb-TinyBERT-L-4* calibrated. Low potential of NLI labelling. 5 points in 0.0.

In the case of **meaning detection**, benefits of NLI labelling would be most visible for model *s-t/all-MiniLM-L6-v2* (Figure 6.7), since most of the sentences marked as contradictions have a human score of 0, but model scores them higher. The same applies to sentences marked as entailment with a human score of 1 and model scoring them lower. The old model can better detect incorrect meaning but still has many incorrectly scored answers (Figure 6.9). Model *stsb-Tiny-BERT-L-4* has a better ability to detect incorrect meaning (Figure 6.8). However, there is a substantial part of wrongly scored answers marked as contradictions, so the potential to improve performance is still there. NLI labelling effect would be minimal on model *c-e/stsb-roberta-large* (Figure 6.10) since most of the contradictions are already marked as 0.

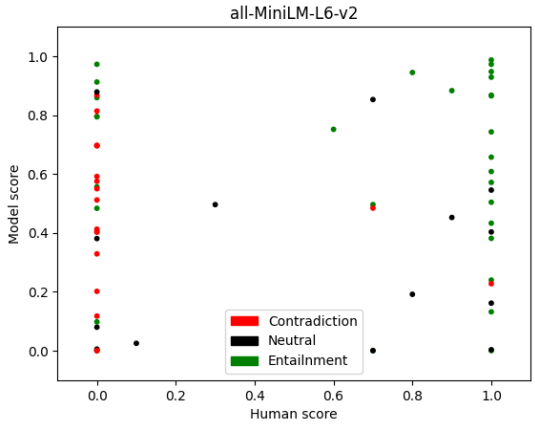

Figure 6.7: Meaning detection. Model *s-t/all-Mini-LM-L6-v2*. High potential effect of NLI labelling.

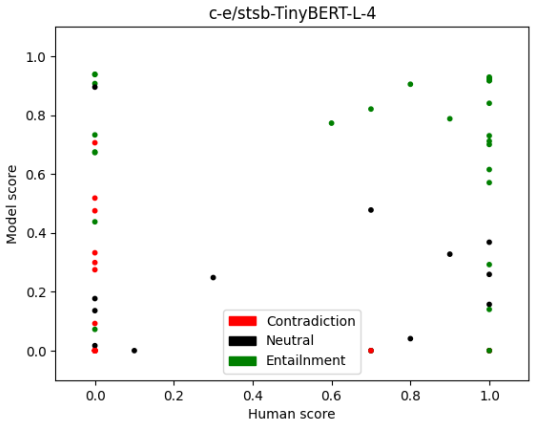

Figure 6.8: Meaning detection. Model *c-e/stsb-TinyBERT-L-4*. High potential effect of NLI labelling.

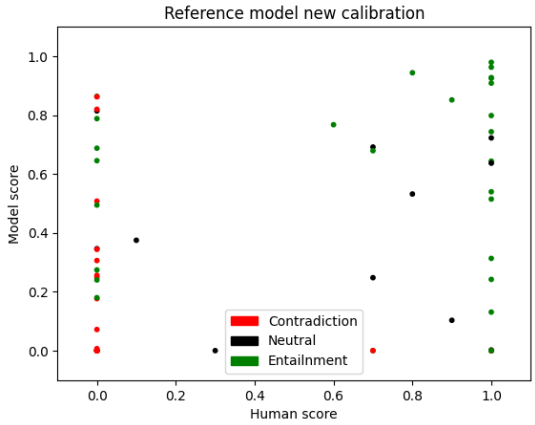

Figure 6.9: Meaning detection. Old model with a new calibration. Medium effect of NLI labelling.

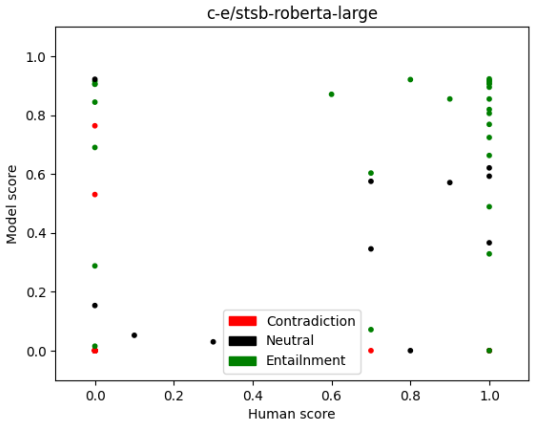

Figure 6.10: Meaning detection. Model *c-e/stsb-roberta-large*. Most of contradictions are marked as 0, low effect of NLI labelling.

The **correct grammar** is very important in the language learning domain. However, all models have problems with the detection of incorrect grammar from our dataset (Figure 6.11, Figure 6.12). The most frequent grammatical mistake is incorrect word ordering and verb form. We suggest implementing a grammatical model which will be able to find mistakes, classify them and change the score accordingly.

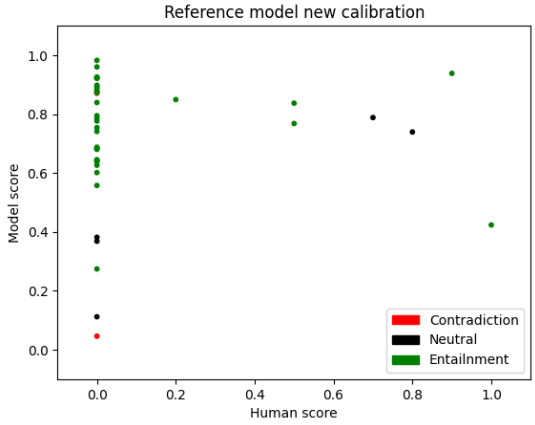

Figure 6.11: Incorrect grammar detection. The old model calibrated. Poor detection of incorrect grammar.

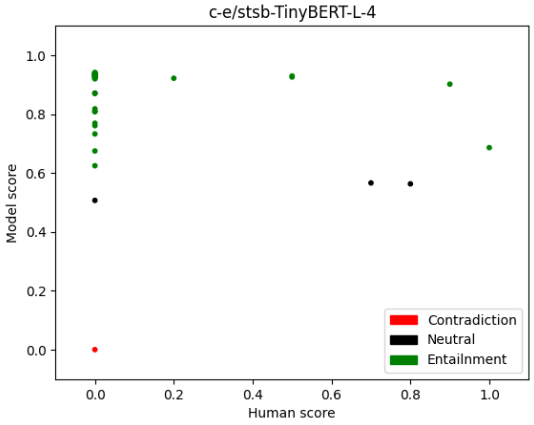

Figure 6.12: Incorrect grammar detection. Model *c-e/stsb-TinyBERT-L-4*. Poor detection of incorrect grammar.

Chapter 7

Conclusion

Implementation of the model. This thesis aimed to construct and implement a chosen method for evaluating the correctness of a student's sentence-alike answer to a question in a language learning application. The objective was successfully accomplished by choosing and implementing the model *cross-encoder/stsb-TinyBERT-L-4*, which performs the best for the domain. After the calibration to the validation data (linear regression), the model outperforms the previously used system based on model *sentence-transformers/bert-base-nli-stsb-mean-tokens* by 27.8% on Mean Square Error. Calibration of the old model increases its performance by 18.5%. We used human-annotated logs from the application (1000 student answers) to evaluate performance and calibration parameters.

Disadvantages and alternative. The disadvantage of the new cross-encoder model is that it can no longer store pre-computed sentence embeddings for correct answers. Therefore model has to run for each pair of compared sentences. This will increase computation time. However, our chosen model is 5-times faster than reference model. The alternative is to use the old model (bi-encoder) with our calibration, which performs best among examined bi-encoder.

NLI labels. We examined the usage of NLI labels (entailment, contradiction, neutrality) to improve performance of sentence similarity further. We created our dataset with sentence pairs focused on negation (8 sentences), wrong grammar (42) and meaning (63) because logs from the app did not include these phenomena.

The benefit of NLI labelling varies model-by-model because some models can detect negation and contradictions easier, but the potential is very high even for models with the best ability. There are many ways how to include the label in the final score. We are convinced that NLI with a combination of STS deserves further research.

Grammar. Sentence similarity models perform very poorly in the detection of incorrect grammar. We recommend including a grammatical model that will detect incorrect grammar, categorise the error and maybe even show the mistake to the student.

Before the work on the thesis, I did not have any experience with NLP and machine learning. I am very thankful that I was motivated to dive deep into the subject, which is very interesting and motivated me to choose machine learning as my following field of study.

Bibliography

[1] Bhardwaj, A., Di, W. and Wei, J. *Deep Learning Essentials*. 1st ed. Birmingham, England: Packt Publishing, january 2018. ISBN 9781785887772.

[2] Bishop, C. M. *Pattern Recognition and Machine Learning*. 1st ed. Springer, 2006. ISBN 978-0387-31073-2. Available at: http://research.microsoft.com/en-us/um/people/cmbishop/prml/.

[3] Bojanowski, P., Grave, E., Joulin, A. and Mikolov, T. Enriching Word Vectors with Subword Information. *CoRR*. 1st ed. 2016, abs/1607.04606. Available at: http://arxiv.org/abs/1607.04606.

[4] Bowman, S. R., Angeli, G., Potts, C. and Manning, C. D. A large annotated corpus for learning natural language inference. *CoRR*. 2015, abs/1508.05326. Available at: http://arxiv.org/abs/1508.05326.

[5] Cer, D. M., Diab, M. T., Agirre, E., Lopez-Gazpio, I. and Specia, L. SemEval-2017 Task 1: Semantic Textual Similarity - Multilingual and Cross-lingual Focused Evaluation. *CoRR*. 1st ed. 2017, abs/1708.00055. Available at: http://arxiv.org/abs/1708.00055.

[6] Chandrasekaran, D. and Mago, V. Evolution of Semantic Similarity—A Survey. *ACM Comput. Surv.* 1st ed. New York, NY, USA: Association for Computing Machinery. feb 2021, vol. 54, no. 2. DOI: 10.1145/3440755. ISSN 0360-0300. Available at: https://doi.org/10.1145/3440755.

[7] Devlin, J., Chang, M., Lee, K. and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR*. 1st ed. 2018, abs/1810.04805. Available at: http://arxiv.org/abs/1810.04805.

[8] Ganegedara, T. *Natural Language Processing with TensorFlow: The Definitive NLP Book to Implement the Most Sought-After Machine Learning Models and Tasks*. 2nd ed. Packt Publishing, Limited, 2022. Expert insight. ISBN 9781838641351. Available at: https://books.google.sk/books?id=ONYRzwEACAAJ.

[9] Harispe, S., Ranwez, S., Janaqi, S. and Montmain, J. Semantic Similarity from Natural Language and Ontology Analysis. *CoRR*. 1st ed. 2017, abs/1704.05295. Available at: http://arxiv.org/abs/1704.05295.

[10] Kingma, D. P. and Ba, J. Adam: A Method for Stochastic Optimization. In: M.Sc., D. P. K., ed. *International Conference on Learning Representations* [online]. 2015. DOI: 10.48550/arXiv.1412.6980. Available at: https://arxiv.org/abs/1412.6980.

[11] Korman, D. Z., Mack, E., Jett, J. and Renear, A. H. Defining textual entailment. *Journal of the Association for Information Science and Technology*. 1st ed. 2018, vol. 69, no. 6, p. 763–772. DOI: https://doi.org/10.1002/asi.24007. Available at: https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/asi.24007.

[12] Le, Q. V. and Mikolov, T. Distributed Representations of Sentences and Documents. *CoRR*. 1st ed. 2014, abs/1405.4053. Available at: http://arxiv.org/abs/1405.4053.

[13] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M. et al. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR*. 1st ed. 2019, abs/1907.11692. Available at: http://arxiv.org/abs/1907.11692.

[14] Loshchilov, I. and Hutter, F. Fixing Weight Decay Regularization in Adam. *CoRR*. 2017, abs/1711.05101. Available at: http://arxiv.org/abs/1711.05101.

[15] Mikolov, T., Chen, K., Corrado, G. and Dean, J. *Efficient Estimation of Word Representations in Vector Space*. arXiv, 2013. DOI: 10.48550/ARXIV.1301.3781. Available at: https://arxiv.org/abs/1301.3781.

[16] Nair, V. and Hinton, G. E. Rectified Linear Units Improve Restricted Boltzmann Machines. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. Madison, WI, USA: Omnipress, 2010, p. 807–814. ICML'10. ISBN 9781605589077.

[17] Pennington, J., Socher, R. and Manning, C. D. GloVe: Global Vectors for Word Representation. In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, p. 1532–1543. Available at: http://www.aclweb.org/anthology/D14-1162.

[18] Quiza, R. and Davim, J. P. Computational Methods and Optimization. In: Davim, J. P., ed. *Machining of Hard Materials*. London: Springer London, 2011, p. 177–208. DOI: 10.1007/978-1-84996-450-0_6. ISBN 978-1-84996-450-0. Available at: https://doi.org/10.1007/978-1-84996-450-0_6.

[19] Reimers, N., Beyer, P. and Gurevych, I. Task-Oriented Intrinsic Evaluation of Semantic Textual Similarity. In: *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*. Osaka, Japan: The COLING 2016 Organizing Committee, December 2016, p. 87–96. Available at: https://aclanthology.org/C16-1009.

[20] Reimers, N. and Gurevych, I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *CoRR*. 2019, abs/1908.10084. Available at: http://arxiv.org/abs/1908.10084.

[21] Rothman, D. *Transformers for natural language processing: Build innovative deep neural network architectures for NLP with python, pytorch, tensorflow, Bert, Roberta, and more*. Packt Publishing, 2021.

[22] Shibuya, N. *Transformer's Encoder-Decoder: Let's Understand The Model Architecture* [online]. December 2021. 2022-12-22 [cit. 2022-12-22]. Available at: https://kikaben.com/transformers-encoder-decoder.

[23] Song, K., Tan, X., Qin, T., Lu, J. and Liu, T. MPNet: Masked and Permuted Pre-training for Language Understanding. *CoRR*. 1st ed. 2020, abs/2004.09297. Available at: https://arxiv.org/abs/2004.09297.

[24] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L. et al. Attention Is All You Need. *CoRR*. 2017, abs/1706.03762. Available at: http://arxiv.org/abs/1706.03762.

[25] Wikipedia. *Spearman's rank correlation coefficient. Wikipedia, The Free Encyclopedia* [online]. 2023 [cit. 2023-05-07]. Available at: http://en.wikipedia.org/w/index.php?title=Spearman's%20rank%20correlation%20coefficient&oldid=1148440452.

[26] Williams, A., Nangia, N. and Bowman, S. R. A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference. *CoRR*. 1st ed. 2017, abs/1704.05426. Available at: http://arxiv.org/abs/1704.05426.

[27] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M. et al. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR*. 1st ed. 2016, abs/1609.08144. Available at: http://arxiv.org/abs/1609.08144.