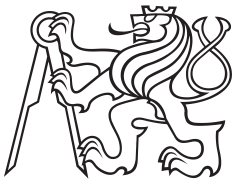
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Semantic Clustering of Twitter Data

Jan Petrov

Supervisor: Ing. Jan Drchal, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence
May 2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Petrov** Jméno: **Jan** Osobní číslo: **492165**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Umělá inteligence**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Sémantické shlukování dat z Twitteru

Název diplomové práce anglicky:

Semantic Clustering of Twitter Data

Pokyny pro vypracování:

The task of the thesis is to experiment with methods of semi-supervised semantic clustering with a focus on Czech Twitter data. The work should aim at applications in semi-supervised topic detection.
1) Research state-of-the-art NLP methods as well as methods of semi-supervised clustering.
2) Create an appropriate Czech dataset. This will most likely involve the development of an annotation tool or the use of machine translation methods.
3) Train and evaluate multiple approaches, including neural architectures based on Set Transformer.

Seznam doporučené literatury:

[1] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).
[2] Lee, Juho, et al. "Set transformer: A framework for attention-based permutation-invariant neural networks." International Conference on Machine Learning. PMLR, 2019.
[3] Lee, Juho, Yoonho Lee, and Yee Whye Teh. "Deep amortized clustering." arXiv preprint arXiv:1909.13433 (2019).
[4] Ibrahim, Rania, et al. "Tools and approaches for topic detection from Twitter streams: survey." Knowledge and Information Systems 54.3 (2018): 511-539.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jan Drchal, Ph.D. centrum umělé inteligence FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **02.02.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

Ing. Jan Drchal, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Datum převzetí zadání

Podpis studenta

Acknowledgements

Dear teachers and scientists,

Many thanks for the transformative years.

With gratitude

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Methodological guideline on the observance of ethical principles in the preparation of university theses.

Prague, 20 May 2022

Abstract

We address the task of semantic clustering, using 203K tweets published at Twitter in the Czech language. We investigated neural networks models for converting text data into vectors that encode semantic information. We developed a software tool that augments syntactic search by multiple semantic methods for finding content-related tweets. Using the tool, we annotated 18K tweets and created a dataset of 65 clusters on particular publicly relevant topics. We based evaluation of multiple automatic clustering methods on the dataset. We also created a list of key lemmas characterizing the automatically found clusters.

Keywords: semantic similarity, clustering, Transformer, BERT

Supervisor: Ing. Jan Drchal, Ph.D.
Centrum umělé inteligence FEL
Karlovo náměstí 13
Praha 2

Abstrakt

Zabýváme se úlohou sémantického shlukování na podkladě 203K tweetů zveřejněných na Twitteru v českém jazyce. Zkoumáme modely neuronových sítí pro převod textových dat do vektorů nesoucích sémantickou informaci. Vyvinuli jsme softwarový nástroj, který doplňuje syntaktické vyhledávání o různé sémantické metody nacházející obsahově související tweety. S jeho použitím jsme anotovali 18K tweetů a tím vytvořili datovou sadu 65 skupin adresujících konkrétní veřejně významná témata. Na jejím základě jsme vyhodnocovali více metod automatického shlukování. Vytvořili jsme též seznam klíčových lemat, které tématicky charakterizují nalezené shluky.

Klíčová slova: sémantická podobnost, shlukování, Transformer, BERT

Překlad názvu: Sémantické shlukování dat z Twitteru

Contents

Figures

Tables

Chapter 1

Introduction

In the past ten years, machine learning and neural network methods have achieved multiple breakthroughs in most natural language processing (NLP) tasks. Multiple models, pre-trained on huge amounts of text, became available that can convert words (or other language units such as sub-word tokens or sentences) to vectors (e.g., with 768 dimensions) encoding various aspects of *semantic* information (meaning). For most major NLP tasks, such as translation, sentiment classification or question answering, vector representation provides major advantage over syntactic methods, which process text data as a mere sequence, or sub-sequences, of characters.

We applied the semantic methods to Twitter data: tweets published by a group of selected Czech users mostly in the last six years. Twitter is the social network most used, at least in the Czech Republic, for communicating topics of public interest and general importance. In addition, the limit of 280 characters favors high information density. Accordingly, Twitter is a reasonably accessible resource of manageable data size that can provide insights in social issues and tendencies. Moreover, techniques developed on Twitter can be reasonably extended to newspaper headlines and leads.

To facilitate discovery and analysis of socially important issues, we aimed at the task of finding groups of tweets related by their topics (semantic clustering and subsequently topic discovery). Using multiple neural-network models, we converted 203,057 tweets from the provided dataset to high-quality vector representations. Thereafter, we developed a tool (web application) for semantic search and annotations that makes extensive use of the computed vector representations. The purpose of the tool is twofold:

- to provide multiple semantic search methods when finding all tweets related to a particular topic; thus, also the tweets can be found that do

not conform to the syntactic pattern, but semantically address the same topic as tweets already found by syntactic methods; and

- to facilitate manual annotation; reviewing 18,371 tweets, we defined 65 groups of tweets covering particular important topics for further use, in this thesis and in general.

We experimented with multiple techniques of automatic semantic clustering, both classic (pre-neural) and neural-network, the latter represented by the recent Amortized Clustering approach based on the Set Transformers architecture. We used the annotated groups of tweets to evaluate the automatically generated clusters and for fine-tuning the Set Transformer network (which we pre-trained at first on batches sampled at random from the provided dataset and clusters computed by the classic methods). In particular, we evaluated generated clusters according to their:

- correspondence to human-annotated clusters, examining whether current automatic methods provide clusters that would include mostly all tweets, and only those tweets, to particular newsworthy topics; and
- potential of use for topic-detection. To discover important topics, we generated a list of key lemmas shared by tweets within generated clusters.

This thesis consists of Part I and Part II and the following chapters:

- In **Chapter 2** we address the fascinating field of representing words (and other language units such as sub-word tokens or sentences) by vectors (embeddings) that encode semantic properties. We proceed from models (e.g., word2vec) trained to pre-compute distinct vectors for distinct words to the BERT family of models, which consider context, representing the same language unit differently in different contexts.
- In **Chapter 3** we explore network architectures most relevant to NLP tasks and this thesis: recurrent networks, Transformers (underlying BERT), and Set Transformers (underlying Amortized Clustering).
- In **Chapter 4** we outline two classical clustering methods and analyze Amortized Clustering in detail; we applied all three methods.
- In **Chapter 5** we describe the provided Twitter dataset and the procedure used to convert provided tweets to vectors.
- In **Chapter 6** we address the tool for semantic search and annotations we developed, both the software solution and features available to users. We also mention our work on manual annotations and its results.
- In **Chapter 7** we summarize our experiments with K-Means and Deep Amortized Clustering, evaluate results of this thesis and propose further research goals.

Part I

Theoretical Background

Chapter 2

Representing Words and Other Language Units by Vectors

Neural networks, and most other machine learning methods (such as *clustering*, addressed in chapter 3), cannot operate directly on syllables, words, sentences or paragraphs. They accept only numbers for input, or their vectors, usually of a pre-determined size. Accordingly, language units need to be converted to vectors first, before further stages of an NLP pipeline can follow. We present in this chapter various methods for converting words and other language units to vectors, generally proceeding in the order of increasing complexity.

2.1 One-Hot Encoding

Every word is assigned a unique index number, a natural number representing word position in the vocabulary of size $|V|$ (where, e.g., $|V| = 30\,000$). For instance, the word *cat* can be assigned 1 774 and the word *computer* 15 795. Then, each word is represented by a vector from $\mathbb{R}^{|V|}$, with as many elements (dimensions) as the vocabulary size. All elements of the vector are set to 0.0, except that the element at the position equal to the index of the represented word has the value of 1.0. Given the example above, the word *cat* is represented by the vector:

$$\mathbf{v}_{cat}^{\mathsf{T}} = \mathbf{e}_{1774}^{\mathsf{T}} = [\,0.0 \quad 0.0 \quad \ldots \quad 0.0 \quad 0.0 \quad 1.0 \quad 0.0 \quad 0.0 \quad \ldots \quad 0.0 \quad 0.0\,],$$

where the 1.0 value is at the position 1 774.

One-hot representation has two major shortcomings, however:

1. Vectors with tens of thousands of dimensions are needlessly large.
2. The representation cannot capture any degree of similarity between words. Each word (its one-hot vector representation) is orthogonal to every other word, whether both words are similar or dissimilar, or even synonyms or antonyms. In one-hot representation, distance between words, whether *dog* and *doggy*, *excellent* and *great*, *great* and *awful*, or *dog* and *excavator*, is always the same.

2.2 Neural Network Word Embeddings

Bengio et al published already in 2003 [6] a method for representing each word by a vector from $\mathbb{R}^d$ with much fewer dimensions (e.g., $d = 512$) than $|V|$, the size of the vocabulary. They proposed a 2-layer neural network trained as a language model, trained to predict $\mathrm{P}(w_t \,|\, w_{t-1}, w_{t-2}, \ldots, w_{t-n})$, that is, the probability that a particular word comes in the corpus just after a particular sequence of words (where n is a pre-determined number, e.g., 4). The neural network is to answer, for instance, the following question: What is the probability regarding the sequence *"I am glad to* • *"* that the word *see* (or *postpone* or *hippopotamus* or any other word in the vocabulary) comes at the place of •?

The network processes inputs into outputs in the following layers and steps (let us take a simple feed-forward variant and not complicate notation by addressing a training batch of multiple inputs and targets):

1. **First layer = Embedding Layer.** Word vectors coincide with trainable parameters of the first layer. There are $|V|$ words, and every word is represented by a vector from $\mathbb{R}^d$. Accordingly, the whole vocabulary (embedding layer) can be repreaccuracsented by $\mathbf{M} \in \mathbb{R}^{|V| \times d}$ matrix of trainable parameters.
2. **Representation of inputs using the embedding layer.** Each of n input words is encoded as a vector from $\mathbb{R}^d$. A word, having the vocabulary index i (e.g., 1 774), is represented by the vector coinciding with i-th line of the $\mathbf{M}$ matrix.
3. **Second Layer.** n vectors from the previous step, each from $\mathbb{R}^d$, are concatenated into a single vector from $\mathbb{R}^{nd}$. This concatenated vector is fed into a fully-connected layer, thus multiplying the matrix $\mathbf{F} \in \mathbb{R}^{|V| \times nd}$ of trainable parameters.

Note: We prefer a simple architecture here, although [6] does not adhere to any particular architecture, and more sophisticated neural networks may yield better results.

4. **Softmax.** The resulting vector from $\mathbb{R}^{|V|}$ is run through softmax to obtain a probability for each form |V| that it comes in the corpus just after the input sequence of words.

Training is based on a large text corpus, with the following elements:

- The **input** consists of n consecutive words (their representations) sampled from the corpus.
- The **target** is the word (its representation) that comes in the corpus just after the input.
- The **loss function** is defined as the usual *negative log softmax*: $L = -log\left(\frac{exp(f_T)}{\sum_j exp(f_j)}\right)$, where f_j denotes j-th element of the vector with logits entering softmax, T denotes the target class index, and thus f_T denotes the logit whose index equals that of the target class.
- The whole network is **trained at once**. In each training pass, both **M** matrix (see numbered item 1. above) and **F** matrix (see numbered item 3. above).

Word vectors (coincident with rows of the M matrix, see numbered item 1. above), first initiated at random values, gradually change during the training. Vectors representing words that occur in similar words sequence within the corpus (such words usually have related meaning) tend to get closer in the $\mathbb{R}^d$ space.

It is a common technique to represent a word or other language unit as a vector coincident with a particular row of the embedding layer of a neural network (see numbered item 1. above). We will return to the technique in section 2.5 below. Now, let us consider computational demands related to rendering probabilities for a large number of classes (e.g., when the vocabulary size $|V| = 30\,000$, or even $150\,000$). In particular, let us consider the number of parameters in the second layer (numbered item 3. above) and *softmax* computed over, e.g., $30\,000$, or even $150\,000$, classes (numbered item 4. above).

2.3 Word2Vec and FastText

Mikolov et al published their word2vec algorithm in two breakthrough papers from 2013 [28, 30]. They devised a considerably more efficient procedure for computing word vectors, compared to 2.2 above. This allowed *Mikolov et al* to train already in 2013 on larger corpora (e.g., full Wikipedia), calculate first high-quality word vectors, achieve state-of-the-art results at that time, and demonstrate that word vectors can encode more meaning than expected before.

Authors experimented with two opposite approaches:

1. **Continuous bag-of-words** model (CBOW) that answers the question: If you know the surrounding words (e.g., you know what 20 words are at one of 10 positions just preceding or 10 positions just following the central surrounded word) what is the probability for a particular word from the vocabulary to be the central surrounded word?
2. **Continuous skip-gram** model that answers the question: If you know the central surrounded word, what is the probability for a particular word from the vocabulary that it is one of the surrounding words (e.g., occurs as one of 10 words just preceding or 10 words just following the central surrounded word)?

We address the continuous skip-gram model below, which achieved better semantic results.

Every word from the vocabulary is, again, represented by a word vector of d dimensions (where d, e.g., 512, is significantly lower than $|V|$, the vocabulary size), initialized at random at first. The probability that a particular word w surrounds the central word c is based on $\mathbf{v}_w^\mathsf{T}\,\mathbf{v}_c$, the dot product of vectors representing these two words (the higher the dot product, the higher the probability). Word vectors gradually change during training. In particular, vectors that represent words often (or rarely) found near each other in the corpus tend to change so that their dot product increases (or decreases).

The following batch loss is minimized during the training (the training is based on negative sampling, a simplification of contrastive estimation method):

$$\frac{1}{|\mathcal{B}|}\sum_{c\in\mathcal{B}}\left(\sum_{\substack{-w\le j\le w\\ j\neq 0}} -\log\sigma(\mathbf{v}_c^\mathsf{T}\mathbf{v}_{\overline{c+j}}) + \sum_{a\in\mathcal{A}} -\log\sigma(-\mathbf{v}_c^\mathsf{T}\mathbf{v}_a)\right)$$

where:

1. $\mathcal{B}$ is the training batch, a set of positions of central words within the corpus. Batch sampling is basically uniform, but frequent words are sampled with somewhat reduced probability, which still exceeds that of less frequent words. In detail, probability of sampling a word occurring in the corpus n times is given by the formula $1 - \sqrt{t/n}$, where t is a suitable constant, such as 10 000. This accelerates the training and renders better results, since training on many co-occurences with high-frequency words, such as *the*, would not be very fruitful.

2. $v_{\overline{c}}$ and $v_{\overline{c+j}}$ are vectors (each consisting of d trainable parameters) representing words that are at the c-th or $(c+j)$-th position of the corpus. By $\overline{n}$ we denote the word that is at n-th position of the corpus (and usually also at its many other positions).

   Note: Authors of [30] use two different vectors for each word, a complication neither strictly necessary for computing word2vecs nor addressed here.

3. w denotes the window size around the central word (10 can be a reasonable value).

4. $-\log\sigma$ is negative logarithm of the sigmoid function (applied in the formula to the dot product of the vectors):

$$-\log\sigma(x) = -\log\left(\frac{1}{1+e^{-x}}\right) = -\log(1) + \log(1+e^{-x}) = \log(1+e^{-x})$$

5. $\mathcal{A}$ denotes words obtained by sampling at random from all words in the corpus. Sampling probability of a word w that occurs in the corpus n_w times is given by the formula $n_w^{0.75}/Z$, where Z is the normalizing constant $(= \sum_{z \in V} n_z^{0.75})$. Thus, the higher the frequency, the higher the sampling probability, but sampling probabilities of frequent and infrequent words are, again, somewhat squeezed together, by the operation of the exponent. For small corpora, the optimal size of $\mathcal{A}$ can be 15 samples for each positive word-pair ($|\mathcal{A}| = 15 \cdot 2w$), but for larger corpora, the multiple of 5 ($|\mathcal{A}| = 5 \cdot 2w$) or even less samples may suffice.

   As words from $\mathcal{A}$ are sampled at random from the whole corpus, they are highly unlikely to occur near the central word, thus establishing **negative samples**. Accordingly, vector $\mathbf{v}_{\overline{c}}$ (for the word at c-th position in the corpus) and vector $\mathbf{v}_a$ (for the word a randomly chosen as one of negative samples in this pass) are meant to be altered during the training so that their dot product is low (see the *minus* sign within the last σ function in the formula).

Word2vec became famous and widely used mainly because of the following advantages:

1. The algorithm can compute word vectors fast, requiring neither the second neural network layer nor *softmax* computation over large number of classes (see section 2.2 above). This made it possible to train on large datasets and thus obtain vectors of the state of the art quality.

2. Authors made *pre-computed* vectors available to the public. Thus anyone could, and can, convert a text sequence into the vector sequence and feed it into a neural network of chosen design, focusing on the upstream NLP task. Good results are usually obtained already when pre-computed vectors are taken as fixed, but the possibility remains to fine-tune the vectors during neural network back-propagation.

3. It was expected that vectors representing words occurring in similar contexts would be similar (cosine similarity $= \frac{\mathbf{v_1}^\mathsf{T}\mathbf{v_2}}{\|\mathbf{v_1}\|_2 \, \|\mathbf{v_2}\|_2} = \cos(\theta)$ is usually used here as a measure). Yet word2vec papers have shown that, in addition, well-trained $\mathbb{R}^d$ space is even more structured. Some vectors (directions) within the space tend to indicate an important semantic or syntactic relationship, such as positive-negative, country-capital or masculine-feminine. For example, a question regarding analogy such as *X is to Paris as Germany is to Berlin* can be formalized as

$$v_X - v_{Paris} \approx v_{Germany} - v_{Berlin}$$
$$v_X \approx v_{Germany} - v_{Berlin} + v_{Paris}$$

   That is, if we take the vector representing the word *Germany*, subtract from it the vector representing the word *Berlin*, and add the vector representing the word *Paris*, obtaining thus the resulting vector, what vector (representing what word from the vocabulary) has the smallest cosine distance to it? (The vector representing the word *France*, indeed.)

These relationships (substructures) within the $\mathbb{R}^d$ space are somewhat surprising given that the only explicit aim during the training is to increase (or decrease) the dot product of vectors representing co-occurent (or not co-occurent) words. Accordingly, attempts have been published to explain why, or under what conditions, these substructures emerge during the training [25, 1]. In addition, the chance to obtain meaningful vector representations for other phenomena sparked word2vec-style training in other domains, such as biology [3].

Word2vec implementation was further developed by the Facebook research group. It has made several tweaks [20] and enriched word vectors with sub-word information [7]. Pre-computed word vectors trained using this improved method, known as **FastText**, are available at the fasttext.cc webpage for 157 different languages. FastText surpasses word2vec and GloVe (word vectorisation algorithm from Stanford University [31]) in various analogy and classification tasks [29], as well as in stability [8] (that is, in vectors having similar values irrespective of the random seed during training).

Accordingly, FastText is still the state-of-the-art solution when dealing with isolated words (see the word vector arithmetic above), or in certain applications with limited use of computational resources (see, e.g., DAN in [9]). Yet as regards most NLP tasks, contextual embeddings have prevailed in the last years. We will turn to them in section 2.5.

2.4 Subword Language Modelling. WordPieces.

We have been dealing so far with vectors that represent words (noting that FastText vectors are enriched by sub-word information [7]). It is due now to address vectors representing smaller language units, such as letters, syllables, or other sub-word parts. The main advantages of smaller-than-word language units are twofold:

1. **Morphologically rich languages.** In many languages, words have different forms according to their grammatical function or binding within the sentence. While we write *with a computer*, *from a computer*, or *on a computer* in English (always the same spelling *computer*), we write *s počítačem*, *od počítače*, or *na počítači* in Czech (the root *počítač* followed by various suffixes). Training on sub-word vectors allows to encode faster into vectors that words *počítačem*, *počítače* and *počítači* are related, and that the core meaning is borne by the root part *počítač*.
2. **Misspelled and rare words.** It is challenging for the pure word-vector approach to deal with misspelled words, such as *elepant*. *Elepant* and *elephant* are simply different words, and the word *elepant* occurs so rarely in the corpus (potentially just once) that it is impossible to train a reasonable word vector representing it. The same may hold for rare word variants, even if correctly spelled, such as *scarceness*. The sub-word approach yields better accuracy for these words, provided they share sub-word units with higher-frequency words (such as *elephant*, *scarce* or *scarcity*).

One possible approach is to use the smallest unit possible, a single character, for the first layer with embeddings (see section 2.2 above). Such character embeddings were used in the ELMo network, a predecessor of BERT. ELMo, too, computed context-sensitive word vectors, but was based on the bi-directional LSTM architecture and not on Transformers yet [32]. Detailed explanation of the architectures is available in Chapter 3.

The current state of the art, however, is a pre-determined number of sub-word units of variable character lengths. Then, depending on pre-definition

of these sub-word units, the sentence *Jet makers feud over orders.* may be tokenized to *_J et _makers _fe ud _over _orders.* (Please see below for the explanation of the _ sign for tokens denoting word beginnings.) This mid-point solution, with multiple-character sub-word units, retains flexibility while being more efficient, carrying more meaning than singular characters [46]. Arguably the most widely employed algorithm for sub-word tokenization are BPE and WordPiece[36], which is used also for the BERT-family models (see section 2.5 below).

The number of desired tokens (token inventory size) is n, a pre-determined number, e.g., 200 000. The optimization problem is to define n tokens so that the corpus coded as sequence of tokens has the minimal token length (e.g., the word *improve*, if it can be best coded by WordPieces *im*, *prove* included in the inventory, has the token length of 2). The algorithm in the following greedy style is used. We address the BPE variant here, whereas the current WordPiece algorithm uses somewhat more complex criterion, [36, 46, 38]:

1. Initially, the token inventory includes all language characters as tokens (and nothing more).
2. Two tokens from the inventory are identified that occur most often as a sequence in the corpus.
3. Concatenation of the two tokens identified in step 2 is added to the inventory as a new token.
4. The corpus is changed so that any occurrence of two consecutive tokens identified in step 2 is substituted by the new token created in step 3.
5. The algorithm terminates if the inventory already includes n items, otherwise the next cycle of steps 2 – 5 is run.

For instance, if the corpus consists of the sentence *_ t h i s _ i s _ a _h a t* (usually a special character is added at the beginning of each word so that, e.g., *are* is assigned a different token as a standalone word and as a part of another word), then *i s* is identified in step 2, token *is* is created in step 3, and the corpus is changed to *_t h is _ is _a _h a t* in step 4 See [38] for minimal implementation in Python (that is more efficient than the illustrative example above) and [36] for further improvements.

A pre-trained BERT model is usually made available to the public together with a pre-computed token dictionary and tokenizer that were used for training. A tokenizer is an executable code that can be used during fine-tuning and deployment to convert any input text into tokens (into their indices within the token inventory).

Tokenization usually takes much less time than subsequent processing steps. To make the process even faster, Google has recently published an algorithm that achieves further speedups (8.2x compared to HuggingFace and 5.1x compared to TensorFlow), based on prefix tries and altered Aho-Corasick algorithm [39].

2.5 Contextual Vectors. BERT Family of Models.

Deliberately simple models for producing word vectors (see section 2.3 above) could be trained on large datasets already in 2013. Later, the trend of ever-larger neural networks and increasing power of GPUs made it viable to train on large datasets even more expressive representations. Thus in 2018 researchers managed to publish two models: ELMo [33], based on RNNs (see detailed description of the recurrent architecture in section 3.1 below), and, more importantly, BERT [13], based on Transformers (see detailed description of the Transformer architecture in section 3.2 below). Both ELMo and BERT encode a language unit not *per se*, but as used in the context of the given sequence.

While there is a single non-contextual word vector for the word *tree* (thus it can be pre-computed for all future uses), contextual representation of the word is sequence dependent and differs, e.g., in each of the following three sentences: *I sit under the tree. I cut down our tree. I queried ancestors in the family tree.* Contextual embeddings can naturally disambiguate homonyms (the word *tree* has a different meaning in sentences 1 and 2 versus in sentence 3, as follows from the context). But their function is wider: contextual vectors representing the word *tree* differ also between sentence 1 and sentence 2.

Especially the BERT model has considerably improved state of the art results in various NLP tasks. The $\text{BERT}_{\text{LARGE}}$ variant (with 340M parameters) has exceeded, on average, pre-contextual accuracy in GLUE tasks ([45]) by 8.1 percentage points, and for question answering, the improvement was 33.6 percentage points. The improvement reached by the $\text{BERT}_{\text{BASE}}$ variant (with 110M parameters) was smaller, but it still exceeded contemporary state-of-the-art results by a wide margin. BERT variants published in the following years (see subsection 2.5.4 below) have improved results even further.

2.5.1 BERT Forward Computation

BERT (bidirectional encoder representations from Transformers) is the encoder-only model. It encodes n input tokens in n contextual embeddings.

In particular, BERT does not include a decoder, whose task is to take encoder results and generate ("decode"), usually in a one-by-one manner, tokens of the output sequence (e.g., in translation tasks), whose number and structure usually differs from the inputs. Nevertheless, BERT can be made a part of an encoder–decoder architecture, which goes beyond the topic of this thesis.

Text inputs (e.g. sentences) are processed by the tokenizer: each input is converted in the sequence of tokens and each token is represented by its vocabulary number. In addition the BERT model expects that the sequence starts with the special token [CLS] (a special item added to the WordPiece dictionary under a unique ID that has no counterpart in text data) and ends with the special token [SEP]. BERT can be used for tasks concerning a pair of sentences (pair of contiguous text sequences), such as whether one sentence follows from another; an additional [SEP] token is used to separate both sentences.

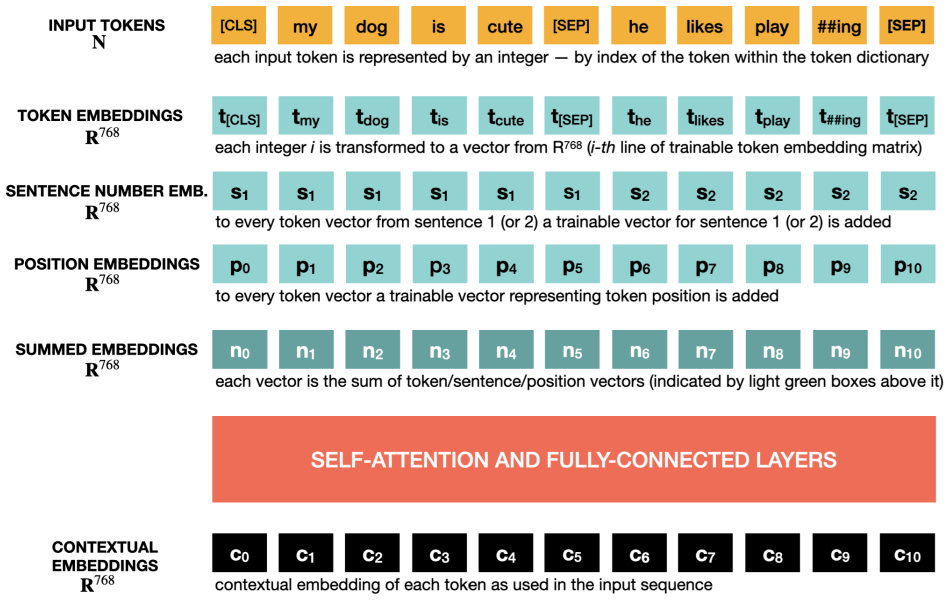


Figure 2.1: BERT Transforming Input Tokens into Output Contextual Embeddings. *Inspired by figure 2 in [13].*

As indicated in figure 1.1 above, transformation of input tokens into contextual embeddings proceeds in the following steps:

1. **Converting input token, sentence information and position information into a non-contextual vector.** Each of ℓ input tokens (where ℓ is the length of the input sequence, and every input token is an integer representing position of the token within the token dictionary) is transformed into non-contextual vectors first, using the three trainable embedding matrices:

- Token embedding matrix of $|\mathrm{V}| \times 768$, where $|\mathrm{V}|$ is the WordPiece dictionary size and 768 is the dimensionality of embeddings, a hyperparameter chosen by authors of BERT. (The value of 768 holds for the $\mathrm{BERT}_{\mathrm{BASE}}$ variant.)
- Sentence embedding matrix of 2×768, where the first line represents the first, and the second line the second, input sentence.
- Position embedding matrix of 512×768, where 512 is the maximum length of input sequence measured in tokens. (Need for this position token follows from the nature of Transformers and will be addressed in section 3.2 below.)

The non-contextual vector is the sum of (1) trainable embedding representing token ID, (2) trainable embedding distinguishing whether the token comes from the first or the second input sentence, and (3) trainable embedding distinguising position of the token within the input.

2. **Using Transformer layers with self-attention mechanism to obtain contextual vectors.** Every non-contextual vector $\mathbf{n}_i$, $i \in \{1, \cdots, \ell\}$, obtained in step 1 above, is run through the self-attention layer. The layer enables each vector to "look at" itself and any other vector and copy its information. This attention is *soft*, i.e., can be distributed, e.g., vector $\mathbf{n}_{10}$ can look from 40% at itself, from 42% at $\mathbf{n}_9$ and from 18% at $\mathbf{n}_1$.

   The self-attention layer includes a triplet of trainable matrices (Q, K a V) used to multiply each input $\mathbf{n}_i$, $\forall i \in \{1, \cdots, \ell\}$, and thus produce for the particular $\mathbf{n}_i$ three vectors: $\mathbf{q}_i$ (query), $\mathbf{k}_i$ (key) and $\mathbf{v}_i$ (value). For every query-key pair $\mathbf{q}_i$ and $\mathbf{k}_j$ (query for i-th input and key for j-th input), logits are computed as the dot product: $s_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j$. The softmax score $\overline{s}_{ij} = \frac{e^{s_{ij}}}{\sum_{x \in \{1,\cdots,\ell\}} e^{s_{ix}}}$ then denotes the amount by which i-th input vector $\mathbf{n}_i$ is to be enriched by value $\mathbf{v}_j$ (which $\mathbf{v}_j$ was produced from $\mathbf{n}_j$ using the V matrix, as described above). Thus, the resulting i-th contextual vector $\mathbf{c}_i = \mathbf{n}_i + \sum_{x \in 1,\cdots,\ell} \overline{s}_{ix}\, \mathbf{v}_x$.

   This description of computation in stage 2 demonstrates aims at the fundamental logic of generating context-dependent representations, while omitting a number of details, such as:

   - the attention layer includes not just one (single-head attention), but multiple (12 in $\mathrm{BERT}_{\mathrm{BASE}}$ and 24 in BERT_{LARGE}) Q, K and V matrices, producing multiple $\mathbf{q}_i$, $\mathbf{k}_i$, $\mathbf{v}_i$ values for each input vector $\mathbf{n}_i$ (multi-head attention);
   - each resulting $\mathbf{c}_i$ vector is then separately processed though a feed-forward layer, and its output is added to the $\mathbf{c}_i$ vector;
   - layer normalization layers are used; and
   - there is not just one attention/feed-forward/normalization block, but inputs are consecutively processed by more such blocks (12 in $\mathrm{BERT}_{\mathrm{BASE}}$ and 24 in $\mathrm{BERT}_{\mathrm{LARGE}}$).

Please find a more detailed explanation, with matrix notation, of the Transformer encoder in section 3.2 below.

3. **Adding the appropriate head.** Stage 2 above outputs $\mathbf{c}_1, \cdots, \mathbf{c}_\ell$ vectors (each from $\mathbb{R}^{768}$ in $\text{BERT}_{\text{BASE}}$ variant or from $\mathbb{R}^{1024}$ in $\text{BERT}_{\text{LARGE}}$ variant). These vector embeddings can be sometimes directly used further (e.g., for a clustering task after the BERT model has been finetuned). Nevertheless, for most NLP tasks, and also for pre-training, the final layer called *head* needs to be added. For instance, the *pre-training head* consists of a dense layer with 768 (or 1024) input dimension and $|V|$ (size of the token vocabulary) output dimension, followed by softmax (and cross-entropy loss). Thus the final output at the pre-training consists of vectors $\mathbf{h}_1, \cdots, \mathbf{h}_\ell \in \mathbb{R}^{|V|}$.

2.5.2 BERT Pre-Training

Training of the BERT model usually proceeds in two stages: initial pre-training and subsequent fine-tuning. Pre-training starts with the initial random state of the model and requires vast amounts of data (BERT has been pre-trained on 3.3M words) and of computational resources (such as dozens of TPUs). Two pre-training tasks are processed concurrently with the same input sequences:

- **Masked word prediction.** On average, 15% of input tokens used in training are masked or mangled: a token at such randomly chosen position is substituted by the special [MASK] token (80% probability), left intact (10% probability) or substituted by a random token (10% probability). The BERT model is then asked to recover the original token at each of these chosen positions, using the context of the whole input sequence. For instance, the input sequence of 99 tokens contained tokens *dog*, *on*, *case*, *fast* at positions 21, 38, 45 and 73, but these positions were randomly chosen for masking/mangling. The model then minimizes the loss by trying to predict from the context that tokens *dog*, *on*, *case*, *fast* fit best these positions. That is, the model is trained to output from its pre-training head (see stage 3 above) vector $\mathbf{h}_{21}$ ($\mathbf{h}_{38}$, $\mathbf{h}_{45}$, $\mathbf{h}_{73}$) with element 144 (51, 833, 2143) approaching 1.0 and other elements approaching 0.0 (assuming that 144, 51, 833 and 2143 are positions of words *dog*, *on*, *case* and *fast* within the WordPiece dictionary).
- **Next sentence classification.** In addition to token mangling, the BERT model is asked to render a binary classification whether the second sentence follows the first or is randomly chosen: with 50% probability the second sentence actually follows the first one in the training text and with 50% probability it is taken from a random position. The prediction is made using the vector $\mathbf{c}_0$ from the second stage (e.g. followed by the

dense layer $768 \to 1$ and sigmoid function) that corresponds to the initial [CLS] token (as explained above, this token is always pre-pended to the sequence and has no corresponding part in the input text).

Pre-training of BERT is bidirectional and unsupervised, and its key purpose is transfer learning.

- **Bidirectional** means that, in order to predict the mangled token, BERT uses the whole context, both preceding and subsequent tokens. This bidirectionality increases accuracy of the model and quality of its language model.
- **Unsupervised** means that no human labeling is needed (unlike in training of usual CNN models for image classification tasks). BERT pre-training as well as training of word2vec or FastText) is fascinating for using easily available resources (e.g., texts scraped from the internet) without any further requirement on human knowledge or labor. Text manipulations needed to prepare training data (such as mangling randomly chosen words) are fully automatic.
- **Transfer learning** means that the model pre-trained on one tasks is then used for another. Mostly, we are not really interested in predicting what words were at mangled positions. Nevertheless, pre-training on this task (which is not directly applicable but requires no labeled data) teaches the model to store in parameters rather complex knowledge of language as used. Pre-trained once on 3.3M words on Google's hardware and made available to the public, the BERT model can be then fine-tuned for a particular task.

2.5.3 BERT Fine-Tuning

Fine-tuning a BERT model for a particular task usually requires much less computation than the pre-training stage. Also, it requires much less compute *and labeled data* than training a randomly initialized model for the task. To fine-tune, one needs to discard the pre-training head (see subsection 2.5.1, step 3, above), whose parameters serve the specific pre-training task (masked word and next-sentence prediction). The structure of the new head depends on the particular NLP task; mostly, it is just a fully connected layer followed by softmax (and cross-entropy loss). There are two options for using the pre-trained model with the new head:

1. to train the new head only. In this case, only forward passes are computed regarding Transformer layers. Accordingly, this option is cheaper than full backpropagation (already BERT $_{\text{BASE}}$ has 110M parameters), but usually renders sub-standard results.

2. to train the new head and also backpropagate gradients in Transformer layers (the term *fine-tuning* mostly denotes only altering parameters in Transformer layers). To fine-tune, and not to destroy, parameters learned during pre-training, one can recommend a reduced learning rate and warm-up epochs (i.e, the learning rate schedule that starts at zero and linearly approaches the target level).

These are examples of NLP tasks that can be solved by fine-tuning a BERT model (see [13], appendix B, for further details):

- **Sentiment classification**. The task is to classify the sentiment of the input sentence as *positive*, *neutral* or *negative*. Either the $\mathbf{c}_0$ output (representing the [CLS] input token) or the average of all $\mathbf{c}$s is fed into the head consisting of 768×3 (or 1024×3, for $\text{BERT}_{\text{LARGE}}$) dense layer and softmax.
- **Natural Language Inference**. The task is to classify if the second sentence (separated by the [SEP] token) is the *entailment* of (follows from), *contradiction* to, or *neutral* to the first sentence. Either the $\mathbf{c}_0$ output (representing the [CLS] input token) or the average of all $\mathbf{c}$s is fed into the head consisting of 768×3 (or 1024×3, for $\text{BERT}_{\text{LARGE}}$) dense layer and softmax.
- **Extractive question answering**. The task is to decide where in the second sentence/paragraph (separated by the [SEP] token) is the answer to the question posed in the first sentence. Two separate 768×1 dense layers, S and E, are used. For each input token in the second sentence/paragraph, its contextual embedding $\mathbf{n}$ is fed into S (or E), and softmax over the results determines the probability that the answer starts (or ends) at a particular token.

  Unlike such *extractive* question answering task, *abstractive* question answering requires that the model generates the answer (using the encoder–decoder architecture instead of a single BERT model).

2.5.4 BERT Variants

After the original BERT model [13], a number of its variants were made available as pre-trained models and published as research papers. Apart from Sentence Transformers and ColBERT models, addressed in section 2.6, we mention some variants below:

- **RoBERTa** [26], which improves on BERT results in particular by: training on 10x more text (160GB for RoBERTa vs 16GB for BERT),

training for considerably more iterations (it has shown that the original BERT model was somewhat under-trained), using a better word-masking procedure (dynamic masking vs static masking), and doing without the next sentence classification subtask.

- **XLM-RoBERTa** [12], a multilingual model trained (unlike English-only BERT and RoBERTa) on 2494 GB of text from 100 languages; 16.3 GB (0.65%) was in Czech. XLM-Roberta has demonstrated that a multilingual model can be competitive to monolingual models, and that multilingual models can largely learn a shared language representation.
- **RobeCzech** [40], a RoBERTa-style model trained solely on Czech data that surpasses XLM-Roberta$_{\text{LARGE}}$ in 7 out of 8 NLP tasks measured in the paper (even though RobeCzech has 125M parameters whereas XLM-Roberta$_{\text{LARGE}}$ 559M).
- **Longformer** [5], a model designed to process long sequences. It is based on a local windowed attention to achieve linear time and memory complexity in sequence length ℓ, whereas the standard attention architecture, with its everyone-looks-at-everyone nature, is $\mathcal{O}(\ell^2)$. In addition, a larger position embedding matrix in longformers supports 4096 token positions (BERT and RoBERTa only 512); in one ablation, the number of token positions was extended even to 16K.

2.6 Sentence Vectors

Many real-world applications deal with sentences or paragraphs, not with word and sub-word units, addressed so far. The application can be to cluster given sentences (paragraphs) in multiple coherent groups or to find sentences (paragraphs) most semantically similar to the given one. With a large corpus, it usually is prohibitively time consuming to run each sentence pair through a fine-tuned neural network (BERT can accept a pair of sentences separated by the [SEP] token). Frequently, the only workable solution is to pre-calculate a vector for each sentence, and than calculate cosine similarity of such vectors as needed.

The following are the main approaches to getting from word or sub-word vectors to sentence (or paragraph) vectors:

1. **Averaging word or sub-word vectors.** The basic approach is to average all vectors (whether computed by FastText, GloVe, or pre-trained BERT model) that represent words (or sub-word tokens) of the particular sentence (or paragraph). This single averaged vector, representing the whole sentence (or paragraph), is then used for upstream NLP tasks.

However, results of this approach are usually considerably below the current state of the art. The BERT pre-training task (see subsection 2.5.2 above) is not aimed at computing embeddings that make a good sentence representation when averaged. Accordingly, averaging outputs of a pre-trained BERT usually does not yield better results than averaging non-contextual vectors, such as FastText or GloVe [34].

2. **Fine-tuning a single BERT network.** The quality of BERT sentence embeddings (computed as the average of all output embeddings) can be significantly improved by fine-tuning BERT on tasks where the head operates on the average of all Transformer outputs [9]. Fine-tuning on a wide range of such single-sentence NLP tasks teaches the network to produce high-quality sentence representations that are generally useful also for other tasks. Further fine-tuning for the particular task is, of course, possible.

3. **Fine-tuning Siamese BERT networks.** This approach (see [34]) is aimed at two-sentence tasks, such as calculating semantic similarity. The term *Siamese* denotes (at least for explanatory purposes) two parallel networks with shared weights, each of them processing one of the input sentences to obtain its sentence vector (by averaging token vectors, as described above in item 2). Usually, the implementation consists of a single network, and each of the sentences is processed through it; the gradient then propagates back through each of both sentence vectors and reaches the same BERT (Transformer) parameters.

   High-accuracy results have been obtained by fine-tuning a pre-trained BERT model on natural language inference (NLI) datasets [34], in particular on Stanford NLI (SNLI), containing 570 000 sentence pairs, and on MultiNLI, containing 430 000 sentence pairs, where each sentence pair is assigned a label *entailment* (one sentence follows from the other), *contradiction*, or *neutral*. The main approach was to:

   - use the Siamese network to get a vector for the first sentence (vector $\mathbf{u} \in \mathbb{R}^d$) and for the second sentence (vector $\mathbf{v} \in \mathbb{R}^d$) of the sentence-pair drawn from SNLI/MultiNLI.
   - compute additional vector as difference of $\mathbf{u}$ and $\mathbf{v}$ (vector $\mathbf{u} - \mathbf{v}$)
   - concatenate these three vectors, $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{u} - \mathbf{v}$ into $\mathbf{t} \in \mathbb{R}^{3d}$
   - run $\mathbf{t}$ through a fully connected layer to obtain vector $\mathbf{l} \in \mathbb{R}^3$ (i.e., multiply the trainable matrix of parameters $\mathbf{M} \in \mathbb{R}^{3 \times 3d}$ by $\mathbf{t}$)
   - run $\mathbf{l}$ through softmax to finally obtain $\mathbf{p} \in \mathbb{R}^3$, denoting the probability of *entailment*, *contradiction*, or *neutral* for the two input sentences.
   - compare $\mathbf{p}$ against the label from SNLI/MultiNLI and train the network using the standard negative log loss over $\mathbf{p}$ (cross-entropy loss over $\mathbf{l}$).

Pre-trained neural networks that can compute high-quality sentence vectors, also in a multilingual setting, are available at `sbert.net` and `huggingface.co` webpages.

Finally, we briefly mention the ColBERT model [21], a good alternative to sentence vectors for some tasks. ColBERT represents a computational middle ground between (1) running a full BERT forward computation whenever a pair-of-sentences task needs to be evaluated and (2) using as simple an interaction as cosine similarity on pre-computed sentence vectors. The ColBERT model is usually trained by finetuning BERT and used to pre-compute for a particular document (e.g, the document number 1730) its embeddings $\mathbf{e}_1^{(1730)}, \cdots, \mathbf{e}_l^{(1730)}$, where l is the pre-set number of per-document embeddings. The query is converted in embeddings $\mathbf{q}_1, \cdots, \mathbf{q}_l$, and the similarity score between the query and the particular document $d \in \{1, \cdots, n\}$ (where n is the number of documents) is determined as the sum of the highest cosine similarities.

$$\sum_{\mathbf{q}\in\{\mathbf{q}_1,\cdots,\mathbf{q}_\ell\}} \max_{\mathbf{e}\in\{\mathbf{e}_1^{(d)},\cdots,\mathbf{e}_l^{(d)}\}} \frac{\mathbf{q}^\mathsf{T}\mathbf{e}}{\|\mathbf{q}\|_2\,\|\mathbf{e}\|_2}$$

The score regarding the query can be computed also for other documents (usually for all documents in the dataset), especially when their ColBERT embeddings have already been pre-computed. The system can, e.g., display documents with top 10 scores.

Chapter 3

Sequential Data. RNNs. Transformers. Set Transformers.

We treat here the Transformer architecture in more detail, as it is a basis both for BERT-family models (see 2.5) and for deep amortized clustering we will focus on in the next chapter.

3.1 Recurrent Neural Networks

Transformer architecture (section 3.2 below) is a reaction to, and improvement over, recurrent neural networks (RNNs). These naturally model sequential data (such as texts, if text units are represented by vectors, as addressed in chapter 1) and are still in use, in particular for smaller datasets, as Transformers often require pre-training on large data to achieve superior results.

3.1.1 RNN as a Computation Graph

A clear understanding of RNNs is, in my opinion, acquired more easily without using somewhat confusing concepts of *"recurrence"* or *"unrolling a single RNN cell in time"*. In fact, the key aspect of RNNs is that for each input sequence a particular computation graph is built with as many computation nodes as there are items in the sequence. The following figure depicts such computation

graph built for a sequence with 3 items (whereas if the sequence had 5 items, the graph would be built with 5 cells).

The key aspects of the computation graph are as follows:

- Each RNN cell is merely a computation node, transforming inputs into outputs. In particular, the cell contains no trainable parameters of its own.
- There is a single trainable weights matrix W (we omit the bias term for simplicity) that is shared by all cells.
- Each cell receives the input vector $\mathbf{i}_n$ and a state vector $\mathbf{s}_n$, concatenates them into vector $\mathbf{c}_n$ and multiplies the trainable weights matrix W by the concatenation $\mathbf{c}_n$. The result is both the output of the current cell and the state of the next cell ($\mathbf{o}_n = \mathbf{s}_{n+1}$).

  We use here the formulation with a single weight matrix W and calculation of $f(\mathrm{W}\,\mathbf{c}_n)$, where f is the activation function, instead of the equal formulation with matrices A and B (where $[\mathrm{A}\,\mathrm{B}] = \mathrm{W}$) and calculation of $f(\mathrm{A}\,\mathbf{i}_n + \mathrm{B}\,\mathbf{s}_n)$

  As the first cell has no previous cell to receive its state from, the initial state needs to be chosen arbitrarily, e.g., as a zero vector.

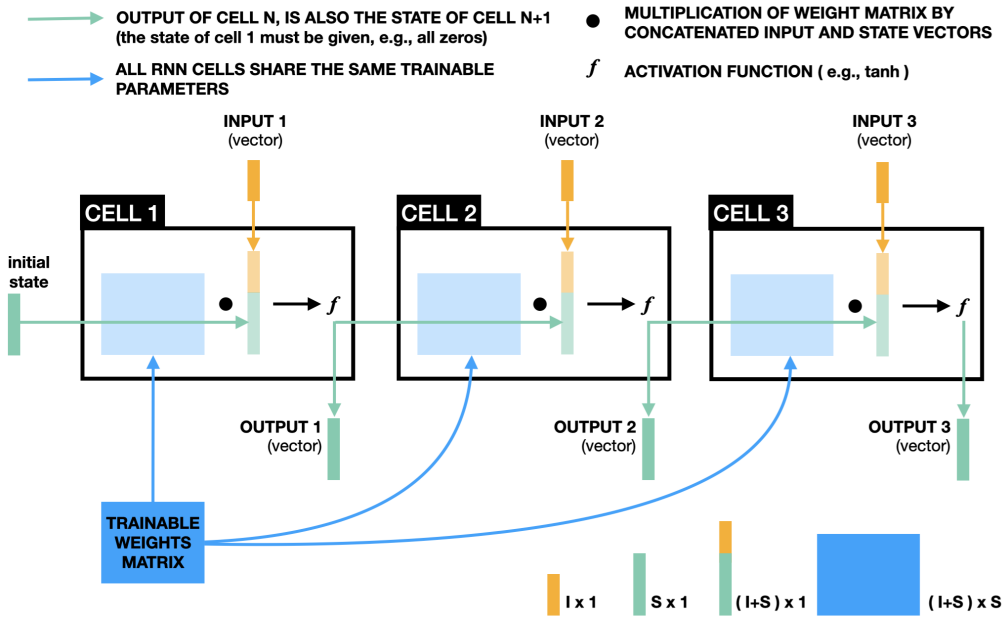


Figure 3.1: RNN Computation Graph: for input sequence of 3 items (vectors)

- Whereas I, the dimension of input vectors, is given, S, dimension of the state, is a hyperparameter, chosen according to requirements of the upstream task or with respect to the balance of memory and computation requirements and network representation power.

- It depends on the task whether to use only the last output (output 3 in the example of a sequence above) or all outputs. The last output only may be fed into a dense layer and softmax if the task is to classify the whole input sequence (e.g., sentiment classification). Every output needs to be fed into a dense layer and softmax if the task requires to classify every input word (e.g. semantic tagging).
- Standard rules for gradient backpropagation in computation graphs apply. If, in the example above (see also figure 2.1), output 3 is followed by the final layer L consisting of softmax and cross-entropy loss, then:
  1. $\mathcal{L}^{\nabla}_{\mathrm{L}}$ (gradient of loss w.r.t. input of the final layer L) equals $\mathbf{s}-\mathbf{t}$ (output of the softmax minus the vector, usually one-hot, representing the target).
  2. $\mathcal{L}^{\nabla}_{f_3}$ (gradient of loss w.r.t. input of the activation in cell 3) equals $f'_3(\bullet) \odot \mathcal{L}^{\nabla}_{\mathrm{L}}$ (derivatives of the activation function applied element-wise to input elements multiplied element-wise by $\mathcal{L}^{\nabla}_{\mathrm{L}}$).
  3. $\mathcal{L}^{\nabla}_{c_3}$ (gradient of loss w.r.t. concatenated vector $\mathbf{c}$ in cell 3) equals the product $\mathrm{W}^{\mathsf{T}}\,\mathcal{L}^{\nabla}_{f_3}$, whereas $\mathcal{L}^{\nabla}_{\mathrm{W}_3}$ (contribution of layer 3 to the gradient of loss w. r. t. weight matrix W) equals the outer product $\mathcal{L}^{\nabla}_{f_3}\,\mathbf{c}_3^{\mathsf{T}}$.
  4. Please note that the vector $\mathcal{L}^{\nabla}_{c_3}$ is a concatenation of
     - vector $\mathcal{L}^{\nabla}_{i_3}$, which can be backpropagated into inputs of the RNN layer (if the network consists of more layers); and
     - vector $\mathcal{L}^{\nabla}_{s_3}$, which is backpropagated into the previous cell. There, $\mathcal{L}^{\nabla}_{f_2}$, $\mathcal{L}^{\nabla}_{c_2}$ and $\mathcal{L}^{\nabla}_{\mathrm{W}_2}$ can be then computed using steps 2 and 3 (and subsequently $\mathcal{L}^{\nabla}_{f_1}$, $\mathcal{L}^{\nabla}_{c_1}$ and $\mathcal{L}^{\nabla}_{\mathrm{W}_1}$ can be computed using steps 2 and 3 again.)
  5. As the same weight matrix is used in ("flows into") each of n cells, $\mathcal{L}^{\nabla}_{\mathrm{W}} = \sum_{i=1}^{n} \mathcal{L}^{\nabla}_{\mathrm{W}_i}$. Gradient of loss w. r. t. weight matrix equals the sum of contributions from each cell.

3.1.2 Bidirectional and Multilayer RNNs

The architecture illustrated above does not enable looking back. For instance, the computation of output 2 is based only on inputs 1 and 2, but cannot make use of input 3. It is often beneficial, however, to use the whole context (e.g., classifying a word in a sentence into one of linguistic categories is more accurate when both preceding and following words are taken into account). Accordingly, the predictive power of RNNs is increased by using

1. a bidirectional layer, consisting of a forward layer (which receives input elements in the order from the first to the last) and a backward layer

(which receives input elements in the order from the last to the first); every output of the forward layer is summed (or, in some architectures, concatenated) with the corresponding output of the backward layer; and

2. more such bidirectional layers, with output of a RNN cell in one layer being the input of a corresponding RNN cell in another layer. (It is usual to include skip connections and layer normalization, too, but we omit these in the figure below.)

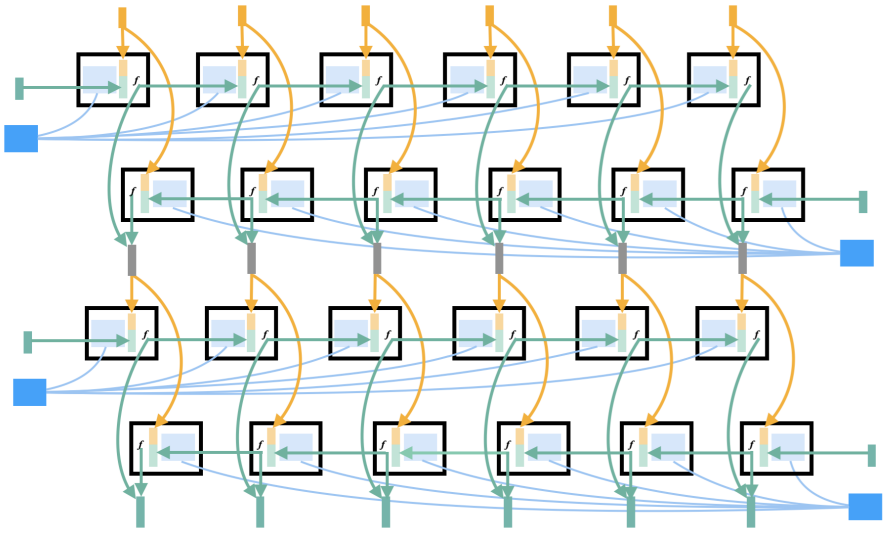


Figure 3.2: RNN with Two Bidirectional Layers: processing the input sequence of 6 elements at the top into 6 outputs at the bottom

3.1.3 Simple RNN, GRU and LSTM

As the explanation above focused on the structure and function of an RNN network, it used the most basic kind of an RNN cell. However, this simple RNN cell is rarely, if ever, employed in real-world applications. One of the key reasons is that consecutively multiplying vectors by the same matrix W can easily result in exploding or vanishing gradients (see, e.g., [17]). To address the issue, various architectures of more complex cells have been published. The LSTM variant (*long short-term memory*), devised already in 1997 and updated in 1999 (see [18] and [16]), is still the state of the art solution, even though some simplifications, such as the GRU cell (*gated recurrence unit*), see [10] and [11], can frequently achieve comparable results with somewhat lower computational demands.

The following key points regarding the LSTM cell compared to the simple RNN cell:

- Whereas the simple RNN cell gets input $\mathbf{i}_n$ and state $\mathbf{s}_n = \mathbf{o}_{n-1}$ vectors and produces the single vector $\mathbf{o}_n = \mathbf{s}_{n+1}$, the LSTM cell receives input $\mathbf{i}_n$, state $\mathbf{s}_n$ and memory $\mathbf{m}_{n-1}$ vectors and produces $\mathbf{o}_n = \mathbf{s}_{n+1}$ and $\mathbf{m}_n$ vectors.

- Whereas a simple RNN layer has a single W trainable matrix, a LSTM layer has 4: $\mathrm{W_f}$, $\mathrm{W_m}$, $\mathrm{W_i}$ and $\mathrm{W_o}$. (We omit here bias terms accompanying each of the trainable matrices.) As described below, each of these matrices is multiplied by $\mathbf{c}_n$ (a concatenation of input $\mathbf{i}_n$ and state $\mathbf{s}_n = \mathbf{o}_{n-1}$ vectors, see subsection 3.1.1 above) to produce the *f*orget mask, *m*emory storage candidate, *i*nput mask and *o*utput mask.

- The forget mask is calculated as $f_t = \sigma(\mathrm{W_f}\,\mathbf{c}_t)$. The sigmoid function σ makes each f_t element lie between 0 (full forgetting of information at the position) and 1 (full retention).

- The candidate for being stored in the memory (the new potential memory contents) is computed as $\widetilde{\mathbf{m}}_n = \tanh(\mathrm{W_m}\,\mathbf{c}_n)$.

- The input mask, determining to what extent the candidate will be in fact stored in the memory, is computed as $i_t = \sigma(\mathrm{W_i}\,\mathbf{c}_t)$.

- $\mathbf{m}_n = \mathbf{m}_{n-1} \odot f_t + \widetilde{\mathbf{m}}_t \odot i_t$, where $\odot$ denotes element-wise multiplication. In words, the memory vector resulting from n-th LSTM cell is determined by: (1) the previous memory vector and partial forgetting of its contents according to the forget mask; and (2) adding the new contents according to the input mask.

- It remains to compute $\mathbf{o}_n$, the output vector produced by cell n. It is determined as a masked readout from now-computed memory: $\mathbf{o}_n = \tanh(\mathbf{m}_n) \odot \sigma(\mathrm{W_o}\,\mathbf{c_t})$.

3.2 Transformer Encoders and Self-Attention

Recurrent neural networks (in LSTM or GRU variant) are still a powerful and used technology. They have been, however, largely superseded by the new Transformer architecture, originating in the *Attention Is All You Need* breakthrough paper [42] from 2017, one of the most cited resources in the history of neural-networks research. Popularity of the Transformer architecture has been increasing fast in the last 5 years, and now Transformers are a major architecture also in computer vision (see, e.g., [14]). Accordingly, one can reasonably claim that current major structural architectures of neural networks (or their major layers) are: dense, convolutional (CNN), recurrent (RNN), and Transformer.

3.2.1 Transformer Model and its Advantage

Transformer architecture was devised in [42] to process sequential data and have more favourable properties regarding:

1. **Parallelization.** RNNs are less suitable for parallelization as the computation represented by the RNN cell needs to be applied to input elements sequentially: computing output $n+1$ depends on the state vector computed in step n. This sequential nature of RNNs, even if alleviated to some extent by certain "tricks" (see, e.g., [22]), limits the amount of training data or training steps that can be processed in a still reasonable time frame.

   To the contrary, Transformer architecture is well-suited for training even with dozens of GPUs or TPUs. Given these resources, Transformer models, such as BERT (see section 2.5 above), can be trained even on full Wikipedia and Book Corpus datasets (3.3 billion words in total, see [13]). Whereas the BERT model was originally trained in days, recent experiments have demonstrated (see [48]) that with 1024 TPUs, batch size of 32k and *LAMB* optimizer (which is more suitable for large batches than *Adam*) training time can be reduced to 101 minutes while achieving superior F1 score (or to 76 minutes while still achieving a better F1 score than the original BERT).

2. **Long-term dependencies.** RNNs accumulate information while consecutively processing elements of the input sequence. This makes RNNs more efficient at representing relationships of elements closer in the sequence, whereas information regarding a distant element may be gradually overshadowed.

   To the contrary, Transformers can explicitly represent relationships between any pair of elements within the input sequence (this, however, makes complexity of Transformers quadratic in input length, whereas such complexity of RNNs is linear). In addition, Transformers use multiple attention heads, which allows to explicitly model different kinds of mutual relationships.

3.2.2 Top-Level Structure of Transformer Encoder

The *Attention Is All You Need* model consists of both encoder and decoder (see also section 2.5 above). However, given the assignment regarding this thesis (encoding text tokens into vectors to be clustered, with no need to generate another sequence of tokens), we will focus below on the encoder part only.

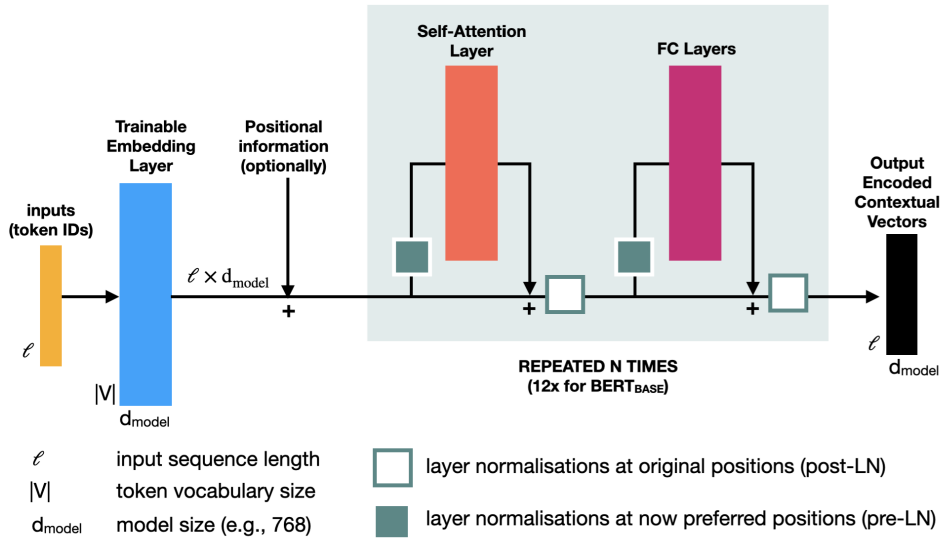


Figure 3.3: General Structure of Transformer Encoder

Token vectors resulting from the embedding layer can be enriched with the positional information (we will address this at the end of this section) or some other additional information depending on the task (see sentence number embeddings in the BERT model, section 2.5 above). Resulting vectors are then processed by the self-attention layer, the key component of the architecture (addressed in detail below). Output of the self-attention layer is then run through a couple of fully-connected layers, which operate separately on each of ℓ rows (i.e., separately on each of $1 \times \ell$ row vectors going into the fully-connected layers) as follows:

$$\mathbf{v}_{1\times d_{model}} \rightarrow \mathrm{FC}_1 \rightarrow \mathbf{v}_{1\times 4d_{model}} \rightarrow \mathrm{ReLU} \rightarrow \mathrm{dropout} \rightarrow \mathrm{FC}_2 \rightarrow \mathbf{v}_{1\times \mathrm{d_{model}}}$$

Original Transformers included layer normalizations [4] just after the output of self-attention and just after the output of fully connected layers is added to skip connections (post-LN transformer). However, placing layer normalizations just before self-attention and just before fully-connected layers instead (pre-LN Transformer), which leaves one path for gradient backpropagation free, tends to increase stability of training and reduce the need for warm-up training steps (see [47]).

3.2.3 Single-Head Attention Layer

The key principle of self-attention is described in section 2.5 above. Trainable matrices $\mathrm{W_Q}$, $\mathrm{W_K}$ and $\mathrm{W_V}$ produce from each input vector its query, key

and value vectors. The query for that input vector is then used to "look at" keys for this and every other input vector. The higher the dot product between the query of the looking vector and the key of the looked-at vector, the higher is the value of the looked-at vector added to the looking vector. This everyone-looks-at-everyone architecture is, unsurprisingly, implemented using matrix operations. These operations are shown in figure 2.4 below, a detailed look inside the red-salmon box *self-attention layer* from figure 2.3 above:

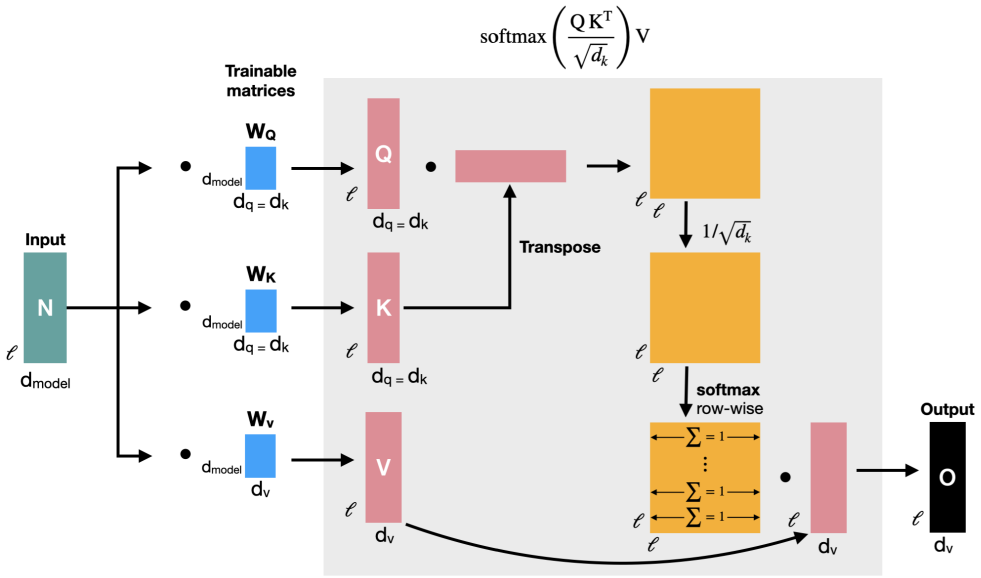


Figure 3.4: Self-Attention Layer

Division by the $\sqrt{d_k} = \sqrt{d_q}$ term is used to balance the effect that dot products of larger vectors (whose elements still have a similar distribution) tend to get larger, thus *"pushing the softmax function into regions where it has extremely small gradients"* [42].

3.2.4 Multi-Head Attention Layer

While we have addressed a single-head attention layer in figure 2.4. above, multi-head attention layers are mostly used instead (e.g., $\text{BERT}_{\text{BASE}}$ has 12 attention heads). Each head of the multi-head layer (and its trainable parameters) can specialize in different kinds of relationships. In text models, some heads usually attend to most adjacent tokens, some heads to tokens in particular syntactic relations (such as objects or adverbial modifiers) and some heads to least frequent words in the sequence [44].

Technically, multi-head attention layer with h heads ($h > 1$) includes not a single triplet $\mathrm{W_Q}$, $\mathrm{W_K}$, $\mathrm{W_V}$ of trainable matrices, but h of them. Each of these h triplets is, as shown in figure 2.5 below, separately applied to the input, producing h separate triplets Q, K and V. Every such Q, K, V triplet is separately run through the softmax$\left(\frac{\mathrm{Q\,K^T}}{\sqrt{d_k}}\right)$V computation, producing h outputs.

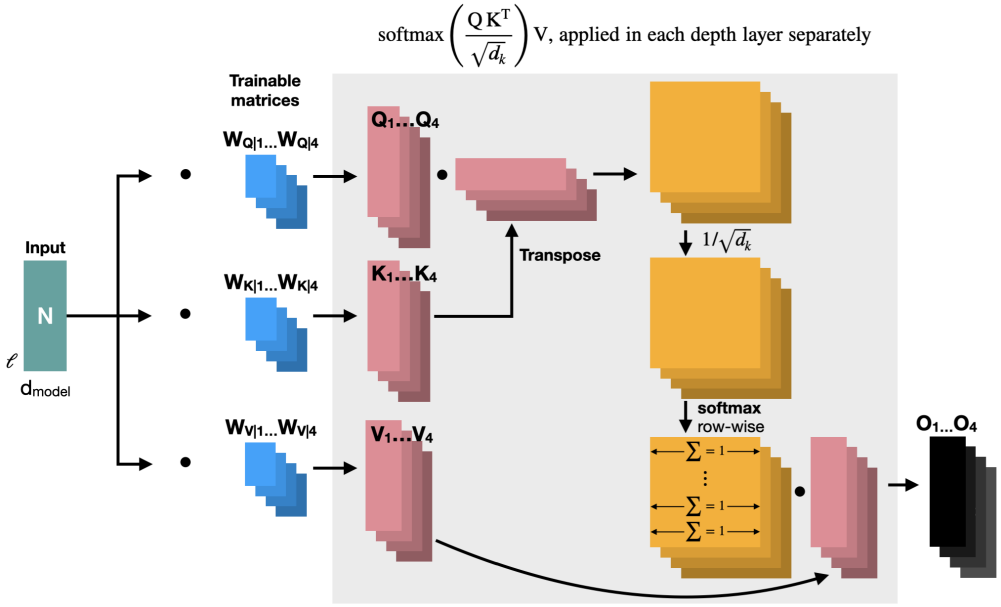


Figure 3.5: Multi-head Self-Attention: example with 4 heads.

Usually, $d_k = d_q$ and d_v dimension are smaller for multi-head attention. In both [13] and [42], $d_k = d_q = d_v = d_{model}/h$. E.g., for $\mathrm{BERT_{BASE}}$, $d_{model} = 768$, $h = 12$ and $d_k = d_q = d_v = 64$. Thus, multi-head attention requires similar computational resources as a single-headed layer with $d_k = d_q = d_v = d_{model}$.

The multi-head attention layer needs to output, just like a single-head attention layer, a single matrix with dimensions $\ell \times d_{model}$. Yet as seen above, multi-head computation results in h output matrices $\mathrm{O}_1, \cdots, \mathrm{O}_h$, each with dimensions $\ell \times d_v$. Therefore, these h matrices $\mathrm{O}_1, \cdots, \mathrm{O}_h$ are then concatenated into a single matrix C with dimensions $\ell \times hd_v$. Moreover, a multi-head layer includes, unlike a single-head layer, an additional $\mathrm{W_O}$ trainable matrix with dimensions $hd_v \times d_{model}$ (there is exactly one $\mathrm{W_O}$ matrix per multi-head layer, irrespective of $h, h > 1$). The product $\mathrm{C\,W_O}$ (calculated even in cases when $hd_v = d_{model}$) then gives the final output with dimensions $\ell \times d_{model}$.

3.2.5 Encoding Positions

In recurrent networks (section 3.1), already the computation process generates sufficient information concerning mutual positions of items within the input sequence. For instance, calculation on the first input $\mathbf{i}_1$ encodes the state $\mathbf{o}_1 = \mathbf{s}_2$ used for calculation in the second RNN cell.

To the contrary, computation in Transformers, due to its parallel and permutation-equivariant nature, has no means to encode mutual distance. If no input vector $\mathbf{n}_i, i \in 1, ..., \ell$, (i.e., no line of the N matrix) contains a positional information, nor does a triplet $\mathbf{q}_i$, $\mathbf{k}_i$, $\mathbf{v}_i$ (trainable matrices $\mathrm{W_Q}$, $\mathrm{W_K}$, $\mathrm{W_V}$ multiply each of input vectors separately). Nor can positional information emerge from a singular calculation with $\mathbf{q}$, $\mathbf{k}$ and $\mathbf{v}$ vectors (if none of them includes any) or from summing $\sum_{x \in \{1,\cdots,\ell\}} (\mathbf{q}_i \cdot \mathbf{k}_x)\mathbf{v}_x$ over the whole length of the input sequence (permutation invariance of sum).

Nevertheless, considering mutual (relative) positions of input elements can considerably increase accuracy. Especially for language models, such as BERT, positional information makes the difference between having to see the whole input as a single bag of words and between the ability to understand it as a sequence. Positional encodings enable to consider, e.g., that within the sentence *"A kid started to play a game with another kid."*, the first word (*"a"*) is, also because of its close position, more related (and needs to attend more) to the 2nd word (*"kid"*) then to the 7th (*"game"*) or to the last word (*"kid"*).

As positional information cannot be generated by the Transformer (BERT) computation, it needs to be already included in input vectors. Two methods are used:

1. **Positional embeddings**, as in the BERT family of models [13]. These positional embeddings are trained together with other parameters of the model. The disadvantage is that availability of positional information depends on length of training sequences and is limited to the pre-determined size of the trainable matrix with positional embeddings.
2. **Wave addition**, as in the *Attention is all you need* paper [42]. To every even element $2i$ of every input vector is, depending on its p (position within the input sequence), added the value $sin(\frac{p}{10000^{2i/d_{model}}})$ and to its every odd element $2i+1$ is added the value $cos(\frac{p}{10000^{2i/d_{model}}})$.

3.3 Set Transformer. Generalized Attention.

Authors of [23] have developed *Set Transformers* to process set-structured data using the Transformer architecture (see section 3.2 above). As a set is defined solely by its elements without any ordering, the calculation needs to be permutation invariant: it needs to yield the same result irrespective of ordering (numbering) of input elements. The set approach is highly relevant to clustering, as cluster is a subset of the whole dataset.

The architecture achieves permutation invariance in the following steps:

1. Input vectors are encoded first using the Transformer encoder *without* any positional information (see section 3.2.5 above). For such Transformer it holds that permuting its inputs permutes correspondingly its outputs, but does not otherwise change them (*permutation equivariance*).
2. Outputs of step 1 are pooled together using a permutation invariant operation. The result of the operation is thus the same irrespective of permuting outputs from step 1, or permuting inputs to the model.
3. Set Transformers finally "decode" the pooling result in a desired output according to the nature of the task. (This decoding is of a different kind than the decoding addressed in the encoder-decoder part of the *Attention is all you need* paper, see [42] and subsection 3.2.2 above.)

Authors use the abbreviation:

- SAB (Set Attention Block) to denote a plain multi-head attention layer with a single input matrix (see subsection 3.2.4 above);
- PMA (Pooling by Multihead Attention Layer) to denote a particular permutation invariant operation that pools together inputs to the PMA layer (more on this below);
- rFF to denote fully connected layer(s) that accepts a matrix as its input and processes each of its rows *independently and identically* [23]. Fully connected layers from section 3.2.2 satisfy this property.

The architecture transforms the input N into the output O according to the following scheme:

$$\mathrm{N}_{\ell\times d} \to \mathrm{SAB}_{\ell\times d} \to \mathrm{SAB}_{\ell\times d} \to \mathrm{PMA}_{k\times d} \to \mathrm{SAB}_{k\times d} \to \mathrm{FFr}_{k\times d} = \mathrm{O}_{k\times d}$$

where lower indices indicate dimension of the matrix at the output of each block (ℓ denotes the number of input vectors, d their dimension and k the

fixed hyperparameter of the PMA layer). Naturally, SAB operations, both to the left and to the right of PMA can be each repeated n_1 and n_2 times (here: $n_1 = 2$, $n_2 = 1$).

We have dealt with attention layers accepting a single input matrix so far (the matrix was then multiplied by one or more triplets of trainable matrices W_Q, W_K, W_V to produce one or more triplets of Q, K, V). The *Set Transformer* paper employs a more general architecture with two separate inputs: a query input N_Q and key/value input $N_K = N_V$. Both inputs can differ both in sequence length and in vector dimension:

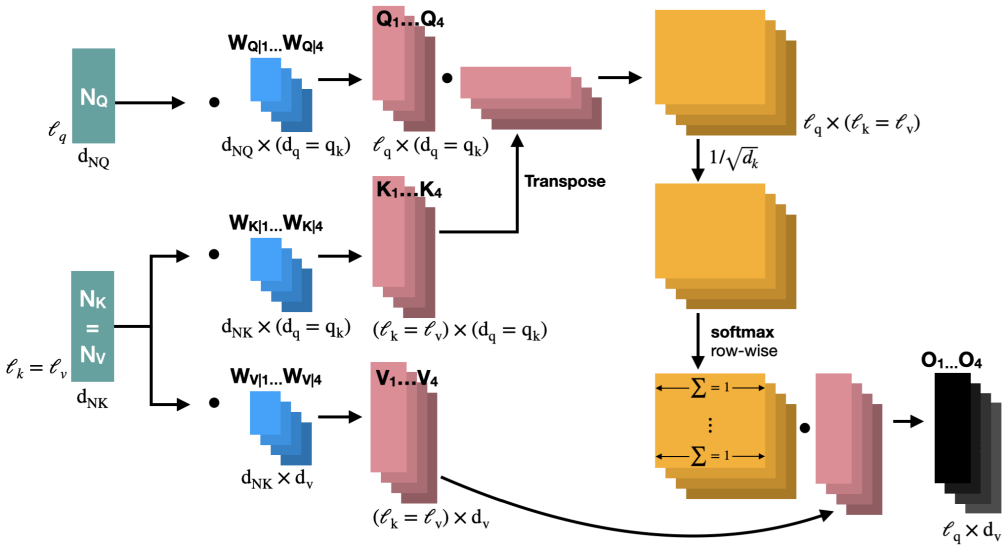


Figure 3.6: Generalized Multi-Head Attention with Two Inputs.

Whereas every input vector "looks at" every input vector in the original architecture (as addressed in subsections 3.2.3 and 3.2.4 above), we can have now query–input vectors that look at key/value–input vectors (but not vice versa and not at themselves). One can even consider an architecture with three distinct inputs (query–input, key–input and value–input), provided that key–input and value–input have the same sequence length, but we can see no use for it with regard to this thesis.

This two-inputs architecture enables to define the PMA operation. Let us denote by:

- MAB(X,Y) the generalized two-inputs attention module (it holds that SAB(X) = MAB(X,X)); and
- $Z \in \mathbb{R}^{\ell \times d}$ the input to PMA, which results from the previous SAB layer(s).

The task of PMA is to pool ℓ input vectors from $\mathbb{R}^d$ (lines of Z) into k vectors from $\mathbb{R}^d$, where $k \ll \ell$ (k, the fixed hyperparameter of PMA, is much smaller

than ℓ, in some cases just 1). To this aim, PMA includes k trainable inducing vectors from $\mathbb{R}^d$ (forming a trainable matrix $\mathrm{S} \in \mathbb{R}^{k \times d}$), and each of them "looks at" every input vector (row of Z). Accordingly, $\mathrm{PMA}(\mathrm{Z}) = \mathrm{MAB}(\mathrm{S}, \mathrm{Z})$.

Moreover, the two-inputs architecture enables to define ISAB. It overcomes the disadvantage that although the SAB can, in theory, accept any input of length ℓ, its everyone-looks-at-everyone architecture has $\mathcal{O}(\ell^2)$ time and memory complexity. Accordingly, SAB calculation for large inputs (e.g., large sets of vectors to be clustered) can get intractable.

Let us denote the two-input generalized attention module as MAB(X,Y), again, and consider that ISAB includes (unlike SAB) m trainable inducing vectors from $\mathbb{R}^d$, forming a trainable matrix $\mathrm{T} \in \mathbb{R}^{m \times d}$, where $m \ll \ell$. Then, given its input $\mathrm{N} \in \mathbb{R}^{\ell \times d}$, ISAB is defined as:

$$\mathrm{ISAB}(\mathrm{N}) = \mathrm{MAB}(\mathrm{N},\ \mathrm{MAB}(\mathrm{T}, \mathrm{N}))$$

The inner MAB does the computation of complexity $\mathcal{O}(m\ell)$ (each of m lines of matrix T "looks at" every of ℓ lines of matrix N) and outputs the matrix $\mathrm{H} \in \mathbb{R}^{m \times d}$. The outer MAB does the computation of complexity $\mathcal{O}(\ell m)$ (each of ℓ lines of matrix N "looks at" every of m lines of matrix H) and outputs the result $\in \mathbb{R}^{\ell \times d}$. Accordingly, ISAB computation, going through the m bottleneck (where m is a fixed hyper-parameter), has $\mathcal{O}(\ell)$ time and memory complexity. When ISABs replace SABs, the neural network can otherwise remain the same:

$$\mathrm{N}_{\ell \times d} \to \mathrm{ISAB}_{\ell \times d} \to \mathrm{ISAB}_{\ell \times d} \to \mathrm{PMA}_{k \times d} \to \mathrm{ISAB}_{k \times d} \to \mathrm{FFr}_{k \times d} = \mathrm{O}_{k \times d}$$

Chapter 4

Clustering: Classical and Neural Network Approaches

Clustering is a standard non-supervised machine learning method used to partition input datapoints, each from $\in \mathbb{R}^n$, into subsets (clusters, groups) where each subset includes mutually similar items (datapoints that are in some metrics sufficiently close to each other). Such datapoints can be, for instance, vectors that represent sentences, as described in chapter 1 (then, finding clusters of similar vectors results in finding similar sentences within the dataset).

We can distinguish:

- **Classical clustering methods**, which make no use of neural networks (see, e.g., `scikit-learn.org/stable/modules/clustering.html` for overview of various classical methods). Out of them, we choose K-Means and DBSCAN, both standard algorithms for topic detection (see [19]). K-Means is the prototypical and simplest clustering algorithm, a baseline serving as a good illustration of clustering in general. DBSCAN is a newer classical algorithm published in 1996 (see [15]) that has some favorable properties and has been given the test of time award by ACM in 2014 (see also [35]).

- **Clustering methods using neural networks** (also called *deep clustering*). Literally dozens of papers have been published in the last ten years that propose various variants of neural network clustering. A good overview is available at `github.com/zhoushengisnoob/DeepClustering`. We focus in this thesis on deep amortized clustering, as published in [24], which applies the Set Transformer architecture (see section 3.3 above).

4.1 K-Means

The K-Means algorithm receives as its input **(a)** p datapoints (each $\in \mathbb{R}^n$) and **(b)** the specific number $k \in \mathbb{N}$ of clusters to be found. Thereupon, the algorithm randomly initializes k centroids, each $\in \mathbb{R}^n$, and repeats the 2 following steps:

1. assign each datapoint to that centroid which is closest to it (usually according to Euclidian distance, but other metrics can be used, or even a kernel trick can be used and the distance of two points measured as if they were remapped into a higher-dimensional space)

2. recalulate the centroid position so that it lies in the center of datapoints assigned to it (see step 1); that is, the centroid moves to the position in $\mathbb{R}^n$ that is the average of $\mathbb{R}^n$ values of assigned datapoints

until the assignment of datapoints remains stable (that is, until the assignment of any datapoint to its closest centroid does not change in step 1) or until a pre-determined number of iterations is reached.

When the algorithm terminates, we get k clusters (each cluster comprising of datapoints that are closest to a particular centroid), as is illustrated in the following figure (where datapoints are from $\mathbb{R}^2$ and k has been set to 3):

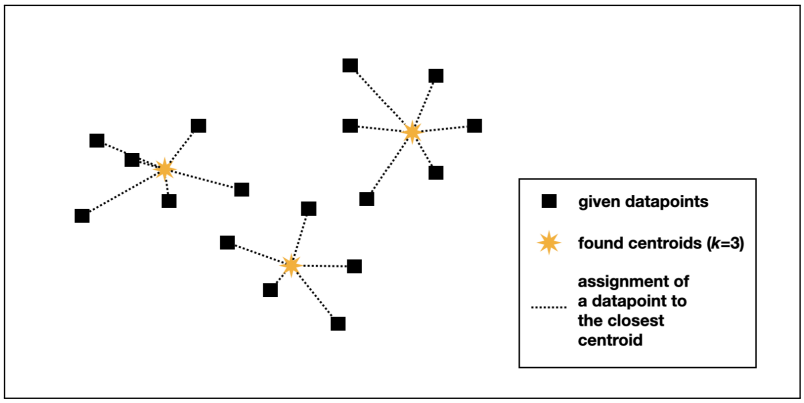


Figure 4.1: K-Means Clustering

Finding exact positions of centroids that minimize the sum over all datapoints of the datapoint distance to its closest centroid is an NP-hard problem. The basic K-Means algorithm, proceeding greedily as described above, can end up in a local minimum that can be, in theory, arbitrarily far from the global minimum. However, the variant based on specific initialization of centroids, named K-Means++, gives a certain guarantee and renders better applied results [2].

While the aim of this introduction is not to delve into details of the algorithm, we need to mention here its two major disadvantages (inherent both to K-Means and K-Means++) that can considerably deteriorate the quality of returned clusters, a contrast to algorithms specified later in this thesis:

1. The number k of clusters to be found is a hyperparameter, an input to the algorithm. The algorithm is unable to make an inference based on mutual positions of datapoints in the $\mathbb{R}^n$ space, but is forced to find k clusters irrespective of their true number.

2. The algorithm assumes clusters whose shape approximately resembles a circle (ball, hyper-ball); this results form the rule that each datapoint is assigned to the closest centroid (see step 1 above). In many real-world applications, however, the clusters or their parts may be oval or bent (e.g. "banana-shaped").

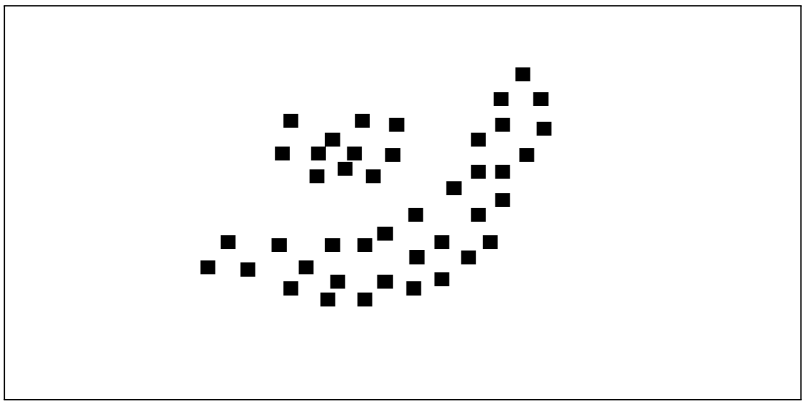


Figure 4.2: Clusters Unsuitable for K-Means

4.2 DBSCAN

The DBSCAN algorithm (where DBSCAN stands for *density-based spatial clustering of applications with noise*) approaches clustering diferently. It discovers the number of clusters from data and can find clusters of irregular shapes. It is based on the insight that a cluster is a denser cloud of points separated from other clouds by less dense areas. Accordingly, hyperparameters are:

1. ε — a maximum distance (e.g. Euclidian or cosine) between two datapoints that makes them directly connected within the same cluster; and

2. p — the minimum number of datapoints that need to be within the same ε-ball in order to represent *core points* of the same cluster.

Every input datapoint is assigned into exactly one of the three following categories:

1. core point — is a point that has in its ε-neighborhood at least $p-1$ other points (or, in other words, is one of at least p points within some ε-ball)
2. fringe point — is a point that has in its ε-neighborhood less than $p-1$ other points but has there at least one core point
3. noise point — is a point that is neither a core point, nor a fringe point.

In the following figure, each of core points B, I, N or O has at least 3 neighbors in its ε-neighborhood (as p is set to 4 here, any 4 points within the same ε-ball are core points). On the other hand, each of fringe points A, J, H or L has less points in its ε-neighborhood, but still at least one core point. The noise point Q has another point in its ε-neighborhood, but only a fringe point L.

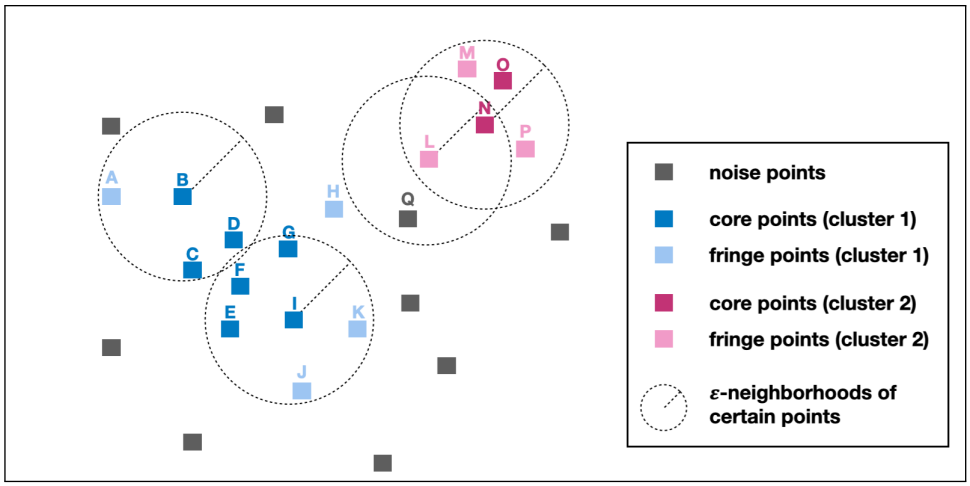


Figure 4.3: DBSCAN Clustering: ε indicated in the figure, $p = 4$

Whereas K-Means is forced to assign every point to one of clusters, DBSCAN assumes that some points (i.e., noise points) are not a good fit in any cluster. If, however, it was required to assign every point, noise points could be assigned in the second pass (e.g., according to closest core or fringe point).

4.3 Deep Amortized Clustering

Deep Amortized Clustering, as a neural network method, needs to be trained to provide desirable clusters. Neural methods are not based on a fixed clustering strategy, but attempt at learning the clustering algorithm or, at least, the distance metrics most appropriate for points in the dataset. The Deep Amortized Clustering approach [24] extends Set Transformers [23], explained in detail in section 3.3 above. Whereas the clustering mechanism proposed in the Set Transformer paper [23] required a given (fixed) number of clusters, deep amortized clustering does not depend on such hyperparameter, inferring one cluster after another while there remain unclustered datapoints. There are 2×2 basic sub-variants of deep amortized clustering:

- **Anchored filtering or minimum loss filtering.** The anchored filtering algorithm is given an anchor, a yet unclustered datapoint, to find the cluster that includes that anchor. On the other hand, the minimum loss filtering approach does not receive such anchor, but decides in each iteration by itself what cluster to return.
- **Parametric distribution assumed or not assumed.** The loss function for deep amortized clustering can, but does not need to, incorporate the assumption that all clusters have been generated by the same parametric family of distributions (such as $\mathcal{N}$ with $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ different from cluster to cluster).

In this thesis, we focus on anchored filtering without assuming a parametric distribution. The reason is that anchored filtering *"is beneficial for harder datasets"* ([24], section 3.1), and it allows to choose a particular datapoint (e.g., representing a particular tweet) and request the model to find similar ones (those that constitute the same cluster). Whereas assuming a family of parametric distributions can increase accuracy for certain toy datasets generated by the family [24], it is questionable that clusters in real-world datasets (such as collected tweets) conform to a particular shape.

The anchored filtering algorithm gets ℓ yet-unclustered datapoints from $\mathbb{R}^d$ forming the matrix $\mathrm{X} \in \mathbb{R}^{\ell \times d}$. Moreover, the algorithm gets the index number $a \in \{1, ..., \ell\}$, either sampled at random or chosen manually, of the input vector (which is the a-th row of X) that will serve as the anchor in this iteration. Given these inputs, the algorithm finds the cluster as follows:

1. First, the X matrix is encoded by ISAB (defined in section 3.3 above), or more such stacked blocks, into $\mathrm{H_X} = \mathrm{ISAB(X)} \in \mathbb{R}^{\ell \times d}$. The a-th row of $\mathrm{H_X}$ is denoted as $\mathbf{h}_a \in \mathbb{R}^d$. It equals the anchor vector (the a-th row of X) after its was transformed, as a part of X, by the ISAB operation(s).

2. $\mathrm{H}_{\mathrm{X}|a} = \mathrm{MAB}(\mathrm{H}_{\mathrm{X}}, \mathbf{h}_a) \in \mathbb{R}^{\ell \times d}$ is then computed. This MAB operation (defined in section 3.3 above) enables each encoded vector from step 1 (each row of the H_{X} matrix) to interact with the anchor vector, as encoded, and incorporate information regarding mutual relation.

3. $\mathrm{H}_{\mathrm{X}|a}$ is then compressed into a single row vector $\mathrm{H}_{\mathrm{c}} \in \mathbb{R}^{1 \times d}$ by being fed into the PMA pooling operation (defined in section 3.3 above); this PMA has one inducing vector only ($k = 1$), but experimenting with a higher k is possible:

$$\mathrm{H}_{\mathrm{c}} = \mathrm{PMA}(\mathrm{H}_{\mathrm{X}|a}) \in \mathbb{R}^{1 \times d}$$

4. Every vector from $\mathrm{H}_{\mathrm{X}|a}$ (step 2) then "looks at" the compressed information in H_{c} (step 3) in another MAB operation:

$$\mathrm{H}_{\mathrm{M}} = \mathrm{MAB}(\mathrm{H}_{\mathrm{X}|a}, \mathrm{H}_{\mathrm{c}}) \in \mathbb{R}^{\ell \times d}$$

5. This H_{M} is then run through: ISAB, or more such stacked blocks; rFF layer (a fully connected layer $d \times 1$ that operates on each row separately, see section 3.3 above); and sigmoid function σ. The computation results in $\mathbf{m}$, a vector with a mask value for every of ℓ input datapoints:

$$\mathrm{H}_{\mathrm{M}\ \ell \times d} \to \mathrm{ISAB}_{\ \ell \times d} \to \mathrm{rFF}_{\ \ell \times 1} \to \sigma_{\ \ell \times 1} = \mathbf{m} \in (0, 1)^{\ell}$$

The i-th element of vector $\mathbf{m}$, a value between 0 and 1, is a prediction from the model whether the datapoint i (i-th row of X) should be included in the cluster together with the anchor a. A common threshold is to classify the datapoint i as a member of the cluster iff $\mathbf{m}_i > 0.5$.

Upon finding the new cluster, rows representing the now-clustered datapoints can be taken out of X, producing the new matrix $\mathrm{X}_2 \in \mathbb{R}^{\ell_2 \times d}$, where ℓ_2 is smaller than ℓ by the number of now-clustered datapoints. The algorithm can thereafter successively process $\mathrm{X}_2, \mathrm{X}_3, \cdots$ matrices, with ever smaller first dimension, until all datapoints are clustered.

During training, the model gets a training batch with ℓ datapoints and a randomly sampled anchor index $a \in \{1, ..., \ell\}$, and it calculates the mask vector $\mathbf{m} \in (0, 1)^{\ell}$. To compute the loss, vector $\mathbf{m}$ is compared, according to the binary cross-entropy criterion, against the "true mask" vector $\mathbf{t} \in \mathbb{R}^{\ell}$ such that $\forall i \in \{1, ..., \ell\}$ $\mathbf{t}_i = 1$ if the datapoint i truly is in the same cluster as the anchor, otherwise $\mathbf{t}_i = 0$:

$$\mathcal{L} = -\frac{1}{\ell} \sum_{i=1}^{\ell} \left(\mathbf{t}_i \log \mathbf{m}_i + (1 - \mathbf{t}_i) \log \left(1 - \mathbf{m}_i\right) \right)$$

Part II

Application to Twitter Data

Chapter 5

Introduction. Representing Tweets by Vectors

Methods addressed in this part II of this thesis are applied to the dataset of 203k tweetsf. The task is to convert the tweets to high-quality sentence vector representations, create an annotation tool providing various semantic search capabilities, use the tool to annotate several dozens of clusters of topic-related tweets, and train a model on these annotations that would be able to find meaningful clusters by itself.

The access to the computational infrastructure of the OP VVV funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics" is gratefully acknowledged. The access involved multiple CPU cores for experiments with classical clustering methods and high-end GPUs, NVIDIA A100 and V100, for fast computing vector representations and training multiple variants of Deep Amortized Clustering models.

The annotation tool, too, benefits from an access tof the GPU, which usually reduces the time needed to respond to the semantic query from 7 seconds to less than a second, provided that vectors representing the dataset can safely fit into the GPU memory. As the current representation needs 415 MB memory only (203k tweets, 512 dimensions each, 4 bytes per dimension), the annotation tool is available to run fast even on low-end GPUs.

5.1 Twitter Dataset

The dataset of 203,057 documents was provided by Jan Drchal, who scraped tweets published by selected Czech users and reactions to those tweets. The source is one pandas pickle file, with tweet texts and various related metadata.

The key metainformation is the twitter ID, a `uint64` value that uniquely identifies each tweet ever published. The twitter ID can be used as a key when joining data stored in multiple tables regarding the same tweet.

More than 96% of tweets in the dataset were published after the beginning of 2015; the oldest tweet is from 12 Feb 2009, the newest tweet from 7 March 2022. Almost all tweets are in Czech, with rare exceptions of tweets in English. The advantage of tweet-length documents is they mostly fit in the limit of 512 tokens set by most BERT models (see section 2.5). Out of 203,057 tweets in the dataset, only 548 exceed the limit.

5.2 Creating Vector Representations

The tweets were converted to vectors according to results according to the methodology published in the FacTeR-Check paper [27]. Its authors take 4 multilingual Sentence Transformer models (see section 2.6) freely available at `www.sbert.net` (paraphrase-xlm-r-multilingual-v1, stsb-xlm-r-multilingual, paraphrase-multilingual-MiniLM-L12-v2 and paraphrase-multilingual-mpnet-base-v2) and finetune them further using the multilingual STS dataset. The STS dataset consists of 5749 train, 1500 dev and 1379 test pairs of sentences, labeled from 0 to 5 according to the similarity of sentences in the pair. Authors created the multilingual STS dataset by automatically translating STS, mostly with Google Translate, in 14 other languages, including Czech. The 4 resulting models are available at `huggingface.co/AIDA-UPM`.

Following the methodology proposed in [27], we have converted every of 203,057 tweets into 4 different vectors using each of the 4 models, with appropriate tokenizers. Every model returns a vector with 768 dimensions per input, only paraphrase-multilingual-MiniLM-L12-v2 outputs have 384 dimensions; thus, the 4 representations of the same tweet have $2688 = 3 \times 768 + 384$ dimensions in total. For each tweet, its four representations were concatenated together, producing a matrix $\mathrm{M} \in \mathbb{R}^{203057 \times 2688}$. Thereafter, dimensionality was reduced from 2688 to 512 using the principal component analysis (PCA), and all vectors were normalized (so that cosine similarity of these vectors equals their dot product: $\mathrm{cossim}(\mathbf{v}_1, \mathbf{v}_2) = \frac{\mathbf{v_1}^\mathsf{T}\mathbf{v_2}}{\|\mathbf{v_1}\|_2 \, \|\mathbf{v_2}\|_2} = \frac{\mathbf{v_1}^\mathsf{T}\mathbf{v_2}}{1 \cdot 1} = \mathbf{v_1}^\mathsf{T}\mathbf{v_2}$), producing the final matrix $\mathrm{F} \in \mathbb{R}^{203057 \times 512}$.

This pre-computed $\mathrm{F} \in \mathbb{R}^{203057 \times 512}$, representing the whole dataset, is then loaded by the subsequent tasks: the annotation tool and clustering scripts. We observed, without quantifying the effect, that the annotation tool started to respond more accurately to semantic search queries after we put in use vectors computed by ensembling described above.

Chapter 6

Semantic Search and Annotations

6.1 Semantic Search and Annotation Tool

We have created an annotation tool that loads, when started, the twitter dataset and vector representations R (see chapter 5) and provides the syntactic and various semantic search features. The primary use case is definition of a cluster of tweets concerning the same topic (e.g. all tweets related to vaccination). The search can be done incrementally (the user can add further tweets to the cluster by running further syntactic and semantic searches), and the tool enables manual labeling, too (the user can exclude the found tweet from the cluster she is defining).

1. The **frontend** part is an asynchronous single-page web applicaton (SPA) that communicates with the backend via `fetch`. It is written in a clean and modularized TypeScript code; we have largely refactored the first version of the application that was in pure JavaScript. Although graphical elements (e.g. buttons and tooltips) resemble the Bootstrap style, no external .css library is used (whereas the first version of the application depended on Bootstrap).

   The frontend part is the main driver of the user experience, gathering inputs from the user, requesting computations from the backend and displaying results in a user-friendly manner. We describe user interface and functionality in section 6.2 below.

2. The **backend** part is an API that is driven by requests from the frontend. To illustrate, when the frontend reqests a search for tweets similar to the given ones (whose twitter IDs are transferred in the request), the

backend runs requested NLP computations (usually on the GPU) and responds to the frontend with results (here: IDs and texts of similar tweets). The backend is coded in Python and depends on Flask library for server functionality, PyTorch library for NLP computations, and Plotly library for visualisation of scatterplot and timeline (these two kinds of visualisations are the only ones where user experience is generated by the backend).

6.2 Workflow and User Interface

The user begins the session by entering one or more syntactic searches; after each search (whether syntactic, COS, LDA, NN1 or NN2), the user can exclude any found tweet as irrelevant to the topic; non-excluded tweets from multiple searches add together. After the user has found some non-excluded tweets, she can apply the COS semantic query: the tool returns 10 tweets with smallest mean cosine distance to tweets included so far. If the user has both included some and excluded other tweets, she can apply further semantic search methods, LDA, NN1 and NN2, which make use of the contrast between included and excluded tweets (explained below). At the end of the session, the user has, and can save to the server, the group of included tweets and labeled excluded tweets.

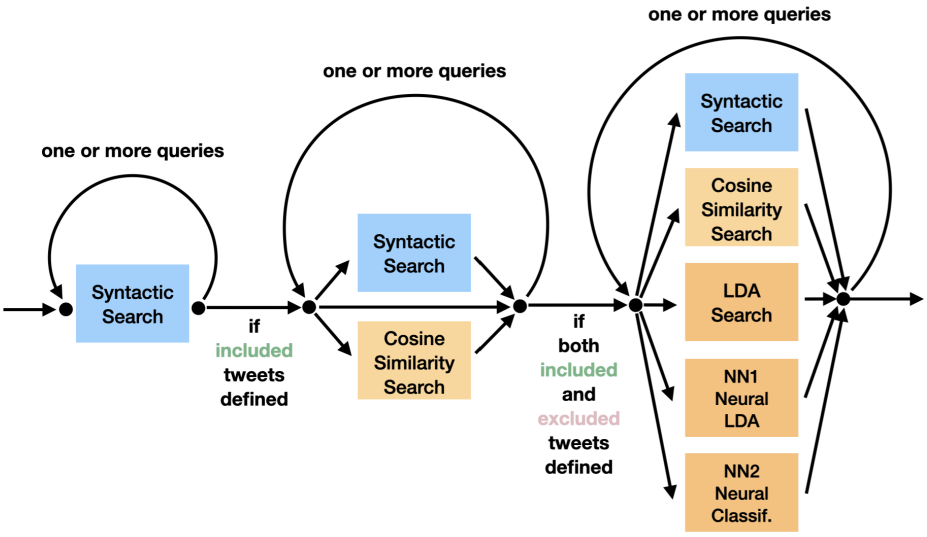


Figure 6.1: Annotation Tool Workflow

We address below key elements of user interface and workflow and then describe the provided semantic search options.

Syntactic search and its results. After the user has entered a syntactic search, tweets conforming to the pattern are listed, all having green background at first, i.e., being included by default in the group.

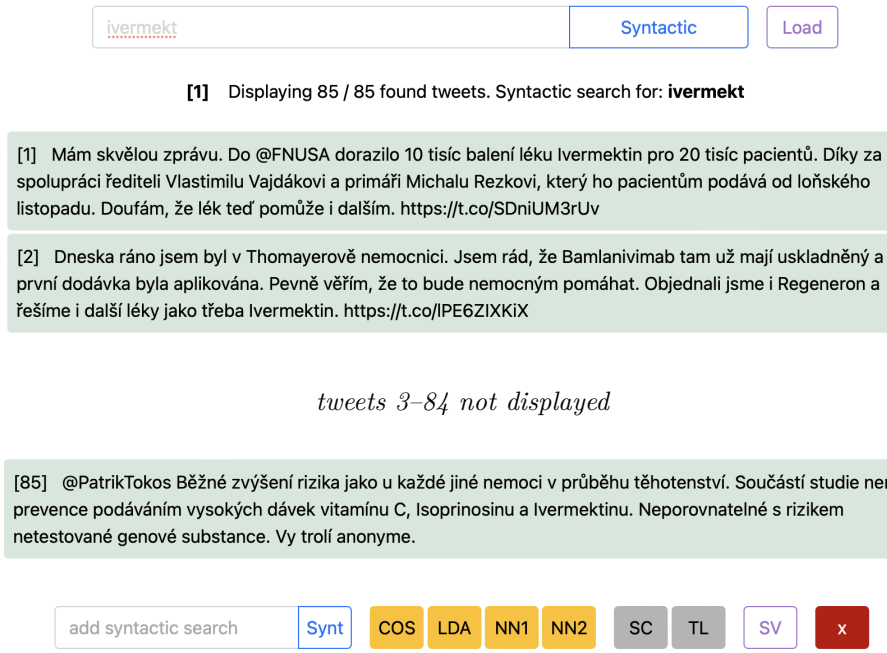
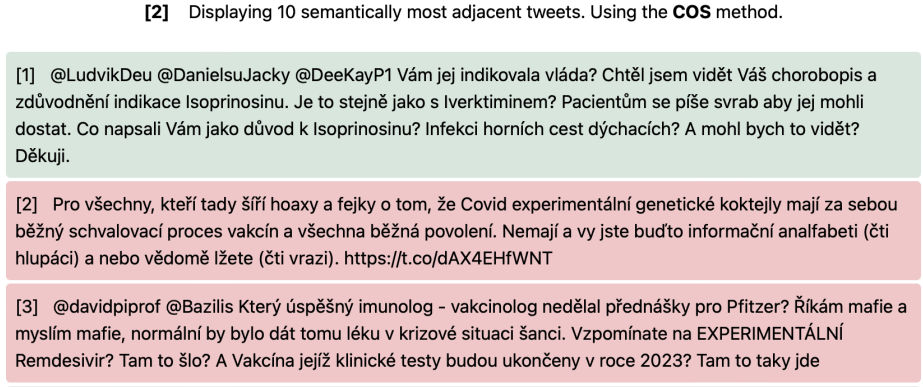


Figure 6.2: Search Results

Additional search and excluding tweets. The user has decided to use the COS query now. The query displays 10 tweets most semantically similar to those included so far (green tweets). The number 10 is chosen to allow the user to label all tweets at once without feeling discomfort; if the user is willing to see more than 10 COS-similar tweets, he or she can repeat the COS search. Furthermore, the user has decided, after seeing the results of the COS query, to exclude some tweets as irrelevant to the group (she is interested in tweets related to ivermectin and isoprinosin, medicines believed by some to be effective against Covid, but not in vaccination in general).



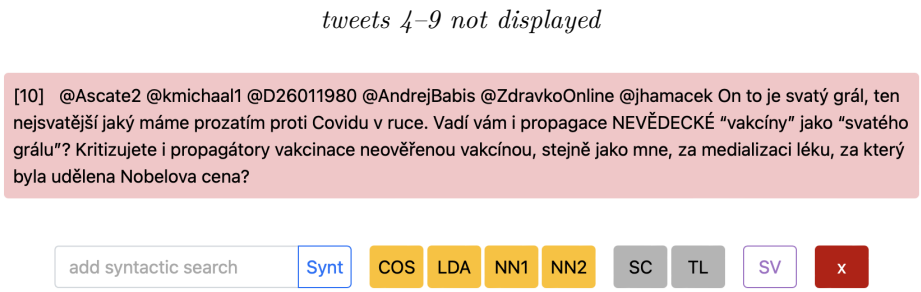


Figure 6.3: Annotation App: Manual (Re)labeling

Additional search(es). After defining enough positive (green) and negative (red) labels, it may be a good time to try NN1 or NN2 methods. Whereas COS aims only at similarity to green tweets, LDA, NN1 and NN2 consider also user's negative choice, both similarity to green tweets and dissimilarity to the red ones.

- **LDA** variant is based on the linear discriminant analysis. It computes a single vector $\mathbf{v}$ from the $\mathbb{R}^{512}$ space so that vectors for included tweets (green tweets) and vectors for excluded tweets (red tweets) are best separable (most distinct) after being projected on $\mathbf{v}$. More specifically, LDA finds the vector $\mathbf{v}$ maximizing the criterion $\frac{(\mu_g-\mu_r)^2}{\sigma_g^2+\sigma_r^2}$, where μ and σ^2 denote the average and variance of green projections and red projections. (A vector from $\mathbb{R}^{512}$ is, by its projection on $\mathbf{v} \in \mathbb{R}^{512}$, reduced to a single number from $\mathbb{R}$.)

  Scikit Learn is used for the task. Accordingly, the computation is made on the CPU, uses eigenvectors and takes more than 6 seconds on the given dataset (the NN1 method, addressed below, provides similar results, but much faster). The frontend then presents to the user 10 tweets classified as the most green-like by the lda.fit method (its operation is described in the following paragraph).

- **NN1** variant is based on the linear discriminant analysis again. This time, however, the searched vector $\mathbf{v}$ coincides with parameters of a neural network that is iteratively trained (in less than a second) to maximize the criterion $\frac{|\mu_g-\mu_r|}{\sigma_g+\sigma_r}$. After the vector $\mathbf{v}$ is found (as an approximation, but much faster), vectors that represent all unclassified tweets (yellow tweets) are projected on $\mathbf{v}$, and thus reduced to certain points on the $\mathbb{R}$ axis. Yellow tweets reduced to the 10 leftmost (if $\mu_g < \mu_r$) or the 10 rightmost (if $\mu_g > \mu_r$) points are presented to user.

- **NN2** This variant uses a neural network that is trained (in less than a second) to classify all labeled tweets to match given labels. Thereafter, the network is used to classify all tweets unlabeled yet (that is, yellow

tweets). The user is presented 10 tweets with the highest softmax value for the green class.

History and save. If the user is not satisfied with the last search(es) she can backtrack (possibly more times) by pressing the red-cross button. If the user is satisfied with the cluster defined so far (usually after trying semantic searches by NN1 and NN2 methods that do not display additional tweets belonging to the cluster) she can save them (SV button): the cluster name entered by the user in a modal window, twitter IDs of positive (green) tweets and twitter IDs of negative (red) tweets are sent to the backend and stored at the server.

Timeline. The user can display a timeline (as an interactive Plotly graph with zooming and panning features) regarding included (green) tweets by clicking on the TL button (the code from Jan Drchal is used for this visualisation). The x-axis corresponds to dates the tweets were published, and the y-axis show number of tweets at particular dates. Colors distinguish tweet authors, and upon hovering over a colored box, full details concerning the tweet pop up, including its text.

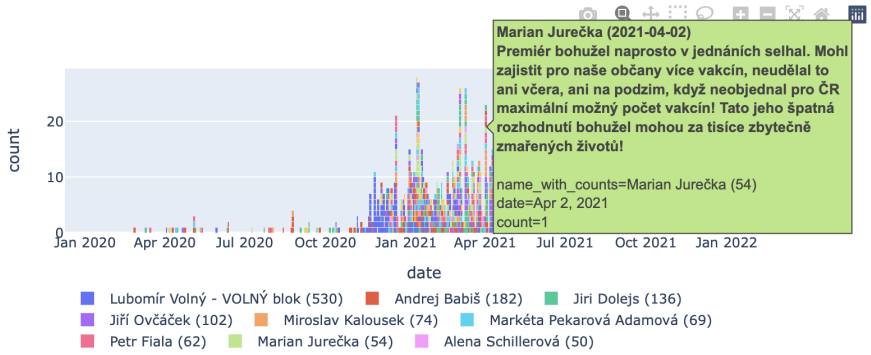


Figure 6.4: Timeline Output

Scatterplot. The user can display a scatterplot (as an interactive Plotly graph) regarding all tweets 203,057 tweets from the dataset. All the tweets (their vector representations) are projected on a 2D plane, two vectors resulting from the 3-class linear discriminant analysis. The figure corresponds to this plane (and its x and y axes to the two vectors found by LDA). Every green, red and yellow dot represents an included, excluded or unlabeled tweet, as projected. Again, the user can hover over any dot and see the full tweet text. This enables to easily see changing semantic nature of unlabeled tweets: they get similar to included tweets as hover over dots closer to the green clump, similar to excluded tweets as we move towards the red clump and unrelated as we move towards the center of the yellow clump.

Figure 6.5: Scatterplot Output

6.3 Annotating Groups of Tweets

Using the annotation tool, we defined 65 groups, mostly regarding topics that have raised wider public interest. See Appendix B. for more information regarding the 65 groups. In our experience, it was rather easy to come with interesting topics at the beginning, but as the number of annotated clusters increased, thinking about another topic got more difficult. Accordingly, we got inspired by key lemmas of K-Means clusters (see section 7.2 below) when defining last 20 groups (46-65).

Although 65 clusters may not seem as much, the result required our review of 18,371 tweets in total. We labeled 14,068 times a tweet as included in one of 65 groups (average group size 216); if, however, we exclude the 3 largest groups (refugees, migrants: 2712, vaccination: 1682, respirators: 1004), the average drops to 139. We labeled 4,303 tweets as excluded (not relevant for the topic defined).

We preferred reasonably broad topics, also because the advantage of the semantic over syntactic approach may be rather small as regards exceedingly narrow topics (those with just a few tweets). Nevertheless, we could not objectively decide, e.g., whether to define a wider cluster regarding vaccination in general (more than 1000 tweets) or, as in section 6.2 above, a narrower cluster concerning ivermectin and isoprinosin only (which already has more than 100 tweets).

Clearly, defining a topic (as a superset or subset) is somewhat arbitrary,

and often it comes as no surprise that a particular tweet (e.g. *Employees at Škoda have received a bonus.*) may be annotated in one group (all tweets concerning Škoda), but clustered into another (with other tweets concerning bonuses). This ambiguity results in lower scores achieved by automatic clustering methods (if evaluated on real-world data) even if they manage to find reasonable clusters.

Chapter 7

Tweet Clustering

7.1 K-Means: Quantitative Analysis

We clustered 203,057 tweets in the dataset with the K-Means algorithm (subsection 4.1), trying different cluster numbers (1,000, 2,000, ..., 20,000). The computation was made feasible by K-Means with minibatches [37]. Even then, clustering of 203,057 tweets into 12,000 clusters required 28.75 minutes on 8 cores of the Intel Xeon 6146 CPU. K-Means algorithm managed to find well-varied clusters of reasonable sizes.

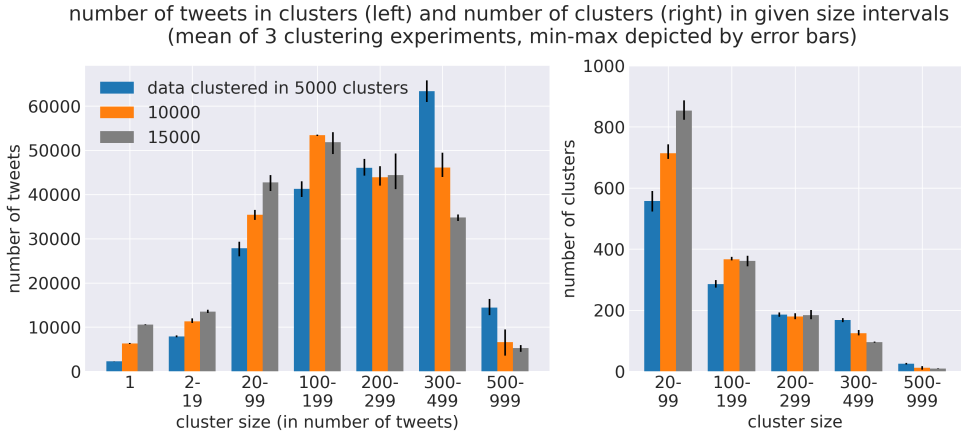


Figure 7.1: Distribution of Tweets and Clusters according to Size Interval

This result is largely superior to the DBSCAN algorithm (see subsection 4.2), which clustered most tweets in either one lump cluster (of more than

100,000 items) or one of many tiny clusters (of a single-digit size), in spite of grid search for suitable parameters.

We further compared K-Means clusters against first 45 manually annotated groups (see section 6.3). Usual methods such as NMI or Rand index are not applicable to comparison between clustering of the full dataset (resulting from K-Means) and its small subset (covered by manual annotations); these methods measure correspondence of two complete clusterings of the same set. We measured the Jaccard index, also known as intersection over union (IOU in short). When we denote the particular annotated group as set A and the particular K-Means cluster as set C, then $\mathrm{IOU}(A, C) = \frac{|A \cap C|}{|A \cup C|}$.

For each annotated group A from all annotated groups $\mathcal{A}$, we chose such K-Means cluster C from all clusters $\mathcal{C}$ that maximizes $\mathrm{I}(A, C)$ and then calculated the average a over all annotated groups: $a = \frac{1}{\mathcal{A}} \sum_{\mathrm{A} \in \mathcal{A}} \max_{\mathrm{C} \in \mathcal{C}} \mathrm{IOU}(\mathrm{A}, \mathrm{C})$, repeating the experiment 3 times. While we generally normalize vectors resulting from Sentence Transformers (see section 5.2 above) before using them further, experiments were run also on pre-normalization embeddings, without a notable difference.

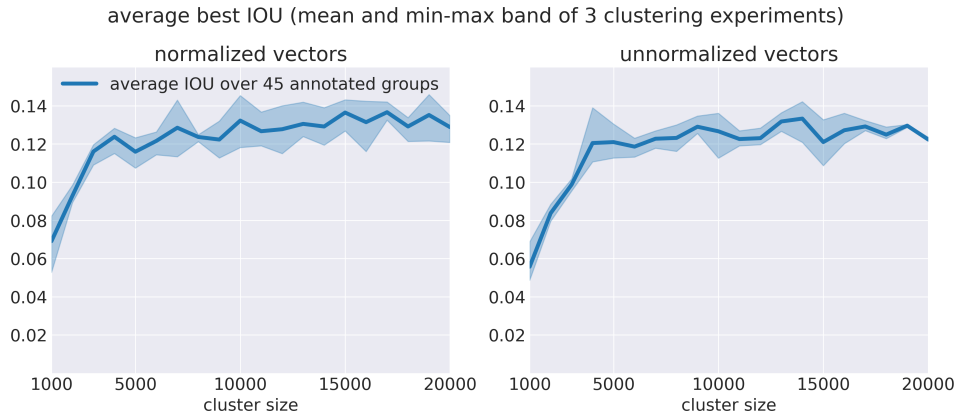


Figure 7.2: Overlap Between Annotated Groups and K-Means Clusters

Apparently, K-Means does not generate clusters resembling human-annotated groups (see also discussion in subsection 6.3). Nevertheless, the resulting clusters are mostly coherent (related to the same topic) and useful.

7.2 K-Means: List of Topics

Although the K-Means algorithm cannot be reasonably employed to find (almost) all tweets related to particular topics, it can still support the topic

discovery task. To find topics common to tweets within the same K-Means cluster, we lemmatized each of 203,057 tweets, i.e., converted each word in each tweet in its basic form; whereas a word may take just a few forms in English (e.g., *house* and *houses* or *write*, *writes*, *wrote* and *written*), the number of forms can easily exceed 10 in Czech, given its varied system of morphology. We used the MorphoDiTa (see [41]), in particular its API made publicly available by Institute of Formal and Applied Linguistics at `lindat.mff.cuni.cz/services/morphodita`.

We partitioned the dataset by K-Means in 15,000 clusters; then we selected all clusters (there were 251) with 30 to 100 tweets provided that their average length exceeded 140 characters (longer tweets usually have higher quality while exceedingly short tweets tend to be mere shouts). For each group, we found key lemmas, i.e., lemmas used at least 10 times within the group that are not one of 300 most frequent Czech words, see the Czech frequency dictionary at `wiki.korpus.cz/doku.php/seznamy:srovnavaci_seznamy`). We did not experiment with criteria weighting number of uses by general rareness of the lemma in the corpus.

The following table includes example of our results, based on syntactic similarity over semantically defined clusters. The table shows key lemmas translated in English for 8 randomly chosen groups. A larger table, concerning 100 randomly chosen groups not translated in English, is included in Appendix C. In our opinion, the table provides a reasonable overview of topics in the dataset, also because each cluster is characterized by multiple words. In fact, we used the table to find new topics to annotate 20 clusters in addition to 45 we annotated originally.

group id	**size**	**key lemmas**
72	60	bus driver union transport bus strike county
74	48	Easter feast Easter wishing
78	87	SPD CSSD *(political parties)* reject government
81	56	Britain Brexit Boris Johnson Great British mine
138	68	concert beneficial evening friend good
170	79	food my state agriculture quality
233	47	influence formation temperature anthropogenic
245	96	my can negotiations must

Table 7.1: Key Lemmas for Randomly Chosen Groups (translated)

7.3 Deep Amortized Clustering

We further applied the Deep Amortized Clustering method (section 4.3) specified in the assignment. Given our large and noisy dataset, we devised the following binary classification training approach (altering the training procedure specified in the [24] regarding the Omniglot dataset): the network needs to classify which points in the training bag are in the same cluster as the anchor point. Since we confirmed early that training on 65 annotated groups, with 40 groups in the training partition, results in serious overfitting, we proceeded in multiple steps:

1. **Pre-training.** We sampled 3 random center points (out of 203,057 in the dataset), finding for each center point 100 points closest to it (including itself). We defined the first group of 100 points as positive; the other 100 + 100 were negative, together with 400 noise points sampled at random from the whole dataset, completing the bag size of 800; the anchor point was sampled at random from positive points.

   Alternatively, we sampled a random center point and found $k = 3600$ nearest points; we defined the 300 points at the very center as positive and 500 hundred points at the fringe as negative (varying the k parameter can change the general distance of the fringe).

2. **Further Training.** We used clusters found by K-Means. We selected a random point from the whole dataset and 800 nearest point. The selected point and other points in the same K-Means cluster were positive, the other negative. The anchor point was sampled at random from all positive points.

3. **Fine-tuning.** We repeated the approach in step 2 to fine-tune the model on the annotated training group (with the only difference that the selection of the initial random point was limited to those included in any of annotated training groups).

In spite of dozens of variants, Deep Amortized Clustering results remained considerably below the K-Means baseline and the best average IOU (see section 7.1) below 8 %. In particular, fine-tuning on 40 annotated training clusters resulted in no significant improvement on annotated validation clusters. The following observations may be of interest:

- The model predicted annotated cluster membership with 69% accuracy: if given a certain central point belonging to yet-unseen annotated group and 800 closest points, the model predicted regarding each point whether it is in the same cluster as the central point with the 69% accuracy.

- The model predicted K-Means cluster membership with 76% accuracy. We did not partition K-Means clusters in training and validation subsets (as making predictions regarding K-Means cluster was not the final goal). Nevertheless, we used a small model (with reduction to 64 or 128 dimensions, 4 ISAB layers in total, each with 4 inducing points, amounting to 3.4MB of parameters) that could not memorize the dataset. This 76% accuracy transferred to the NMI (see [43]) 43.3 % (average of 10 experiments, standard deviation 1.9%), comparing the clustering produced by the model and by K-Means.
- The results indicate that the clustering task is more difficult than the classification. As regards Deep Amortized Clustering, a grave misprediction at one step can wrongly drag in many non-belonging datapoints, making them irrelevant (unreachable as anchor points) in any further step (clustering decision). Accordingly we consider a potential for augmenting the clustering procedure in section 7.5 below.

7.4 Conclusion

Having extensively researched semantic (vector) representation of text data and relevant neural network architectures, we experimented with multiple Sentence Transformers models and converted the provided dataset into vectors. The quality of these vectors proved during clustering and while working with the semantic search and annotation tool. We developed the tool and used it to create the training and validation dataset of 65 clusters by manually annotating 18k tweets.

The tool allowed us to demonstrate that low tens of datapoints (gathered at first by syntactic search methods) sufficently define the semantic core and allow to find semantically related tweets. For finer semantic control, negative samples can be used. In our opinion, the tool is an example of a pleasant and effective interaction with AI technology that does not automatize human work but augments it.

We demonstrate that reaching agreement between automatic clustering and manual annotations is rather demanding; one tweet usually relates to more topics and there are more reasonable annotations of the same dataset. Accordingly, the intersection-over-union metrics between our manual annotations and automatically found clusters was low in spite of high topical coherence of automatically generated clusters, apparent both visually and from the list of key lemmas we created automatically to discover further topics.

The K-Means algorithm proved to be an efficient clustering tool for the given task (if sufficiently high number of clusters was set). Other classical

clustering methods and the neural Deep Amortized Clustering method were less useful. Accordingly, we propose several potentially promising goals for future work and research.

7.5 Future Research Goals

- Working on a smaller dataset, presumably with 10k – 20k tweets, is likely to advance faster. Creating such smaller dataset by sampling its tweets at random from the full dataset would likely result in too many topics scattered over too many small groups. However, as the K-Means algorithm already partitioned the full dataset (203,057 items) in thousands of reasonable clusters, we can select at random these clusters and append them to the smaller dataset being built until its requested size is reached.
- A smaller dataset could be annotated (manually partitioned) in full, potentially by multiple annotators, which would enable to measure inter-annotator agreement. We hypothesize that, given multiple topics available in each tweet (see section 6.3), the agreement will be rather between annotators facing the similar set-up as clustering methods in this thesis (no prior list of topics is given and full non-overlapping partitioning is required).
- We appreciate that Amortized Clustering gives the flexibility of selecting anchor points: seed points from which particular cluster are generated. However, we believe that the current procedure (successive random sampling of anchor points and filtering out predicted points from the dataset until none is left) can be improved. If every datapoint in the dataset is tried as an anchor point first, the predictions could potentially be combined together into more accurate clustering, whether with exclusive or overlapping clusters.
- Naturally, other clustering methods can prove more suitable for the given task and the kind of dataset. We are currently considering methods for overlapping clustering and graph neural networks.

Appendices

Appendix A

Frequently Used Abbreviations

BERT	Bidirectional Encoder Representations from Transformers (NLP model)
CNN	Convolutional Neural Network(s)
NLP	Natural Language Processing
NN	Neural Network(s)
RNN	Recurrent Neural Network(s)

Appendix B

Annotated Groups

We originally annotated groups 1–45. After reviewing topics generated by K-Means and lemmatization (subsection 7.2, Appendix C.), we annotated groups 46–65. incl denotes number of tweets included in the group. excl denotes tweets we reviewed and labeled as excluded.

#	incl	excl	topic
1	90	138	presidential abolition or amnesty
2	58	30	lung ventilators
3	85	36	discount from public transport fares
4	1004	53	respirators
5	1682	13	vaccination
6	261	71	teacher wages
7	19	97	tax breaks for movie producers
8	2712	22	refugees, migrants (but not asylum)
9	594	75	Brexit
10	749	57	global climate and warming
11	111	44	the Novichok case
12	155	56	the Vrbětice case
13	150	61	subsidies for the Agrofert company
14	77	19	bark beetle
15	150	108	tornado at South Moravia
16	274	131	the Stork Nest case
17	83	57	the Becva poisoning case
18	100	91	reconstruction of D1 highway
19	124	43	isoprinosin a ivermectin
20	86	249	results at Olympic games
21	42	153	actual tennis resuls
22	42	153	human rights violations in China

23	138	0	inflation
24	492	26	budget deficit
25	103	58	the lithium case
26	69	62	*[similar topic as 3]*
27	750	65	declared state of emergency in Czechia
28	229	371	enlargement of nuclear energy production
29	89	0	solar energy ”barrons“ and ”tunnelling“
30	86	58	smart quarantine (COVID-related)
31	49	73	cryptocurrencies, Bitcoin
32	753	92	mandatory electronic record of sales
33	43	17	J. Nohavica (Czech musician)
34	60	90	the Danube-Odra-Labe river canal
35	56	33	5G networks
36	120	44	artificial intelligence
37	141	61	cyber-security and cyber-attacks
38	109	55	the new Building Act, amendment, changes
39	45	61	deferment of loan and mortgage payments
40	265	87	COVID compensation bonus for entrepreneurs
41	66	45	tougher criminal penalties for animal cruelty
42	23	25	the Charlie Hebdo case
43	362	25	bankruptcy of the OKD company
44	56	124	the criminal case of David Rath
45	140	81	coal mining and its reduction
46	138	42	M. Horáková (former Czechoslovak politician)
47	54	35	current financial rating of the Czech Republic
48	59	84	currency exchange-rate interventions
49	44	53	currency exchange offices
50	100	29	bio-fuels
51	74	70	appointment of professors and rectors
52	35	36	murder of J. Kuciak (Slovak journalist)
53	19	45	money laundering
54	36	49	industry 4.0
55	44	83	vocational training, dual education
56	38	67	inclusive schooling
57	39	53	mandatory high-school graduation in math
58	38	35	(electronic) highway vignettes
59	31	8	lower tax rate on draft beer
60	61	79	(dual) food quality
61	36	33	working conditions at Amazon
62	51	50	opening the embassy at Jerusalem
63	208	72	adopting Euro, joining the Eurozone
64	54	33	Senate (M. Vystrčil) trip to Taiwan
65	17	37	purchase of U.S. military helicopters
	14068	**4303**	

Appendix C

Key Lemmas for 100 K-Means Clusters

The 100 clusters were sampled at random from the set found by K-Means with all clusters with 30 to 100 tweets whose average length exceeds 140 character; 251 such clusters were found in the dataset. See section 7.2.

cluster id	size	key lemmas
2	81	Brusel můj evropský český chtít ala
5	82	republika prezident Zeman Miloš hrad pražský střed hodina jmenovat
6	52	důchod můj plat mluvit vláda
9	70	SPD můj hnutí Matteo Salvini
10	37	digitální danit firma platit řešení globální zdanění
14	83	volba volit jít moci můj kandidát
16	56	evropský premiér český zájem aba Babiš
17	77	můj Německo německý děkovat
20	57	doprava všechen železnice dálnice správa silnice ministerstvo
22	39	Maďarsko můj
24	35	můj svoboda připomínat památka
27	35	Ukrajina Ukraine
31	51	všechen můj děkovat zdravotní
32	66	český Český stát
37	52	Francie Franc EmmanuelMacron
39	84	2021 miliarda rozpočet 2020 vláda státní
40	35	zemřít čest velký
43	30	Německo Polsko Hitler
44	36	odbor mzda růst tripartita minimální news JMalacov via
47	40	můj ala budoucnost země

51	30	vrtulník nákup můj miliarda
53	78	svatý pražský mše Duka svatá kardinál katedrála Marie
56	80	děkovat můj slovenský všechen Slovensko poděkovat
59	45	obrana HDP můj výdaj NATO rozpočet
63	42	novela zákon CNB trh banka návrh
64	31	SPD děkovat pomáhat lid podpor podnět
65	44	mzda odbor minimální zvýšit koruna růst via min
67	59	německý Německo český spolupráce můj SRN
72	60	autobus řidič odbor doprava autobusový stávka kraj
73	48	odbor mzda růst procento pět via
74	48	Velikonoce všechen svátek velikonoční přát nadát
79	74	odbor tripartita růst Creative mzda Kasparova-Jan CSSD Hospodark
80	35	firma vláda testování muset ala moci
81	56	Británie brexit Boris Johnson Velká britský můj
83	97	lékař Company lék
84	30	miliarda obec státní růst kraj
87	39	republika prezident Miloš Zeman jmenovat škola návrh profesor vysoká
88	50	můj bůh ala modlit všechen
93	76	Francie francouzský islám muslim muslimský islámský útok
94	42	uhlí velký těžba zdroj
95	51	hod odbor tripartita sociální zasedání hospodářský strakovek Rada dohoda 2016
96	57	ruský vláda můj Rusek diplomat český
99	31	volba Slovensko Slovák můj
100	48	republika pražský prezident hrad Miloš Zeman přijmout žádost
102	42	Irák Isa
105	47	agentura Český ratingový stabilní hodnocení můj republika
107	44	potravinový banka potravina dík pomoc
108	67	vakcína evropský dávka dodávka stát
119	33	Aircraft Industrie odbor
121	72	odbor práv moci Company ala zaměstnanec
128	37	důchodový reforma důchod spravedlivý komise systém
129	49	antivirus vir program odbor
130	94	technologie inovační nový strategie Future
131	97	soud spravedlnost právní premiér rozhodnutí

132	94	svátek odbor obchod zaměstnanec chtít volno Company
133	54	Olaf zpráva hnízdo čapí zveřejnit podvod
134	84	mzda minimální odbor nízký via tripartita
137	79	můj připomínat ala
139	66	evropský Babiš zájem střet premiér komise
140	86	NATO můj český výročí Česko
142	67	Mnichov bavorský Bavorsko můj
147	35	mzda zaměstnanec
148	31	prezident republika Miloš Zeman říjen
152	50	můj zahraniční politik rozhovor politika
153	45	svoboda demokracie hodnota můj muset
154	42	Německo německý můj
155	70	biopalivo miliarda zdanit ušetřit zdanění podpor můj
157	72	politik politický odbor ala tripartita
159	82	romský ROM Rom ala problém kultura
163	45	mediální mediálně gramotnost gramotný zavorek Company
164	42	odbor via Kovo AndrejBabis tripartita Soucek Jaroslav mpo tweetovat
166	34	prezident Pan hrad pražský hod
168	57	můj Irák
178	60	žádost program miliarda COVID mil
179	37	hasič všechen dík děkovat policista
181	53	korporátní korporace cenzura Company
182	42	banka úvěr moratorium splátka odklad firma
187	56	odbor tripartita technologie mpo tweetovat JMalacov SvazPrumysl Hospodark
190	72	můj ala jít
191	50	JMalacov jhamacek AndrejBabis solidarita danit
194	36	SPD hnutí islamizace jediný můj islámský bojovat
197	100	stavební zákon řízení návrh vláda ala
198	36	zaměstnanec odbor vláda náhrada
201	48	danit miliarda stát milión
202	46	mzda Slovensko odbor minimální růst KOZSlovakRep slovenský
206	31	kniha
209	30	danit snížení mzda návrh příjem sleva superhrubý
211	79	ala chtít evropský reforma muset Evropa
216	80	vláda bezpečnost firma aba Dukovany země
217	49	Izrael můj prezident spolupráce
218	76	CSSD plat odbor mzda všechen chtít

221	45	odbor tripartita hod strakovek SvazPrumysl 2019 via 2020
222	71	senát můj výbor usnesení aba
223	50	země vztah
227	46	Vánoce všechen přát vánoční svátek rád
228	42	euro konvergence reálný eurozóna ala Euro odbor
231	70	senior očkování všechen můj dík
237	71	100 výročí prezident český můj republika
238	53	prezident summit země
241	78	dítě rodina rodič moci chtít

Appendix D

Bibliography

[1] Sanjeev Arora, Yuanzhi Li, Yingyu Liang, Tengyu Ma, and Andrej Risteski. A latent variable model approach to PMI-based word embeddings. *Transactions of the Association for Computational Linguistics*, 4:385–399, 2016.

[2] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006.

[3] Ehsaneddin Asgari and Mohammad RK Mofrad. Continuous distributed representation of biological sequences for deep proteomics and genomics. *PloS one*, 10(11):e0141287, 2015.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[5] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[6] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[7] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[8] Angana Borah, Manash Pratim Barman, and Amit Awekar. Are word embedding methods stable and should we care about it? *arXiv preprint arXiv:2104.08433*, 2021.

[9] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Céspedes, Steve Yuan, Chris Tar, et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.

[10] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[11] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[12] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. Unsupervised cross-lingual representation learning at scale. *arXiv preprint arXiv:1911.02116*, 2019.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[14] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[16] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[17] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[19] Rania Ibrahim, Ahmed Elbagoury, Mohamed S Kamel, and Fakhri Karray. Tools and approaches for topic detection from twitter streams: survey. *Knowledge and Information Systems*, 54(3):511–539, 2018.

[20] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[21] Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pages 39–48, 2020.

[22] Oleksii Kuchaiev and Boris Ginsburg. Factorization tricks for lstm networks. *arXiv preprint arXiv:1703.10722*, 2017.

[23] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, pages 3744–3753. PMLR, 2019.

[24] Juho Lee, Yoonho Lee, and Yee Whye Teh. Deep amortized clustering. *arXiv preprint arXiv:1909.13433*, 2019.

[25] Omer Levy, Yoav Goldberg, and Ido Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the association for computational linguistics*, 3:211–225, 2015.

[26] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[27] Alejandro Martín, Javier Huertas-Tato, Álvaro Huertas-García, Guillermo Villar-Rodríguez, and David Camacho. Facter-check: Semi-automated fact-checking through semantic similarity and natural language inference. *arXiv preprint arXiv:2110.14532*, 2021.

[28] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[29] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. Advances in pre-training distributed word representations. *arXiv preprint arXiv:1712.09405*, 2017.

[30] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[31] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[32] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations, 2018.

[33] ME Peters, M Neumann, M Iyyer, M Gardner, C Clark, K Lee, and L Zettlemoyer. Deep contextualized word representations. arxiv 2018. *arXiv preprint arXiv:1802.05365*, 12, 1802.

[34] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[35] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3):1–21, 2017.

[36] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152. IEEE, 2012.

[37] David Sculley. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*, pages 1177–1178, 2010.

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

[39] Xinying Song, Alex Salcianu, Yang Song, Dave Dopson, and Denny Zhou. Fast wordpiece tokenization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 2089–2103, 2021.

[40] Milan Straka, Jakub Náplava, Jana Straková, and David Samuel. Robeczech: Czech roberta, a monolingual contextualized language representation model. In *International Conference on Text, Speech, and Dialogue*, pages 197–209. Springer, 2021.

[41] Milan Straka and Jana Straková. Morphodita: Morphological dictionary and tagger. 2014.

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[43] Nguyen Xuan Vinh, Julien Epps, and James Bailey. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *The Journal of Machine Learning Research*, 11:2837–2854, 2010.

[44] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. *arXiv preprint arXiv:1905.09418*, 2019.

[45] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[46] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[47] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. In *International Conference on Machine Learning*, pages 10524–10533. PMLR, 2020.

[48] Yang You, Jing Li, Jonathan Hseu, Xiaodan Song, James Demmel, and Cho-Jui Hsieh. Reducing bert pre-training time from 3 days to 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.