CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Cybernetics

Master's Thesis

Semantic Sentence Similarity for Intent Recognition Task

Tomáš Brich
tomas.brich@seznam.cz

May 2018
Supervisor: Ing. Jiří Spilka, Ph.D.

Acknowledgement / Declaration

First, I would like to thank my thesis supervisor Ing. Jiří Spilka, Ph.D. for a lot of valuable advice and for all the time he spent helping me finish this thesis. Further, I would like to thank Ing. Jan Šedivý, Ph.D. for a great leadership during my work at eClub, where I started working on this thesis. Other thanks belong to the members of the Alquist chat bot team, who kindly provided me with their implementation of an intent recognition system and a dataset to compare it with my results. I would also like to thank the authors of the embedding algorithms, as they always helped me when I had a question about their work. Finally, I would like to thank my friends and family for all the support during my studies.

Abstrakt / Abstract

Modul pro rozpoznání úmyslu je základní součástí jakéhokoliv question-answering bota (např. Amazon Echo). Tato práce implementuje modul pro rozpoznání úmyslu, založený na větných předlohách, který je silně závislý na efektivitě text embedding algoritmů. Tato práce proto poskytuje komplexní přehled nynějších word a sentence embedding algoritmů. Dále provádí unikátní porovnání těchto algoritmů, týkající se jejich trénovacích schopností, výkonu a hardwarových nároků. Tato práce dále implementuje dvě metody komprese embedding modelů (promazávání slovníku a vektorovou kvantizaci) za účelem jejich použití v mobilních aplikacích.

Embedding algoritmus StarSpace dosáhl v experimentech nejlepších výsledků. Zkoumané metody pro kompresi modelů se ukázaly být velmi výkonné, přičemž dokázaly zmenšit velikost modelů 100-1000 krát bez viditelného zhoršení výsledků. Komprimovaný StarSpace model byl proto využit pro výsledný modul pro rozpoznání úmyslu, který byl schopen překonat systém používaný v Alquist social botovi (druhé místo v Alexa prize soutěži, 2017), přičemž byl méně komplexní.

Klíčová slova: zpracování přirozeného jazyka; text embedding; sémantická podobnost textů; rozpoznání úmyslu; komprese vektorů.

Překlad titulu: Sémantická podobnost vět pro úlohu rozpoznání úmyslu

An intent recognition module is a core component of any question-answering bot (e.g. Amazon Echo). This thesis implements a template-based intent recognition system, which heavily relies on the performance of text embedding algorithms. The thesis therefore provides a comprehensive overview of the state-of-the-art word and sentence embedding algorithms. Further, it performs a unique comparison of the algorithms in terms of their training properties, performance, and hardware requirements. This work further implements two model compression techniques (vocabulary pruning and vector quantization) to make the models more suitable for mobile applications.

The StarSpace embedding algorithm performed the best in the experiments. Further, the compression methods proved to be very powerful, being able to reduce the size of the models 100-1000 times without any notable loss of performance. Thus, a compressed StarSpace model was used to create the resulting intent recognition module that was able to outperform the currently used system in the Alquist social bot (second place in the 2017 Alexa prize contest) while being less complex.

Keywords: natural language processing; text embedding; semantic textual similarity; intent recognition; vector compression.

Contents /

Tables / Figures

Chapter 1
Introduction

1.1 Problem definition

An automatic intent recognition module is one of the fundamental blocks of any conversational system. It is required to determine the intent of the user, i.e. what the user wants the system to do, and how the system can offer help. Intent recognition modules are used in a wide variety of tasks, one large set of examples being the internet search engines, where the engine provider first needs to correctly decide on the user's intent, before trying to match specific entities in the query. In more complicated conversational systems, where the machine is supposed to have a complex conversation with the user, the intent recognition module can be used to direct the system towards the relevant branches of the code, helping the algorithm to change its behavior based on the user's input.

The general approach for creating an intent recognition system is to use algorithms, that can determine the degree of similarity between different sentences. The given sentence query can therefore be compared to a predefined set of intent templates (i.e. manually written sentences for each intent), and the intent of the template with the highest similarity is considered as the correct one. Uncertainty can also be incorporated into the decision process by introducing a similarity threshold. A simple way of comparing two sentences is to convert them into vectors and then to find either the Euclidean distance between the two vectors or computing their cosine similarity.

The disadvantage of the template-based intent recognition system is that a list of templates that sufficiently covers the domain in question has to be maintained. The complexity of the intent recognition process also grows linearly with the number of templates used. On the other hand, introducing a new intent into the system becomes as simple as adding a new group of templates.

Another approach to creating an intent recognition module is to use text embeddings as an input to a classifier, which is trained to output an intent for the query sentence. This approach is currently used in conversational applications like Google Home and Amazon Echo. The disadvantage of this approach is that a large labeled dataset is needed in order to train such a classifier. Another drawback is that when a new intent is introduced into the system, the whole classifier needs to be retrained in order to incorporate the changes.

1.2 State-of-the-art

Learning vector space language models has a long history in natural language processing. Converting a body of text to a vector representation can be as simple as using a *Count Vectorizer*, which transforms a collection of text documents into a matrix of word or word n-gram counts. A more sophisticated approach is to use the TF-IDF algorithm (Jones, 1972 [1]; Ramos, 2003 [2]), which can determine the importance of a term inside a document, based on the occurrence counts of the term in other documents in the training dataset.

However, since these text vectorizers work only with the counts of tokens, the different texts converted to vectors by these algorithms can be compared only by the amount of the same words present in the sentences. The text embedding algorithms, on the other hand, have the ability to capture the semantic meaning of the original text, by learning each word representation based on its context. Given the information contained within the embedding vectors, it is possible to compare words or sentences based on their semantic meaning and perform various analogy tasks. For example the analogy “Paris is to France as London is to England” can be encoded in the vector space as *Paris - France = London - England* (cf. Figure 1.1). Thanks to this property, text embeddings find usage in applications like text similarity tasks, document classification or ranking, toxic comments detection and filtering, internet search engines, machine translation and many others.

The text embedding algorithms have caught a major attention especially after Mikolov et al., 2013 [3] utilized the idea of using hierarchical softmax (Morin et al., 2005 [4]) in a shallow neural network in order to effectively learn word embedding models in an unsupervised manner on very large corpora, creating the Word2Vec embedding algorithm.

The more recent work in this area includes the FastText algorithm (Bojanowski et al., 2016 [5]; Joulin et al., 2016 [6]), which enriches the word embeddings with character-level information. The Sent2Vec algorithm (Pagliardini et al., 2017 [7]) builds upon FastText in order to expand the word embeddings to a larger sentence context. The StarSpace algorithm (Wu et al., 2017 [8]) uses a similar architecture as FastText with the possibility to encode different entities in the same vectorial space, which makes the comparison between them easier.

Other embedding algorithms discussed in this thesis are the GloVe algorithm (Pennington et al., 2014 [9]), that instead of a neural network utilizes a co-occurrence matrix factorization method, and the InferSent algorithm (Conneau et al., 2017 [10]), which uses a supervised layer trained on natural language inference data, built on top of pre-trained word embedding models.

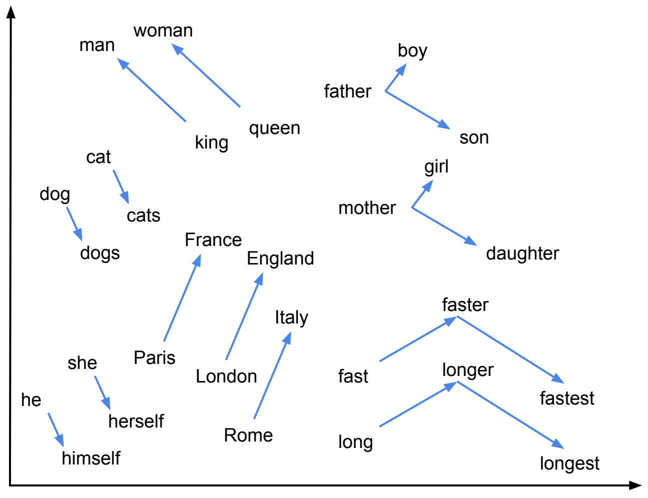


Figure 1.1. Examples of the relations that word embeddings are able to capture in the vectorial space.

1.3 Outline, goals and contribution

This thesis utilizes the ability of text embedding models to determine the semantic similarity between sentences in order to build a template-based intent recognition system (described in Section 1.1) for a question-answering (QA) bot (e.g. Amazon Echo or Google Home, cf. Figure 1.2). Since this approach is heavily dependent on the performance of the used embedding model, a large part of this thesis focuses on the research of the current state-of-the-art embedding algorithms, discussed in Chapter 2.

The authors of the embedding algorithms usually provide publicly available models, trained on large corpora. Even though there were attempts at comparing the algorithms, such comparisons are often made using these pre-trained models, which were trained on different datasets and under different conditions, making the comparison biased. Therefore, in this thesis, the algorithms are compared in terms of their ability to train on different types of corpora with matching training hyperparameters. The results of these experiments, as well as the hardware requirements of the embedding algorithms, are shown in Chapter 3.

Since the embedding models trained on large corpora are usually several gigabytes large, Chapter 4 explores the possibility of reducing the size of the models in order to make them suitable for mobile applications. Finally, Chapter 5 explores the resulting intent recognition module created for the QA bot.

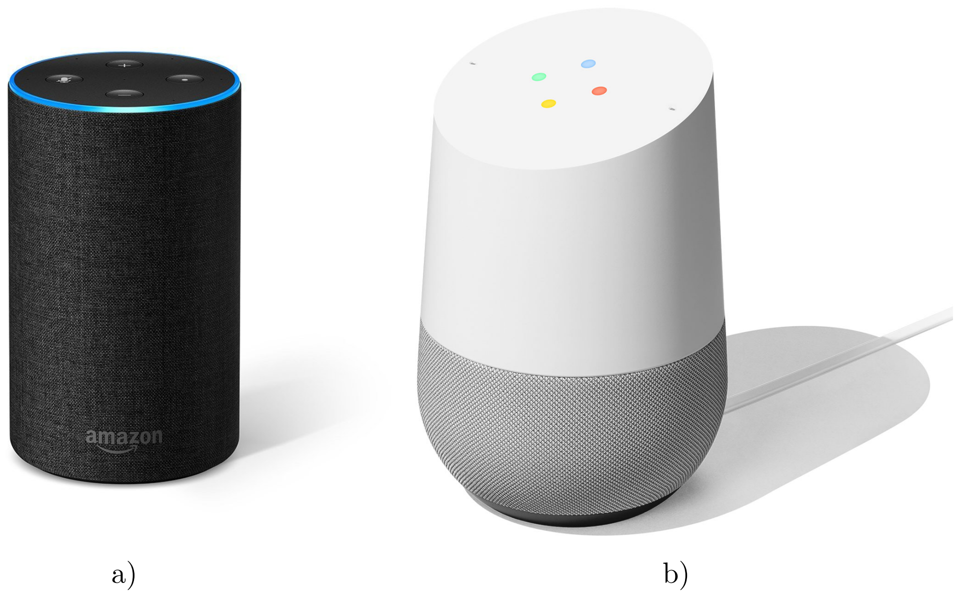


Figure 1.2. Amazon Echo (a) and Google Home (b) devices.

Chapter 2
Embedding algorithms

A selection of the current state-of-the-art word and sentence embedding algorithms was studied. All the used algorithms had their implementations publicly available. Four word embedding and two sentence embedding algorithms were used.

The word embedding algorithms used are:

- Word2Vec
- FastText
- StarSpace
- GloVe

The sentence embedding algorithms used are:

- Sent2Vec
- InferSent

This chapter explores the theory behind each of the embedding algorithms, giving a comprehensive summary of their main features. Section 2.7 also explores the softmax approximations, as they are used in most of the discussed algorithms.

2.1 Word2Vec

There are two main variants of the Word2Vec algorithm (Mikolov et al. [3]). The Continuous Bag-of-Words (CBOW) variant and the Skip-Gram (SG) variant.

The basic idea behind the CBOW algorithm is to try to predict a probability of a word given its context. The context can be an arbitrary number of words. The number of words in the context is given by a context window, which says how large neighborhood of the given word will be used as its context.

The model is in the form of a shallow neural network, with one input, one output, and one hidden layer. The representation of a CBOW model can be seen in Figure 2.1.

Both the context words and the output word are represented in the network as one-hot vectors of size $|\mathcal{V}|$ (i.e. size of the vocabulary), indicating the position of the word in the vocabulary. The context words are fed into the input layer, and the network is trained using a regular back-propagation algorithm to output the target word. Note that each word in the vocabulary has two different representations of dimension equal to the number of neurons in the hidden layer. One representation corresponds to the rows of the input weight matrix W and the second one corresponds to the columns of the output weight matrix W'. For a word w, we therefore receive an input vector $\mathbf{v}_w$ and an output vector $\mathbf{v}'_w$.

There is a linear activation on the hidden layer neurons. The output layer neurons use softmax as their activation to model the probability of a word given context. For a single neuron in the output layer corresponding to the word w_O, we obtain

$$p(w_O|w_I) = \sigma({\mathbf{v}'_{w_O}}^T \mathbf{v}_{w_I}) = \frac{\exp({\mathbf{v}'_{w_O}}^T \mathbf{v}_{w_I})}{\sum_{w_j \in \mathcal{V}} \exp({\mathbf{v}'_{w_j}}^T \mathbf{v}_{w_I})}, \tag{2.1}$$

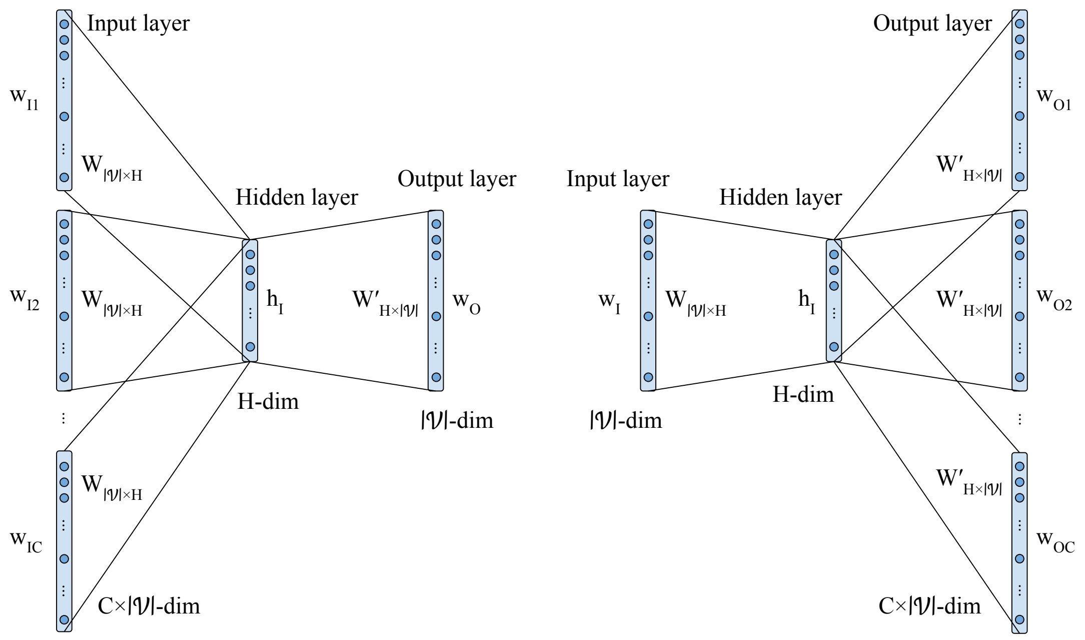


Figure 2.1. The scheme of a general CBOW (left) and Skip-Gram (right) neural networks. $|\mathcal{V}|$ denotes the vocabulary size, C is the context window size, H is the embedding dimension, W and W' are the input and output matrices and w_I and w_O are the input and output words.

where w_I is an input (context) word. In a multi-word context case, the vector $\mathbf{v}_{w_I}$ is computed as an average over all the context words.

The Word2Vec algorithm uses a negative log loss function as its objective as

$$L = -\log p(w_O|w_I). \tag{2.2}$$

To make the computation of the softmax feasible for a large number of training samples, an approximation needs to be used, like hierarchical softmax or negative sampling. Those two approaches are described in Section 2.7.

The disadvantage of CBOW is that since it uses the average of the context of a word during calculation of the hidden activation, it is unable to capture multiple semantic meanings of a single word. The Skip-Gram model, while being slower to learn, solves this problem.

As seen in Figure 2.1, the Skip-Gram architecture is similar to the one of CBOW, but instead of trying to predict a word given its context, it tries to predict the context of a word. During the training process, there will be a separate error for each context word and those error vectors are added element-wise to obtain the resulting error, used during the back-propagation.

The context window works a bit differently in the Skip-Gram model. Instead of having a constant window, a maximum window range C is chosen and for each training word, a context window in the interval $[1, C]$ is randomly selected. Another difference is that the CBOW model usually uses the output matrix W' as the resulting word embeddings, while the Skip-Gram model uses the input matrix W. This is only a convention, however, since it should be possible to use any of the matrices for both CBOW and Skip-Gram as the resulting vector embeddings.

2.2 FastText

Most of the state-of-the-art word embedding techniques assign a distinct vector to each word in the vocabulary. This approach is problematic for morphologically rich languages (like Czech or German), where each word can have many forms. Some of these forms can be very rare and can occur only a few times or not at all in the training corpus. In order to tackle this issue, FastText (Bojanowski et al. [5]) proposes a method for using character-level information instead.

FastText is essentially an extension of Word2Vec. As opposed to Word2Vec, which treats each word as an atomic entity, FastText represents each word as a bag of character n-grams. In addition to the n-grams, the word itself is added to the bag in order to learn its own representation. Each word is changed to include special characters $<$ and $>$, which denote the start and the end of the word, respectively. It is important to note, that a given word and an n-gram, which represents the same character sequence as this word (e.g. the word "<her>" and the character 3-gram "her"), will have different vector representations.

Softmax on the output is modeled by the logistic loss function $\ell(x) = \log(1 + e^{-x})$. The objective function for FastText Skip-Gram (already including the negative sampling) for one training sample therefore becomes

$$L = \sum_{w_O \in \mathcal{C}_{w_I}} \ell(s(w_I, w_O)) + \sum_{w_n \in \mathcal{W}_{\text{neg}}} \ell(-s(w_I, w_n)), \tag{2.3}$$

where $\mathcal{C}_w$ is the set of context words of the word w, $\mathcal{W}_{\text{neg}}$ is the set of current negative samples, and $s(\cdot,\cdot)$ is a scoring function, which represents the score of similarity between two word representations as

$$s(w, c) = \sum_{g \in \mathcal{G}_w} \mathbf{v}'^{T}_{g} \mathbf{v}_c, \tag{2.4}$$

where $\mathcal{G}_w \subset \mathcal{G}$, $\mathcal{G}$ is the character n-gram vocabulary, and $\mathbf{v}'_g$ is therefore the input vector representation of the character n-grams present in the word w. This model allows sharing of the vector representations of the character n-grams across words. A word embedding is then constructed as the sum of its own representation and the representation of its character n-grams.

FastText has several advantages over Word2Vec:

- It generates better word embeddings for rare words. Even though some words do not appear often in the training corpus, its character n-grams can still be shared with other words, which can result in a reasonable embedding for the rare word. This phenomenon manifests itself mainly when FastText is used on morphologically rich languages, as shown in Figure 2.2.
- It can generate embeddings for out-of-vocabulary (OOV) words, since it can still create the embedding as the sum of the character n-grams present in the unknown word. Bojanowski et al. [5] report surprisingly good performance in tasks of word similarity for OOV words. The performance improvement after utilizing this ability of FastText can also be seen in the experiments in Chapter 3.
- Thanks to the ability to create embeddings for OOV words, the algorithm is also more robust towards the size of the training data. This can be used for training the algorithm for tasks where there are not any large training corpora available. This is also demonstrated in Chapter 3.

- It can be faster to learn than Word2Vec. Even though training a FastText model on the same corpus size as Word2Vec will be slower, the model can achieve mostly constant performance with much smaller training corpus (or with less training epochs) than Word2Vec, thanks to the properties discussed in the above points.

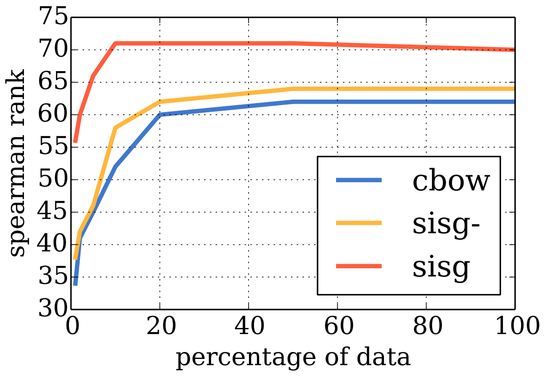


Figure 2.2. Influence of the size of the training data on the performance on a word similarity task (German GUR350 dataset). Here, “sisg” (Subword Information Skip-Gram) corresponds to the FastText algorithm, where OOV words are computed using character n-grams and in “sisg-”, the OOV words are replaced with zero vectors. Retrieved from [5] (p. 7).

2.3 StarSpace

The idea behind the StarSpace algorithm (Wu et al. [8]) is to encode entities of different types (e.g. words, sentences, documents, document labels or classes, user definitions, etc.) into a common vectorial space. Each entity is described by a set of features from the vocabulary (bag-of-features). The embedding of a multi-feature entity is constructed as a sum of the embeddings of its features. The entities can then be easily compared by using a predefined similarity function (e.g. cosine similarity) to solve various tasks, including text classification, ranking, document recommendations, article search and semantic similarity. Starspace uses the following loss function as its objective:

$$L = \sum_{\substack{(a,b)\in E^+ \\ b^-\in E^-}} L^{\text{batch}}(s(a,b), s(a,b_1^-), \ldots, s(a,b_k^-)), \tag{2.5}$$

where E^+ is a task-specific generator of positive examples, E^- is a generator of negative examples utilizing negative sampling, L^{batch} is a loss function that compares the positive pair with the negative pairs and $s(\cdot,\cdot)$ is a similarity function, which is then used to compare the entities during testing.

The StarSpace algorithm implements two options for the loss function L^{batch}. Either negative log loss of softmax, as used in the previously discussed algorithms, or the margin ranking loss, i.e. $\max(0, \mu - s(a,b))$, where μ is the margin parameter. The similarity function also has two options, either simple dot product or the cosine similarity as

$$s(a,b) = \frac{\mathbf{v}_a{}^T \mathbf{v}_b}{\|\mathbf{v}_a\| \|\mathbf{v}_b\|}. \tag{2.6}$$

In the mode for unsupervised learning of word embeddings, a target word is selected as the b entity, and the entity a represents the words in the context of the target word. Therefore, if we select L^{batch} to be the negative log loss of softmax and the similarity function $s(\cdot,\cdot)$ as a dot product, the resulting objective function will be equal to the objective of FastText (2.3) without considering the character n-grams.

2.4 GloVe

The current word embedding models are usually trained by either co-occurrence matrix factorization or by a neural network using local context windows (Word2Vec, FastText, etc.). The matrix factorization models use the statistical data contained within the corpus, but they tend to fail in capturing the semantics of the words. On the other hand, the context window methods are good at capturing the meaning, but they do not utilize the valuable statistical data. The GloVe (Global Vectors) algorithm (Pennington et al. [9]) tries to show that it is possible to use matrix factorization methods to keep the statistical co-occurrence data while being able to capture the meanings of words.

GloVe is able to capture the relationships between words by using the ratios of their co-occurrences with other words ($P_{w_i w_k}/P_{w_j w_k}$). The co-occurrence probability is defined as

$$P_{w_i w_k} = \frac{X_{w_i w_k}}{\sum_{w_k \in \mathcal{V}} X_{w_i w_k}}, \tag{2.7}$$

where $X_{w_i w_k}$ is the number of times word w_k occurs in the context of word w_i, drawn from the co-occurrence matrix X.

For words w_k related to the word w_i, we can expect a high ratio and for words w_k related to word w_j, we can expect the ratio to be small. For words completely unrelated or related to both w_i and w_j, the ratio should be close to one.

The algorithm creates two word vector matrices, W and W'. These two matrices are equivalent in case that X is symmetric and they differ only because of their random initialization. Pennington et al. [9] state that training multiple instances of a neural network and then combining the results can be less prone to overfitting and can improve the performance for certain types of neural networks. Even though GloVe uses matrix factorization instead of a neural network model, this approach is adopted here. The resulting embeddings are therefore constructed by summing the matrices W and W' and the vectors $\mathbf{v}_w$ and $\mathbf{v}'_w$ correspond to these two matrices, in contrast to the previously discussed algorithms.

GloVe proposes a weighted least squares regression model as

$$J = \sum_{w_i, w_j \in \mathcal{V}} r\left(X_{w_i w_j}\right) \left(\mathbf{v}_{w_i}{}^T \mathbf{v}'_{w_j} + b_{w_i} + b'_{w_j} - \log X_{w_i w_j}\right)^2, \tag{2.8}$$

where b_w and b'_w are biases for $\mathbf{v}_w$ and $\mathbf{v}'_w$, and $r(\cdot)$ is a weighting function

$$r(x) = \begin{cases} (x/x_{\max})^\alpha & \text{if } x < x_{\max}; \\ 1 & \text{otherwise,} \end{cases} \tag{2.9}$$

where x_{max} denotes the co-occurrence count, after which the weight will not increase anymore. This is done in order not to overweight frequent co-occurrences. Pennington et al. [9] empirically found $\alpha = 3/4$ to perform the best on their models.

2.5 Sent2Vec

The Sent2Vec algorithm (Pagliardini et al., 2017 [7]) can be seen as a modification of Word2Vec CBOW, where the word vectors are optimized in order to subsequently construct sentence embeddings. The most important differences are:

- Instead of a fixed context window C, the context always covers the whole sentence of the target (output) word.
- CBOW uses frequent word subsampling, deciding to discard each word w with a probability proportionate to its frequency in the training corpus. As this deprives the sentence of important syntactical features, Sent2Vec does not use this kind of subsampling.
- Sent2Vec supports the possibility to also encode word n-grams, which can improve performance on certain tasks.

The sentence embedding is defined as a simple average over all the words in the sentence as

$$\mathbf{v}_S = \frac{1}{|R(S)|} \sum_{w \in R(S)} \mathbf{v}_w, \tag{2.10}$$

where $R(S)$ is the set of word n-grams present in the sentence S (i.e. the set of words in a sentence in case only unigrams are considered).

The objective function uses softmax output approximated by negative sampling. The unsupervised training objective for one training sentence S is formulated as

$$L = \sum_{w_O \in S} \ell\left(\mathbf{v}_{w_O}{}^T \mathbf{v}'_{S\setminus\{w_O\}}\right) + \sum_{w_n \in \mathcal{W}_{\text{neg}}} \ell\left(-\mathbf{v}_{w_n}{}^T \mathbf{v}'_{S\setminus\{w_O\}}\right). \tag{2.11}$$

2.6 InferSent

The InferSent algorithm exploits word embeddings previously trained on large corpora in an unsupervised manner (e.g. Conneau et al. [10] use the publicly available GloVe vectors) while building a supervised classifier on top. The classifier is trained on the SNLI corpus (Bowman et al. [11]), which is a large collection of human-written sentence pairs (i.e. the premise and the hypothesis pairs), manually labeled for the task of natural language inference (i.e. recognizing textual entailment — classes “entailment”, “neutral” and “contradiction”).

The model is trained in a way that separates the encoding of the sentences in the sentence pairs. A shared sentence encoder is used, which outputs a vector for the premise $\mathbf{u}$ and the hypothesis $\mathbf{v}$. After that, three methods are used to extract relations between the premise and the hypothesis — concatenation $(\mathbf{u}, \mathbf{v})$, element-wise product $\mathbf{u} \odot \mathbf{v}$ and absolute difference $|\mathbf{u} - \mathbf{v}|$. The resulting vector $(\mathbf{u}, \mathbf{v}, |\mathbf{u} - \mathbf{v}|, \mathbf{u} \odot \mathbf{v})$ is used as an input to a 3-class classifier with a softmax output layer, which is trained to output the classes defined in the SNLI dataset (cf. Figure 2.3).

Conneau et al. [10] use a multi-layer perceptron with a single hidden layer of 512 neurons as the classifier while experimenting with different architectures for the shared sentence encoder. They further conclude that a bi-directional LSTM network with max pooling outperforms other network architectures. The proposed network scheme can be seen in Figure 2.4.

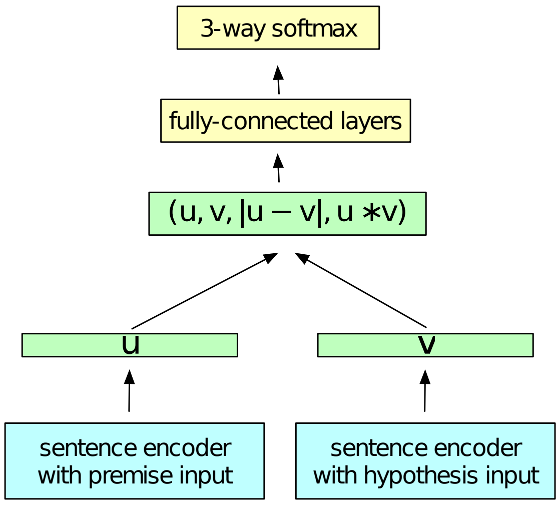

3-way softmax
fully-connected layers
(u, v, |u − v|, u ∗v)
u
v
sentence encoder with premise input
sentence encoder with hypothesis input

Figure 2.3. Natural Language Inference (NLI) training scheme. The premise and hypothesis sentences are taken from the SNLI Corpus before encoding them to vectors **u** and **v**, respectively. The combinations of these vectors are then fed to a 3-class classifier (e.g. multi-layer perceptron network) that is trained to output classes “entailment”, “neutral” and “contradiction”. Retrieved from [10] (p. 3).

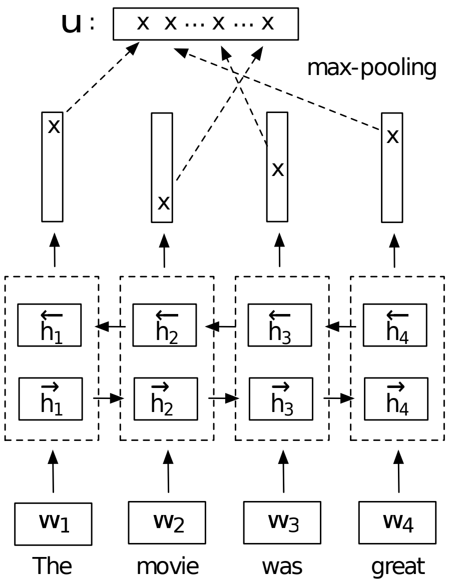


Figure 2.4. Bi-directional LSTM network with max-pooling scheme used as a sentence encoder in InferSent. A sentence is fed as a sequence of T words into a forward and a backward LSTM, which read the sentences in two opposite directions. For $t \in [1, \dots, T]$, the vector $\mathbf{h}_t$ is the concatenation of the hidden representations in the t-th layer of the two networks. The concept of max-pooling (i.e. selecting the maximum value over each dimension of the hidden units) is used to create a fixed-size vector as a sentence representation. Retrieved from [10] (p. 3).

2.7 Softmax approximations

Most of the discussed algorithms use softmax function on their output in order to model the probability of a word given context. All these models need to update both the input weights W and the output weights W' for each training sample. Updating of the input weight matrix is cheap as it uses linear activation, but updating the softmax function for the output weight matrix is very computationally expensive. The complexity of updating one output word vector $\mathbf{v}'_w$ for each training sample is $\mathcal{O}(|\mathcal{V}|)$, when using the naive implementation of softmax. This makes the training unfeasible for large training corpora. The following approximations solve this problem by limiting the number of required operations.

2.7.1 Hierarchical softmax

Hierarchical softmax uses a binary tree, where each word in the vocabulary is represented as one leaf (cf. Figure 2.5). It can be proven that such a tree will have $|\mathcal{V}| - 1$ inner nodes.

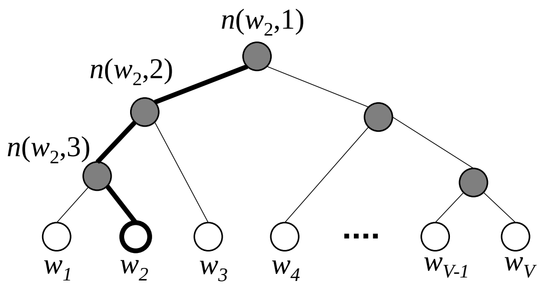


Figure 2.5. Hierarchical softmax binary tree example. The white units represent the leaf nodes (i.e. words in the vocabulary) and the dark units represent the inner nodes. Retrieved from [12] (p. 10).

In the hierarchical softmax model, the output word vectors $\mathbf{v}'_w$ are replaced with vector representations of the $|\mathcal{V}| - 1$ inner nodes $\mathbf{v}'_{n(w,d)}$, where $n(w,d)$ represents the node at depth d on the path from root to the word w. The probability of a word being the output word can be computed by following a path from the root to the corresponding leaf as

$$p(w = w_O) = \prod_{d=1}^{D(w)-1} \sigma\left(\llbracket n(w, d+1) = l(n(w,d)) \rrbracket \cdot {\mathbf{v}'_{n(w,d)}}^T \mathbf{v}_{w_I}\right), \tag{2.12}$$

where $l(n)$ is the left child of the node n, and

$$\llbracket x \rrbracket = \begin{cases} 1 & \text{if } x \text{ is true;} \\ -1 & \text{otherwise.} \end{cases} \tag{2.13}$$

The probability of a word being the output word is therefore defined as the probability of a random walk from the root to the corresponding leaf, where the probability of going left or right at each inner node is given by $\sigma(\llbracket\cdot\rrbracket \cdot {\mathbf{v}'_n}^T \mathbf{v}_{w_I})$. The probability of the word w_2 from the example in Figure 2.5 can therefore be computed as

$$p(w_2 = w_O) = \sigma\left({\mathbf{v}'_{n(w_2,1)}}^T \mathbf{v}_{w_I}\right) \cdot \sigma\left({\mathbf{v}'_{n(w_2,2)}}^T \mathbf{v}_{w_I}\right) \cdot \sigma\left(-{\mathbf{v}'_{n(w_2,3)}}^T \mathbf{v}_{w_I}\right). \tag{2.14}$$

Thanks to this approach, the probability of a single outcome $p(w = w_O)$ only depends on the internal nodes that lie on the path from the root to the leaf denoting w, effectively reducing the complexity of computing the softmax from $\mathcal{O}(|\mathcal{V}|)$ to $\mathcal{O}(\log_2 |\mathcal{V}|)$. Exploring the tree with a depth-first search also allows for discarding the branches with a small probability.

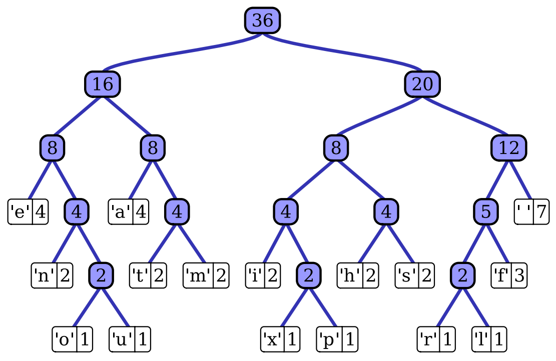


Figure 2.6. Wikipedia example of a Huffman tree constructed from characters in the sentence “this is an example of a huffman tree”.

As for the tree itself, the usual approach in the embedding algorithms is to create a Huffman tree (cf. Figure 2.6), which minimizes the average path length from the root to the leaf.

2.7.2 Negative sampling

Negative sampling solves the problem of having a large output weight matrix by choosing only a small subset of weights for the update. There is always a small number of negative samples (negatives) selected at random for each training sample. By negative is considered a word, for which the network is supposed to output a zero (i.e. the training sample and a negative sample do not share a context). Now the network will be updating the weights only for the positive sample (i.e. the output word) and the selected negatives.

The objective function in Word2Vec then changes from (2.2) to

$$L = -\log\sigma({\mathbf{v}'_{w_O}}^T \mathbf{v}_{w_I}) - \sum_{w_j \in \mathcal{W}_{\text{neg}}} \log\sigma(-{\mathbf{v}'_{w_j}}^T \mathbf{v}_{w_I}), \tag{2.15}$$

where $\mathcal{W}_{\text{neg}}$ denotes the current set of negative samples.

The experiments reported in Mikolov et al. [3] show that for smaller corpora, 5-20 negative samples for each training sample work well. For larger datasets, it can be even less than five negative samples, which drastically reduces the number of weights modified.

The probability of a word w_i to be selected as a negative is usually related to its frequency in the corpus as

$$p(w_i) = \frac{f(w_i)^\alpha}{\sum_{w_j \in \mathcal{V}} f(w_j)^\alpha}, \tag{2.16}$$

where $f(w)$ is the number of occurrences (frequency) of the word w in the training corpus, and α is a parameter, which can be chosen empirically (e.g. Mikolov et al. [3] used $\alpha = 3/4$). It is interesting to note, that this value of α was also used in the GloVe weighting function in Section 2.4.

In practice, a table is constructed from the corpus, containing words corresponding to the square root of their frequency. The negatives are then sampled uniformly from the table.

Chapter 3
Model training comparison

The embedding algorithms discussed in the previous chapter are usually provided with a model pre-trained on a large corpus. These models were however trained on different datasets, with different training parameters, and under different conditions, making them incomparable in terms of performance of the original algorithm. In this chapter, the models are therefore trained on the same datasets and with the same hyperparameters. The resulting models are compared by their performance on the independent STS Benchmark testing dataset (described in Section 3.1), based on both the training time needed and the number of data passes (i.e. epochs). The performance evaluation process is described in Section 3.2. The goal of the experiments performed in Section 3.4 is to explore the ability of the embedding algorithms to train on a small, well defined and well preprocessed dataset, while the experiments in Section 3.5 use a large dataset in order to compare the training capabilities of the algorithms with an increasing number of training samples.

The algorithms compared in the following experiments are Word2Vec CBOW and Skip-Gram, FastText Skip-Gram, StarSpace and the unigram variant of Sent2Vec. Negative sampling was used for all these algorithms, but the hierarchical softmax (HS) variant of Word2Vec Skip-Gram was also added to the comparison. The exact training parameters are described in the following sections.

3.1 Data

3.1.1 STS Benchmark

The studied embedding models were evaluated on the STS Benchmark dataset (Cer et al., 2017 [13]), which consists of three predefined parts: training, development and testing sets. Each part includes a given number of sentence pairs (see Figure 3.1 for the exact sentence pair counts). The pairs were manually scored on a scale between 0 and 5. A higher number denotes a higher degree of semantic similarity between the two sentences.

	train	dev	test	total
news	3299	500	500	4299
captions	2000	625	625	3250
forum	150	375	254	1079
total	5749	1500	1379	8628

Table 3.1. Number of sentence pairs in different parts of the STS Benchmark dataset. “Train”, “dev” and “test” represent the training, development and testing sets, respectively.

The STS training dataset was used for training the embedding models described in Section 3.4.

3.1.2 C4Corpus

A larger dataset was needed for training the embedding algorithms, in order to compare them based on the size of the training data, as described in Section 3.5. For this purpose, an English part of the C4Corpus (Habernal et al., 2016 [14]) dataset was used. The C4Corpus dataset is a preprocessed version of the Common Crawl[1]) dataset, which is an open collection of web crawl data. Details of the part[2]) of the C4Corpus used in Section 3.5 and its subsets are presented in Table 3.2.

data percentage	samples	tokens	vocabulary
100%	20.82	456.0	2.23
80%	16.65	364.3	1.95
60%	12.49	273.1	1.65
40%	8.33	181.9	1.30
20%	4.16	90.7	0.86
10%	2.08	45.4	0.56
1%	0.21	4.4	0.14

Table 3.2. Part of the C4Corpus dataset and its subsets used for training in Section 3.5. "Samples" represent the number of training samples (i.e. number of lines in the training file — can comprise of single words or sentences) and "tokens" represent the total amount of tokens (i.e. words, punctuation and other symbols) present in the corpus. All the values shown are in millions.

3.1.3 Data preprocessing

The training datasets as well as the testing samples were lower-cased and tokenized using the *TweetTokenizer* from the *NLTK* Python package[3]). This tokenizer utilizes common tokenization techniques, like splitting word contractions and separating punctuation symbols from words. It also processes the text in order to find patterns, which are common in online chats, like emojis, phone numbers, etc. Ultimately, the choice of a tokenizer did not play a large role, as most of the tokenizers available in the *NLTK* package performed very similar.

The stochastic gradient descent (SGD) algorithm, that the researched embedding algorithms use for optimization based on their objective functions, also needs the training dataset to be shuffled in order to mitigate the possibility of getting stuck in local minima. The STS Benchmark dataset is by default unordered and the training samples are independent, therefore there is no need to shuffle the samples. The C4Corpus dataset, on the other hand, is an ordered set of textual web data, therefore it was shuffled randomly and each of the embedding algorithms was trained on the same shuffled set. The embedding algorithms handle any potential further sample shuffling between training epochs themselves.

The Sent2Vec algorithm required the training dataset to be in the form of one sentence per line. Since the C4Corpus dataset included whole paragraphs of text, the English *Punkt* sentence tokenizer from the *NLTK* package was used to split the sentences.

[1]) `http://commoncrawl.org/`
[2]) Lic_by-nc-nd_Lang_en_NoBoilerplate_true_MinHtml_true-r-00017.seg-00000
[3]) `https://www.nltk.org/`

3.2 Evaluation

The models were evaluated based on two criteria: the performance of the embedding models on a semantic textual similarity (STS) task and the hardware requirements of the algorithms during sentence encoding.

The **performance on an STS task** is evaluated using the STS Benchmark dataset described in Section 3.1. The sentences from the testing set were transformed to a vector representation, using different embedding algorithms. In case of the word embedding algorithms, the sentence embeddings were constructed as the average over the words present in the sentence (out-of-vocabulary words were skipped). Each sentence pair was given a score as a cosine similarity of the two encoded sentences. The Pearson and the Spearman correlation coefficients between the vector of the manual scores provided in the STS Benchmark dataset and the vector of output cosine similarities are then considered as resulting scores of the models.

Because the evaluation result in the form of correlation is not as intuitive as classic accuracy scores, a simple visual comparison of Pearson and Spearman correlation coefficients is provided in Figure 3.1. The Pearson coefficient evaluates the linear relationship between two continuous variables (how they change together at a constant rate), while the Spearman coefficient evaluates the monotonic relationship (i.e. the variables change together, but not necessarily at a constant rate). It is important to note, that this thesis focuses mostly on the Pearson correlation coefficient, but since the Spearman coefficient is also widely used in NLP applications, it was included in the results for future reference.

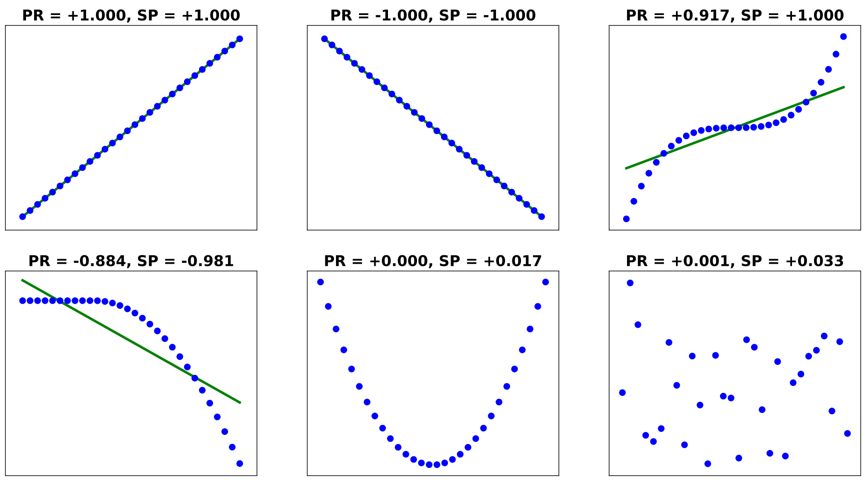


Figure 3.1. Visual comparison of Pearson (PR) and Spearman (SP) correlation coefficients between x and y vectors. The green line in some of the plots serves only as a reference to a linear relationship.

Since the goal of this thesis is to create an intent recognition system, which should be able to work in a reasonable time, the deciding factor for a suitable embedding algorithm for this task will also be its **hardware requirements** (mainly CPU and memory usage), which are discussed in Section 3.6.

3.3 Pre-trained models

Most of the embedding algorithms discussed in this thesis provide a model trained on large corpora. Even though these models are mostly incomparable due to different training conditions, they can serve as a reference to the models trained in sections 3.4 and 3.5. The information about these models is therefore provided in Table 3.3.

algorithm	training corpus	dim	vocab	size [GB]
Sent2Vec unigram	Twitter	700	1.0	13.0
	Wikipedia	600	0.5	4.8
	Toronto Books	700	0.2	1.7
Sent2Vec bigram	Twitter	700	1.8	23.0
	Wikipedia	700	1.2	16.0
	Toronto Books	700	0.8	6.8
InferSent	SNLI	4096	-	0.2
	GloVe Common Crawl	300	2.2	5.6
Word2Vec SG	Google News	300	3.0	3.6
FastText CBOW	Wikipedia	300	2.5	8.5
GloVe	Wikipedia + Gigaword	300	0.4	1.0

Table 3.3. Parameters of the publicly available pre-trained models for different embedding algorithms. The InferSent model is trained on the SNLI corpus but uses a GloVe pre-trained model. “Sent2Vec unigram” and “Sent2Vec bigram” represent the word unigram and bigram variants of the algorithm. “Dim” represents the dimension of the embedding vectors and “vocab” is the approximate number of tokens in the vocabulary in millions.

The performance of the pre-trained models on the STS Benchmark testing set is shown in Table 3.4.

algorithm	training corpus	Spearman	**Pearson**
Sent2Vec unigram	Twitter	<u>0.728</u>	<u>0.755</u>
	Wikipedia	0.640	0.638
	Toronto Books	0.704	0.726
Sent2Vec bigram	Toronto Books	0.690	0.716
InferSent	SNLI + GloVe vectors	0.685	0.710
Word2Vec SG	Google News	0.579	0.622
FastText CBOW OOV	Wikipedia	0.539	0.483
FastText CBOW	Wikipedia	0.582	0.584
GloVe	Wikipedia + Gigaword	0.438	0.408

Table 3.4. Performance of the publicly available pre-trained models on the STS Benchmark testing set. “FastText OOV” represents the FastText algorithm, in which the out-of-vocabulary word embeddings were constructed using the character n-grams. The OOV words were skipped otherwise.

The models trained using the word bigram variant of the Sent2Vec algorithm for both the pre-trained models and the models trained in this thesis always achieved worse results on the STS Benchmark dataset than the unigram models. The bigram variant was therefore not used in the experiments.

The InferSent model, while achieving better results than any of the word embedding models shown in Table 3.4, was also not used in the following experiments. This was done because the algorithm was specifically designed to be trained in a supervised manner on the SNLI corpus (see Section 2.6 for more details), making any attempts to compare its training capabilities with the other studied embedding algorithms impossible.

It is important to note, that the performance of the InferSent algorithm shown in this thesis is significantly lower than the one reported in the comparison on the STS Benchmark website[1]). This was because their experiments with InferSent utilized the STS Benchmark training dataset in order to train a regression model to output the similarity scores. This can, of course, be performed for any of the embedding algorithms to improve their performance on the respective domain, but it would defeat the purpose of the comparison in this thesis. This again shows, how inconclusive the currently available comparisons of the embedding algorithms can be.

Both the FastText and the Word2Vec algorithms performed better than reported on the STS Benchmark website. This might be caused by the fact, that the authors of the comparison were not very clear on the pre-processing of the dataset for these algorithms. The significant performance boost of FastText was also possibly caused by the FastText authors releasing a new set of pre-trained models (Mikolov et al., 2017 [15]), which were used in this thesis.

As seen from the Table 3.4, the pre-trained GloVe model performed very poorly on the STS task compared to the other algorithms. Even though multiple public models are available for the GloVe algorithm, neither of them was able to achieve better results. It soon became apparent, that GloVe was not suited for this kind of task, and therefore any other experiments with this algorithm were dropped in this thesis.

3.4 Training on the STS Benchmark dataset

A significant problem in creating a domain-specific intent recognition module is the lack of training data. The models trained on large corpora can be general enough to perform well under many different domains, but this does not always hold true and such models also tend to be unnecessarily large.

When creating an intent recognition module for a QA bot used for a constrained task (e.g. a bot automating a shopping process), the vocabulary required to cover the domain in question will be constrained as well. Large training corpora for such tasks are usually hard or impossible to acquire. This section therefore explores the ability of the different embedding algorithms to train on a relatively small dataset, containing roughly thousands of samples (as opposed to millions or billions of samples usually needed to properly train an embedding model). For this purpose, the STS Benchmark training dataset was used, containing around 11 500 samples (cf. Table 3.1), comprising of roughly 130 thousand tokens. The general training hyperparameters for all the algorithms were set as follows:

- No minimum number of word occurrences. Only for the comparison purposes, the algorithms should not discard any words with low frequency in the training corpus.
- Initial learning rate of 0.1. All the algorithms utilize a stochastic gradient descent (SGD) algorithm with linearly decaying learning rate. The only exception is StarSpace, which uses the *AdaGrad* algorithm by default, but it was switched to the regular SGD for the purpose of the experiments.
- Embedding dimension H of 300.

[1]) `http://ixa2.si.ehu.es/stswiki/index.php/STSbenchmark`

- Context window of five (i.e. five past and five future words). This does not apply to the Sent2Vec algorithm, which uses its context window to always cover the whole sentence.
- Negative sampling with five negative samples per training sample. The only exception being the hierarchical softmax variant of Word2Vec.
- The algorithms were trained on eight threads of an *Intel Core i7-3632QM 2.2 GHz* CPU.

Other algorithm-specific hyperparameters were left as default. Character n-grams in FastText were set to n being in the interval $[3, 6]$, since it achieved the best performance in the experiments reported by Bojanowski et al. [5]. StarSpace was set to use the margin ranking loss with the margin parameter μ of 0.05, using the cosine similarity as the similarity function s.

3.4.1 Results and discussion

The results of the trained models are shown in Figure 3.2. The exact results together with the training times are also presented in Table D.1.

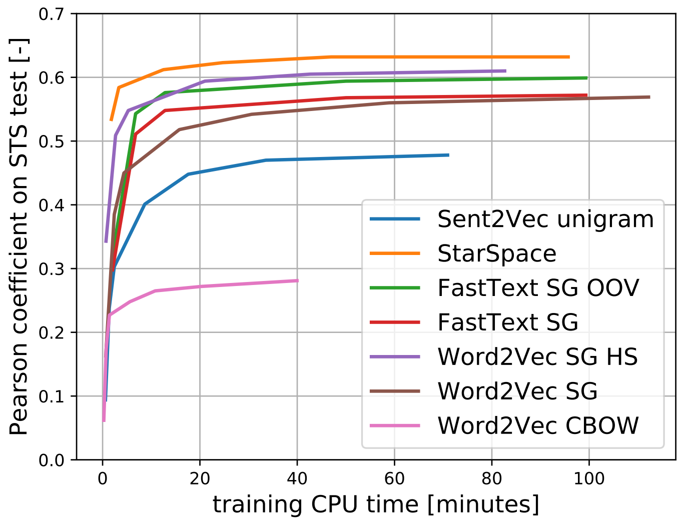


Figure 3.2. Performance (Pearson correlation coefficient) of the models trained on the STS Benchmark training set as a function of training CPU time.

As seen from the graph in Figure 3.2, the StarSpace algorithm was able to outperform all the other compared embedding algorithms, both in terms of data passes and training time. Intuitively, the StarSpace algorithm was expected to perform roughly on par with the FastText algorithm. From the experiments, it seems, however, that the margin ranking loss and the cosine similarity used in StarSpace bring a slight boost over the softmax and a simple dot product used in FastText. This begs the question, whether the algorithms tend to overfit in terms of the Euclidean norm of the embedding vectors, since using a cosine similarity, which is a normalization of the dot product based on the Euclidean norm, seems to bring better results.

The results also show that the Skip-Gram variant of Word2Vec and FastText achieve roughly the same performance in the long run when the ability of FastText to construct the out-of-vocabulary (OOV) word embeddings is not used. FastText is, however, able to converge to this performance much faster, which is consistent with the theory discussed in Section 2.2. FastText achieves a slightly better performance when constructing the OOV word vectors from the character n-grams. This boost in performance is not as significant as shown in Figure 2.2, most probably because the n-grams do not bring so much information into the English language as to the morphologically rich German language.

The Sent2Vec model trained on the STS Benchmark was not able to get anywhere near the performance of the publicly available models, possibly because the algorithm requires much larger training corpus in order to train properly. This is further examined and discussed in the next section. Because of that, the Sent2Vec algorithm is not applicable in situations when a large training dataset is not available or when the publicly available general pre-trained model is too large for the intended purpose.

The CBOW variant of the Word2Vec algorithm was falling far behind the Skip-Gram variant. The algorithm also showed a minor boost in performance after utilizing the hierarchical softmax instead of the negative sampling approach. This was expected, since the hierarchical softmax is able to update all weights in the neural network for each training step, while negative sampling updates only a small subset. However, even though computing the hierarchical softmax for each training sample is slower than using the negative sampling, it can be seen from the graph, that even when compared based on the training time, the hierarchical softmax is still superior in terms of performance on the STS task.

3.5 Training on the C4Corpus dataset

In Section 3.4, the embedding algorithms were examined based on their ability to train on a small (but fixed size) dataset, where each algorithm was trained until its performance converged. On the other hand, the algorithms can be compared in terms of their ability to train on datasets of varying sizes, while fixing the number of data passes (i.e. epochs), effectively creating learning curves of the embedding algorithms.

In this section, the algorithms are trained on the C4Corpus dataset and its subsets. The algorithms were set to perform only two epochs for each subset of the data. This was done in order to rule out possible optimization differences in the used implementations of the embedding algorithms. Even though a proper performance comparison should be done based on the training times of the algorithms, this comparison was already done in Section 3.4 and such comparison is not the goal of the experiments performed in this section. Constraining the algorithms by the training time could also force some of the algorithms not to finish a full data pass, rendering the learning curves incorrect. This section therefore assumes that the size of the training corpus affects only the best performance of the embedding algorithms and not their convergence abilities.

The training hyperparameters of the embedding algorithms were set the same as for the experiments in Section 3.4, with a few exceptions:

- 10 negative samples per training sample were used for the negative sampling. Even though it is counter-intuitive to increase the number of negative samples for a larger training corpus, this was done to match the hyperparameters of the pre-trained FastText model.
- The algorithms were trained on 10 threads of an *Intel Xeon E5-2690 v4 2.6 GHz* CPU.

3.5.1 Results and discussion

The performance of the trained models on the STS task is shown in the graph in Figure 3.3. The exact results together with the training times are also presented in Table D.2.

The graph in Figure 3.3 shows that the performance of the Sent2Vec algorithm grows significantly with the amount of data used. Starting on par with Word2Vec CBOW with only 1% of the dataset (4.4 million training tokens), the performance of Sent2Vec gradually surpasses all of the word embedding algorithms, with the exception of StarSpace, which still achieved better performance than all of the other algorithms.

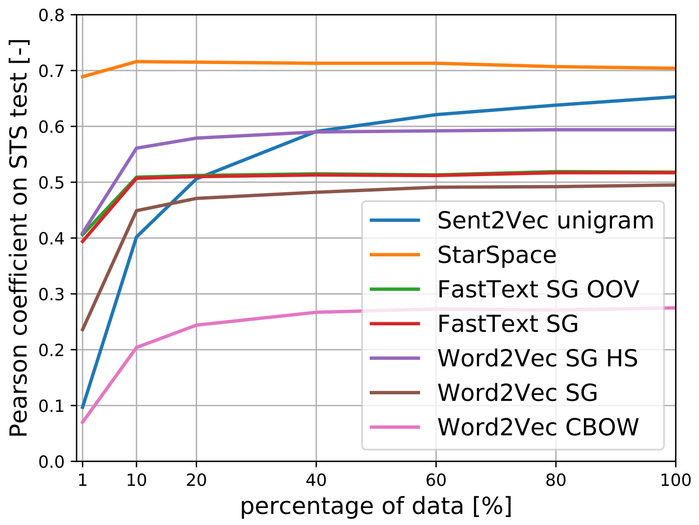


Figure 3.3. Learning curves for the models trained on different percentages of the C4Corpus dataset.

The results of the word embedding algorithms are consistent with the experiments in Section 3.4. All the word embedding algorithms were able to achieve their best performance on the STS task with a relatively small amount of data compared to the Sent2Vec algorithm, with FastText and StarSpace needing only 10% (45 million training tokens) to converge. The performance of the FastText algorithm with and without constructing the out-of-vocabulary word embeddings became almost identical, which suggests that the vocabulary of the C4Corpus dataset sufficiently covered the tokens contained in the STS Benchmark testing set.

The results clearly show that two epochs are not enough for the embedding algorithms to achieve their best possible performance (with the current hyperparameters). The algorithms were therefore also trained on the full dataset with an increasing number of epochs until their performance on the STS Benchmark testing set stopped improving.

The results presented in Table 3.5 show that the performance of StarSpace and the CBOW variant of Word2Vec improved by roughly 10% over the models trained on the STS Benchmark training dataset. Other than that, using a significantly larger dataset than in Section 3.4 did not improve the performance of the word embedding models on the STS task. In case of the hierarchical softmax variant of the Word2Vec algorithm, the performance actually dropped slightly. However, this was probably caused by the fact, that the training and testing sets of the STS Benchmark dataset have for the

algorithm	epochs	CPU time [minutes]	Spearman	**Pearson** (PR 2 epochs)
Sent2Vec unigram	10	7304	0.691	0.709 (0.653)
StarSpace	1	7051	0.673	0.708 (0.704)
FastText SG OOV	10	2777	0.582	0.582 (0.518)
FastText SG	10	2777	0.581	0.581 (0.517)
Word2Vec SG HS	4	1283	0.590	0.597 (0.594)
Word2Vec SG	20	3865	0.572	0.569 (0.495)
Word2Vec CBOW	17	751	0.311	0.309 (0.275)

Table 3.5. Performance of the models trained on the C4Corpus after reaching convergence. The models were trained repeatedly with an increasing number of epochs until their performance on the STS Benchmark testing set stopped improving. The Pearson correlation coefficients for 2 training epochs are also shown for comparison.

most part very similar vocabularies, not leaving much space for improvement on larger datasets.

On the other hand, the performance of the Sent2Vec algorithm significantly increased, slightly surpassing even the StarSpace algorithm. This shows that Sent2Vec requires much larger dataset than the word embedding algorithms in order to train properly. It still achieved worse performance than the public models trained on Toronto Books and Twitter corpora (0.726 and 0.755 Pearson coef. respectively), as seen in Table 3.4, showing that the Sent2Vec algorithm still has room for improvement on even larger datasets. The performance of the word embedding models trained on the C4Corpus was satisfactory and roughly on par with the pre-trained models.

3.6 CPU and memory requirements

In this section, the memory and CPU requirements of the embedding algorithms during both training and testing (i.e. sentence encoding) are discussed. The resource requirements during the training of the models are not as important for the purpose of this work, nevertheless, they are shown to provide the full overview of the selected algorithms.

Since the intent recognition module developed in this thesis will be used in a question-answering system, the module needs to be able to transform the sentences it receives in a reasonable time. The CPU requirements during the computation of the sentence embeddings are therefore an important factor in deciding which of the algorithms will be used in the resulting system. The memory requirements of the uncompressed models are also discussed, since the resulting module can be used on mobile devices with a limited RAM.

3.6.1 Training complexity

Taking the Word2Vec algorithm as a baseline for most of the other embedding algorithms, first, the training complexity of Word2Vec will be discussed, before comparing it to the CPU requirements of the other used algorithms.

As stated in Mikolov et al. [3], the training complexity O for Word2Vec can be expressed as

$$O = E \times |\mathcal{T}| \times Q, \tag{3.1}$$

where E denotes the number of training epochs, $|\mathcal{T}|$ denotes the number of tokens (words) in the training dataset, and Q differs by the algorithm used. For CBOW it is

$$Q = H \times (C + \log_2 |\mathcal{V}|), \tag{3.2}$$

where H is the embedding dimension, C is the context window size, and $|\mathcal{V}|$ denotes the size of the vocabulary. For Skip-Gram,

$$Q = H \times (C + C \times \log_2 |\mathcal{V}|), \tag{3.3}$$

where C now denotes the maximum context window size.

Bojanowski et al. [5] state, that the FastText algorithm is roughly 30% slower to train than Word2Vec on the same amount of training data for character n-grams with n ranging between 3 and 6. This is due to the extra time needed to train the embeddings for the character n-grams. The training times on the C4Corpus dataset presented in Table 3.6 show that the proportional training time difference between Word2Vec and FastText decreases with increasing amount of training data. This is to be expected, since the number of distinct character n-grams stops increasing at some point, while the vocabulary still grows.

Sent2Vec in the word unigram variant can be seen as an extension of Word2Vec CBOW. The only difference that should affect the CPU usage is the dynamic context window in Sent2Vec, which stretches over the whole sentence. Mikolov et al. [3] states, that they managed to get the best performance from Word2Vec on a word similarity task for the context window of four (i.e. four history and four future words are taken as the context for each word). This context window size was also used in the experiments in sections 3.4 and 3.5. As an average sentence length in written English ranges roughly between 15 and 25 words, depending on the text source[1]), which translates to a context window in the interval $[7, 12]$, we can say that the training time of Sent2Vec compared to Word2Vec CBOW should increase by approximately 75-200%. The experiments on the C4Corpus show roughly 160% CPU time increase over the Word2Vec algorithm for the full dataset, which is consistent with this theory. Again, the time difference decreases with an increasing amount of data used for training.

The StarSpace algorithm, while achieving the best results of the word embedding algorithms, required by far the largest amount of time to complete each training epoch. Even when the algorithm was set to use the dot product and the softmax function instead of the margin ranking loss, the training time improved only by approximately 10%. This could be caused by the fact, that StarSpace was not optimized to learn word embeddings in an unsupervised manner. Otherwise, the training times of StarSpace should be in theory on par with the other word embedding algorithms.

algorithm	100% data	80% data	20% data
Sent2Vec unigram	1310	1067	319
StarSpace	14074	11425	3074
FastText SG	592	519	140
Word2Vec SG	507	389	75
Word2Vec CBOW	87	76	22

Table 3.6. Training times of the embedding models trained on the C4Corpus dataset in minutes. The times were recorded for two data passes (i.e. epochs). The full results are presented in Table D.2.

[1]) `https://top.quora.com/How-long-is-the-average-sentence`

The memory requirements during the training of the embedding algorithms mostly consist of loading the training corpus and storing the weight matrices. The memory needed for the training corpus can be significantly reduced by loading the dataset in batches, instead of keeping it loaded as a whole. The RAM requirements are therefore dominated by the need to store the weight matrices, which are constantly getting updated. For most of the algorithms, this consists of storing two matrices of $|\mathcal{V}| \times H$ double precision numbers, with the FastText algorithm needing additional space for storing embeddings of the character n-grams. During the training of the algorithms on the C4Corpus dataset, FastText usually required roughly 50% more memory than the other embedding algorithms.

3.6.2 Text encoding complexity

The models of all the word embedding algorithms come as a text file (or binary for reduced size and faster loading), where each row contains a token (word) and its vector representation. The tokens are usually sorted by the frequency in the training corpus, with the most frequent tokens being on top of the file. The memory requirements during the encoding phase are dominated by the need to load these files into RAM to have a fast access to the word embeddings. The same applies to the two sentence embedding algorithms, but InferSent requires to additionally load the parameters of the BiLSTM network.

During a sentence encoding, all of the algorithms except for InferSent first need to access the word vector entries for each word in the given sentence. The sentence embedding is then computed as an average over the word vectors, requiring

$$(|S| - 1) \times H + H = |S| \times H \tag{3.4}$$

floating point operations, where $|S|$ is the number of words in the encoded sentence, and H is the embedding dimension.

As seen in Table 3.7, all of the word embedding algorithms together with Sent2Vec take roughly the same amount of time to encode the sentences present in the STS Benchmark dataset, which is consistent with the theory. The FastText algorithm in the example required roughly twice the amount of time when constructing the out-of-vocabulary word embeddings, compared to the case when the OOV words were skipped. This should, however, be heavily dependent on the amount of OOV words in the encoded dataset.

algorithm	time [s]
InferSent	316.81
Sent2Vec unigram	1.59
StarSpace	1.88
FastText OOV	4.58
FastText	2.20
Word2Vec	2.40

Table 3.7. Comparison of the embedding algorithms based on the time needed to encode the whole STS Benchmark dataset, consisting of over 17 thousand sentences (cf. Table 3.1). The times recorded are for the models trained on the STS Benchmark training set, except for InferSent, which uses the pre-trained model. The individual variants of Word2Vec and FastText are not distinguished here, as they should have no effect on the encoding complexity.

Table 3.7 clearly shows that the InferSent algorithm requires significantly more time to encode the sentences than the other embedding algorithms. This is caused by the fact, that while the other algorithms construct the embeddings by simply looking up the word vectors in the trained model, InferSent additionally needs to use the sentence embeddings (constructed for example from the GloVe model) as an input to the bi-directional LSTM network encoder and compute the resulting sentence embeddings in the network. The LSTM network also outputs embedding vectors of much higher dimension (e.g. Conneau et al. [10] used 4096-dimension vectors) than the other embedding algorithms, requiring more floating point operations for any further task. This makes the InferSent algorithm unsuitable for the intent recognition module developed in this thesis.

Chapter 4
Model compression

The general embedding models trained on large corpora tend to have several gigabytes in size (min. 1 GB, max. 23 GB, cf. Table 3.3 for details), making them unsuitable for mobile applications, where both the storage memory and RAM are limited. The larger the model is, the longer it also takes to load into RAM before it can be used for creating the sentence embeddings. This is not a problem for a server-based service, since the model is only loaded once and is used as long as the server is running. However, when used on the client side, this would result in tedious waiting times for every launch of the application. Even the smaller models designed for tasks on a constrained domain are usually several hundred megabytes large, which is still notable size for a mobile device. It is therefore important to reduce the size of the models as much as possible.

This chapter explores two embedding model compression methods, namely vocabulary pruning and vector quantization. These compression methods were inspired by Joulin et al. [16].

4.1 Data and evaluation

The experiments described in the following sections were all performed on a publicly available English Wikipedia FastText model, which was stripped of the character n-gram embeddings for simplicity. The experiments were performed using FastText only, since it is a core component of Sent2Vec and StarSpace. Further, as all the embedding algorithms are very similar in nature, it is reasonable to assume that the compression process will perform the same for any other embedding model.

The compressed models were evaluated on the STS Benchmark testing dataset in the same way as in the previous chapter (see Section 3.2 for details). Because the vocabulary of the STS Benchmark testing set contains approximately only 12 500 words, the FastText model used for experiments was further stripped to only 200 thousand most frequent words (i.e. top 200 thousand entries in the embedding model file) for time purposes. This reduced the Pearson correlation coefficient on the testing dataset by only 0.002 (cf. Table 4.1), suggesting that the dataset did not contain most of the discarded words.

model	dim	vocab	size	Pearson
FastText without n-grams	300	2.5 mil.	6.6 GB	0.584
+ only first 200k words	300	0.2 mil.	523 MB	0.582

Table 4.1. Information about the pre-trained FastText model used for the compression experiments. The model was stripped of its character n-gram embeddings and further reduced to only 200 thousand most frequent words. The reported performance (Pearson correlation coefficient) was computed on the STS Benchmark testing set.

4.2 Vocabulary pruning

The size of an embedding model grows linearly with the size of the vocabulary, each new word adding H double precision numbers to the model. For most of the constrained-domain tasks, it is pointless to keep all of the millions of words that the public pre-trained models contain. It is therefore useful to prune the vocabulary based on the specific task. This section explores two methods of vocabulary pruning — either based on the frequency of the words in the training corpus or based on the norm of the embedding vectors.

Pruning **based on the frequency** of the words in the original training corpus of the embedding model is probably the most intuitive approach to vocabulary pruning. It is also by far the simplest approach to perform, since the words in the embedding model files are by convention sorted by their occurrence count in the training dataset. Creating a new model with only K most frequent words therefore becomes as simple as selecting K top words from the model file. This was actually already performed in Section 4.1, where the first 200 thousand words were taken from the FastText model, effectively reducing the size of the model by over 90% without any notable performance drop on the given STS task (cf. Table 4.1).

On the other hand, Joulin et al. [16] propose a different approach of pruning the vocabulary, which is **based on the Euclidean norm** of the embedding vectors. While the frequency pruning will probably keep a lot of insignificant words like "the" or "is" (i.e. stop words), keeping K words with the highest embedding norms should preserve the most discriminative words (i.e. the words with the highest importance in the training dataset). This can, however, pose a problem in the case that the training dataset includes samples that would not contain any of the K highest-norm embeddings. An additional constraint is therefore needed, which forces the algorithm to keep at least one word from each of the training samples. The problem can be therefore formally defined as

$$\max_{\mathcal{V}_P \subseteq \mathcal{V}} \sum_{w \in \mathcal{V}_P} \|\mathbf{v}_w\|_2 \quad \text{s.t.} \quad |\mathcal{V}_P| \leq K, \quad P\mathbf{1}_{\mathcal{V}_P} \geq \mathbf{1}_{\mathcal{T}}, \tag{4.1}$$

where $\mathcal{V}_P$ represents the pruned vocabulary as a subset of the original vocabulary $\mathcal{V}$, $\mathbf{v}_w$ is the embedding vector of the word w, and P is a matrix such that $P_{jk} = 1$ if k-th feature is in the j-th training sample.

For this purpose, Joulin et al. [16] utilize a simple greedy strategy. For each sample in the training dataset (i.e. sentence or document), if $\mathcal{V}_P$ does not contain any of the tokens in the training sample, the token with the highest norm is added to $\mathcal{V}_P$. After traversing all training samples, if $|\mathcal{V}_P| < K$, more tokens with the highest embedding norm are added independently of the training dataset.

4.2.1 Results and discussion

The STS Benchmark training set was used for the pruning based on vector norms to perform the pruning on the respective domain, since its testing set was used for the performance evaluation of the compressed models. The constraint for keeping the highest norm vectors was therefore to cover the whole STS Benchmark training set, i.e. at least one token had to be kept for each sentence from the set. This translated into the smallest number of tokens kept to exactly $K = 3413$ for the used training set.

The performance comparison of the two pruning methods can be seen in Figure 4.1. The exact results together with the sizes of the compressed models are presented in Table 4.2.

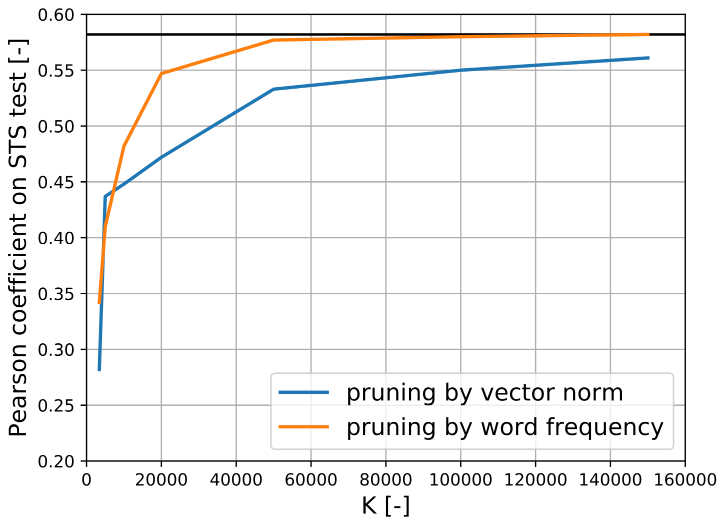


Figure 4.1. Performance of the models with pruned vocabulary on the STS Benchmark testing set. The black line in the graph represents the performance of the original model (PR = 0.582).

K	size [MB]	Spearman (norm)	Spearman (freq)	**Pearson** (norm)	**Pearson** (freq)
3 413	9	0.419	0.395	0.282	<u>0.342</u>
5 000	13	0.460	0.452	<u>0.437</u>	0.410
10 000	26	0.475	0.502	0.448	<u>0.482</u>
20 000	53	0.490	0.550	0.472	<u>0.547</u>
50 000	132	0.534	0.572	0.533	<u>0.577</u>
100 000	263	0.549	0.576	0.550	<u>0.580</u>
150 000	393	0.557	0.578	0.561	<u>0.582</u>

Table 4.2. Results of the vocabulary pruning experiments. “Norm” and “freq” represent the vector norm and the word frequency pruning variants, respectively. The reported performance (Pearson correlation coefficient) was computed on the STS Benchmark testing set. The size of the original model was 523 MB with a Pearson coefficient of 0.582.

Figure 4.1 clearly shows that pruning based on the word frequency achieves much better performance on the STS task than pruning based on the vector norm. Even though the full original model was already stripped to only the first 200 thousand words before the experiments, further discarding another 150 thousand least frequent words from the model did not noticeably reduce the performance on the STS Benchmark dataset.

Further, to strictly follow Joulin et al. [16], the model should be trained and pruned on the same domain. The FastText model, on which the experiments presented in Figure 4.1 were performed, was trained on English Wikipedia, while the pruning is performed based on the STS Benchmark training set. These two datasets might put emphasis on different words in the sentences and because of that, the embedding vector norms from one dataset might not match the importance of words from the other. A FastText model trained on the STS Benchmark training set, previously trained in Section 3.4, was therefore used to verify the results.

K	size [MB]	Spearman (norm)	Spearman (freq)	**Pearson** (norm)	**Pearson** (freq)
1 513	4	0.416	0.410	0.319	<u>0.356</u>
5 000	13	0.414	0.492	0.342	<u>0.461</u>

Table 4.3. Results of the vocabulary pruning experiments on a FastText model trained on the STS Benchmark training dataset. The model used was previously trained in Section 3.4, specifically the one trained for 64 epochs. The size of the original model was 33 MB with a vocabulary of roughly 12 500 tokens and a Pearson coefficient on the STS Benchmark dataset of 0.511.

Table 4.3 reveals that thanks to unifying the domain of the training sets, the vocabulary pruning based on vector norms did require fewer words for covering all the sentences in STS Benchmark training dataset (1 513 compared to the 3 413 tokens needed in the previous experiment). However, the simpler approach of pruning based on the word frequency still brought better results on the STS task. It is important to note, that Joulin et al. [16] tested the compression methods on a FastText model trained for a document classification task (cf. Joulin et al. [6]). Therefore, the reason for the poor performance of the norm-based pruning might be that while it is suitable for the document classification task, where a single word can be descriptive enough for the document to be correctly classified, it fails on the semantic similarity task, where it throws out the important syntactic features of the compared sentences.

4.3 Quantization

Vector quantization is a common technique in the domain of signal processing, which can also be used for lossy data compression. The basic idea behind vector quantization is to use a clustering algorithm to find groups of close datapoints (vectors) in the original dataset and approximate those groups (clusters) by their centroids. The set of centroids is referred to as the *codebook* and it is typically significantly smaller than the original dataset. The vectors are then replaced by the index of their closest centroid in the codebook, which greatly reduces the size of the model.

However, if used in the basic form on a high-dimensional embedding model, vector quantization would replace each embedding vector with a single centroid, effectively grouping the words in the vocabulary by their semantics and replacing them with an "average" word. This would greatly reduce the performance on the STS task, unless a large number of centroids was used, which would render the compression useless. Instead, the embedding vectors are first split into *sub-vectors* of dimension $D_{\mathrm{SV}} \ll H$, which are then used as an input to a clustering algorithm in order to fit D_{CB} centroids to the sub-vectors. The original embeddings are therefore replaced not by one index, but by a sequence of centroid indices based on the sub-vectors the embedding vector contains. The compressed vector can then be decoded by simply traversing the sequence of indices and concatenating the respective codebook entries. A visualization of the vector quantization process is shown in Figure 4.2, with a specific example shown in Figure 4.3.

Both the size of the sub-vectors D_{SV} and the amount of the centroids to fit (i.e. the size of the codebook) D_{CB} are hyperparameters that influence the compression rate and the quality of the compressed embeddings. Intuitively, the performance of the compressed models should be better with larger D_{CB} and smaller D_{SV}, since it will increase the approximation precision. At the same time, however, the compression rate

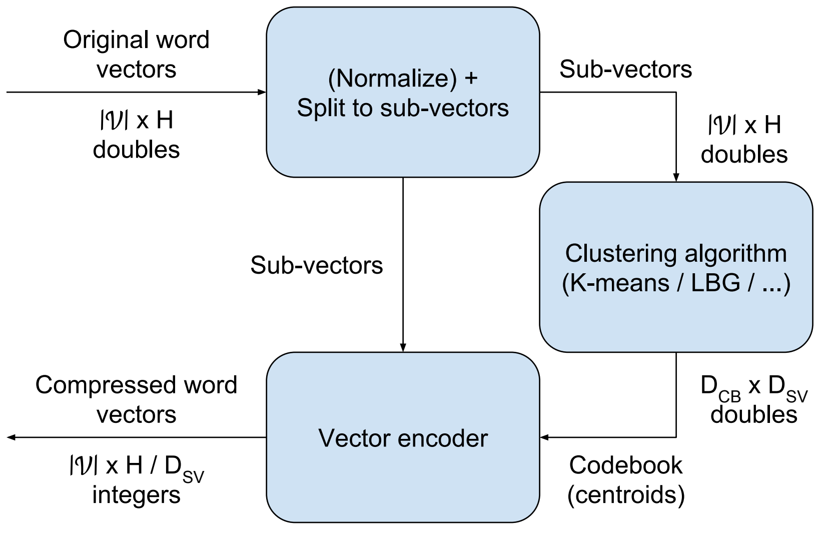
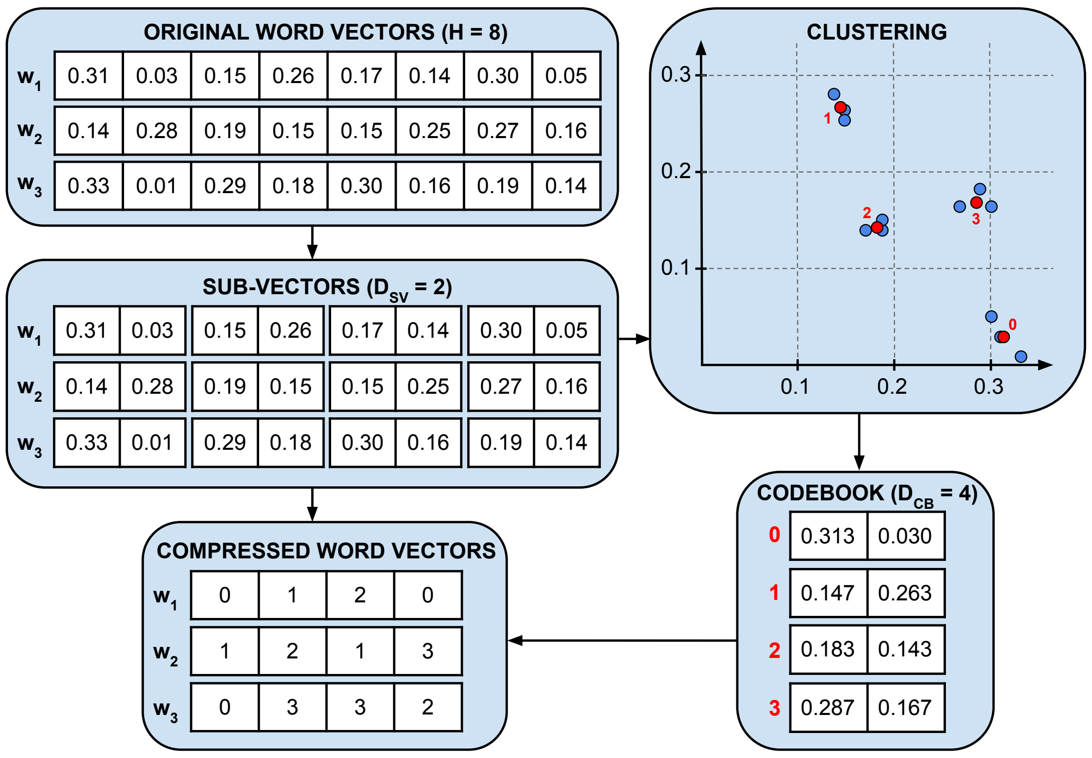

Original word vectors
$|\mathcal{V}|$ x H doubles
(Normalize) + Split to sub-vectors
Sub-vectors
$|\mathcal{V}|$ x H doubles
Clustering algorithm (K-means / LBG / ...)
Sub-vectors
D_{CB} x D_{SV} doubles
Compressed word vectors
$|\mathcal{V}|$ x H / D_{SV} integers
Vector encoder
Codebook (centroids)

Figure 4.2. Scheme of the vector quantization process. $|\mathcal{V}|$ is the number of words in the vocabulary, H is the embedding dimension, D_{SV} is the size of the sub-vectors and D_{CB} is the codebook size (i.e. the number of centroids).



ORIGINAL WORD VECTORS (H = 8)

w_1	0.31	0.03	0.15	0.26	0.17	0.14	0.30	0.05
w_2	0.14	0.28	0.19	0.15	0.15	0.25	0.27	0.16
w_3	0.33	0.01	0.29	0.18	0.30	0.16	0.19	0.14

SUB-VECTORS (D_{SV} = 2)

w_1	0.31	0.03	0.15	0.26	0.17	0.14	0.30	0.05
w_2	0.14	0.28	0.19	0.15	0.15	0.25	0.27	0.16
w_3	0.33	0.01	0.29	0.18	0.30	0.16	0.19	0.14

CODEBOOK (D_{CB} = 4)

0	0.313	0.030
1	0.147	0.263
2	0.183	0.143
3	0.287	0.167

COMPRESSED WORD VECTORS

w_1	0	1	2	0
w_2	1	2	1	3
w_3	0	3	3	2

Figure 4.3. Example of a vector quantization process for an embedding model with only three words. The blue circles in the graph denote the sub-vectors, the red circles denote the cluster centroids. The output of the algorithm is the set of compressed word vectors together with the codebook.

will decrease, because the sequence of indices that describes the embedding vectors will get longer. The size of the codebook should always be negligible compared to the compressed embedding model.

The choice of the clustering algorithm used for creating the codebook is also essential for the quality of the compressed model. Joulin et al. [16] utilize the k-means algorithm, which is one of the most common clustering algorithms. However, in this work, the more complex Linde-Buzo-Gray (LBG, Linde et al., 1980 [17]) algorithm was used instead, as it should improve the quality of the fitted centroids. The centroid splitting property of the LBG algorithm also suits the purpose of fitting the codebook indices into different-size integers (e.g. into an 8-bit integer if $D_{\mathrm{CB}} \leq 256$). A simplified pseudo-code of the LBG algorithm can be written as follows:

```
LBG(data, CB_size)
  INIT: CB = [MEAN(data)]
  while length(CB) < CB_size:
     CB_new = []
     for each centroid c in CB:
        CB_new += [c * (1 + eps), c * (1 - eps)]
     CB = K-MEANS(data, CB_new)
  return CB
```

Another important difference to the quantization approach used by Joulin et al. [16] is that they incorporate the algorithm into the training process of the FastText algorithm, which lets the neural network adapt to the quantized networks, bringing a minor boost in performance of the compressed model. The quantization experiments in this thesis are however performed on already trained models.

There are several extensions to the quantization algorithm that can further improve its performance. In this thesis, two approaches are explored: normalizing the embedding vectors before the clustering and using a distinct codebook for each sub-vector position in the embedding vectors.

Since the norms of the embedding vectors tend to vary by a large margin (e.g. the norms in the used FastText model range between 1 and 66), they can bring a severe bias to the clustering algorithm. **Normalizing the embeddings to unit length** before splitting them to sub-vectors can solve this problem. The original norms of the vectors are then stored in the compressed model, and the vectors are then simply multiplied by them during the decoding process. Joulin et al. [16] suggest to further compress the stored norms by a separate quantizer. However, as the size increase of the compressed model by adding one double precision number to each word is negligible, the norms were not quantized in the following experiments.

The second extension of the quantization algorithm is to create a **separate codebook for each sub-vector position**. This should improve the accuracy of the approximation in case that the embedding vectors are not homogeneous across their dimensions, which would result in highly distinct sub-vectors.

4.3.1 Results and discussion

A new quantization module was implemented from scratch in the *Python* programming language. The LBG algorithm was also implemented, inspired by a *Python* implementation available on GitHub[1]), while optimizing the algorithm using the *NumPy* package. Thanks to the changes, the optimized LBG algorithm achieved a speedup of over 50 times compared to the available implementation. However, even with the significant speedup, the LBG algorithm was unable to fit the centroids in a reasonable time, when all of the 200 thousand embedding vectors (i.e. up to 30 million sub-vectors for

[1]) `https://github.com/internaut/py-lbg`

$D_{\mathrm{SV}} = 2$) were used as an input. A random sample of 10 thousand vectors was therefore selected from the embedding model, from which the centroids were computed. This set of vectors was fixed for the purpose of the experiments.

First, the **basic vector quantization** algorithm without normalizing the embedding vectors was examined based on the sub-vector and codebook sizes. The results of the experiment can be seen in Figure 4.4. The exact results together with the sizes of the compressed models are also presented in Table E.3.

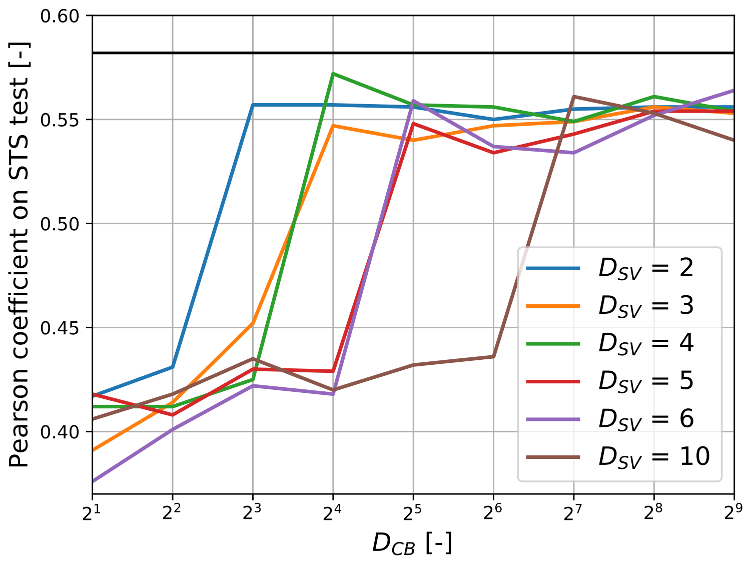


Figure 4.4. Performance of the basic vector quantization (i.e. without normalizing the embedding vectors or creating distinct codebooks for each sub-vector position) on the STS Benchmark testing set. The black line in the graph represents the performance of the uncompressed model (PR = 0.582).

Figure 4.4 shows that each sub-vector size required a certain amount of codebook entries, for which the performance of the resulting compressed model significantly increase. After this point, the performance of the models did not further improve with larger codebooks. As expected, the required number of centroids increases with the size of the sub-vectors, with the $D_{\mathrm{SV}} = 2$ model requiring only 8 (2^3) codevectors, while the $D_{\mathrm{SV}} = 10$ model requiring 128 (2^7) codevectors. However, since the size of the codebook is negligible compared to the compressed embedding model, the model with sub-vectors of size 10 achieves its best performance with a better compression rate than the other models. The resulting model for $D_{\mathrm{SV}} = 10$ and $D_{\mathrm{CB}} = 2^7$ is only 21 MB large (cf. Table 4.4), which translates to a size reduction of almost 96% of the original 523 MB model.

It is important to note, that any codebooks of the size smaller than 8 (2^3) will not further reduce the size of the resulting models, since a model with $D_{\mathrm{CB}} = 8$ already uses only one-digit indices. Such models were however included in the comparison, since it brings an interesting insight into how the compressed models will perform under such circumstances. The codebook with less than 256 (2^8) centroids will also not reduce the RAM requirements of the resulting models, since the CPUs cannot address anything smaller than a byte, which makes storing the centroid indices in anything smaller than an 8-bit integer impossible.

Next, the same experiment was performed, but with the **embedding vectors being normalized** to unit length, before passing them to the LBG algorithm. The results of this experiment can be seen in Figure 4.5. The exact results with the sizes of the compressed models are also presented in Table E.4.

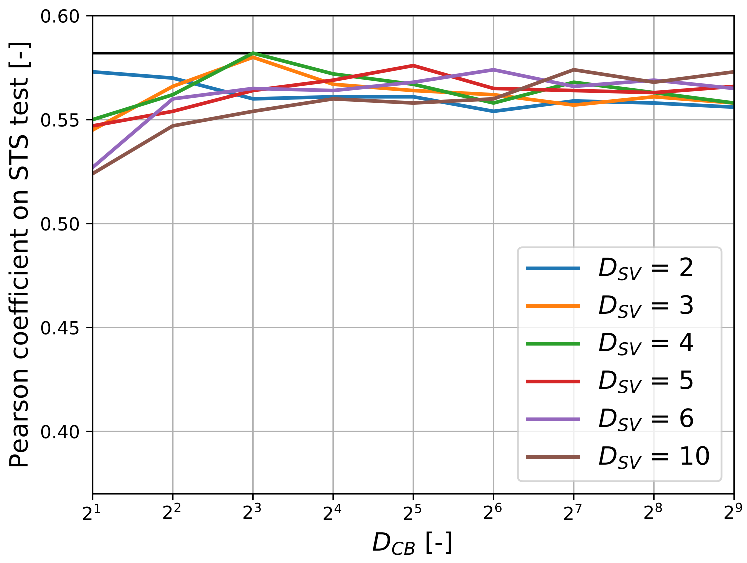


Figure 4.5. Performance of the vector quantization with normalized embedding vectors on the STS Benchmark testing set. The black line in the graph represents the performance of the uncompressed model (PR = 0.582).

Figure 4.5 shows that normalizing the embedding vectors to unit length greatly improved the performance of the resulting compressed models. The performance boost is most significant for smaller codebook sizes and longer sub-vectors, with most of the models reaching roughly constant performance when only eight (2^3) centroids were used.

Another important thing to note is that the low-performance regions for too small codebooks, which were present in the previous experiment, were almost completely eliminated by the normalization. However, each sub-vector size still shows a small performance peak around the codebook size after which its performance stopped improving in the previous experiment. These centroid numbers can therefore be considered to be the optimal settings for the respective sub-vector dimension on this particular FastText model.

The results of the models with a sub-vector size of $D_{\text{SV}} = 2$ were, however, the most interesting, since these models were able to achieve their best performance with only two centroids. Even though the compression rate of such a model is not as significant as for other models (e.g. the $D_{\text{SV}} = 10$, $D_{\text{CB}} = 2^7$ model still achieved better performance while being over 60% smaller, cf. Table 4.4), it is interesting that the embedding vectors can be replaced by a sequence of alternating only two different codevectors without any noticeable loss of performance.

The last experiment was performed for the variant of quantization algorithm in which a **distinct codebook** was created for each sub-vector position. Since normalizing the embedding vectors to unit length proved to bring a significant performance boost, the normalization was also performed for this experiment. The results can be seen in

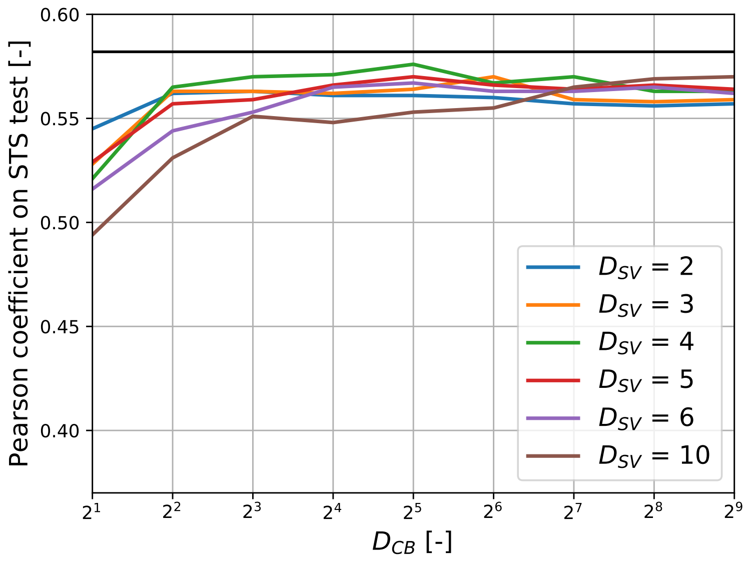


Figure 4.6. Performance of the vector quantization with distinct codebooks for each sub-vector position (and normalized embedding vectors) on the STS Benchmark testing set. The black line in the graph represents the performance of the uncompressed model (PR = 0.582).

Figure 4.6. The exact results with the sizes of the compressed models are also presented in Table E.5.

Introducing distinct codebooks for each sub-vector position into the algorithm seems to suppress the minor fluctuations in the performance of the resulting models, which can be seen in Figure 4.5. It did, however, reduce the performance of the models with smaller codebooks, while the performance of the models with larger codebooks stayed roughly equal.

Using multiple codebooks introduces higher complexity into the system, increasing both the loading times of the models and the times needed to transform text to the vector representation. Since this approach did not bring any performance improvements, it was not further used in this thesis.

quantization	size [MB]	Spearman	**Pearson**
basic	62 / 21	0.557 / 0.543	0.557 / 0.561
norm	66 / 24	0.558 / 0.570	0.560 / 0.564
norm + distinct CBs	65 / 24	0.561 / 0.568	0.563 / 0.565

Table 4.4. Comparison of selected quantized models, specifically ($D_{\mathrm{SV}} = 2$, $D_{\mathrm{CB}} = 8$ / $D_{\mathrm{SV}} = 10$, $D_{\mathrm{CB}} = 128$). The results for all quantized models are presented in the tables in Appendix E.

Chapter 5
Intent recognition module

This chapter introduces the resulting intent recognition system. Section 5.1 describes the important modules and implementation details of the system. Section 5.2 then discusses the embedding model chosen for the system and the process of evaluating its performance on an intent recognition task.

5.1 Implementation details

The whole project was developed in the *Python* programming language. In order to make it as easy as possible to switch between different embedding algorithms, a module of embedding wrappers was implemented. The wrappers handle the loading of the models and any preprocessing of the input sentences needed before transforming them to the vector representation. A special class was created for working with the quantized models.

Both the module with embedding wrappers and the module for model compression, described in Chapter 4, were the most crucial parts of the resulting system. Creating a template-based intent recognition module was then quite straightforward. The module loads the predefined template sentences from a JSON-format file, encoding them using the currently used embedding model and storing only the resulting vectors, as the original text is not needed. The intent detection process is then as simple as comparing the encoded sentences with the templates, e.g. using a cosine similarity, and selecting the intent associated with the most similar template. The system therefore utilizes the *1-NN* classification strategy.

The complexity of the intent recognition process grows linearly with the number of templates used, with the algorithm requiring to compute the cosine similarity between the query sentence and each of the templates. For a large number of templates (or intent classes), it is possible to reduce this complexity by quantizing the templates. Instead of computing the cosine similarity individually for each template, the cosine similarity can be pre-computed between the sub-vectors of the query sentence and the centroids present in the codebook of the quantized templates. This creates a matrix M of dimension $D_{\text{CB}} \times (H/D_{\text{SV}})$, where each column denotes the similarities between one sub-vector of the query sentence and all the centroids. The cosine similarity for a template is then computed by traversing its centroid indices and simply summing the corresponding entries in the matrix M.

The resulting intent recognition module together with embedding wrappers and the model compression module are also available on GitHub[1]).

5.2 Evaluation

Based on the experiments performed in this work, the StarSpace algorithm was chosen for the resulting intent recognition module, specifically the one trained on the C4Corpus

[1]) `https://github.com/Tiriar/intent-reco`

in Section 3.5. The model was compressed by selecting only first 50 thousand word vectors and quantizing it using a quantizer with $D_{\mathrm{SV}} = 10$ and $D_{\mathrm{CB}} = 128$. This reduced the model from the original 7.2 GB to only 6.1 MB.

In order to test the model on real intent recognition data, instead of the sentence similarity task, a new dataset (cf. Table 5.1) was kindly provided by the Aquist social bot team (Pichl et al., 2018 [18]), which achieved the second place in the 2017 Alexa prize social bot contest[1]). The dataset was created by collecting and manually labeling conversations of real users with the chatbot (see Table 5.2 for examples). The Alquist team also provided their implementation of an intent recognition system for comparison purposes. Their system uses a supervised convolutional network built on top of GloVe embedding vectors, which directly classifies the intent based on the input sentence embedding vector.

intent	training set	testing set	total
Change subject	205	98	303
Deep conversation	952	398	1350
Greeting	192	99	291
News	2473	1200	3673
Repeat	206	90	296
Sports	355	152	507
total	4383	2037	6420

Table 5.1. Unique sample counts per intent present in the Alquist dataset. While the full dataset contained more intents, their counts were largely unbalanced, with some intents having less than 10 samples. All intents with less than 200 samples were therefore stripped from the dataset.

Firstly, an experiment on the full Alquist dataset was carried out in order to see how the template-based system performs on the dataset with different amounts of templates. An increasing number of samples was randomly chosen from the dataset while observing the accuracy and the F1 score on the rest of the samples.

As seen from the graph in Figure 5.1, the performance on each of the intents reaches convergence when roughly 30 samples are chosen as templates. The *Greeting* intent is the only one that causes problems for the intent recognition system. This is possibly caused by the large variety of sentences present in this intent set, since apart from regular greetings, it also contains any user replies to an opening line from the Amazon Echo device (e.g. even sentences like “Who are you?”, “I am going to find you.”, etc.).

For the comparison of the system developed in this work and the system used by the Alquist chatbot, the dataset was split into training and testing parts (cf. Table 5.1). While the convolutional network was trained using the full training set, the template-based system only used a certain number of samples from the set as templates. The performance of the systems is presented in Table 5.3.

As seen from Table 5.3, when using more than 20 templates per intent, the template-based recognition system outperforms the convolutional network used in the Alquist social bot, while being less complex. Thanks to the embedding model compression, the resulting models are also very similar in size (6.1 MB for the StarSpace model, 9.9 MB for the convolutional network model).

[1]) `https://developer.amazon.com/alexaprize/2017-alexa-prize`

intent	examples
Change subject	Let's switch the subject. I don't want to talk about this anymore. I want to change to a different topic. Can we talk about anything else?
Deep conversation	Tell me something about the meaning of life. Let's chat about the human nature. Can I talk to you about death? Let's talk about love.
Greeting	Hello. / Hi. / Hey. / Greetings. Good morning, how are you? What a lovely day. How's it going?
News	Tell me about that UBER story. Give me latest news from CNN. Read me the top headlines. I want to talk about the current events.
Repeat	Repeat. / Once again. Say it again please. Alquist, can you repeat that?
Sports	Boston Celtics. / NHL. / NBA. Tell me something interesting about sports. Can we talk about football? Who won the basketball league playoffs?

Table 5.2. Examples of sentences present in the Alquist dataset.

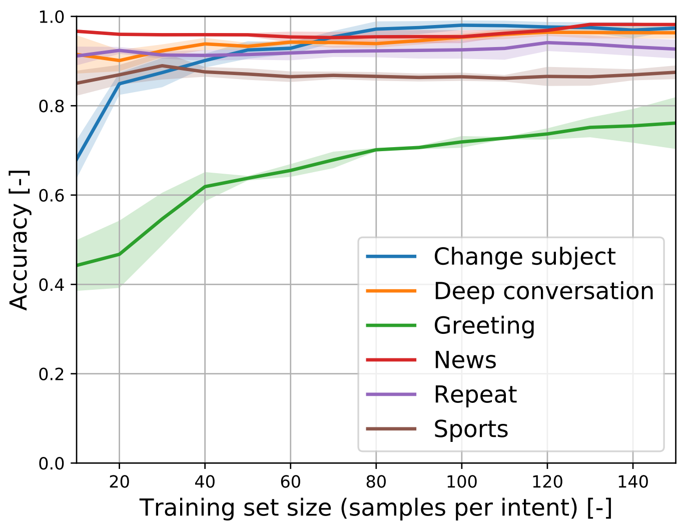


Figure 5.1. Performance of the template-based intent recognition system on the Alquist dataset with an increasing amount of samples randomly taken as the templates. The colored area around the lines denotes the standard deviation of multiple measurements.

model	accuracy	F1 score
Convolutional network	0.912	0.752
5 templates per intent	0.842	0.712
10 templates per intent	0.890	0.798
20 templates per intent	0.917	0.846
30 templates per intent	0.924	0.858
40 templates per intent	0.931	0.869
50 templates per intent	0.933	0.873

Table 5.3. Comparison of the convolutional network trained on the Alquist training set and the template-based system, which uses different amounts of templates per intent from the training set. Since a random set of templates was chosen for each run, the values in the table are averages over 10 measurements.

The biggest advantage of the template-based system is however that while the embedding algorithm requires a large amount of plain text in order train, the resulting system does not need nearly as much labeled data as the convolutional network. The only labeled samples the system requires are the ones that are then used as templates. These templates can therefore be manually written based on the specific task, making the system more transparent, with an option to easily add new intents by writing a new set of templates. The system can therefore be easily updated with new intents during runtime, while the convolutional network would need to be retrained in order to incorporate the new intents.

Chapter 6
Conclusion

This work provides a comprehensive summary of the current state-of-the-art word and sentence embedding algorithms, together with the most important techniques used to make their training on large corpora possible, in a way that makes the algorithms easier to understand and compare to one another.

The performance and the hardware requirements were examined on large datasets. The results clearly show that the StarSpace algorithm provides the best performance on the sentence similarity task, while it also takes the longest time to train, both on smaller and larger datasets. However, the sentence embedding algorithm Sent2Vec achieved performance on par with StarSpace, when trained on larger corpora. It also shows promise for improving even further with an increasing amount of training data. The experiment also verified that FastText is able to achieve its best performance with the least amount of training data compared to the other algorithms.

Further, the vocabulary pruning and vector quantization techniques were implemented in order to explore the possibility of compressing the embedding models. The experiments show that simple pruning using word frequency in the training corpus outperforms more complex pruning techniques on the semantic textual similarity task. The vocabulary pruning is able to significantly reduce the size of a general model for a more specific task (in extreme cases over 50 times). The vector quantization technique also provides significant benefits, being able to further reduce the size of the models by another 5-20 times, without any noticeable loss of performance.

Finally, the template-based intent recognition system was designed and implemented. The system uses a StarSpace model trained on a large corpus, compressed from 7 GB to only 6 MB using the developed compression module. The performance of the system was then compared to another intent recognition module, used by the Alquist social bot team (second place in the 2017 Alexa prize competition). The module developed in this thesis outperformed the convolutional network on a manually labeled set of human conversations with the Alquist social bot while being less complex and needing a significantly smaller amount of labeled data.

6.1 Further work

The experiments with the embedding algorithms performed in this thesis brought a lot of new insights, that are beyond the scope of the current work. The algorithms were compared out-of-box with the same hyperparameters. However, in the further work, one can go even deeper in this direction and truly examine the possibilities of the algorithms. It would be interesting to combine the basic features of the algorithms (e.g. using a StarSpace algorithm enriched by the character n-gram embeddings from FastText) and see the results.

The results of the experiments in Chapter 3 also showed that using the cosine similarity instead of the dot product during the training process of StarSpace improves the performance of the resulting model. The cosine similarity could therefore be also introduced into the other algorithms. However, it requires changes in a core implementation of the embedding algorithms in order to have a full control over the underlying processes.

References

[1] K. S. Jones. *A Statistical Interpretation of Term Specificity and its Application in Retrieval*, Journal of Documentation (vol. 28, pp. 11–21), 1972.

[2] J. Ramos. *Using TF-IDF to Determine Word Relevance in Document Queries*, Proceedings of the 1st Instructional Conference on Machine Learning, 2003.

[3] T. Mikolov, K. Chen, G. Corrado, J. Dean. *Efficient Estimation of Word Representations in Vector Space*, arXiv preprint, 2013. `https://arxiv.org/abs/1301.3781`.

[4] F. Morin, Y. Bengio. *Hierarchical Probabilistic Neural Network Language Model*, Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics (AISTATS) (pp. 246–252), 2005.

[5] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov. *Enriching Word Vectors with Subword Information*, arXiv preprint, 2016. `https://arxiv.org/abs/1607.04606`.

[6] A. Joulin, E. Grave, P. Bojanowski, T. Mikolov. *Bag of Tricks for Efficient Text Classification*, arXiv preprint, 2016. `https://arxiv.org/abs/1607.01759`.

[7] M. Pagliardini, P. Gupta, M. Jaggi. *Unsupervised Learning of Sentence Embeddings using Compositional n-Gram Features*, arXiv preprint, 2017. `https://arxiv.org/abs/1703.02507`.

[8] L. Wu, A. Fisch, S. Chopra, K. Adams, A. Bordes, J. Weston. *StarSpace: Embed All The Things!*, arXiv preprint, 2017. `https://arxiv.org/abs/1709.03856`.

[9] J. Pennington, R. Socher, C. D. Manning. *Global Vectors for Word Representation*, Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP) (pp. 1532–1543), 2014.

[10] A. Conneau, D. Kiela, H. Schwenk, L. Barrault, A. Bordes. *Supervised Learning of Universal Sentence Representations from Natural Language Inference Data*, arXiv preprint, 2017. `https://arxiv.org/abs/1705.02364`.

[11] S. R. Bowman, G. Angeli, C. Potts, C. D. Manning. *A Large Annotated Corpus for Learning Natural Language Inference*, Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP) (pp. 632–642), 2015.

[12] X. Rong. *Word2Vec Parameter Learning Explained*, arXiv preprint, 2014. `https://arxiv.org/abs/1411.2738`.

[13] D. Cer, M. Diab, E. Agirre, I. Lopez-Gazpio, L. Specia. *SemEval-2017 Task 1: Semantic Textual Similarity Multilingual and Cross-lingual Focused Evaluation*, Proceedings of the 11th International Workshop on Semantic Evaluation (pp. 1–

14), 2017.
`http://ixa2.si.ehu.eus/stswiki`.

[14] I. Habernal, O. Zayed, I. Gurevych. *C4Corpus: Multilingual Web-size corpus with free license*, Proceedings of the 10th International Conference on Language Resources and Evaluation (LREC) (pp. 914–922), 2016.

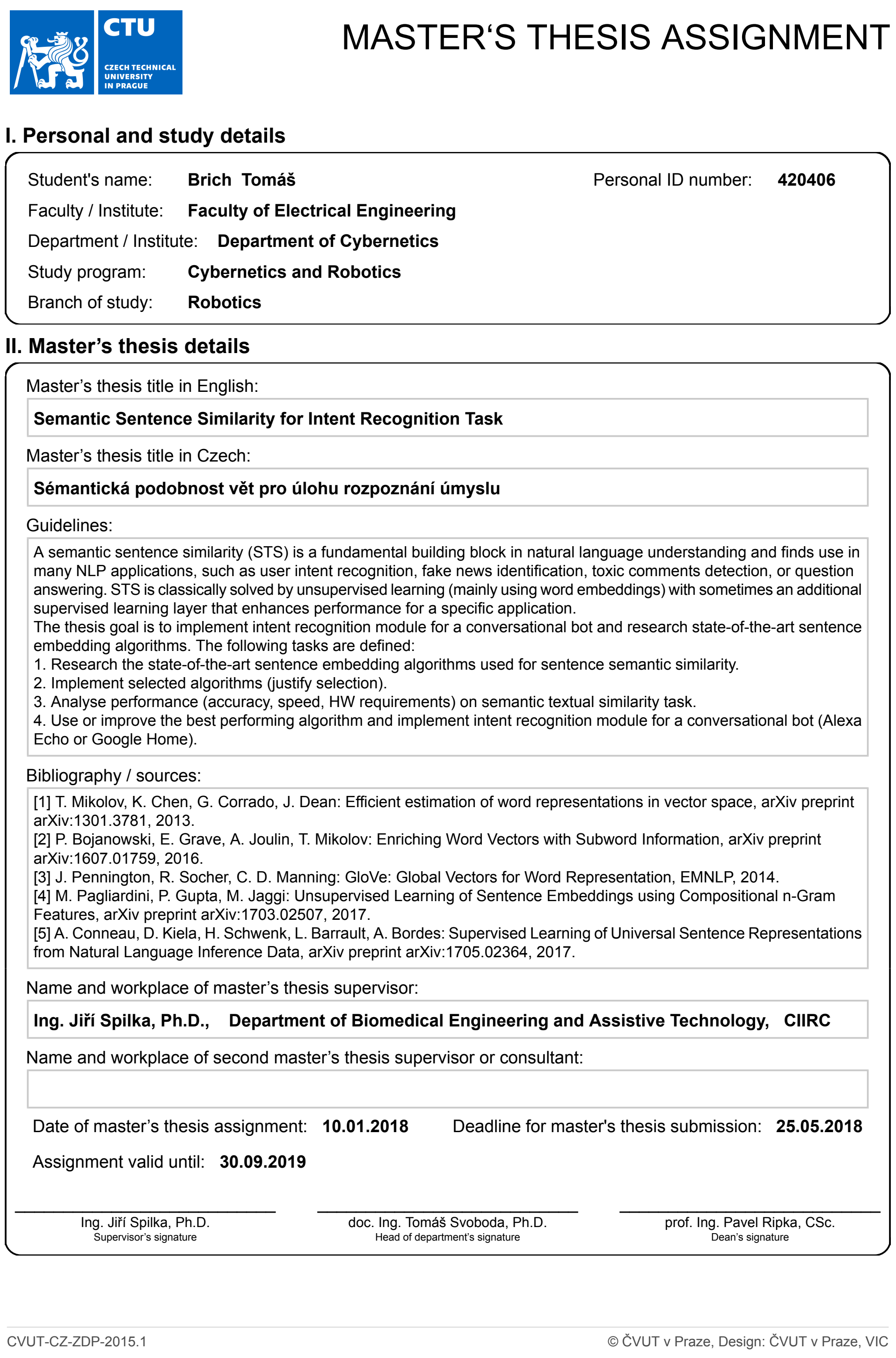
[15] T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, A. Joulin. *Advances in Pre-Training Distributed Word Representations*, arXiv preprint, 2017.
`https://arxiv.org/abs/1712.09405`.

[16] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, T. Mikolov. *FastText.zip: Compressing text classification models*, arXiv preprint, 2016.
`https://arxiv.org/abs/1612.03651`.

[17] Y. Linde, A. Buzo, R. Gray. *An Algorithm for Vector Quantizer Design*, IEEE Transactions on Communications (vol. 28, pp. 84–95), 1980.

[18] J. Pichl, P. Marek, J. Konrád, M. Matulík, H. L. Nguyen, J. Šedivý. *Alquist: The Alexa Prize Socialbot*, arXiv preprint, 2018.
`https://arxiv.org/abs/1804.06705`.

Appendix A Specification



CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Brich Tomáš**	Personal ID number:	**420406**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Cybernetics and Robotics**		
Branch of study:	**Robotics**		

II. Master’s thesis details

Master’s thesis title in English:

Semantic Sentence Similarity for Intent Recognition Task

Master’s thesis title in Czech:

Sémantická podobnost vět pro úlohu rozpoznání úmyslu

Guidelines:

A semantic sentence similarity (STS) is a fundamental building block in natural language understanding and finds use in many NLP applications, such as user intent recognition, fake news identification, toxic comments detection, or question answering. STS is classically solved by unsupervised learning (mainly using word embeddings) with sometimes an additional supervised learning layer that enhances performance for a specific application.
The thesis goal is to implement intent recognition module for a conversational bot and research state-of-the-art sentence embedding algorithms. The following tasks are defined:
1. Research the state-of-the-art sentence embedding algorithms used for sentence semantic similarity.
2. Implement selected algorithms (justify selection).
3. Analyse performance (accuracy, speed, HW requirements) on semantic textual similarity task.
4. Use or improve the best performing algorithm and implement intent recognition module for a conversational bot (Alexa Echo or Google Home).

Bibliography / sources:

[1] T. Mikolov, K. Chen, G. Corrado, J. Dean: Efficient estimation of word representations in vector space, arXiv preprint arXiv:1301.3781, 2013.
[2] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov: Enriching Word Vectors with Subword Information, arXiv preprint arXiv:1607.01759, 2016.
[3] J. Pennington, R. Socher, C. D. Manning: GloVe: Global Vectors for Word Representation, EMNLP, 2014.
[4] M. Pagliardini, P. Gupta, M. Jaggi: Unsupervised Learning of Sentence Embeddings using Compositional n-Gram Features, arXiv preprint arXiv:1703.02507, 2017.
[5] A. Conneau, D. Kiela, H. Schwenk, L. Barrault, A. Bordes: Supervised Learning of Universal Sentence Representations from Natural Language Inference Data, arXiv preprint arXiv:1705.02364, 2017.

Name and workplace of master’s thesis supervisor:

Ing. Jiří Spilka, Ph.D., Department of Biomedical Engineering and Assistive Technology, CIIRC

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **10.01.2018** Deadline for master's thesis submission: **25.05.2018**

Assignment valid until: **30.09.2019**

Ing. Jiří Spilka, Ph.D.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor’s signature	Head of department's signature	Dean’s signature

CVUT-CZ-ZDP-2015.1

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Appendix B
Contents of the attached CD

- code/	Directory containing the Python codes.
- data/	Needed data files.
- starspace_C4C_50k.txt	StarSpace model used in the resulting intent recognition system.
- starspace_C4C_50k_cb.txt	Codebook of the quantized StarSpace model.
- templates.json	Example template JSON file.
- utils/	Utility modules.
- embedding_wrappers.py	Embedding algorithm wrappers.
- lbg.py	LBG algorithm.
- utils.py	Various utility functions.
- utils_data.py	Functions for data loading and preprocessing.
- utils_plotting.py	Functions for plotting the results.
- utils_sent2vec.py	Special functions used by the Sent2Vec wrapper.
- alquist_convnet.py	Convolutional network training from the Alquist team.
- alquist_data_stats.py	Module for getting Alquist dataset statistics.
- alquist_templates.py	Module for plotting the template learning curve on the Alquist dataset.
- compare_intent_supervised.py	Module for comparing performance of supervised intent recognition systems.
- compare_intent_unsupervised.py	Module for comparing performance of unsupervised intent recognition systems.
- compare_sts.py	Module for comparing embedding algorithms on the STS Benchmark dataset.
- compare_sts_infersent.py	Module for training a regression model for InferSent on the STS Benchmark dataset.
- intent_query.py	Interactive intent recognition module.
- model_compression.py	Module for compressing embedding models.
- results.txt	File containing unprocessed experiment results.
- thesis/	Directory containing the Plain TEX thesis project.
- Brich_DP_2018.pdf	This thesis in PDF.

Appendix C
Abbreviations and symbols

C.1 Abbreviations

CBOW	Continuous Bag-of-Words model.
CPU	Central Processing Unit.
CTU	Czech Technical University in Prague.
HS	Hierarchical Softmax.
LBG	Linde-Buzo-Gray clustering algorithm.
LSTM	Long Short-Term Memory network.
NLP	Natural Language Processing.
NLTK	Natural Language Toolkit Python package.
OOV	Out-of-Vocabulary word — a word that was not present in the training corpus during the training of the embedding model.
PR	Pearson correlation coefficient.
RAM	Random-Access Memory.
SG	Skip-Gram model.
SGD	Stochastic Gradient Descent.
SNLI	Stanford Natural Language Inference.
SP	Spearman correlation coefficient.
STS	Semantic Textual Similarity.
QA	Question-Answering task.

C.2 Symbols

$\mathcal{T}$	Training dataset (corpus).
$\mathcal{V}$	Vocabulary.
w	Word in a vocabulary.
w_I/w_O	Input / output words in the neural network.
W/W'	Input / output weight matrix of the neural network.
$\mathbf{v}_w/\mathbf{v}'_w$	Vector representation of the word w in input / output weight matrix.
$\mathcal{C}_w$	Context of the word w.
C	(Maximum) context window size.
H	Embedding dimension.
E	Number of training epochs.
$\mathcal{G}$	Character n-gram vocabulary used in FastText.
$\mathcal{G}_w$	Character n-grams present in the word w. $\mathcal{G}_w \subset \mathcal{G}$.
E^+/E^-	Generators of positive and negative examples used in StarSpace.
μ	Margin parameter used in the margin ranking loss in StarSpace.
X	Co-occurrence matrix used in GloVe.
$X_{w_i w_k}$	Number of times word w_k occurs in the context of word w_i in the co-occurrence matrix X.
$P_{w_i w_k}$	Co-occurrence probability of words w_i and w_k, defined in Section 2.4.
S	Sentence from a training corpus — used in Sent2Vec.
$R(S)$	Set of word n-grams present in the sentence S.
$\mathbf{u}, \mathbf{v}$	Embeddings of a premise and a hypothesis from the SNLI corpus used in InferSent.
$\mathbf{h}_t$	Concatenation of the hidden representations in the t-th layer of the BiLSTM network used in InferSent.
$s(\cdot,\cdot)$	Function denoting similarity between two words in FastText or two entities in StarSpace, defined in the respective sections.
$r(\cdot)$	Weighting function used in the objective of GloVe, defined in Section 2.4.
$\sigma(\cdot)$	Softmax function, defined in Section 2.1.
$\ell(\cdot)$	Logistic loss function, defined in Section 2.2.
L	Loss function (used as an objective by the embedding algorithms).
$n(w,d)$	Inner node at depth d on the path from root to the word w in a tree used for hierarchical softmax.
$D(w)$	Depth of a word w in a tree used for hierarchical softmax.
$[\![\cdot]\!]$	Function used in hierarchical softmax, defined in Section 2.7.
$\mathcal{W}_{\text{neg}}$	Set of negative samples used during negative sampling.
$f(w)$	Number of occurrences (frequency) of a word w in a training corpus.
O	Time complexity function.
K	Number of words that are kept after vocabulary pruning.
$\mathcal{V}_P$	Pruned vocabulary, subset of $\mathcal{V}$.
P	Matrix such that $P_{jk} = 1$ if k-th word in the vocabulary is in the j-th training sample in the training dataset.
D_{SV}	Sub-vector size used for vector quantization.
D_{CB}	Codebook size (i.e. number of centroids) used for vector quantization.
M	Matrix of pre-computed cosine similarities between a query sentence vector and template codebook centroids.

Appendix D
Embedding models training results

epochs	Sent2Vec	W2V SG HS	W2V SG	W2V CBOW	FastText OOV	FastText	StarSpace
1	—	—	—	—	—	—	0.506 / 0.534
2	—	—	—	—	—	—	0.546 / 0.584
8	—	—	—	—	—	—	0.580 / 0.612
16	0.100 / 0.094	0.364 / 0.343	0.205 / 0.163	0.086 / 0.062	0.339 / 0.320	0.323 / 0.298	0.592 / 0.623
32	—	—	—	—	—	—	0.604 / 0.632
64	0.235 / 0.238	0.508 / 0.509	0.412 / 0.385	0.199 / 0.174	0.552 / 0.543	0.529 / 0.511	0.602 / 0.632
128	0.301 / 0.303	0.540 / 0.548	0.475 / 0.450	0.248 / 0.227	0.580 / 0.576	0.558 / 0.548	—
512	0.398 / 0.401	0.579 / 0.594	0.533 / 0.518	0.257 / 0.248	0.597 / 0.594	0.576 / 0.568	—
1024	0.431 / 0.448	0.587 / 0.605	0.552 / 0.542	0.273 / 0.265	0.601 / 0.599	0.580 / 0.572	—
2048	0.452 / 0.470	0.589 / 0.610	0.566 / 0.560	0.283 / 0.272	—	—	—
4096	0.473 / 0.478	—	0.571 / 0.569	0.285 / 0.281	—	—	—
1	—	—	—	—	—	—	1.80
2	—	—	—	—	—	—	3.33
8	—	—	—	—	—	—	12.50
16	0.58	0.70	0.65	0.25	2.03	2.03	24.69
32	—	—	—	—	—	—	46.97
64	1.37	2.67	2.40	0.73	6.78	6.78	95.68
128	2.40	5.28	4.38	1.40	12.82	12.82	—
512	8.67	21.02	15.77	5.62	50.02	50.02	—
1024	17.60	42.68	30.62	10.80	99.32	99.32	—
2048	33.55	82.63	58.90	20.37	—	—	—
4096	70.90	—	112.20	39.95	—	—	—

Table D.1. Results of the models trained on the STS Benchmark training set. The first part of the table shows the performance of the models on the STS Benchmark testing set in the form of Spearman / **Pearson** correlation coefficients. The second part then shows the respective training CPU times in minutes (trained on eight threads of an *Intel Core i7-3632QM 2.2 GHz* CPU). "W2V" stands for the Word2Vec algorithm, "SG" and "CBOW" then represent the Skip-Gram and the Continuous Bag-of-Words variants, "HS" stands for hierarchical softmax. "FastText OOV" represent the FastText model for which the out-of-vocabulary words were constructed using the character *n*-gram embeddings. The OOV words were skipped otherwise. Only the Skip-Gram variant of FastText was used in the experiments, refer to Section 3.4 for details.

data %	Sent2Vec	W2V SG HS	W2V SG	W2V CBOW	FastText OOV	FastText	StarSpace
100	0.639 / 0.653	0.585 / 0.594	0.518 / 0.495	0.296 / 0.275	0.536 / 0.518	0.534 / 0.517	0.674 / 0.704
80	0.626 / 0.638	0.583 / 0.594	0.515 / 0.492	0.294 / 0.271	0.537 / 0.519	0.535 / 0.517	0.674 / 0.707
60	0.612 / 0.621	0.582 / 0.592	0.512 / 0.491	0.297 / 0.273	0.535 / 0.513	0.533 / 0.512	0.677 / 0.713
40	0.586 / 0.591	0.581 / 0.590	0.505 / 0.482	0.296 / 0.267	0.533 / 0.515	0.531 / 0.513	0.679 / 0.713
20	0.514 / 0.506	0.568 / 0.579	0.494 / 0.471	0.274 / 0.244	0.532 / 0.512	0.530 / 0.510	0.683 / 0.715
10	0.423 / 0.402	0.555 / 0.561	0.473 / 0.449	0.237 / 0.204	0.532 / 0.509	0.529 / 0.507	0.686 / 0.716
1	0.108 / 0.097	0.423 / 0.409	0.275 / 0.236	0.084 / 0.070	0.430 / 0.406	0.422 / 0.394	0.664 / 0.689
100	1309.53	658.50	506.85	86.98	591.92	591.92	14074.33
80	1066.73	446.73	388.52	75.92	519.38	519.38	11424.97
60	837.43	337.38	240.03	66.02	373.87	373.87	8559.02
40	588.85	193.32	155.52	45.63	258.33	258.33	5616.17
20	319.22	100.47	75.23	22.48	140.02	140.02	3074.95
10	177.02	77.35	44.53	13.07	73.25	73.25	1550.20
1	14.83	5.92	4.63	1.08	8.22	8.22	151.17

Table D.2. Results of the models trained on the C4Corpus dataset. The first part of the table shows the performance of the models on the STS Benchmark testing set in the form of Spearman / **Pearson** correlation coefficients. The second part then shows the respective training CPU times in minutes (trained on 10 threads of an *Intel Xeon E5-2690 v4 2.6 GHz* CPU). "W2V" stands for the Word2Vec algorithm, "SG" and "CBOW" then represent the Skip-Gram and the Continuous Bag-of-Words variants, "HS" stands for hierarchical softmax. "FastText OOV" represent the FastText model for which the out-of-vocabulary words were constructed using the character *n*-gram embeddings. The OOV words were skipped otherwise. Only the Skip-Gram variant of FastText was used in the experiments, refer to Section 3.5 for details.

Appendix E
Vector quantization results

D_{CB} / D_{SV}	2	3	4	5	6	10
512	0.554 / 0.556	0.553 / 0.553	0.554 / 0.554	0.552 / 0.554	0.558 / 0.564	0.539 / 0.540
256	0.554 / 0.556	0.552 / 0.556	0.556 / 0.561	0.553 / 0.554	0.552 / 0.552	0.544 / 0.553
128	0.554 / 0.555	0.551 / 0.549	0.548 / 0.549	0.548 / 0.543	0.537 / 0.534	0.543 / 0.561
64	0.550 / 0.550	0.548 / 0.547	0.551 / 0.556	0.537 / 0.534	0.536 / 0.537	0.458 / 0.436
32	0.555 / 0.556	0.541 / 0.540	0.553 / 0.557	0.544 / 0.548	0.546 / 0.559	0.455 / 0.432
16	0.551 / 0.557	0.555 / 0.547	0.561 / 0.572	0.451 / 0.429	0.443 / 0.418	0.447 / 0.420
8	0.557 / 0.557	0.467 / 0.452	0.450 / 0.425	0.455 / 0.530	0.548 / 0.422	0.460 / 0.435
4	0.450 / 0.431	0.433 / 0.414	0.437 / 0.412	0.434 / 0.408	0.417 / 0.401	0.432 / 0.418
2	0.443 / 0.417	0.414 / 0.391	0.446 / 0.412	0.451 / 0.418	0.409 / 0.376	0.423 / 0.406
512	118	78	59	48	40	25
256	110	73	56	45	38	23
128	98	65	49	39	33	21
64	89	59	45	37	30	19
32	83	53	42	34	28	18
16	72	45	37	30	26	16
8	62	42	32	26	22	14
4	62	42	32	26	22	14
2	62	42	32	26	22	14

Table E.3. Results of the models created by compressing the first 200 thousand vectors in the pre-trained FastText model using the basic vector quantization. The first part of the table shows the performance of the models on the STS Benchmark testing set in the form of Spearman / **Pearson** correlation coefficients. The second part then shows the sizes of the respective models in megabytes. The size of the uncompressed model was 523 MB with the performance of 0.578 Spearman / 0.582 Pearson.

D_{CB} / D_{SV}	2	3	4	5	6	10
512	0.553 / 0.556	0.555 / 0.558	0.555 / 0.558	0.559 / 0.566	0.563 / 0.565	0.565 / 0.573
256	0.555 / 0.558	0.557 / 0.561	0.560 / 0.563	0.560 / 0.563	0.563 / 0.569	0.561 / 0.568
128	0.555 / 0.559	0.555 / 0.557	0.563 / 0.568	0.560 / 0.564	0.560 / 0.566	0.570 / 0.574
64	0.552 / 0.554	0.558 / 0.562	0.557 / 0.558	0.561 / 0.565	0.569 / 0.574	0.558 / 0.560
32	0.557 / 0.561	0.559 / 0.564	0.562 / 0.567	0.570 / 0.576	0.562 / 0.568	0.552 / 0.558
16	0.558 / 0.561	0.557 / 0.567	0.570 / 0.572	0.562 / 0.569	0.558 / 0.564	0.554 / 0.560
8	0.558 / 0.560	0.569 / 0.580	0.580 / 0.582	0.565 / 0.564	0.566 / 0.565	0.546 / 0.554
4	0.560 / 0.570	0.556 / 0.566	0.559 / 0.562	0.556 / 0.554	0.549 / 0.560	0.541 / 0.547
2	0.567 / 0.573	0.537 / 0.545	0.551 / 0.550	0.553 / 0.547	0.529 / 0.527	0.525 / 0.524
512	121	81	62	51	43	28
256	115	77	59	48	41	27
128	102	68	52	43	37	24
64	92	62	48	40	34	22
32	88	59	46	38	32	21
16	76	54	41	34	29	20
8	65	45	35	29	25	17
4	65	45	35	29	25	17
2	65	45	35	29	25	17

Table E.4. Results of the models created by compressing the first 200 thousand vectors in the pre-trained FastText model using the vector quantization with normalized embedding vectors. The first part of the table shows the performance of the models on the STS Benchmark testing set in the form of Spearman / **Pearson** correlation coefficients. The second part then shows the sizes of the respective models in megabytes. The size of the uncompressed model was 523 MB with the performance of 0.578 Spearman / 0.582 Pearson.

D_{CB} / D_{SV}	2	3	4	5	6	10
512	0.554 / 0.557	0.555 / 0.559	0.559 / 0.563	0.559 / 0.564	0.558 / 0.562	0.562 / 0.570
256	0.553 / 0.556	0.556 / 0.558	0.562 / 0.563	0.562 / 0.566	0.562 / 0.565	0.563 / 0.569
128	0.553 / 0.557	0.555 / 0.559	0.565 / 0.570	0.563 / 0.564	0.559 / 0.563	0.558 / 0.565
64	0.556 / 0.560	0.565 / 0.570	0.566 / 0.567	0.561 / 0.566	0.561 / 0.563	0.552 / 0.555
32	0.558 / 0.561	0.561 / 0.564	0.571 / 0.576	0.568 / 0.570	0.564 / 0.567	0.557 / 0.553
16	0.558 / 0.561	0.558 / 0.562	0.568 / 0.571	0.566 / 0.566	0.563 / 0.565	0.549 / 0.548
8	0.561 / 0.563	0.556 / 0.563	0.570 / 0.570	0.559 / 0.559	0.557 / 0.553	0.547 / 0.551
4	0.562 / 0.562	0.560 / 0.563	0.567 / 0.565	0.552 / 0.557	0.551 / 0.544	0.537 / 0.531
2	0.552 / 0.545	0.539 / 0.528	0.535 / 0.521	0.532 / 0.529	0.521 / 0.516	0.505 / 0.494
512	120	81	63	51	44	28
256	113	77	59	48	41	27
128	100	68	52	43	37	24
64	92	63	48	40	34	23
32	86	59	46	38	33	22
16	76	52	41	34	29	20
8	65	45	35	29	25	17
4	65	45	35	29	25	17
2	65	45	35	29	25	17

Table E.5. Results of the models created by compressing the first 200 thousand vectors in the pre-trained FastText model using the vector quantization with normalized embedding vectors and distinct codebook for each sub-vector position. The first part of the table shows the performance of the models on the STS Benchmark testing set in the form of Spearman / **Pearson** correlation coefficients. The second part then shows the sizes of the respective models in megabytes. The size of the uncompressed model was 523 MB with the performance of 0.578 Spearman / 0.582 Pearson.