Project: Swedish Tokenizer and Embedder for NLP

Victor Arvidsson
DV2586 Deep Machine Learning
Blekinge Tekniska Högskola
Karlskrona, Sweden
viar19@student.bth.se

Ahmad Al-Mashahedi
DV2586 Deep Machine Learning
Blekinge Tekniska Högskola
Karlskrona, Sweden
aham19@student.bth.se

I. Introduction

This paper presents a Swedish word embedder that is built from the ground up as an alternative solution for pre-trained multilingual transformer embedders such as the ones found in GPT-2, BERT, and other popular alternatives [1] [2]. These embedders are often trained on large datasets with different languages, however, the ratio of English to non-English vocabulary is imbalanced which leads to degraded performance when predicting on non-English vocabulary [3]. A wholly Swedish focused embedder, in theory, should yield better results on Swedish vocabulary. Our embedder uses byte-pair encoding (BPE) to better utilize space and eliminate the problem of out of vocabulary tokens. This, in addition to word2vec inspired contextual word learning using skip-grams, yielded very promising results.

II. Background

A. Byte Pair Encoding

The natural language processing (NLP) variant of byte pair encoding (BPE) is a sub-word tokenization technique that is a sort of middle ground between word and character tokenization, where the problems lie in large vocabularies with many out of vocabulary tokens or very long sentences with less meaningful tokens respectively. Byte pair encoding aims to alleviate these problems by splitting up words into sub-word tokens, for example, the word ”bytes” could be split up into ”byt” and ”es”, thus keeping some context whilst still having the form of a word [4]. The BPE algorithm first adds a boundary token ”`<\w>`” to the end of each word, then proceeds to split up the words in the corpus into individual character tokens, i.e ”`b y t e s <\w>`”. By counting the frequency of each token pair, we can select the most common pair to merge into a single token. This counting and merging is one iteration of BPE, and the merged token is added to our vocabulary, e.g. ”`b y \textbf{te} s <\w>`. This is done for a specified amount of iterations or until there are no more possible iterations left. The result is a corpus of byte-pair encoded tokens that is then used as a table for new corpuses. The big advantage of BPE is that it can manage out of vocabulary words by merging them into tokens it has previously merged through the BPE table, and if those tokens don’t exist, for example for a word in another language, it can keep the word as individual characters instead since they are added as tokens to the BPE table as well [4]. Moreover, padding and end tokens are added manually to the BPE table, the purpose of the end token is to enable the embedding model to learn contextually if the word is at the end of the sentence or not. The padding token is used in later stages by a transformer to pad the sentence vectors so that they are all the same input size.

B. Tokenizer

The tokenizer’s task is to convert the BPE tokens into numerical representations that can be fed into the embedding model. In the case of our tokenizer, it represented the tokens as index values corresponding to the same tokens in the BPE table. The final output of the tokenizer is a numerically represented corpus of tokens that will then be used for the embedder model.

C. Embedding

An embedding is a numerical mapping of the input domain to a N-dimensional space for further use in models. For example, one-hot encoding is one such embedding. Onehot-encoding represents a document as a series of binary vectors, with each vector corresponding to one word, where that one word is assigned the value one and all the other words are assigned zeros. One-hot encoding is effective but basic, it fails to capture the meaningful semantic relations between words and also within words [5]. To capture more data intrinsic to the semantics of the sentence, neural network embedders are utilized. Models such as word2Vec and GloVe use techniques that allow an embedding model to capture more meaningful data about the sentences [6], [7]. One such technique is using skip-grams [6]. The idea is to find words related to a target word, i.e., plane and sky. It works by specifying a window size which is then used to collect the context words around the target word. An example can be seen in Table I. Negative skip-grams samples are retrieved from the words that are outside of the window. For larger documents, negative skip-grams can also be constrained by a window. The generated skip-gram examples are then fed into an embedder model that is able to learn the captured data from examples and place semantically related words near each other in a N-dimensional latent space. Such a model is constructed using TensorFlow’s embedding layer. This is seen as a classification problem. The training set

TABLE I
A TABLE SHOWING THE PROCESS OF USING SKIP-GRAMS AND NEGATIVE SKIP-GRAMS ON "THE QUICK BROWN FOX", WITH A WINDOWS SIZE = 2

Target Word	Positive Skip-Gram	Negative Skip-Gram
The	(The, quick), (The, brown)	(The, fox)
quick	(quick, The), (quick, brown) (quick, fox)	()
brown	(brown, quick), (brown, The), (brown, fox)	()
fox	(fox, brown), (fox, quick)	(fox, The)

for the model is made of a target word, one positive skip-gram, and a number of negative skip-grams for the input and a one-hot encoded skip-gram array as the label, e.g., `(The, (quick, fox)), (1, 0)` where the zeroes are negative skip-grams and one is the positive skip-gram. The embedding layer is then trained for a specified amount of epochs until a desirable accuracy has been reached.

III. METHOD

In order to train the tokenizer and embedder, a corpus is needed. Språkbanken, provided by Göteborgs Universitet, provides tokenized corpuses in Swedish [8].

For small scale testing, we use the 1734 year's Swedish law. This only has 98,120 tokens, which means that development and testing will be quicker. The file format for the corpus is XML, which is loaded using the built in XML parser in Python. The words in the corpus are extracted and combined into their sentences.

For our final results, we use a corpus of the entire Swedish Wikipedia. This contains 194 million tokens. The corpus is loaded from a text file, with one token per row. This means that we treat each token as a sentence up until the tokenization step, where we join the tokens into sentences. Due to the time constraints of the project, we decided to only use 10% of the Wikipedia corpus, which resulted in a total of 19.4 million tokens.

The first step in the process, after loading the corpus, is to use Unicode NFKC normalization to ensure that all tokens characters are represented in a predictable and normalized way [9].

After loading and preprocessing the corpus, the byte pair encoding is performed. To do this, the occurrence of each word in the corpus is counted. A word boundary token is also added to the end of each word, represented by `</w>`. Each word is also separated into its individual characters by a space.

To perform the BPE, the occurrence of each pair of tokens is counted for the entire corpus, using the counting of the occurrence of each word generated in the previous step. The most frequent pair is then selected to be merged. This pair is added to the BPE-structure, and all of these pairs are merged into single tokens in the entire corpus. This process is then repeated for a specified number of iterations, which generates a number of pairs. The output from this step is a list of the tokens to merge, and in which order to merge them. The order is important, since later steps might depend on tokens being merged in previous steps.

From the BPE, a vocabulary is created, which contains all of the tokens that can be tokenized. The tokens that are out of vocabulary are all tokenized to the same value. The vocabulary is constructed by taking all printable ascii characters, excluding white space characters. The vocabulary is extended with åäö, as well as the word and sentence boundary tokens `</w>` and `[END]`. Finally, all of the BPE merged pairs are added to the vocabulary. Each token in the vocabulary also receives an consecutive integer id ≥ 2. Two special values, 0 and 1, are reserved for padding ,`<pad>`, and out of vocabulary, `<oov>`, tokens respectively.

The next step is to apply the BPE to the entire corpus and then tokenize it. Since in the previous step, the corpus was summarized by the unique words, we need to reapply the BPE to the entire corpus. We apply the BPE one step at a time to the entire corpus, merging the pairs in the BPE iteratively. Besides inserting the `</w>` word boundary token at the end of each word, a `[END]` sentence boundary token is also inserted at the end of each sentence. Once the entire corpus has been merged using the BPE, the vocabulary is used to convert each token into its integer representation.

To create a training set for the embedder to train on, we use the tokenized corpus to generate skip-grams and negative skip-grams. For each target word, we generate one positive skip-gram, and four negative skip-grams. The idea is to train the model to predict if two tokens are skip-grams of each other. This makes the model learn which tokens that are related, and places these close to each other in the embedding space.

For the embedding model, we use two embedding layers, one for the context and one for the target token. To get the output, the dot product is applied between these embedding layers.

The model is trained for a maximum of 100 epochs, with an early stopping condition when the accuracy has improved less than 0.5% over the last two epochs. The model uses the Adam optimizer and categorical cross entropy as the loss function.

To extract the embedder, we extract the weights from the target embedding layer. This gives a matrix where each row is an n-dimensional vector corresponding to the token at that index's position in the embedding space. This matrix is saved together with the corresponding tokens, and BPE rules as separate TSV files.

An overview of the parameters and metadata used for the different stages can be seen in Table II

To evaluate the embedder, we use the SuperSim v1 word relatedness data set [10]. This contains pairs of words together with a score indicating how similar they are. For each of these word pairs, for the ones that have complete tokens in our vocabulary, we calculate the cosine similarity between the two words. We then compare our calculated similarities with the labeled ones, and we expect that they should follow a linear relationship. We can validate this by using the Pearson correlation coefficient.

TABLE II
HYPERPARAMETERS AND METADATA FOR THE PROCESS.

Number of tokens	19,390,072
Number of tokens after adding sentence boundaries	19,579,155
Number of sentences	938,761
BPE iterations	10,000
Tokens in vocabulary	10,075
Skip-gram window size	1
Skip-gram negative samples per positive sample	4
Examples in training data set	15,810,050
Batch size	8,192
Embedding dimension	500

IV. RESULT

The results are presented for the training on the Wikipedia corpus.

A. Training time

All of the experiments were run on an 8-core, 16-threads, CPU with 64GB RAM. For the training time, there were multiple steps in the process that took a significant amount of time. The presented times are after optimization. Loading the data set took about 1-2 minutes. Performing the BPE iterations took 14 hours, while applying the BPE to the data set took 7 hours. Generating the skip-grams for the training data takes 7 minutes, and training the model takes 20 minutes.

B. Embedding model

The embedding model achieved an accuracy of 94.71% after 12 epochs with a loss of 0.1574, as seen in Fig. 1. The result from running the SuperSim evaluation can be seen in Fig. 2. The Pearson correlation test achieves a statistic of 0.439 with a p-value of $1.63 \cdot 10^{-11}$. We can also see the top 10 closest tokens to any token. Some examples are shown in Fig. 3. An example of a 3D PCA projection of the embedding space, with a selected word and it's closes neighbors, is shown in Fig. 4. Two sentences from a randomly selected Wikipedia article[1], were also run through the tokenizer to get an example of how the text is divided into tokens. The original sentences together with the BPE encoded sentence can be seen below.

Original:

```
Församlingen utgjorde till 1 maj 1917 ett
eget pastorat för att därefter till 2008
vara annexförsamling i pastoratet Kuddby,
Tåby och Å som 1962 utökades med Östra
Stenby, Konungsunds, Furingstads och
Dagsbergs församlingar. Församlingen
uppgick 2008 i Västra Vikbolandets
församling.
```

After tokenization (space separates tokens):

```
församlingen</w> utgjorde</w> till</w>
1</w> maj</w> 1917</w> ett</w> eget</w>
pastorat</w> för</w> att</w> därefter</w>
till</w> 2008</w> vara</w> anne x
församling</w> i</w> pastoratet</w> k ud
d by ,</w> tå by</w> och</w> å</w>
som</w> 1962</w> utökades</w> med</w>
östra</w> sten by ,</w> kon ung sun ds
</w> fur ing stads</w> och</w> dags
bergs</w> församlingar</w> [END]
församlingen</w> uppgick</w> 2008</w>
i</w> västra</w> vik bol an dets</w>
församling .</w> [END]
```

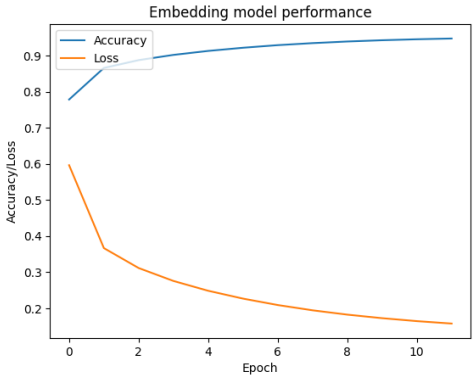

Fig. 1. Plot of accuracy vs loss on 12 epochs for the embedding layer.

V. DISCUSSION

A. Results

The embedding model was able to achieve an accuracy of 94.71% on 12 epochs with early stopping, which is a satisfying result when taking into consideration the amount of tokens the embedder model had to train on. Moreover, it is only natural that some tokens are referenced less within the corpus and

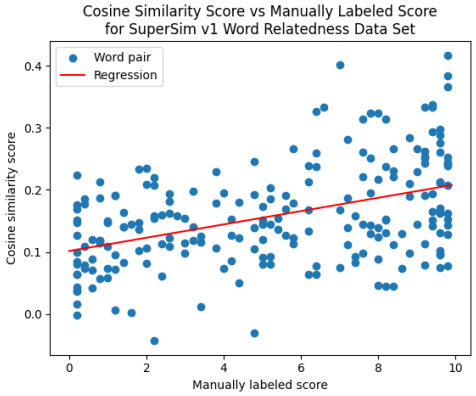

Fig. 2. Result from Running Cosine Similarity on the SuperSim v1 Word Relatedness Data Set.

[1]Tåby församling, https://sv.wikipedia.org/wiki/T%C3%A5by_f%C3%B6rsamling

Nearest points to Illinois</w> in the original space:		Nearest points to minuter</w> in the original space:		Nearest points to arkitektur</w> in the original space:	
georgia</w>	0.643	dygn</w>	0.570	teknik</w>	0.707
kentucky</w>	0.654	månader</w>	0.571	konst</w>	0.715
kalifornien</w>	0.658	sekunder</w>	0.573	kultur</w>	0.749
pennsylvania</w>	0.680	veckor</w>	0.582	teologi</w>	0.750
minnesota</w>	0.693	millimeter</w>	0.617	verksamhet</w>	0.753
virginia</w>	0.707	centimeter</w>	0.620	tradition</w>	0.766
kansas</w>	0.719	stämmor</w>	0.630	industrin</w>	0.774
west</w>	0.720	timmar</w>	0.638	insatser</w>	0.780
colorado</w>	0.723	omgångar</w>	0.645	industri</w>	0.780
florida</w>	0.725	gånger</w>	0.662	kyrkobyggnad</w>	0.792

Fig. 3. Top 10 similar words to three different sample words using cosine distance.

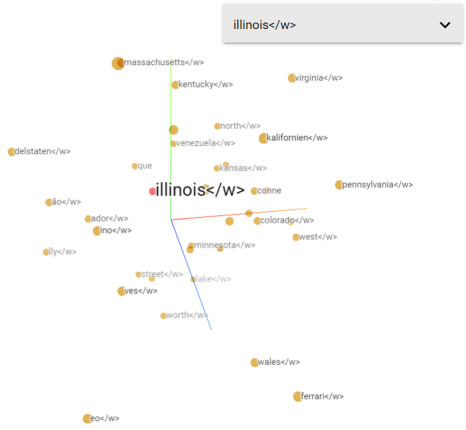

Fig. 4. Visualisation of a 3D PCA projection of the embedding space, with Illinois< /w> and it's closest neighbors selected.

therefore do not yield as much semantic meaning as words that are more common. However, a larger subset of the data set might allow the model to achieve better results because of the larger corpus it would have access to. Because byte pair encoding and tokenizing the data set subset was a time-intensive process, pursuing higher accuracies with even larger subsets was not a feasible option given the time constraints for the project. Even so, we were able to show that our model performs well in simulated real-word scenarios through the comparison with the SuperSim dataset, acting as the ground truth regarding the metric of how words are related. Through Pearson's correlation coeffcient, an adequate correlation was discovered between our model's embedding capabilities and the ground truth dataset, as seen in Fig.2. To further show the capabilities of our swedish embedder model, three words are chosen, for each word, the top 10 nearest words are shown, giving an idea of how the embedder understands the semantics behind the words. Looking at Fig.3, we can see that the top 10 nearest words to "Illinois" are also US states, the top 10 nearest words for "minuter" are measurements in general where the top three are measurements of time, and finally for "arkitektur", the top 10 nearest words reflect various terms and industries where architecture is prevalent within. Through these results, we were able to show that even with 10% of the whole wikipedia dataset, the embedder is able to discern the semantics and contexts of a given input with a high degree of confidence.

B. Metrics

1) Intrinsic evaluation: One method of evaluating embedders is through an intrinsic process where an embedder is evaluated independent of its specified NLP task, rather, through the embedder's intrinsic abilities like word similarity, word analogy and concept categorization [11]. Moreover, an extrinsic version of the evaluation can be used where the embedder is evaluated on its specific domain task rather than just its innate ability for understanding language. This process was however not used in this project because we constrained ourselves to a task agnostic embedding model. Using the word similarity process, we were able to evaluate and test our embedding model for relatedness within a latent vector space using cosine similarity as a metric. A PCA version of this latent space with "`Illinois<w>`" as chosen word is seen in Fig. 4.

2) Cosine similarity: Cosine similarity is way to measure the distance between two vectors, the equation is as follows:

$$cos(a,b) = \frac{a \cdot b}{||a||||b||}$$
$$cos(a,b) \in [-1,1]\mathbb{R}. \quad (1)$$

Cosine similarity is an attractive evaluation option because of its ease of use whilst being computationally inexpensive. Moreover, it is fast which enables more prototyping and experimentation. However, there are a few minor downsides to cosine similarity, one of them being similarity vs. relatedness. Relatedness implies that two words are related to each other in someway, be it literally or conceptually. For example, coffee and cup are related to each other, but not similar as one is a plant or drink, and the other is a man-made object for drinking. Coffee and plant are both related and similar to each other, more so than coffee and cup. Moreover, it is subjective what words are similiar or related to each other, so there isn't a ground truth regarding that. However, in our project, cosine similiarty was adequate enough as we put more importance on the relatedness between the words, rather than how similiar they are, moreover, similiarty and relatedness often overlap with each other thus there isn't always a clear-cut way to differentiate between the two [12].

C. Challenges

1) Data Set Size: Working with a large data set presents a lot of challenges. The first challenge is to simply load the data set into the program. The size of the data set when it is compressed is 3.58GB, and uncompressed it is 62.4GB. This means that we can't easily load the entire file into memory. The format of the file is XML, and it has a lot of superfluous information that we can remove. To reduce the size of the file, we read the file in a streaming fashion and only save the

relevant parts to a text file. This reduces the size of the file to 1.29GB.

2) Counting Words: The second challenge that we faced was counting the words. To count the words, we used the `CountVectorizer` from Scikit-learn. This gives a count of how many times each word appears in each sentence. We could then get the total for each word. The problem we ran into when running on the large data set was that it tried to allocate 4.7PB of memory, which is a bit more than what we had access to. To solve this, we instead used the built in `Counter`. This works since we have one word per "sentance".

3) Tokenizing the Corpus: The third challenge was applying the BPE to the corpus. Our original approach was to iterate over the entire corpus for each merge and perform a replace with regex. This took 45 minutes for the 1734 Law corpus with 1000 tokens, which means that it would take a least 62 days to run on the Wikipedia corpus with 1000 tokens, or 620 days for 10,000 tokens.

Our first approach to this solution was to use a trie to create a single regex string to perform the entire replace in one pass, removing the need to do a pass for each individual merge [13]. The problem with this is that the BPE merges can be dependent on each other, and therefore we need to apply the merges in order, which can't be done with a single pass regex replace.

Our second approach was to edit our regex search string. Our original string was `( |^)a b( |$)`. The alternations `( |^)` and `( |$)` are used to check for word boundaries, and in the written way they need to check for both cases, which slows down computation. In regex, there is a word boundary token `\b`, and using this instead would yield an almost 2x speedup. The problem with using `\b` is that it treats any character not matching `[A-Za-z0-9_]` as a word boundary, which means that our word boundary tokens `</w>` is not handled correctly.

Our third, and final approach was to use a TensorFlow tensor to store the entire corpus and perform the regex replace on the entire corpus at once. This means that we only need to iterate over the merges, and perform the replace once per merge. However, this also means that we need to merge the entire corpus into a single "sentence", which requires introducing sentence boundary tokens to be able to split the merged corpus into sentences again. This reduced the time taken to merge the 1734 Law corpus down to 2 minutes, and also showed time complexity $< O(n)$, which resulted in a total time of just under 7 hours to apply the BPE to 10% of the Wikipedia corpus, for 10,000 tokens.

4) Embedding: For the embedding, the size of the data set was again a limitation. With a window size of 2, and 4 negative skip-grams per positive skip-gram, the total size of the training data grew to over 80GB, which meant that it did not fit into memory, which significantly slowed down computations due to use of the swap-file. The main problem was that the training samples where stored as 64-bit integers, which used a lot of memory to store each sample. Casting these to 16-bit integers reduced the size to about 20GB, which meant that the entire data set fit into memory. Also using a window size of 1 instead of 2 further reduced the size and time to generate the data. Finally, using multiprocessing to parallelize the generation, and thereby utilizing all CPU cores, significantly sped up the generation.

VI. Conclusions

From this project we conclude that using a Swedish corpus to train a tokenizer and an embedder could be beneficial to Swedish NLP tasks. We have also gained experience with handling a large data set and how to overcome the challenges that come with that. Finally, we would have liked to extrinsic testing with an actual transformer model, as well as look at applying the resulting embedder to a real world problem. However, this was not possible due to the time constraint of the project.

References

[1] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv.org*, May 2019. [Online]. Available: https://arxiv.org/abs/1810.04805

[3] S. Katsarou, "Improving multilingual models for the swedish language," *Diva Portal*. [Online]. Available: https://kth.diva-portal.org/smash/get/diva2:1618310/FULLTEXT01.pdf

[4] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016.

[5] TensorFlow, "Word embeddings." [Online]. Available: https://www.tensorflow.org/text/guide/word_embeddings

[6] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv.org*, Sep 2013. [Online]. Available: https://arxiv.org/abs/1301.3781v3

[7] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162

[8] L. Borin, M. Forsberg, and J. Roxendal, "Korp – the corpus infrastructure of språkbanken," in *Proceedings of LREC 2012. Istanbul: ELRA*, vol. Accepted, 2012, p. 474–478.

[9] K. Whistler, "UAX #15: Unicode Normalization Forms." [Online]. Available: https://unicode.org/reports/tr15/

[10] S. Hengchen and N. Tahmasebi, "SuperSim: a test set for word similarity and relatedness in Swedish," Apr. 2021, arXiv:2104.05228 [cs]. [Online]. Available: http://arxiv.org/abs/2104.05228

[11] B. Wang, A. Wang, F. Chen, Y. Wang, and C.-C. J. Kuo, "Evaluating Word Embedding Models: Methods and Experimental Results," *APSIPA Transactions on Signal and Information Processing*, vol. 8, no. 1, 2019, arXiv:1901.09785 [cs]. [Online]. Available: http://arxiv.org/abs/1901.09785

[12] M. Faruqui, Y. Tsvetkov, P. Rastogi, and C. Dyer, "Problems with evaluation of word embeddings using word similarity tasks," *arXiv.org*, Jun 2016. [Online]. Available: https://arxiv.org/abs/1605.02276

[13] E. Duminil, "Answer to "Speed up millions of regex replacements in Python 3"," Mar. 2017. [Online]. Available: https://stackoverflow.com/a/42789508