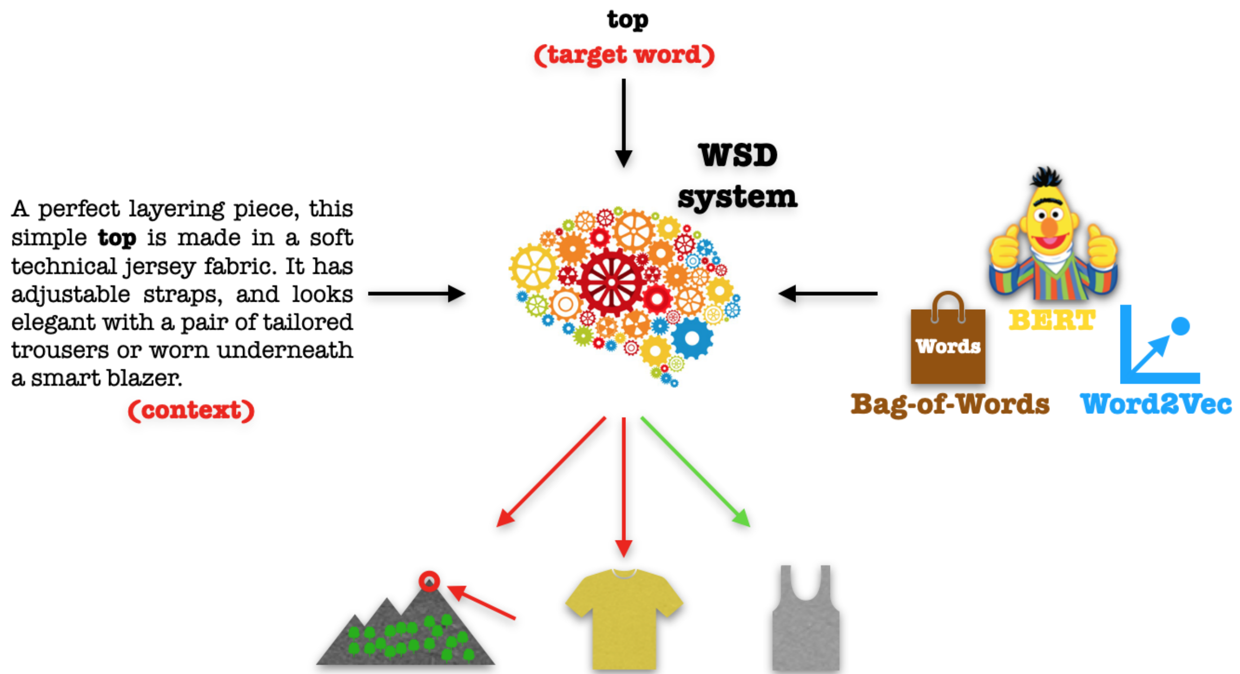
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG



Unsupervised Word-Sense Disambiguation for Product Description Texts

Using contextual representations to disambiguate words with multilingually defined senses in product description texts

Master's thesis in Computer science and engineering

ANDREAS DAHLBERG, OSKAR OLIN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Unsupervised Word-Sense Disambiguation for Product Description Texts

Using contextual representations to disambiguate words with multilingually defined senses in product description texts

ANDREAS DAHLBERG, OSKAR OLIN

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Unsupervised Word-Sense Disambiguation for Product Description Texts
Using contextual representations to disambiguate words with multilingually defined senses in product description texts
ANDREAS DAHLBERG, OSKAR OLIN

© Andreas Dahlberg and Oskar Olin, 2020.

Supervisor: Krasimir Angelov, Department of Computer Science and Engineering
Advisor: Oscar Kalldal, Textual Relations AB
Examiner: Aarne Ranta, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Unsupervised Word-Sense Disambiguation for Product Description Texts
Using contextual representations to disambiguate words with multilingually defined senses in product description texts
ANDREAS DAHLBERG, OSKAR OLIN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As the name suggests, word-sense disambiguation is the task of determining the correct meaning, or sense, of words that can have multiple interpretations. Textual, a company with a product that automatically generates product description texts in multiple languages, can make use of word-sense disambiguation to improve the quality of their texts. In this project, an attempt to solve this task is made. To achieve this, word alignment is used to define and label the senses of words as quadruples of translations in English, Swedish, French and Spanish, making word-sense disambiguation a supervised task. Contextually alike quadruples are then merged using a permutation test and a novel merging algorithm. In word-sense disambiguation it is natural to represent the word along with its context as a vector in a higher-dimensional vector space. For this, different BERT-models are used as well as the simpler Bag-of-Words- and contextual Word2Vec-models. The results on 69 different word types show an average accuracy of 91.97% compared to 58.35% for the baseline classifier, the classifier that always predicts the most frequent sense. On unseen data from new fashion sites, the average accuracy on 8 word types is 85.19% compared to 56.89% for the baseline classifier.

Keywords: word-sense disambiguation, natural language processing, BERT, word alignment, machine learning, artificial intelligence

Acknowledgements

We would like to give our appreciation to our supervisors Krasimir Angelov and Oscar Kalldal for their support and valuable inputs during the project. We would also like to thank Textual for providing us with data and computational resources, making this project possible.

Andreas Dahlberg and Oskar Olin, Gothenburg, June 2020

Contents

List of Figures **xi**

List of Tables **xv**

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Aim and Objectives . . . 2
- 1.3 Outline . . . 2

2 Theory **3**
- 2.1 Word Alignment . . . 3
  - 2.1.1 EM-algorithm . . . 3
  - 2.1.2 IBM Models . . . 5
    - 2.1.2.1 IBM Model 1 . . . 6
    - 2.1.2.2 Higher IBM Models . . . 10
- 2.2 Statistical Machine Translation . . . 10
  - 2.2.1 Decoder . . . 11
  - 2.2.2 BLEU Score . . . 13
- 2.3 Feed Forward Neural Networks . . . 14
  - 2.3.1 Architecture . . . 14
  - 2.3.2 Training . . . 15
- 2.4 Contextual Representations . . . 15
  - 2.4.1 Bag-of-Words . . . 16
  - 2.4.2 Word2Vec . . . 17
    - 2.4.2.1 Skip-gram Model . . . 17
    - 2.4.2.2 Contextual Word2Vec . . . 18
  - 2.4.3 Bidirectional Encoder Representations from Transformers . . . 18
    - 2.4.3.1 Tokenization and Subword Splitting . . . 20
    - 2.4.3.2 Word Embedding . . . 21
    - 2.4.3.3 Attention . . . 21
    - 2.4.3.4 The Encoder . . . 23
    - 2.4.3.5 Pre-training . . . 24
- 2.5 Classification . . . 25
  - 2.5.1 k-Nearest Neighbours . . . 26
  - 2.5.2 Support Vector Machines . . . 26
    - 2.5.2.1 Binary Classification . . . 26

2.5.2.2 The Kernel Trick 28
2.5.2.3 Probabilistic Classifier 29
2.5.2.4 Multiclass Classification 29
2.5.3 Random Forest Classifier 30
2.5.4 Gaussian Naive Bayes Classifier 31
2.6 Self-training 31
2.7 Bayesian Optimization for Tuning Hyperparameters 32

3 Methods **35**
3.1 Datasets 36
3.2 Defining Word Senses 37
3.2.1 Initial Definition of Word Senses Using Word Alignment . . . 37
3.2.2 Permutation Test 40
3.2.3 Merging of Senses 41
3.3 Preparation of Contextual Models 42
3.3.1 Bag-of-Words 42
3.3.2 Word2Vec 43
3.3.3 Pre-training BERT 43
3.3.4 Fine-tuning BERT 44
3.4 Word-sense Disambiguation 45
3.5 Evaluating the Performance on New Fashion Sites 46

4 Results **47**
4.1 Word Alignment 47
4.2 Training BERT 49
4.3 Merging of Senses 50
4.4 Word-sense Disambiguation 51

5 Discussion **59**
5.1 Defining Senses 59
5.2 Comparing and Analyzing the Models 60
5.3 Using Non-annotated Data 61
5.4 Conclusion 61

Bibliography **63**

A Appendix **I**
A.1 Sense Definitions I
A.2 Results for Each Model VII
A.3 Examples of the Permutation Test XVII
A.4 Example Usages of the Final Program XXI

List of Figures

2.1 A possible alignment between a French and English sentence pair. . . 6
2.2 Architecture of a neural network (left) and four examples of activation functions (right). The input is propagated through the network from left to right. The weights (red arrows, not drawn everywhere) are used to compute the value of each neuron in a layer based on the values of the neurons in the previous layer and the chosen activation function. 14
2.3 An illustration of the BERT Base model. The tokenized input sequence at the bottom (in red) is word embedded, before it is fed to the first encoder. After passing through all 12 encoders, the BERT vectors are obtained at the top (in green). 19
2.4 Each word is embedded with the sum of its token embedding, segment embedding and positional embedding. We call this sum the *word embedding* of the word. 21
2.5 Visualization of the first BERT encoder. The input sequence is passed through a multi-headed attention layer followed by a feed forward neural network layer (containing one hidden layer). Both layers are followed by a residual connection and layer normalization. The output is then passed to the second encoder. 24
2.6 k-nearest neighbours. 26
2.7 Support Vector classifier 26
2.8 A simple decision tree. The data contains two features, namely color and shape. Each leaf node is considered a region which is associated with one class. 30

3.1 Bidirectional word alignment between the four languages. 38
3.2 An example showing how the SenseMerge algorithm reduces the graph until there are no edges left. 42
3.3 The fine-tuning structure. 44
3.4 Structure of the word-sense disambiguation procedure for the BERT models. 45

4.1 Word alignment for one text in English to and from the corresponding texts in Swedish, French and Spanish. Red lines are double alignments, while blue arrows are single alignments. 47

4.2 Loss, masked LM accuracy, and next sentence prediction accuracy on test data during pre-training is shown in the left figure. The loss has been moved down for increased visibility. Training loss and validation accuracy during fine-tuning on category classification is shown in the right figure. The loss has been moved up for increased visibility. . . . 49
4.3 t-SNE plot for the word *fastener* before SenseMerge. Every point represents the BERT vector for the word *fastener* in the sentence it is used in. The BERT model used is the pre-trained BERT model. . 50
4.4 Histograms showing the distribution of the statistic (3.1) under H_0 when comparing the five quadruples. The red line is the value of the statistic before permuting the labels. 50
4.5 The most common window of words for the two senses of the word fastener that are merged, see Figure 4.3. 51
4.6 Accuracy of the best model (see Table 4.2) and baseline (BL) for all words. The baseline is the classifier that predicts every sense to be the most frequently occurring sense (see Table A.1). It is clear that the baseline is beaten for all words. 54
4.7 Histogram over a small window of words around *neck* for its two senses. 56
4.8 Histogram over a small window of words around *logo* for its three senses. 56
4.9 Illustrating the difference between the senses ('jacket', 'jacka', 'veste', 'chaqueta') shown in **(a)**, ('jacket', 'jacka', 'veste', 'cazador') shown in **(b)** and ('jacket', 'kavaj', 'blazer', 'americana') shown in **(c)**. . . . 58
4.10 Illustrating the difference between the senses ('hat', 'hatt', 'chapeau', 'sombrero') shown in **(a)**, ('hat', 'mössa', 'bonnet', 'gorro'), shown in **(b)** and ('hat', 'hatt', 'chapeau', 'gorro') shown in **(c)**. 58

A.1 Left picture shows two classes before the permutation test and SenseMerge and right picture after. XVII
A.2 Same as the picture above but here we use $d(s(Q_i), s(Q_j))$ in the numerator instead of $d_{0.5}(s(Q_i), s(Q_j))$ in Equation 3.2. Note that the two classes are not merged which is not what we want. XVII
A.3 Left picture shows two classes before the permutation test and SenseMerge and right picture after. Green class is $\sim N(0, 10^{-2}I)$. Blue is $\sim N(0, 10^{-1}I)$ and within the disc $x^2 + y^2 \leq 0.25$ points are removed with probability 0.5. XVIII
A.4 Same as the picture above but here we use $T = d_{0.5}(s(Q_i), s(Q_j))$ as statistic in Equation 3.2, i.e. without the denominator. Note that the two classes are merged which is not what we want XVIII
A.5 Left picture shows three classes before the permutation test and SenseMerge and right picture after. XIX
A.6 Same as the picture above but here we use $d(s(Q_i), s(Q_j))$ in the numerator instead of $d_{0.5}(s(Q_i), s(Q_j))$ in Equation 3.2. Note that the green and blue class are not merged which is not what we want. . XIX

A.7 Left picture shows five classes before the permutation test and SenseMerge and right picture after. Green is $\sim N([0, 0.015]^T, 10^{-3}I)$, blue is $\sim N([0, 0.015]^T, 10^{-3}I)$, yellow is $\sim N(0, 10^{-3}I)$, black is $\sim N([0, -0.015]^T, 10^{-3}I)$ and red is $\sim N([0, -0.015]^T, 10^{-3}I)$. XX
A.8 Ilustrating the steps in the SenseMerge algorithm for the scenario in the figure above. XX

List of Tables

3.1 Summary of all product description texts. Around 82000 texts were used in the training phase while around 9600 texts were used in the final evaluation. This is excluding headers (titles). 36

4.1 BLEU score of a machine translation model that uses word alignment. Used as an indirect evaluation of the word alignment. *Lexicons* refers to the number of duplicates of Textual's lexicon and *Subtitles* refers to the number of subtitle texts. 48

4.2 Obtained accuracy on 69 different words using the following contextual models: Baseline classifier (BL), Bag-of-Words (BoW), Word2Vec (W2V), BERT Base (BB), BERT Pre-trained (BP), BERT Pre-trained and Fine-tuned (BP&F) and the best of these models (Best). 51

4.3 The gain from including the self-training procedure as a tunable parameter amounts to 0.1% for the best model. 55

4.4 The total number of manually annotated data points, the baseline accuracy and the obtained accuracy when evaluating the program for eight ambiguous words on the new fashion sites; KappAhl, Nike, Gina Tricot and Man of a kind. 56

4.5 The percentage of texts that contains the keywords: *premium quality, suede, boots* or *black* for the two senses of *leather*: $leather_1$ = (*'leather', 'läder', 'cuir', 'piel'*) and $leather_2$ = (*'leather', 'skinn', 'cuir', 'piel'*) 57

4.6 The percentage of texts that contains the keywords: *out, warm, button, pocket, single-breasted* or *long sleeve* for the three senses of *jacket*: $jacket_1$ = ('jacket', 'jacka', 'veste', 'chaqueta'), $jacket_2$ = ('jacket', 'jacka', 'veste', 'cazador') and $jacket_3$ = ('jacket', 'kavaj', 'blazer', 'americano'). 57

A.1 Definition of the senses used to produce the results in Table 4.2. The count and proportion columns refer to the number of texts that the particular sense appears in (in the annotated data). I

A.2 The results from WSD with the Bag-of-Words model for 69 ambiguous words after hypertuning. Each word uses different classifiers (with different parameters that are not displayed here) and are trained in either a supervised manner or using self-training. Note that it was advantageous to use the non-annotated data (self-training) for 39% (27/69) of the words. The *total* column refers to the total number of annotated data points. The baseline classifier is simply taken to be the classifier that always predicts the most frequent occurring sense, see Table A.1. Its performance is given in the column *BL acc.* VII

A.3 The results from WSD with the Word2Vec model for 69 ambiguous words after hypertuning. Each word uses different classifiers (with different parameters that are not displayed here) and are trained in either a supervised manner or using self-training. Note that it was advantageous to use the non-annotated data (self-training) for 48% (33/69) of the words. The *total* column refers to the total number of annotated data points. The baseline classifier is simply taken to be the classifier that always predicts the most frequent occurring sense, see Table A.1. Its performance is given in the column *BL acc.* IX

A.4 The results from WSD with the BERT Base model for 69 ambiguous words after hypertuning. Each word uses different classifiers (with different parameters that are not displayed here) and are trained in either a supervised manner or using self-training. Note that it was advantageous to use the non-annotated data (self-training) for 45% (31/69) of the words. The *total* column refers to the total number of annotated data points. The baseline classifier is simply taken to be the classifier that always predicts the most frequent occurring sense, see Table A.1. Its performance is given in the column *BL acc.* XI

A.5 The results from WSD with the pre-trained BERT model for 69 ambiguous words after hypertuning. Each word uses different classifiers (with different parameters that are not displayed here) and are trained in either a supervised manner or using self-training. Note that it was advantageous to use the non-annotated data (self-training) for 52% (36/69) of the words. The *total* column refers to the total number of annotated data points. The baseline classifier is simply taken to be the classifier that always predicts the most frequent occurring sense, see Table A.1. Its performance is given in the column *BL acc.* XIII

A.6 The results from WSD with the pre-trained and fine-tuned BERT model for 69 ambiguous words after hypertuning. Each word uses different classifiers (with different parameters that are not displayed here) and are trained in either a supervised manner or using self-training. Note that it was advantageous to use the non-annotated data (self-training) for 41% (28/69) of the words. The *total* column refers to the total number of annotated data points. The baseline classifier is simply taken to be the classifier that always predicts the most frequent occurring sense, see Table A.1. Its performance is given in the column *BL acc.* XV

1 Introduction

Word-sense disambiguation (WSD) is an important domain within the field of Natural Language Processing. As the name suggests, WSD is the study of determining the sense of words with multiple meanings, called ambiguous words. What constitutes different senses of a word is the context that they are used in. Simply put, if a word can be used in multiple contexts it follows that it has multiple senses. In written language, the context is determined by the other words in the text and also the order in which they appear. In many cases the context is only given by the immediate surrounding words, but it is also common with longer dependencies. Consider the word *bank* in the following examples

1. "The fisherman jumped off the *bank* and into the water."
2. "The *bank* down the street was robbed!"

In the first example, *bank* refers to a sloping land (especially the slope beside a body of water), while in the second example it refers to a financial institution. To determine the sense in the first one, one might look at the words *fisherman*, *jumped* and *water*. These words are rarely used when talking about financial institutions, but they are more commonly used together with sloping lands. In the same way, the words *street* and *robbed* are more commonly used when referring to financial institutions but not for sloping lands.

1.1 Background

Textual[1] has a product capable of automatically generating product descriptions in up to 15 different languages based on structured product data. In essence, the product is able to process a set of keywords describing a product and in turn produce natural text intended to be read by potential customers on a product page on an e-commerce website. These texts have been proven to perform on-par or better than texts written by humans in several real cases on clients e-commerce websites using AB-testing, and are significantly cheaper and faster to produce. Currently, the text generator is purely rule-based and deterministic. Combining this approach with machine learning, creating a hybrid method, could improve the quality of the generated texts.

[1] www.textual.ai

In the current version of Textual's text generator the user has to provide the sense of ambiguous words. This is problematic since the senses are often polysemous, meaning that the senses are related and similar to each other. This becomes extra problematic if a distinction between the senses is not made in the language(s) that the user speaks, but rather in another language. If the user provides the wrong sense, the English text is not affected but the translations could be wrong. With this in mind, a program that can predict the sense of ambiguous words will be of use. The direct application of this program would be to compare the predicted senses in the generated English texts with the senses provided by the user. If these two differ, an extra check is required by the user.

1.2 Aim and Objectives

The first goal of this project is to annotate data and define senses for fashion words using word alignment on product description texts written in English, Swedish, French and Spanish. The second goal is to train a classifier on the labeled data obtained from the word alignment that can disambiguate all ambiguous words for any given product description text written in English.

1.3 Outline

First, in Chapter 2 we describe the theory behind the methods that we use. In Chapter 3, the word alignment procedure that gives an initial set of possible senses is presented. In a later step we explain how this set of senses can be reduced to more contextually different senses using a permutation test and a novel algorithm that we call SenseMerge. Similarity between senses is measured using a custom measure that considers the Euclidean distance between the BERT vectors of the ambiguous words. Furthermore, we explain how the BERT embeddings can be improved to better capture the jargon in product description texts by continuing the pre-training and fine-tuning on a downstream task. Lastly in this chapter the word-sense disambiguation procedure is explained and also how the program can be evaluated by testing it on manually annotated data from new fashion sites, not seen in the training procedure. Thereafter in Chapter 4, we present our results and give examples illustrating when the senses for a word are truly different senses or only apparent different senses. Finally, in Chapter 5, different aspects of the project is discussed and we suggest improvements that can be made and areas that could be further investigated.

2
Theory

In this chapter the theory behind our methods is described. We start by introducing the word alignment problem and how this can be solved for the IBM models using the Expectation-Maximization (EM) algorithm. This is followed by a short introduction to Statistical Machine Translation, a technique that can be used for indirect evaluation of word alignment. After this we outline the theory behind Neural Networks and also the training procedure. Next we give a thorough description of models that provide contextual representations. These include: Bag-of-Words, Contextual Word2Vec and, in particular, the BERT model and all its components. Lastly, we present several different classifiers and describe how their hyperparameters can be tuned using Bayesian Optimization.

2.1 Word Alignment

The *word alignment* between two parallel texts (the same text expressed in different languages) is a mapping between the words in the two texts. This mapping can intuitively be thought of as which words are translated to what. Word alignment is the key component in statistical machine translation, but it can also be used for finding the senses of a word. Training of word alignment consists of estimating a set of parameters, which includes lexicon translation probabilities. This is usually done in an unsupervised manner by feeding it a training set of parallel texts and then estimating the parameters by using the EM-algorithm [1]. In this section, the EM-algorithm is first explained followed by an explanation of the IBM models which are the most common models for word alignment.

2.1.1 EM-algorithm

The *Expectation-Maximization* algorithm [2] is an iterative method consisting of an E-step (Expectation) and an M-step (Maximization) for finding the parameters that maximizes a likelihood function or a posterior in a model that includes hidden variables.

In the rest of this text we let $\pi(\cdot)$ be a general probability density function with no specific assumptions. We will assume that all random variables are continuous. In the discrete setting, all integrals are replaced by sums. The EM-algorithm can be applied to the situation when

$$\pi(x|\theta) = \int_z \pi(x, z|\theta) dz$$

where x is the observed data, z is a hidden variable and θ are the parameters of the model that we want to find. The goal of the EM-algorithm is to estimate θ by

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\ \pi(x|\theta) = \underset{\theta}{\operatorname{argmax}}\ \log \pi(x|\theta).$$

To do this we first define the Kullback-Leibler divergence between the two distributions q and p as

$$KL[q||p] = -E_q[\log\left(\frac{p(X)}{q(X)}\right)]$$

and then the helper function

$$L(q,\theta) = \int_z q(z) \log\left(\frac{\pi(x,z|\theta)}{q(z)}\right) dz$$

where q is an arbitrary probability density function. By letting

$$\pi_z(z) = \pi(z|x,\theta)$$

we see that the logarithm of the likelihood function can be written as

$$\begin{aligned}
KL[q||\pi_z] + L(q,\theta) &= -\int_z q(z) \log\left(\frac{\pi(z|x,\theta)}{q(z)}\right) dz + \int_z q(z) \log\left(\frac{\pi(x,z|\theta)}{q(z)}\right) dz \\
&= \int_z q(z) \log\left(\frac{\pi(x,z|\theta)}{\pi(z|x,\theta)}\right) dz \\
&= \int_z q(z) \log\left(\frac{\pi(z|x,\theta)\pi(x|\theta)}{\pi(z|x,\theta)}\right) dz \\
&= \int_z q(z) \log \pi(x|\theta) dz \\
&= \log \pi(x|\theta).
\end{aligned}$$

By using that the Kullback-Leibler divergence is positive definite [3, p. 9] we obtain that the following holds for all q

$$\log \pi(x|\theta) = KL[q||\pi_z] + L(q,\theta) \geq L(q,\theta) \tag{2.1}$$

with equality if and only if $q = \pi_z$.

The EM-algorithm improves the lower bound of the log-likelihood function by alternating between maximizing $L(q,\theta)$ over q for a fixed θ and maximizing $L(q,\theta)$ over θ for a fixed q. We start with an initial solution $\theta^{(0)}$. By using that the left-hand side in Equation 2.1 is independent of q and that the Kullback-Leibler divergence is positive definite we get that

$$\underset{q}{\operatorname{argmax}}\ L(q,\theta^{(0)}) = \pi(z|x,\theta^{(0)}) := q^{(0)}.$$

The next step is to maximize $L(q^{(0)},\theta)$ over θ. We can rewrite this function as

$$L(q,\theta) = \int_z q(z) \log\left(\frac{\pi(x,z|\theta)}{q(z)}\right) dz = E_q[\log \pi(X,Z|\theta)] - \int_z q(z) \log q(z) dz$$

and thus

$$\underset{\theta}{\operatorname{argmax}}\ L(q^{(0)}, \theta) = \underset{\theta}{\operatorname{argmax}}\ E_{q^{(0)}}[\log \pi(X, Z|\theta)] := \theta^{(1)}.$$

The E-step in EM is calculating the expectation above and the M-step is maximizing this expectation.

Next we will prove that the EM algorithm finds a stationary point to the log-likelihood function. It is not hard to see that the sequence

$$L(q^{(0)}, \theta^{(0)}),\ L(q^{(1)}, \theta^{(1)}), \ldots = \log \pi(x|\theta^{(0)}),\ \log \pi(x|\theta^{(1)}), \ldots$$

is monotonically increasing. This together with the assumption that $\log \pi(x|\theta)$ is bounded from above implies that the following limit exists

$$\lim_{k\to\infty} \log \pi(x|\theta^{(k)}) := \ell^*.$$

With some additional regularity conditions, described in [4], one can prove that

$$\lim_{k\to\infty} \theta^{(k)} := \hat{\theta}.$$

From Equation 2.1 we have that

$$\log \pi(x|\theta) \geq L(q^{(k)}, \theta)$$

with equality at $\theta = \theta^{(k)}$. This means that

$$\nabla_\theta \log \pi(x|\theta^{(k)}) = \nabla_\theta L(q^{(k)}, \theta^{(k)})$$

so if $\theta^{(k+1)} = \theta^{(k)}$ then we must have

$$\nabla_\theta \log \pi(x|\theta^{(k)}) = \nabla_\theta L(q^{(k)}, \theta^{(k)}) = 0$$

which means that $\theta^{(k)}$ is a stationary point to the log-likelihood function.

2.1.2 IBM Models

We will assume throughout this section that we want to translate from French, the source language, to English, the target language. The data that we will learn this translation from is

$$(\mathbf{e}^{(1)}, \mathbf{f}^{(1)}), \ldots, (\mathbf{e}^{(n)}, \mathbf{f}^{(n)}) \tag{2.2}$$

where $(\mathbf{e}^{(j)}, \mathbf{f}^{(j)})$ is a English-French sentence pair. Furthermore, we assume that every French word can be aligned to only 0 or 1 English words. This means that for an English sentence $\mathbf{e} = (e_1, \ldots, e_J)$, of length J, and a corresponding French sentence $\mathbf{f} = (f_1, \ldots, f_I)$, of length I, the alignment can be represented by an alignment vector $\mathbf{a} = (a_1, a_2, \ldots, a_I)$ where $a_i \in \{0, 1, \ldots, J\}$ and $a_i = 0$ means that the i:th French word doesn't align to any English word. Note that an English word can be aligned to more than one French word. To make this more concrete, consider the example below where the French sentence "Le chat était assis près de la fenêtre"

is translated to the English sentence "The cat sat by the window". The alignment vector is given by $\mathbf{a} = (1, 2, 3, 3, 4, 0, 5, 6)$.

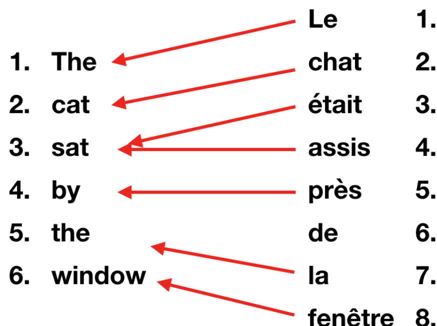


Figure 2.1: *A possible alignment between a French and English sentence pair.*

The idea in word alignment is to assume that the English sentences $\mathbf{e}^{(1)}, \ldots, \mathbf{e}^{(n)}$ are fixed and that the French sentences are obtained from a generative process parameterized by θ. From the observed data given in Equation 2.2 we can then find the likelihood function and maximize it using the EM-algorithm. The main difference between the different IBM models is the complexity of the generative process that generates the French sentences. In this report, we will primarily describe the first model as this is sufficient for an understanding of the general idea.

2.1.2.1 IBM Model 1

In this model we assume that a French sentence $\mathbf{f}$ is generated from an English sentence $\mathbf{e}$ in the following steps:

1. choose a length I for the French sentence, $\pi(I|J) = \epsilon$
2. for $i = 1, \ldots, I$ choose a_i uniformly from $\{0, \ldots, J\}$, $\pi(a_i = j|I, J) = (J+1)^{-1}$
3. for $i = 1, \ldots, I$ generate a French word from the corresponding English word, $\pi(f_i|e_{a_i})$

The first naive assumption made here is that the length of the French sentence is independent of the length of the English sentence and all lengths of French sentences are equally possible, i.e. $\pi(I|J)$ is independent of both I and J. The second naive assumption is that the i:th position in the French sentence is just as likely to be aligned to the last word in the English sentence as to the first word (this is solved in the more advanced IBM models by including positional alignment probabilities). From this generative process we can calculate the probability of observing the French

sentence $\mathbf{f}$ from the English sentence $\mathbf{e}$ as:

$$\begin{aligned}\pi(\mathbf{f}|\mathbf{e}) &= \sum_{\mathbf{a}} \pi(\mathbf{f},\mathbf{a}|\mathbf{e}) = \sum_{\mathbf{a}} \pi(\mathbf{f}|\mathbf{a},\mathbf{e})\pi(\mathbf{a}|\mathbf{e}) \\ &= \sum_{\mathbf{a}} \frac{\epsilon}{(J+1)^I}\pi(\mathbf{f}|\mathbf{a},\mathbf{e}) = \sum_{\mathbf{a}} \frac{\epsilon}{(J+1)^I}\prod_{i=1}^{I}\pi(f_i|e_{a_i}) \\ &= \frac{\epsilon}{(J+1)^I}\prod_{i=1}^{I}\sum_{j=0}^{J}\pi(f_i|e_j). \end{aligned} \tag{2.3}$$

where the sum is over all possible alignments. In the last step we used that

$$\sum_{\mathbf{a}}\prod_{i=1}^{I}\pi(f_i|e_{a_i}) = \prod_{i=1}^{I}\sum_{j=0}^{J}\pi(f_i|e_j) \tag{2.4}$$

The example below with $I = J = 2$ should clarify this equality.

$$\begin{aligned}&\sum_{a_1=0}^{2}\sum_{a_2=0}^{2}\prod_{i=1}^{2}\pi(f_i|e_{a_i}) = \\ &= \pi(f_1|e_0)\pi(f_2|e_0) + \pi(f_1|e_0)\pi(f_2|e_1) + \pi(f_1|e_0)\pi(f_2|e_2) \\ &+ \pi(f_1|e_1)\pi(f_2|e_0) + \pi(f_1|e_1)\pi(f_2|e_1) + \pi(f_1|e_1)\pi(f_2|e_2) \\ &+ \pi(f_1|e_2)\pi(f_2|e_0) + \pi(f_1|e_2)\pi(f_2|e_1) + \pi(f_1|e_2)\pi(f_2|e_2) \\ &= \pi(f_1|e_0)[\pi(f_2|e_0) + \pi(f_2|e_1) + \pi(f_2|e_2)] \\ &+ \pi(f_1|e_1)[\pi(f_2|e_0) + \pi(f_2|e_1) + \pi(f_2|e_2)] \\ &+ \pi(f_1|e_2)[\pi(f_2|e_0) + \pi(f_2|e_1) + \pi(f_2|e_2)] \\ &= (\pi(f_1|e_0) + \pi(f_1|e_1) + \pi(f_1|e_2))(\pi(f_2|e_0) + \pi(f_2|e_1) + \pi(f_2|e_2)) \\ &= \prod_{i=1}^{2}\sum_{j=0}^{2}\pi(f_i|e_j).\end{aligned}$$

The EM-algorithm for word alignment: The parameters θ that we want to find are all lexical translation probabilities $\pi(f|e)$. This is a matrix of size $|V_E||V_F|$ where V_E is the English vocabulary and V_F the French vocabulary. The hidden variables are all the alignment vectors and the observed variables are all French sentences. The likelihood function is given by

$$\pi(\mathbf{f}^*|\mathbf{e}^*,\theta) = \prod_{k=1}^{n}\pi(\mathbf{f}^{(k)}|\mathbf{e}^{(k)},\theta) = \prod_{k=1}^{n}\left(\sum_{\mathbf{a}}\frac{\epsilon}{(J+1)^I}\prod_{i=1}^{I}\pi(f_i^{(k)}|e_{a_i}^{(k)})\right)$$

By $(\mathbf{f}^*,\mathbf{e}^*,\mathbf{a}^*)$ we mean the whole sample, i.e. $\mathbf{f}^* = (\mathbf{f}^{(1)},\ldots,\mathbf{f}^{(n)})$ etc. The first step of an iteration in the EM algorithm is to put $q(\mathbf{a}^*) = \pi(\mathbf{a}^*|\mathbf{e}^*,\mathbf{f}^*,\theta)$. We want to be able to calculate

$$q(a_i = j) = \pi(a_i = j|\mathbf{e},\mathbf{f},\theta) = \frac{\pi(a_i = j,\mathbf{f}|\mathbf{e},\theta)}{\pi(\mathbf{f}|\mathbf{e},\theta)}. \tag{2.5}$$

The numerator can be rewritten as

$$\pi(a_i = j, \mathbf{f}|\mathbf{e}, \theta) = \sum_{\mathbf{a}:a_i=j} \pi(\mathbf{a}, \mathbf{f}|\mathbf{e}, \theta) = \sum_{\mathbf{a}:a_i=j} \frac{\epsilon}{(J+1)^I} \prod_{i'=1}^{I} \pi(f_{i'}|e_{a_{i'}})$$
$$= \frac{\epsilon}{(J+1)^I} \pi(f_i|e_j) \prod_{\substack{i'=1 \\ i' \neq i}}^{I} \sum_{j=0}^{J} \pi(f_{i'}|e_j)$$

where in the last equality we use the same trick as in Equation 2.4. By using Equation 2.3 for the denominator in Equation 2.5, we arrive at

$$q(a_i = j) = \frac{\pi(f_i|e_j)}{\sum_{j=0}^{J} \pi(f_i|e_j)}. \tag{2.6}$$

• **E-step:** In the E-step we calculate the expectation of the full log-likelihood $\log \pi(\mathbf{f}^*, \mathbf{a}^*|\mathbf{e}^*, \theta)$ with respect to q given in Equation 2.6. The full log-likelihood can be written as

$$\log \pi(\mathbf{f}^*, \mathbf{a}^*|\mathbf{e}^*, \theta) = \sum_{k=1}^{n} \log \pi(\mathbf{f}^{(k)}, \mathbf{a}^{(k)}|\mathbf{e}^{(k)}, \theta)$$
$$= \sum_{k=1}^{n} \log \pi(\mathbf{f}^{(k)}|\mathbf{a}^{(k)}, \mathbf{e}^{(k)}, \theta) + C$$
$$= \sum_{k=1}^{n} \sum_{\mathbf{a}} \delta(\mathbf{a}^{(k)}, \mathbf{a}) \log \pi(\mathbf{f}^{(k)}|\mathbf{a}, \mathbf{e}^{(k)}, \theta) + C$$
$$= \sum_{k=1}^{n} \sum_{\mathbf{a}} \delta(\mathbf{a}^{(k)}, \mathbf{a}) \sum_{i=1}^{I} \log \pi(f_i^{(k)}|e_{a_i}^{(k)}) + C$$
$$= \sum_{k=1}^{n} \sum_{\mathbf{a}} \delta(\mathbf{a}^{(k)}, \mathbf{a}) \sum_{i=1}^{I} \sum_{\substack{e \in V_E \\ f \in V_F}} \delta(e_{a_i}^{(k)}, e) \delta(f_i^{(k)}, f) \log \pi(f|e) + C$$

where C is a function not dependent of θ and $\delta(\cdot, \cdot)$ is the Kronecker-delta function. In the last step we used that

$$\log \pi(f_i^{(k)}|e_{a_i}^{(k)}) = \sum_{\substack{e \in V_E \\ f \in V_F}} \delta(e_{a_i}^{(k)}, e) \delta(f_i^{(k)}, f) \log \pi(f|e)$$

Taking the expectation we get

$$E_q[\log \pi(\mathbf{f}^*, \mathbf{a}^*|\mathbf{e}^*, \theta)] = \sum_{k=1}^{n} \sum_{\mathbf{a}} q(\mathbf{a}^{(k)} = \mathbf{a}) \sum_{i=1}^{I} \sum_{\substack{e \in V_E \\ f \in V_F}} \delta(e_{a_i}^{(k)}, e) \delta(f_i^{(k)}, f) \log \pi(f|e) + C$$

- **M-step:** In the M-step we want to maximize the expression above under the constraints

$$\forall e \in V_E : \sum_{f \in V_F} \pi(f|e) = 1 \tag{2.7}$$

where the sum is over all French words. This can be solved using the method of Lagrange multipliers. The Lagrangian is

$$\mathcal{L}(\theta, \lambda) = \sum_{k=1}^{n} \sum_{\mathbf{a}} q(\mathbf{a}^{(k)} = \mathbf{a}) \sum_{i=1}^{I} \sum_{\substack{e \in V_E \\ f \in V_F}} \delta(e_{a_i}^{(k)}, e) \delta(f_i^{(k)}, f) \log \pi(f|e)$$
$$- \sum_{e \in V_E} \lambda_e \Big(\sum_{f \in V_F} \pi(f|e) - 1 \Big) + C$$

and taking the derivative with respect to the parameter $\pi(f|e)$ we get

$$\frac{\partial \mathcal{L}}{\partial \pi(f|e)} = \sum_{k=1}^{n} \sum_{\mathbf{a}} q(\mathbf{a}^{(k)} = \mathbf{a}) \sum_{i=1}^{I} \delta(e_{a_i}^{(k)}, e) \delta(f_i^{(k)}, f) \frac{1}{\pi(f|e)} - \lambda_e. \tag{2.8}$$

We define the counts $c(f|e, \mathbf{f}, \mathbf{e})$ as the expected number of times the French word f aligns to the English word e in the English-French sentence pair $(\mathbf{e}, \mathbf{f})$. Mathematically, this is expressed as

$$c(f|e, \mathbf{f}^{(k)}, \mathbf{e}^{(k)}) = \sum_{\mathbf{a}} q(\mathbf{a}^{(k)} = \mathbf{a}) \sum_{i=1}^{I} \delta(e_{a_i}^{(k)}, e) \delta(f_i^{(k)}, f).$$

Even though it is possible to evaluate this expression, it is computationally expensive since it is a sum with an exponential number of terms (the number of alignments are $(J+1)^I$). By using Equation 2.6 and the same trick as in Equation 2.4 we can rewrite this as

$$c(f|e, \mathbf{f}^{(k)}, \mathbf{e}^{(k)}) = \frac{\pi(f|e)}{\sum_{j=0}^{J} \pi(f|e_j)} \underbrace{\sum_{i=1}^{I} \delta(f, f_i)}_{\text{count of f in } \mathbf{f}^{(k)}} \underbrace{\sum_{j=0}^{J} \delta(e, e_j)}_{\text{count of e in } \mathbf{e}^{(k)}}.$$

Putting the derivative in Equation 2.8 equal to 0 and using the definition of the counts, we get

$$\pi(f|e) = \frac{1}{\lambda_e} \sum_{k=1}^{n} c(f|e, \mathbf{f}^{(k)}, \mathbf{e}^{(k)}) \tag{2.9}$$

Finally, by using the constraints in Equation 2.7 we find that

$$\lambda_e = \sum_{f} \sum_{k=1}^{n} c(f|e, \mathbf{f}^{(k)}, \mathbf{e}^{(k)}). \tag{2.10}$$

This means that the Lagrange multipliers, λ_e, are normalizing factors which ensures that the probabilities sum up to 1.

The complete algorithm: The EM-algorithm for finding the lexical translation probabilities for IBM model 1 can be summarized as:

1. Choose initial parameters, for example $\pi(f|e)^{(0)} = \frac{1}{|V_F|}$ for all French and English words.
2. For every English-French sentence pair in the training data, compute the counts $c(f|e, \mathbf{f}, \mathbf{e})$ using the parameters $\pi(f|e)^{(k)}$.
3. For every English word compute λ_e according to Equation 2.10 and then compute the updated parameters $\pi(f|e)^{(k+1)}$ using Equation 2.9.
4. Repeat steps 2 and 3 until convergence.

Decoding: Now that we know the translation probabilities, we want to be able to extract the alignment vector $\mathbf{a}$ from an English-French sentence pair $(\mathbf{e}, \mathbf{f})$. This is done by finding the most likely alignment, i.e.

$$\hat{\mathbf{a}} = \underset{\mathbf{a}}{\text{argmax}}\ \pi(\mathbf{a}|\mathbf{e}, \mathbf{f}) = \underset{\mathbf{a}}{\text{argmax}}\ \pi(\mathbf{a}, \mathbf{f}|\mathbf{e}) = \underset{\mathbf{a}}{\text{argmax}}\ \prod_{i=1}^{I} \pi(f_i|e_{a_i})$$

so for every $i = 1, 2, \ldots, I$ we choose a_i such that $\pi(f_i|e_{a_i})$ is maximized.

2.1.2.2 Higher IBM Models

As mentioned before, a naive assumption made in model 1 is that the alignment probabilities are uniform, i.e. $\pi(a_i|i, I, J) = (J+1)^{-1}$. In model 2 these alignment probabilities are instead parameters, just like the translation probabilities, that are estimated under the constraint

$$\sum_{j=0}^{J} \pi(a_i = j|i, I, J) = 1.$$

It is likely that the probability mass of $\pi(a_i|i, I, J)$ is centered around values that are close to i. In model 3 the concept of *fertility* is introduced. The fertility of an English word e in a sentence pair $(\mathbf{e}, \mathbf{f})$ is the number of French words that align to it. Words with fertility 0 are called *infertile*. For example, the word *sat* in Figure 2.1 has fertility 2. Note that, since the alignment vector $\mathbf{a}$ is a random variable, the fertilities are also random. For every English word e_i in the vocabulary we have a probability distribution $n(\phi|e_i)$ representing the probability that e_i has fertility ϕ. Similarly to the translation- and alignment probabilities, these are parameters that are estimated. For more information about the IBM models we refer to the paper in [1].

2.2 Statistical Machine Translation

Besides from the obvious reason of using statistical machine translation to translate texts, it can also be used in an indirect way to measure the performance of word alignment. This is because word alignment is the key component in statistical

machine translation. The advantage of evaluating word alignment using machine translation is that it does not require any gold standard (labeled) alignments, something that could be cumbersome to obtain.

In this scenario of machine translation, we are given a French sentence $\mathbf{f}$ that is to be translated to English [5][6]. The most probable English sentence $\hat{\mathbf{e}}$ is chosen according to

$$\hat{\mathbf{e}} = \underset{\mathbf{e}}{\operatorname{argmax}}\ \pi(\mathbf{e}|\mathbf{f}) = \underset{\mathbf{e}}{\operatorname{argmax}}\ \pi(\mathbf{e})\pi(\mathbf{f}|\mathbf{e}).$$

The probability $\pi(\mathbf{f}|\mathbf{e})$ can be calculated using the generative process from any of the IBM models (see Section 2.1.2) together with an already trained word aligner. Using this we get

$$\hat{\mathbf{e}} = \underset{\mathbf{e}}{\operatorname{argmax}}\ \pi(\mathbf{e}) \sum_{\mathbf{a}} \pi(\mathbf{f}, \mathbf{a}|\mathbf{e}) \approx \underset{\mathbf{e}}{\operatorname{argmax}}\ \pi(\mathbf{e}) \max_{\mathbf{a}}\ \pi(\mathbf{f}, \mathbf{a}|\mathbf{e}). \tag{2.11}$$

The approximation made in the equation above removes the need of evaluating exponentially many alignments and thus making the computations tractable. Assuming IBM Model 2 we can write

$$\log(\pi(\mathbf{e})\pi(\mathbf{f}, \mathbf{a}|\mathbf{e})) = \log \epsilon + \log \pi(\mathbf{e}) + \sum_{i=1}^{I} \log(\pi(f_i|e_{a_i})\pi(a_i|i, I, J)). \tag{2.12}$$

In the next section we will describe how this function can be maximized and thus finding an English translation of the French sentence $\mathbf{f}$, a process also known as *decoding*.

2.2.1 Decoder

The maximization of Equation 2.12 over $(\mathbf{e}, \mathbf{a})$ is done using the A^* search algorithm [7]. This algorithm performs a search through the search space by building partial solutions (hypotheses) and storing them in a priority queue. The goal is to find a terminal hypothesis with the highest score. A hypothesis consists of the partially complete English translation and the alignment vector, i.e. $H : e_0 \dots e_k\ ;\ \overline{\mathbf{a_k}}$ where $\overline{\mathbf{a_k}}$ are the alignments for the aligned French words. A terminal hypothesis is a hypothesis where all French words are accounted for in the alignment. The score of a hypothesis H is $g(H) = a(H) + b(H)$ where $a(H)$ is called the prefix score and $b(H)$ is a heuristic. The prefix score is chosen as

$$a(H) = \log \epsilon + \log \pi(\mathbf{e}) + \sum_{i:a_i \in \overline{\mathbf{a_k}}}^{I} \log(\pi(f_i|e_{a_i})\pi(a_i|i, I, J)).$$

and can be thought of as the logarithm of the probability of hypothesis H. The function $b(H)$ is a heuristic that approximates $g(H^*) - a(H)$ where H^* is the terminal hypothesis with the highest score that is reachable from H. In a sense, $b(H)$ is the score of completing H in the best possible way. We require that $b(H^*) = 0$ for a terminal hypothesis H^*. All in all this means that for a good heuristic function $b(H)$ then $g(H)$ is close to $\log(\pi(\mathbf{e})\pi(\mathbf{f}, \mathbf{a}|\mathbf{e}))$ for the terminal hypothesis $H^* : \mathbf{e}\ ;\ \mathbf{a}$

with the highest score $g(H^*) = a(H^*) = \log(\pi(\mathbf{e})\pi(\mathbf{f}, \mathbf{a}|\mathbf{e}))$. The probability of observing the English sentence $\mathbf{e}$, $\pi(\mathbf{e})$, is calculated from an n-gram language model [8, p. 192-227].

From the current hypothesis (partial solution) a new hypothesis is created by performing any of the 4 operations described below. These operations are only valid for an IBM model that uses fertility. It is convenient to think of the hypotheses as nodes in a graph where there is an edge from H_1 to H_2 if H_2 can be reached from H_1 by doing an operation.

- **Add:** Add a new English word and align a French word to it.
- **AddZFert:** Add two new English words. The first has fertility 0 and the second is aligned to exactly one French word.
- **Extend:** Align a French word to the most recent English word.
- **AddNull:** Align a French word to NULL.

In **Add** we only consider the top 10 most likely English words, i.e. highest $\pi(e|f)$, for every unaligned French word. In **AddZFert** the possible infertile English words to be included are the ones that occur frequently and have a high probability of being infertile.

The A^* decoding algorithm can now be described as following:

1. Initialize a priority queue q with $H_0 : e_0\ ;\ \emptyset$, where $e_0 = NULL$.
2. Let H be the hypothesis with highest score $g(H)$ and remove it from the queue.
3. If H is a terminal hypothesis then output H and terminate.
4. Create new hypotheses from H in every possible way by performing the operations **Add**, **AddZFert**, **Extend** and **AddNull** and add them to the queue.
5. Go to step 2

The performance of the decoding algorithm is highly dependent on the heuristic function. With a heuristic function that underestimates the score of completing the hypothesis, shorter translations naturally get a higher score and we might end up with a suboptimal translation. For a heuristic function that always overestimates the score of completing the hypothesis, called an *admissible* heuristic, the optimal solution will always be found [9]. To build a simple heuristic, we first note that all French words must be aligned in a terminal hypothesis. A possible way of completing the hypothesis is then to align every unaligned French word to its most probable English word, i.e.

$$b(H) = \log \left(\prod_{j \in C(H)} \max_{e} \pi(f_j|e) \right)$$

where $C(H)$ is the set of indices of all unaligned French words.

2.2.2 BLEU Score

One of the most popular methods of evaluating a machine translation model is to use the BLEU score [10]. The central idea behind the BLEU score is that the closer a machine translation is to a professional human translation, the better it is. For a French sentence **f** there could be several different correct English translations. For this reason, we let C be a candidate translation (the output from the machine translation model) with m reference/correct translations $R_1, \ldots, R_m$. The BLEU score uses a modified n-gram precision, defined as

$$p_n = \frac{\sum_{n\text{-gram} \in C} count_{clip}(n\text{-gram})}{\sum_{n\text{-gram} \in C} count(n\text{-gram})}. \tag{2.13}$$

The function $count(n\text{-gram})$ is simply the number of times the n-gram occurs in C. The "clipped" n-gram counts are defined as: Let $R_j(n\text{-gram})$ be the number of times the n-gram occurs in R_j and

$$w(n\text{-gram}) = \max\{R_1(n\text{-gram}), \ldots, R_m(n\text{-gram})\}$$

then the clipped n-gram count is

$$count_{clip}(n\text{-gram}) = \min(count(n\text{-gram}), w(n\text{-gram})).$$

This means that the n-gram counts are clipped so that they can not be more than the maximum number of occurrences of the n-gram in the reference translations. If the candidate translation is equal to any of the reference translations then $p_n = 1$ since the counts will never be clipped. Modified n-gram precision for large n better captures the fluency of the candidate translation. Let's consider the following example

C: the cat the cat on the mat
R_1: the cat is on the mat
R_2: there is a cat on the mat

The counts and the clipped counts for the bigrams are

	the cat	cat the	cat on	on the	the mat
count	2	1	1	1	1
count$_{clip}$	1	0	1	1	1

and so the modified bigram precision is $p_2 \approx 0.667$. Usually one looks at p_1, p_2, p_3 and p_4 at the same time when calculating the BLEU score. The final formula for the BLEU score is

$$\text{BLEU} = BP \cdot e^{\frac{1}{4}\sum_{j=1}^{4} \ln p_j} \tag{2.14}$$

where BP is a factor that penalizes short candidate translations, since shorter translations naturally get a higher modified precision.

2.3 Feed Forward Neural Networks

A feed forward neural network is perhaps the simplest example of an artificial neural network. We will first present the architecture of feed forward neural networks and then how they are trained.

2.3.1 Architecture

As illustrated in Figure 2.2, the input is fed to the input layer and is then propagated through the network layer by layer until it reaches the output layer. The layers between the input and output layers are called hidden layers. A feed forward neural network with at least two hidden layers is said to be deep.

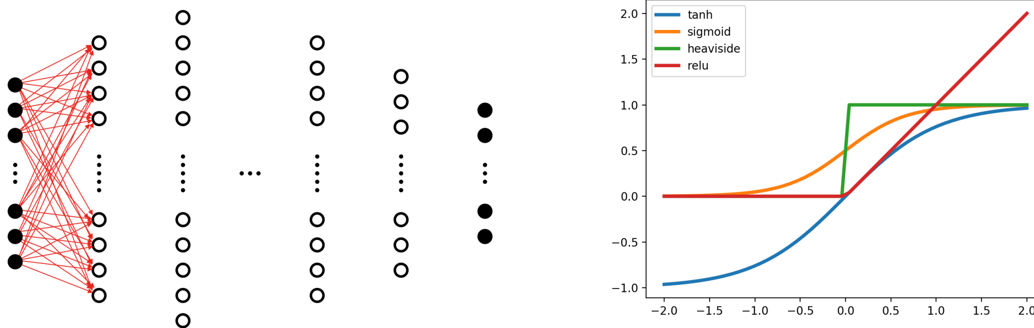


Figure 2.2: *Architecture of a neural network (left) and four examples of activation functions (right). The input is propagated through the network from left to right. The weights (red arrows, not drawn everywhere) are used to compute the value of each neuron in a layer based on the values of the neurons in the previous layer and the chosen activation function.*

The process is as follows. Let $\mathbf{x} = (x_1, ..., x_n)^T$ be the input vector. The j:th neuron in the first hidden layer is then computed with

$$n_j^{(1)} = g\Big(b_j^{(1)} + \sum_k \omega_{jk}^{(1)} x_k\Big) \tag{2.15}$$

where $b_j^{(1)}$ is the bias of the neuron, $\omega_{jk}^{(1)}$ are the weights connecting neuron j to all neurons in the previous layer and g is the activation function (examples of which are illustrated in Figure 2.2). Similarly, the j:th neuron in the ℓ:th layer is computed with

$$n_j^{(\ell)} = g\Big(b_j^{(\ell)} + \sum_k \omega_{jk}^{(\ell)} n_k^{(\ell-1)}\Big). \tag{2.16}$$

This is done for all layers from left to right until the output layer is reached. Here a different activation function is used which depends on the specific task. For classification, one normally normalizes the output using the softmax function, given by

$$n_j^L = \frac{e^{a_j}}{\sum_i e^{a_i}}, \quad \text{where } a_j = b_j^{(L)} + \sum_k \omega_{jk}^{(L)} n_k^{(L-1)}. \tag{2.17}$$

In classification, the number of output neurons is usually the same as the number of classes and the value of an output neuron is interpreted as the probability of classifying the input as the class corresponding to that neuron. The input is then classified as the class with the highest probability.

2.3.2 Training

The parameters to train in a feed forward neural network are the weights connecting the layers and the biases for each neuron (except the ones in the input layer). In supervised learning, the most common approach is stochastic gradient descent by backpropagation. In this approach, errors are first computed for the output layer given a loss function. These errors are then propagated backwards in the network to obtain the errors in the the hidden layers. Examples of loss functions are *quadratic loss* and *cross-entropy loss* given by

$$L_{QL}(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^{C} (y_i - \hat{y}_i)^2 \tag{2.18}$$

$$L_{CEL}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) = -\mathbf{y} \cdot \log \hat{\mathbf{y}} \tag{2.19}$$

where $\hat{\mathbf{y}}$ is the predicted probability vector (given an input $\mathbf{x}$), $\mathbf{y}$ is the target vector and C is the number of classes. The total loss is the mean loss for the data points in the training set. Using this loss, the gradients can be computed and the parameters updated. The update rule of a parameter p (a weight or a bias) is given by

$$p \leftarrow p - \eta \frac{\partial L}{\partial p} \tag{2.20}$$

where $\eta > 0$ is called the learning rate, which is a hyperparameter determining the level of exploration versus exploitation. One normally starts the training with a larger value of η (exploration) and then successively decreasing it to a smaller value (exploitation). It is common to train a network using mini-batches, which means that the network is fed with a portion (selected randomly at each step with a given size) of the data before computing the loss and updating the parameters. This has proven to increase the performance as it increases its ability of escaping local minima. It also has the advantage of a decreased use of RAM in case of heavy computations.

2.4 Contextual Representations

Many previous methods for generating continuous vector representations of words, such as Word2Vec [11][12] and GloVe [13], did not consider the context. For example, the word *right* in the sentence “That is the *right* answer” would have the same

vector representation as in the sentence "Take a *right* turn here". Creating context dependent representations of words is crucial for word-sense disambiguation since the sense of a word is defined by the context it is used in. One of the earlier models is the Bag-of-Words model [14], see Section 2.4.1. This model is very simple but still works well for many applications. Word2Vec (see Section 2.4.2) provides, in its basic form, word embeddings but can also be used to form contextual representations.

More recent models are mainly based on recurrent neural networks with LSTM units (for example ELMo [15]) or the transformer architecture released with the paper *Attention is all you need* [16]. Models based on the transformer architecture include XLNet [17], GPT-2 [18] and BERT [19]. BERT (or Bidirectional Encoder Representations from Transformers) provides contextual representations and is thoroughly explained in Section 2.4.3. This model obtains state-of-the-art results on 11 Natural Language Processing tasks [19].

2.4.1 Bag-of-Words

Bag-of-Words (BoW) [14] is one of the first models used for representing a text as a vector. Due to its simplicity it is easy to implement and can compete in performance with more advanced models on some problems. The idea is to create a vocabulary V and represent a text as a vector in $\mathbb{R}^{|V|}$ where the entry at index i is the number of times the ith word in the vocabulary occurs in the text. For example, suppose we have the three texts:

1. this movie is very scary and long
2. this movie is not scary and is slow
3. this movie is spooky

The vocabulary is $V = \{$this, movie, is, very, scary, and, long, not, slow, spooky$\}$ and the bag of words for the three texts will be

1. (1, 1, 1, 1, 1, 1, 1, 0, 0 ,0)
2. (1, 1, 2, 0, 1, 1, 0, 1, 1, 0)
3. (1, 1, 1, 0, 0, 0, 0, 0, 0, 1)

With a corpus consisting of many different texts the vocabulary will be very large. To reduce the dimension of the feature vectors one usually only looks at the most common words, for example the 90% most common words. A drawback with BoW is that it does not account for word order. For example, rearranging the words in a positive movie review could result in a negative movie review but they would both still have the same feature vector. Another drawback with this model is that similar words, for example *spooky* and *scary*, will be regarded as orthogonal to each other since they represent different entries in the feature vector. This is something that the model in the next section tries to solve.

2.4.2 Word2Vec

The idea of Word2Vec is that each word has a corresponding continuous vector representation where relationships between words can be expressed by mathematical operations. One common example is that the word *king* should be to *man* as *queen* is to *woman*, i.e.

$$\text{W2V}(king) - \text{W2V}(man) \approx \text{W2V}(queen) - \text{W2V}(woman)$$

Another example is the relationship between countries and their capital cities, i.e.

$$\text{W2V}(Germany) - \text{W2V}(Berlin) \approx \text{W2V}(France) - \text{W2V}(Paris)$$

One can see that these vectors could then be used to answer basic questions such as *What is the capital of France?* if the capital of Germany is known. Here the standard measure of dissimilarity between word vectors is the cosine distance. Although the idea of representing words as vectors has a long history, the real breakthrough came with Word2Vec in [11]. Two new architectures for training these representations were presented; *Continuous Bag-of-Words* and *Continuous Skip-gram*. The latter was later refined in [12] and this is the one we will consider here before describing how it can be used for contextual representations.

2.4.2.1 Skip-gram Model

The goal of the skip-gram architecture is to predict the surroundings of a given word and this is done by predicting the most likely words in a range $2c$ around the word. The architecture is often described in terms of a neural network with one hidden layer, but may just as well be described as two matrix multiplications followed by a softmax. The reason to this is that the hidden layer has neither any biases nor activation functions and the output layer has no biases. The input, a one-hot encoding $\mathbf{x} \in \mathbb{R}^{|V|}$, is first projected by a matrix $P \in \mathbb{R}^{|V|\times h}$, where h is the dimension of the word embedding (and the hidden layer). Next, the projection $\mathbf{p} \in \mathbb{R}^h$ is multiplied by $2c$ output matrices $M^{(i)} \in \mathbb{R}^{h\times|V|}$, where $i = \{-c, ..., -1, 1, ..., c\}$. This produces the scores $\mathbf{s}^{(i)} \in \mathbb{R}^{|V|}$. Finally the softmax-function is applied to each $\mathbf{s}^{(i)}$ to produce the output probabilities $\hat{\mathbf{y}}^{(i)} \in \mathbb{R}^{|V|}$, a procedure that can be summarized by the following equations

$$\mathbf{p} = \mathbf{x}P \quad \rightarrow \quad \begin{cases} \mathbf{s}^{(+c)} & = \mathbf{p}M^{(+c)} \quad \rightarrow \quad \hat{\mathbf{y}}^{(+c)} = \text{softmax}(\mathbf{s}^{(+c)}) \\ & \vdots \\ \mathbf{s}^{(+1)} & = \mathbf{p}M^{(+1)} \quad \rightarrow \quad \hat{\mathbf{y}}^{(+1)} = \text{softmax}(\mathbf{s}^{(+1)}) \\ \mathbf{s}^{(-1)} & = \mathbf{p}M^{(-1)} \quad \rightarrow \quad \hat{\mathbf{y}}^{(-1)} = \text{softmax}(\mathbf{s}^{(-1)}) \\ & \vdots \\ \mathbf{s}^{(-c)} & = \mathbf{p}M^{(-c)} \quad \rightarrow \quad \hat{\mathbf{y}}^{(-c)} = \text{softmax}(\mathbf{s}^{(-c)}) \end{cases}. \tag{2.21}$$

If one only wants to predict the surroundings of the word (encoded by $\mathbf{x}$), one looks at the index of the largest value in each $\hat{\mathbf{y}}^{(i)}$ (or $\mathbf{s}^{(i)}$ for that matter). During

training, the objective is to minimize the negative average log loss (similar to the cross-entropy loss in Equation 2.19)

$$-\frac{1}{N}\sum_{n=1}^{N}\sum_{-c\leq i\leq c, i\neq 0}\mathbf{y}_n^{(i)}\cdot\log\hat{\mathbf{y}}_n^{(i)} \tag{2.22}$$

where the loss is averaged over N words to which we want to predict the context. $\hat{\mathbf{y}}_n^{(i)}$ is the probability vector, while $\mathbf{y}_n^{(i)}$ is the corresponding one-hot-encoding of the target. Training is done using stochastic gradient descent. Improvements made in [12] include subsampling of frequent words (downsampling) and replacing the *soft-max*-function with negative sampling. For further details, we refer to the paper in [12].

Once the skip-gram model is trained, the matrix P is kept while the rest of the model is tossed. The rows in P are used as the word embeddings, meaning that the i:th row is the vector representation for word i in the vocabulary. Note that these vector representations are independent of the context that the word is used in.

2.4.2.2 Contextual Word2Vec

From the previous section we saw that the Word2Vec embedding of a word w in a sentence does not capture the context. Since the context is determined by the surrounding words, it is natural to think that the mean of the Word2Vec vectors of the surrounding words to w should be able to represent the context. Furthermore, since the mean does not take the word order into account this means that if a word w^* is always a neighbouring word to w then it is probably not important in determining the sense of w. For this reason it is better to use the following weighted sum of Word2Vec vectors to capture the context of w:

$$\sum_{-c\leq i\leq c} weight(w_i)\cdot W2V(w_i) \tag{2.23}$$

where $w_0 = w$ and $weight(w_i)$ is given by the normalized inverse document frequency (idf):

$$weight(w_i) = \frac{idf(w_i)}{\sum_{\text{-c}}^{c} idf(w_j)}.$$

The inverse document frequency of a word w is the negative logarithm of the percentage of documents (texts) that contains w, namely.

$$idf(w) = -\ln\left(\frac{|\{d\in D : w\in d\}}{N}\right)$$

where D is the set of all documents and N is the total number of documents. The idf puts a high weight on rare words and low weight on common words.

2.4.3 Bidirectional Encoder Representations from Transformers

A recent model that is able to create context dependent representations for words is ELMo [15], but it had the flaw of being unidirectional or only slightly bidirectional,

meaning that it looks only in one direction when predicting the next word conditioned on the previous words in the sentence. The same is true for GPT-2. The model presented in this section is a model from Google Brain called *Bidirectional Encoder Representations from Transformers*, or in short BERT [19] and is truly bidirectional.

Before using any kind of encoders, all words in the input sequence undergo tokenization and subword splitting, described in Section 2.4.3.1. In a second step they undergo word embedding, described in Section 2.4.3.2. This process consists of three parts: token embedding, segment embedding and positional embedding. In Figure 2.3 we see how the tokenized input sequence (in red) is word embedded before being fed into the first encoder.

There are various BERT models that mainly differ in size and the one considered here is *BERT Base* (Uncased). BERT Base features 12 stacked Transformer encoders [16] and an embedding size of 768. The main feature of the encoder is its use of attention which is described in Section 2.4.3.3 followed by a description of the entire encoder in Section 2.4.3.4.

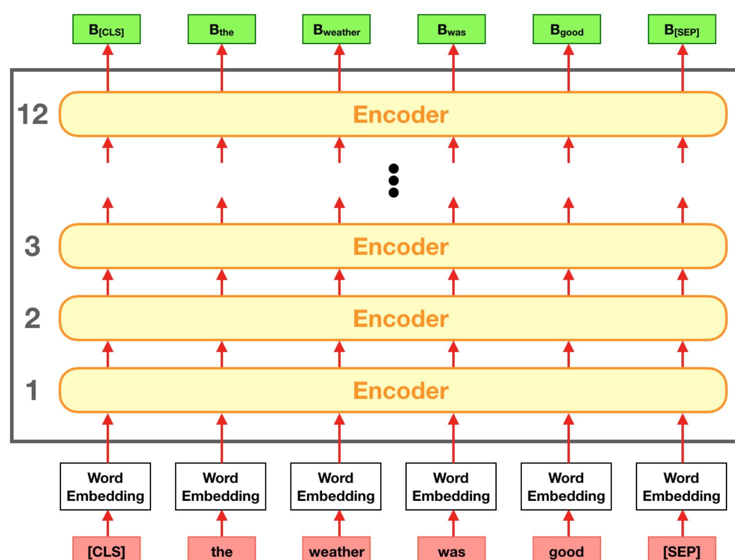


Figure 2.3: *An illustration of the BERT Base model. The tokenized input sequence at the bottom (in red) is word embedded, before it is fed to the first encoder. After passing through all 12 encoders, the BERT vectors are obtained at the top (in green).*

Apart from being deeply bidirectional, BERT also makes use of the vast amount of non-annotated texts on the web by performing a pre-training step, see Section 2.4.3.5, that enables it to learn general contextual embeddings and basic knowledge of language and texts. The same pre-trained BERT model can then be fine-tuned on different supervised tasks, meaning that all the parameters of the BERT model and the attached neural network that is used for the classification are trained.

2.4.3.1 Tokenization and Subword Splitting

In Natural Language Processing, tokenization is the process of chopping a text of words into smaller pieces, called tokens. In BERT, tokenization is done in the following steps:

1. Convert all characters to lower case.
2. Split all punctuation characters by adding a whitespace on both sides. Punctuation characters are all non-letter and non-number ASCII characters.
3. Apply subword splitting to every word. This process is described below.
4. If the text consists of a sentence pair then the two special tokens [CLS] and [SEP] are added as: [CLS] Sen A [SEP] Sen B [SEP] and if the text consists of a single sentence then the two tokens are added as: [CLS] Sen [SEP]. Here, a sentence represents a span of text and not an actual linguistic sentence.
5. Map every subword/token to its corresponding entry in the pre-defined vocabulary. Every token is then represented as a one-hot encoded vector.

Instead of having a huge vocabulary with almost every English word in the corpus, BERT uses a smaller vocabulary with around 30000 subwords. These subwords can then be used to build every other possible English word. For example, the word *playing* can be split up as *play* and *ing*. The advantage of using subwords is that it reduces the number of parameters that has to be learned.

A popular method that does subword splitting is Byte-Pair Encoding. This works by first splitting every word in the corpus into characters and then counting the word frequencies. For example, if our corpus consists of 5 occurrences of the word *low*, 2 occurrences of *lower*, 6 occurrences of *newest* and 3 occurrences of *widest* we get

$$\{l\ o\ w\ :\ 5,\quad l\ o\ w\ e\ r\ :\ 2,\quad n\ e\ w\ e\ s\ t\ :\ 6,\quad w\ i\ d\ e\ s\ t\ :\ 3\}$$

and our vocabulary will consist of the subwords $\{l,\ o,\ w,\ e,\ r,\ n,\ s,\ t,\ i,\ d\}$. The next step is to find the most frequent occuring subword pair and then merge these two subwords to a new subword and add it to the vocabulary. In this example, e and s occurs together in this order 6+3=9 times and is the most frequent occurring pair. These two subwords are then merged together as *es* and the counts are now

$$\{l\ o\ w\ :\ 5,\quad l\ o\ w\ e\ r\ :\ 2,\quad n\ e\ w\ \underline{es}\ t\ :\ 6,\quad w\ i\ d\ \underline{es}\ t\ :\ 3\}$$

and the vocabulary is $\{l,\ o,\ w,\ e,\ r,\ n,\ s,\ t,\ i,\ d,\ es\}$. In the next iteration the subword *es* will be merged with t producing the new subword *est*. This process is repeated until a desired vocabulary size is reached. In the encoding part, when a word is split into subwords, the vocabulary is iterated from the longest subword to the shortest and a check is made to see if any subword is a substring of the word we want to encode. This is repeated until the word is split entirely into subwords. Note that this is always possible since the vocabulary will contain all characters for a sufficiently large corpus.

2.4.3.2 Word Embedding

Each word embedding is a sum of three parts: token embedding, segment embedding and positional embedding (see Figure 2.4). As described in Section 2.4.3.1, a one-hot encoded vector $v \in \mathbb{R}^{|V|}$ is obtained from tokenization using a pre-defined vocabulary of length $|V|$. The token embedding for v is then obtained by multiplying it with a matrix $M \in \mathbb{R}^{|V| \times d_x}$, where $d_x = 768$ is the length of the word embedding vector. All parameters in M are learned in the training process of BERT. In the segment embedding, a vector $E_A \in \mathbb{R}^{d_x}$ is added to every entry in the first sentence and another vector $E_B \in \mathbb{R}^{d_x}$ is added to every entry in the the second sentence. E_A and E_B are both parameters to be learned. The purpose of the segment embedding is to differ between the two sentences. This is especially important in the next sentence prediction pre-training task. Finally, the positional embeddings are

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_x}}\right)$$
$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_x}}\right)$$

where pos is the position of the token within the text and $i \in \{1, 2, \ldots, d_x\}$. This means that for a fixed dimension i, the positional encoding is either a *sin* or a *cos* function. The idea is that by adding these positional embeddings the model should be able to learn relative positions of tokens.

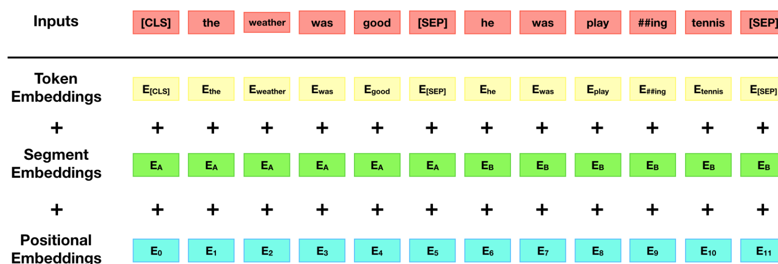


Figure 2.4: *Each word is embedded with the sum of its token embedding, segment embedding and positional embedding. We call this sum the word embedding of the word.*

2.4.3.3 Attention

Attention was first proposed in [20] and later refined in [21] to highly improve the performance of recurrent neural networks in machine translation systems. It is a mechanism that allows extra *attention* to be concentrated at the more relevant parts of the input sequence. When looking at an image, for example, one pays extra attention to the more defining parts while not focusing so much on the background. This is also done automatically when we read texts. We focus more on the more relevant words and less on the less important words. The attention mechanism used in BERT was first proposed in [16] with the introduction of the Transformer. This attention mechanism uses multiple attention heads (multi-headed attention), each of which uses self-attention. We will first describe self-attention, the basic approach, and then move on to multi-headed attention.

Self-attention

Consider an input sequence of vectors $\mathbf{x_1}, ..., \mathbf{x_n}$ of the same length d_x and define $X \in \mathbb{R}^{n \times d_x}$ as the input matrix where the rows are $\mathbf{x_1}, ..., \mathbf{x_n}$. In the same way we define the output matrix $Y \in \mathbb{R}^{n \times d_y}$ with the output sequence vectors $\mathbf{y_1}, ..., \mathbf{y_n}$ of the same length d_y. Note that $d_x = d_y = 768$ in BERT Base, but for the purposes of generality we keep these separate here.

The main idea of self-attention is that each output $\mathbf{y_i}$ depends on all inputs, where the attention mechanism makes sure that the dependence is larger for the more relevant parts. Each input $\mathbf{x_j}$ is first split into key-value-query triplets, namely the vectors $\mathbf{k_j}$, $\mathbf{v_j}$ and $\mathbf{q_j}$. In vector and matrix form, this split is done in the following way

$$\begin{aligned} \mathbf{k_j} &= \mathbf{x_j} W^k \quad \rightarrow \quad K = XW^k \\ \mathbf{v_j} &= \mathbf{x_j} W^v \quad \rightarrow \quad V = XW^v \\ \mathbf{q_j} &= \mathbf{x_j} W^q \quad \rightarrow \quad Q = XW^q \end{aligned} \tag{2.24}$$

where $W^k \in \mathbb{R}^{d_x \times d_{kq}}$, $W^v \in \mathbb{R}^{d_x \times d_y}$ and $W^q \in \mathbb{R}^{d_x \times d_{kq}}$ are matrices that must be learned during the training process and d_{kq} is chosen arbitrarily. We can also see that $K \in \mathbb{R}^{n \times d_{kq}}$, $V \in \mathbb{R}^{n \times d_y}$ and $Q \in \mathbb{R}^{n \times d_{kq}}$.

The next step is to compute the attention scores a_{ij} from the key $\mathbf{k_j}$ and query $\mathbf{q_i}$ using a scoring function $\text{Score}(\mathbf{k_j}, \mathbf{q_i})$. The scoring function used in BERT is the so called *dot product attention* given by

$$a_{ij} = \frac{\mathbf{k_j} \cdot \mathbf{q_i}}{\sqrt{d_{kq}}} \quad \rightarrow \quad A = \frac{QK^T}{\sqrt{d_{kq}}} \tag{2.25}$$

where $A \in \mathbb{R}^{n \times n}$ is the attention score matrix with elements a_{ij}. Dividing by $\sqrt{d_{kq}}$ attempts prevent the gradients from exploding. The next step is to apply the softmax function across each row of A, given by

$$s_{ij} = \frac{\exp(a_{ij})}{\sum_k \exp(a_{ik})} \quad \rightarrow \quad S = \text{softmax}_{\text{row}}(A) \tag{2.26}$$

Finally, the output $\mathbf{y_i}$ computed by a weighted sum of the values $\mathbf{v_j}$, where the weights are s_{ij}.

$$\mathbf{y_i} = \sum_j s_{ij} \mathbf{v_j} \quad \rightarrow \quad Y = SV \tag{2.27}$$

Note that there are only three parameter matrices that have to be trained, namely W^k, W^v and W^q. The entire self-attention procedure can be summarized with the following four equations

$$\begin{aligned} K &= XW^k \\ V &= XW^v \\ Q &= XW^q \\ Y &= \text{softmax}_{\text{row}}\Big(\frac{QK^T}{\sqrt{d_{kq}}}\Big) V \end{aligned} \tag{2.28}$$

Multi-headed Attention

Multi-headed attention is an extension of self-attention. Self-attention is performed multiple times producing multiple *heads*. For each head, different matrices W_h^k, W_h^v and W_h^q are used. The resulting output head matrices Z_h are then concatenated from left to right into the matrix Z (Z_h corresponds to Y in the previous section) in the following way

$$Z = \text{Concat}(Z_1, ..., Z_{n_h}) \tag{2.29}$$

where $Z \in \mathbb{R}^{n \times d_h n_h}$. n_h is the number of heads and d_h corresponds to d_y in the previous section. Note that the head size d_h is equal to the final output size d_y in BERT, but this must not be the case in general. By introducing a new matrix $W^{MH} \in \mathbb{R}^{d_h n_h \times d_y}$, we get the desired output Y in the following way

$$Y = ZW^{MH} \tag{2.30}$$

where W^{MH} is learned during the training process. To summarize, the parameter matrices that are trained are W_h^k, W_h^v, W_h^q and W^{MH} where $h = 1, \dots, n_h$.

The advantage of using multiple heads is an increase in performance due to an expansion of the attention mechanism. This means that the ability to focus on different positions when embedding a word is increased and the model now has has many possible representations given by each head. Another advantage of increasing the complexity in this way is that the training time is not affected in a significant way. This is due to the fact that the calculations for each head are independent and can therefore be parallelized.

2.4.3.4 The Encoder

BERT Base uses 12 Transformer encoders and the first of these is displayed in Figure 2.5. The first input undergoes multi-headed attention with eight heads followed by a residual connection with layer normalization. Then a feed forward neural network with one hidden layer follows, which can be summarized as the function

$$FFN(\mathbf{x}) := \max(0, \mathbf{x}W_1 + \mathbf{b_1})W_2 + \mathbf{b_2} \tag{2.31}$$

where the input vector $\mathbf{x} \in \mathbb{R}^{768}$ should not be confused with the $\mathbf{x_i}$ in Figure 2.5. We also have that $\mathbf{b}_1 \in \mathbb{R}^{3072}$, $\mathbf{b}_2 \in \mathbb{R}^{768}$, $W_1 \in \mathbb{R}^{768 \times 3072}$, $W_2 \in \mathbb{R}^{3072 \times 768}$ are trained during the training process and identical across word positions but differ from one encoder to another. This is then followed by another residual connection with layer normalization before being fed to the next encoder.

For a vector $\mathbf{x}_i$, the residual connection with layer normalization of a layer (either multi-headed attention or feed forward neural network) can mathematically be expressed as

$$\text{LayerNorm}(\mathbf{x} + \text{Layer}(\mathbf{x})) \tag{2.32}$$

For deep neural networks, where only a subset of the layers are necessary, it is sometimes difficult for the network to learn the identity function for the layers that are

not necessary. Residual connections, have been shown to greatly reduce this problem as the network does not have to learn the identity function of these layers but instead just setting them to zero.

The layer normalization function, proposed in [22], is defined as

$$\text{LayerNorm}(\mathbf{z}) = \boldsymbol{\gamma} \odot \left(\frac{\mathbf{z} - \mu_z}{\sigma_z}\right) + \mathbf{b} \tag{2.33}$$

where $\mathbf{b}$ and $\boldsymbol{\gamma}$ are parameters to be learned, $\odot$ is the element-wise multiplication, μ_z is the mean of $\mathbf{z}$ and σ_z is the standard deviation of $\mathbf{z}$. Note that the mean and variance are across the feature dimensions. Layer normalization has been shown to greatly reduce training time [22].

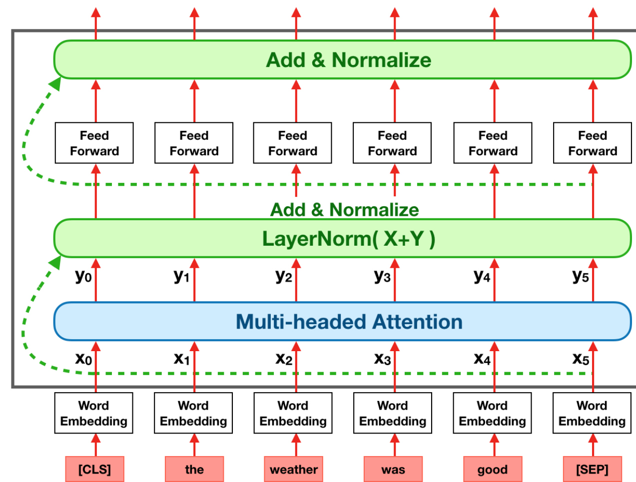


Figure 2.5: *Visualization of the first BERT encoder. The input sequence is passed through a multi-headed attention layer followed by a feed forward neural network layer (containing one hidden layer). Both layers are followed by a residual connection and layer normalization. The output is then passed to the second encoder.*

2.4.3.5 Pre-training

BERT is pre-trained on texts from BooksCorpus (800M words) and the entire English Wikipedia (2500M words) with tables and headers ignored. The two unsupervised tasks that it is jointly trained on are: Masked Language Model (Masked LM) and Next Sentence Prediction (NSP).

Task 1: Masked LM. By training a masked language model, BERT learns to use the neighbouring words, the context, when creating the vector representation of a word. This is done by randomly replacing 15% of all words by the token [MASK] and then try to predict the replaced word by feeding the vector representation of the [MASK] token from the last encoder into a feed forward neural network with a softmax output over the vocabulary. A problem with this approach is that BERT

will only learn to predict the correct token for the masked tokens, while we want it to predict the correct token for all tokens in the input sequence. To solve this, out of the 15% selected tokens:

- 80% are replaced with [MASK]
- 10% are left unchanged
- 10% are replaced by a randomly chosen token from the vocabulary

Task 2: NSP. In the NSP task, BERT receives a pair of sentences, Sen A and Sen B, and should predict whether or not Sen B is the next sentence after Sen A or a random sentence. The goal of this task is to learn relationships among sentences. The training data is generated such that for a pair of consecutive sentences Sen A and Sen B, 50% of the time Sen B is replaced by a randomly chosen sentence from the training data and 50% of the time left unchanged. The actual prediction is done by feeding the output of the [CLS] token from the last encoder into a feed-forward neural network with a sigmoidal output.

The loss function in both tasks is the cross-entropy loss, see Equation 2.19. In BERT, the two pre-training tasks are trained together in order to minimize the combined loss which is the the sum of the loss from task 1 and task 2. The training data for the pre-training of BERT is generated by first tokenizing every sentence, then applying the masking procedure and finally for every pair of consecutive sentences, choose the second sentence randomly 50% of the time and leave it unchanged 50% of the time. Usually the text corpus is also duplicated around 5 times. Below is an example of a training point.

[CLS] The boy sat by the [MASK] and looked out [SEP]
He wants to [MASK] outside and play [SEP]

NSP label: isNext=True
Masked LM label: window, go

2.5 Classification

In this section we describe the general theory behind the classifiers: k-Nearest Neighbours (kNN), Support Vector Machines (SVM), Random Forests (RFC) and Gaussian Naive Nayes (GNB). Different classifiers are more suitable for some problems while other classifiers are better for others. Although it might be intuitive that a certain classifier works well for a certain problem, the only way to find this out is to evaluate its performance on test data and compare the results to the ones from other classifiers.

2.5.1 k-Nearest Neighbours

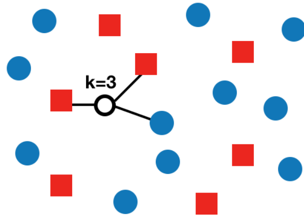


Figure 2.6: *k-nearest neighbours.*

In k-nearest neighbours a feature x is classified using the labels of its k nearest neighbours. In the simple case when $k = 1$, x is then assigned the same class as its nearest neighbour. For $k > 1$, a majority vote among the k nearest neighbours is performed. To compute the distance between two points x and z, the Euclidean distance function is normally used

$$d(x, z) = \sqrt{\sum_i (x_i - z_i)^2} \tag{2.34}$$

Note that there are more distance functions, including the Manhattan and Minkowski distance functions for continuous features and the Hamming distance for discrete features.

The k-nearest neighbour classifier is exemplified in Figure 2.6. Blue circles and red squares represent the two classes, while the unfilled circle represent the data point which is about to be classified. In this example, $k = 3$ so we look at the 3 nearest neighbours. Among these we have two red squares and one blue circle, which means that the point will be classified as a red square.

2.5.2 Support Vector Machines

The support vector machine (SVM) [23] classifier is in its basic form a linear non-probabilistic binary classifier. However, there are methods to make it non-linear, probabilistic and non-binary. This section starts by introducing the basic SVM followed by a description of the kernel trick, which extends it to a non-linear classifier. Furthermore, it is explained how Platt scaling is used to make SVM a probabilistic classifier and finally how multiple (binary) SVMs can be combined to make it suitable for multiclass classification.

2.5.2.1 Binary Classification

We assume that our training data is $\{\mathbf{x_i}, y_i\}_{i=1}^n$ where $\mathbf{x_i} \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$. SVMs try to find a hyperplane that separates the two classes as much as possible. Any hyperplane can be written as $\mathbf{w} \cdot \mathbf{x} + b = 0$ where $\mathbf{w} \in \mathbb{R}^d$ is a normal vector to the hyperplane. Assuming that the two classes are linearly separable and that the plane $\mathbf{w} \cdot \mathbf{x} + b = 0$ separates the two classes, there exists two other parallel hyperplanes $\pi_1 : \mathbf{w} \cdot \mathbf{x} + b = 1$ and $\pi_{-1} : \mathbf{w} \cdot \mathbf{x} + b = -1$ separating the two classes with maximum distance between them. The points in the training data that lies on π_1 or π_{-1} are called support vectors and the distance between π_1 and π_{-1} is called the margin.

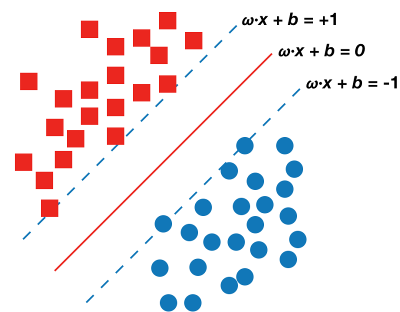


Figure 2.7: *Support Vector classifier*

There could be many values for $\mathbf{w}$ and b that separate the two classes but we want to find the $\mathbf{w}$ and b that maximize the margin, i.e. the distance between π_1 and $\pi_{\text{-}1}$.

Suppose that $\mathbf{x_0} \in \pi_{\text{-}1}$, then there exists $r > 0$, the distance between π_1 and $\pi_{\text{-}1}$, such that the point $\mathbf{x_0} + r\frac{\mathbf{w}}{||\mathbf{w}||} \in \pi_1$, i.e.

$$\mathbf{w} \cdot (\mathbf{x_0} + r\frac{\mathbf{w}}{||\mathbf{w}||}) + b = 1.$$

By using that $\mathbf{w} \cdot \mathbf{x_0} + b = -1$ and solving for r we get that $r = \frac{2}{||\mathbf{w}||}$. Finding the hyperplane that maximizes the margin can thus be formulated as solving the optimization problem

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}||\mathbf{w}||^2 \\ \text{s.t.} \quad & y_i(\mathbf{w} \cdot \mathbf{x_i} + b) \geq 1, \quad i = 1, \dots, n. \end{aligned} \tag{2.35}$$

A test point $\mathbf{x}$ is classified by $\text{sgn}(f(\mathbf{x}))$ where $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$.

Not allowing any training points to be in the margin, the region between π_1 and $\pi_{\text{-}1}$ is called hard-margin SVM and is only solvable when the two classes are linearly separable. When this is not the case, we can relax the constraints in Equation 2.35 by introducing the error terms $\delta_1, \dots, \delta_n$ where δ_i is the distance from π_{y_i} to $\mathbf{x_i}$ if $\mathbf{x_i}$ is in the margin and 0 otherwise. The optimization problem in the soft-margin SVM is

$$\begin{aligned} \min_{\mathbf{w},b,\boldsymbol{\delta}} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_i \delta_i \\ \text{s.t.} \quad & y_i(\mathbf{w} \cdot \mathbf{x_i} + b) \geq 1 - \delta_i, \quad \forall i \\ & \delta_i \geq 0, \quad \forall i. \end{aligned} \tag{2.36}$$

The hyperparameter $C \geq 0$ determines the penalty for a training point to be in the margin. Note that the constraint $y_i(\mathbf{w} \cdot \mathbf{x_i} + b) \geq 1 - \delta_i$ will be an equality if and only if $\mathbf{x_i}$ is in the margin.

Instead of solving the primal problems given in Equation 2.35 and Equation 2.36 it turns out to be more beneficial to instead solve the corresponding dual problem. Below we formulate the (Wolfe) dual problem for hard-margin SVM.

$$\begin{aligned} \max_{\mathbf{w},b,\boldsymbol{\alpha}} \quad & L(\mathbf{w}, b, \boldsymbol{\alpha}) \\ \text{s.t.} \quad & \nabla_{\mathbf{w},b} L(\mathbf{w}, b, \boldsymbol{\alpha}) = 0 \\ & \alpha_i \geq 0, \quad \forall i. \end{aligned} \tag{2.37}$$

where

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}||\mathbf{w}||^2 + \sum_{i=1}^{n} \alpha_i \left(1 - y_i(\mathbf{w} \cdot \mathbf{x_i} + b)\right)$$

is the Lagrangian. The gradient of the Lagrangian with respect to $\mathbf{w}$ and b can be calculated as

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_i \alpha_i y_i \mathbf{x_i} = 0$$
$$\frac{\partial L}{\partial b} = -\sum_i \alpha_i y_i = 0$$

and plugging this into the Lagrangian we obtain

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x_i} \cdot \mathbf{x_j}.$$

Note that this expression is only dependent on the multipliers α_i. The final optimization problem can thus be formulated as

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x_i} \cdot \mathbf{x_j} \\ \text{s.t.} \quad & \alpha_i \geq 0, \quad \forall i \\ & \sum_i \alpha_i y_i = 0. \end{aligned} \tag{2.38}$$

Which can efficiently be solved using Sequential Minimal Optimization (SMO) [23]. It turns out that the support vectors correspond to those training points having $\alpha_i > 0$. From the solution of this optimization problem we can find $\mathbf{w}$ by $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x_i}$ and b by $b = y_k - \mathbf{w} \cdot \mathbf{x_k}$ where $\mathbf{x_k}$ is a support vector. The dual problem and its solution for soft-margin SVM and be found and solved similarly.

2.5.2.2 The Kernel Trick

Since SVMs makes the classification based on the decision function $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$ it is a linear classifier and can only produce linear decision boundaries. To overcome this we can map all points in the original features space, $\mathbb{R}^m$, to a new space, $\mathbb{R}^d$, and hope that the two classes in this new space can easily be separated by a hyperplane. For this reason we define a kernel function as

$$k(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$$

where $\phi : \mathbb{R}^m \to \mathbb{R}^d$. To find the maximum-margin hyperplane in $\mathbb{R}^d$ the inner product in Equation 2.38 is simply changed to $k(\mathbf{x_i}, \mathbf{x_j})$ and the classification is determined by the function $f(\mathbf{x}) = \mathbf{w} \cdot \phi(\mathbf{x}) + b$. To make classifications and find $\mathbf{w}$ and b we never actually need the map $\phi(\mathbf{x})$, all we need is the kernel function $k(\mathbf{x}, \mathbf{y})$. However, not all functions are kernels, Mercer's theorem [23] gives sufficient and necessary conditions for a function to be a kernel function. The most common kernel functions are polynomial kernels $k(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + c)^d$ and the RBF kernel $k(\mathbf{x}, \mathbf{y}) = e^{-\frac{1}{2\sigma^2}||\mathbf{x}-\mathbf{y}||^2}$.

2.5.2.3 Probabilistic Classifier

In some cases we would like to know how accurate a classification is, i.e. for a data point $\mathbf{x}$ we want to calculate $p = P(y = 1|\mathbf{x})$. Normal Support Vector Machines does not give this information, as it just classifies depending on which side of the hyperplane the point is in, but it is still possible to estimate these probabilities using a technique called Platt scaling [24]. We assume that the decision function is $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$ where $\mathbf{w}$ and b are found by solving the corresponding optimization problem for SVMs. The probability that $\mathbf{x}$ belongs to class 1 can be calculated as

$$P(y = 1|\mathbf{x}) = \frac{1}{1 + e^{Af(\mathbf{x})+B}} \tag{2.39}$$

where A and B are scalar parameters and can be estimated using maximum likelihood.

2.5.2.4 Multiclass Classification

Support Vector Machines is by its nature a binary classifier. We can extend SVMs to make classification on the K classes $C_1, C_2, \ldots, C_K$ by for every pair of classes (C_i, C_j) training a binary SVM classifier only on the training points from class C_i and C_j and then for a test point x we classify it by taking a majority vote over all $\frac{K(K-1)}{2}$ classifier. This strategy is known as *one-vs-one* and works quite well but could be computationally expensive when there are many classes. Also, when there is a tie one of the classes is selecteed randomly which is not optimal.

In the previous section we saw that Platt scaling could be used to get probability estimates for binary SVMs. Next we will see that it is possible to get probability estimates also in the multiclass setting. For a point x we define

$$p_k = P(y = k|\mathbf{x}) \quad \text{and} \quad r_{k\ell} = P(y = k|y = k \text{ or } y = \ell, x)$$

where $r_{k\ell}$ can be calculated using (2.39). The goal is to find p_i for all i. From Bayes theorem we obtain the equality

$$r_{k\ell}p_\ell = r_{\ell k}p_k.$$

We want this to hold under the constraint that $\forall i : \ p_i \geq 0$ and they sum up to 1. There is no guarantee that there exists a solution to this. Instead we relax the problem and solve the following optimization problem

$$\begin{aligned} \min_{\mathbf{p}} \quad & \sum_{k,\ell} (r_{k\ell}p_\ell - r_{\ell k}p_k)^2 \\ \text{s.t.} \quad & p_i \geq 0, \quad \forall i \\ & \sum_i p_i = 1. \end{aligned} \tag{2.40}$$

In [25] they describe thoroughly how this optimization problem can be solved efficiently.

2.5.3 Random Forest Classifier

The random forest classifier, first proposed in [26], is based on the concept of decision trees. The goal of decision trees is to split the feature space into regions $R_1, ..., R_J$, where each region R_j is associated with one class, such that the classification error is minimized. The fundamental concept is to build a tree with binary decisions, which results in the classification regions. A simple example is displayed in Figure 2.8, where the data has two different features (color and shape). In the first step the data is split according to shape, while in the second step (of the right node) it is split according to color. The resulting leaf nodes are then considered to be regions of the feature space and are associated with one class according to a majority vote.

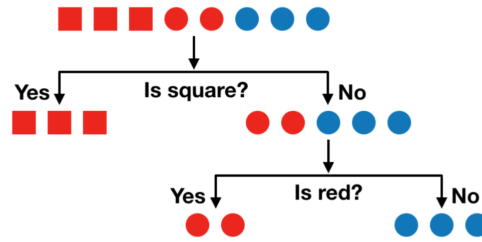


Figure 2.8: *A simple decision tree. The data contains two features, namely color and shape. Each leaf node is considered a region which is associated with one class.*

When splitting a node, the split is done such that the decrease in entropy from the split is maximized. If we let x_j be the feature and v be the value to make the split according to, then the two regions after the split can be expressed as

$$R_1(j, v) = \{\mathbf{x} | x_j < v\}$$
$$R_2(j, v) = \{\mathbf{x} | x_j \geq v\}$$

Now, if we denote the error after the split in region s as $E_s(j, v)$, then we want to find j and v such that

$$\frac{D_1}{D} E_1(j, v) + \frac{D_2}{D} E_2(j, v) \tag{2.41}$$

is minimized. Here, D, D_1 and D_2 are the number of observations in the node we want to split, in region R_1 and in region R_2, respectively. It is natural to use the classification error rate here, but it turns out that two other measures work better for growing trees. These are the Gini index E_{gini} and entropy E_{entr} given by

$$E_{\text{gini}} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$
$$E_{\text{entr}} = -\sum_{k=1}^{K} \hat{p}_{mk} \log_2(\hat{p}_{mk})$$

where $\hat{p}_{mk}$ is the estimated probability of a point belonging to class k in region m and K is the number of classes in the node. For the Gini index it is clear that its terms take values near 0 if $\hat{p}_{mk}$ is close to 0 or 1. The same can be shown for the entropy. After making the first split, the two resulting regions are split again and the process continues until a stopping criterion is reached. Normally this stopping criterion is a minimum number of observations in each node.

One problem with decision trees, and in particular deep decision trees, is that they have a high variance. This means that if we train it on a random sample (with the same size) of the original data set, it might yield significantly different results. In random forest, this problem is dealt with using bagging. This means that multiple trees are used and trained on different bootstrapped samples of the data set. The individual trees are then aggregated such that the final prediction is a majority vote among the predictions of the individual trees, i.e.

$$\hat{y}(\mathbf{x}) = \underset{1 \leq i \leq T}{\text{Majority}}(\hat{y}_i(\mathbf{x})) \tag{2.42}$$

where T is the number of trees and $\hat{y}_i(\mathbf{x})$ is the prediction from tree i. Another problem is that the individual trees are highly correlated with each other. The random forest classifier uses a decorrelation mechanism, in which only a random sample of the features is used each time a node is split. The number of features that are available at each split is usually set to $\sqrt{p}$, where p is the total number of features. This introduces a lot of variability between the trees and therefore decorrelates them.

2.5.4 Gaussian Naive Bayes Classifier

Any Naive Bayes classifier makes the (naive) assumption that the features are conditionally independent given the class. We assume that our features for $\mathbf{x}$ are represented as $\mathbf{x} = (x_1, ..., x_n)$ and the label is $y \in \{y_1, ..., y_m\}$. Using Bayes theorem we get

$$P(y = y_k|\mathbf{x}) = \frac{P(\mathbf{x}|y_k)P(y_k)}{P(\mathbf{x})} = \frac{P(y_k)\prod_i P(x_i|y_k)}{\prod_i P(x_i)} \propto P(y_k)\prod_i P(x_i|y_k) \tag{2.43}$$

So the goal of any Naive Bayes classifier is to find the y that maximizes the probability above, i.e.

$$y = \underset{k}{\text{argmax}}\; P(y_k)\prod_i P(x_i|y_k) \tag{2.44}$$

For continuous features another assumption is made, namely that the features are normally distributed (Gaussian Naive Bayes).

$$P(x_i|y_k) = \frac{1}{\sqrt{2\pi\sigma_{i,k}^2}} \exp\left(-\frac{(x_i - \mu_{i,k})^2}{2\sigma_{i,k}^2} \right) \tag{2.45}$$

where $\mu_{i,k}$ and $\sigma_{i,k}^2$ are the mean and variances associated with the class y_k, respectively. The training of Gaussian Naive Bayes consists of estimating these parameters.

2.6 Self-training

Self-training [27] is a semi-supervised technique that makes use of both labeled and unlabeled data. It works by training a probabilistic classifier on the labeled data and then labels the unlabeled data that the classifier is most certain about. This process

is repeated until all the unlabeled data is labeled or until a maximum number of iterations is reached

```
1: Input: L : labeled data, U : unlabeled data, F : classifier, T : threshold for
   selection and iter_max : maximum number of iterations
2: Output: A trained classifier H
3: procedure SELF-TRAINING(U, L, F, T, iter_max)
4:     t ← 1
5:     while |U| > 0 and t < iter_max do
6:         H ←TrainClassifier(F, L, U)
7:         S ← set()
8:         for x ∈ U do
9:             p ← H(x, prob=True)          ▷ Probability of most likely class
10:            if p ≥ T then
11:                S.add(x)
12:        L ← L + S                         ▷ Union
13:        U ← U − S                         ▷ Difference
14:        t ← t + 1
15:    H ←TrainClassifier(F, L, U)
16:    return H
```

2.7 Bayesian Optimization for Tuning Hyperparameters

The problem of finding a classifier with the best possible hyperparameters is known as hyperparameter optimization. These hyperparameters define the model's architecture. It is important not to confuse hyperparameters with model parameters, as hyperparameters can not directly be learnt from the data. The problem of finding the optimal set of parameters can be formulated as finding the value maximising

$$D : \Theta \to \mathbb{R}_+$$

where Θ is the hyperparameter space, usually a subset of $\mathbb{R}^d$. The function D takes as input a set of hyperparameters and then trains a classifier with these parameters and outputs the model's performance by doing cross-validation on the training data for a chosen metric. Since D is very expensive and time-consuming to evaluate, many clever ways to maximize this function have been developed. Because Θ is usually finite, one naive method is to search the entire space and evaluate D at every point. When Θ is large, this method doesn't work that well. One of the most popular ways of maximising D is by using Bayesian Optimisation, explained in [28]. In Bayesian statistics unknown quantities such as D are modeled as random variables. In this particular method we assume a Gaussian prior for D over Θ, meaning that for every $k \in \mathbb{Z}_+$ and $x_1, \ldots, x_k \in \Theta$ it holds that

$$[D(x_1), \ldots, D(x_k)] \sim N(\mu, \Sigma)$$

where μ is the prior mean. This mean is $\mu(x) = \mu_{\max(m\text{-}1,0)}(x)$ (see Equation 2.46) where m is the number of points that D has been evaluated at. The initial mean μ_0 can be chosen arbitrarily. The covariance matrix is $\Sigma(x_i, x_j) = e^{-||x_i - x_j]]}$ and is chosen such that points closer to each other have similar function values. Suppose D is evaluated at $x_1, \ldots, x_k$, then for a new point x_{k+1} it holds that the following is multivariate normal: $[D(x_1), \ldots, D(x_k), D(x_{k+1})]$ and then the posterior is also normal, i.e.

$$D(x_{k+1})|\,[D(x_1), \ldots, D(x_k)] \sim N\left(\mu_k(x_{k+1}), \sigma_k^2(x_{k+1})\right)$$

where

$$\begin{aligned}\mu_k(x_{k+1}) &= \Sigma(x_{k+1}, x_{1:k})\Sigma(x_{1:k}, x_{1:k})^{-1}(D(x_{1:k}) - \mu_{k\text{-}1}(x_{1:k})) + \mu_{k\text{-}1}(x_{k+1}) \\ \sigma_k^2(x_{k+1}) &= \Sigma(x_{k+1}, x_{k+1}) - \Sigma(x_{k+1}, x_{1:k})\Sigma(x_{1:k}, x_{1:k})^{-1}\Sigma(x_{1:k}, x_{k+1}).\end{aligned} \tag{2.46}$$

The main idea in this optimization method is to evaluate D at a new point that has most potential in improving the highest value for D found so far. This idea can be formulated mathematically as finding a new point $x \in \Theta$ that maximizes the expected improvement

$$EI(x : x_{1:k}) = E\left[\max(D(x) - D^*(x_{1:k}),\ 0)\right]$$

where $D^*(x_{1:k})$ is the largest value of D among $x_1, \ldots, x_k$. By letting $x_{k+1} = x$, the values of D at $x_1, \ldots, x_{k+1}$ are known and the posterior of D over Θ can be updated and the same procedure repeated. This is done a total of N times and the combination of parameters that yields the highest D is returned.

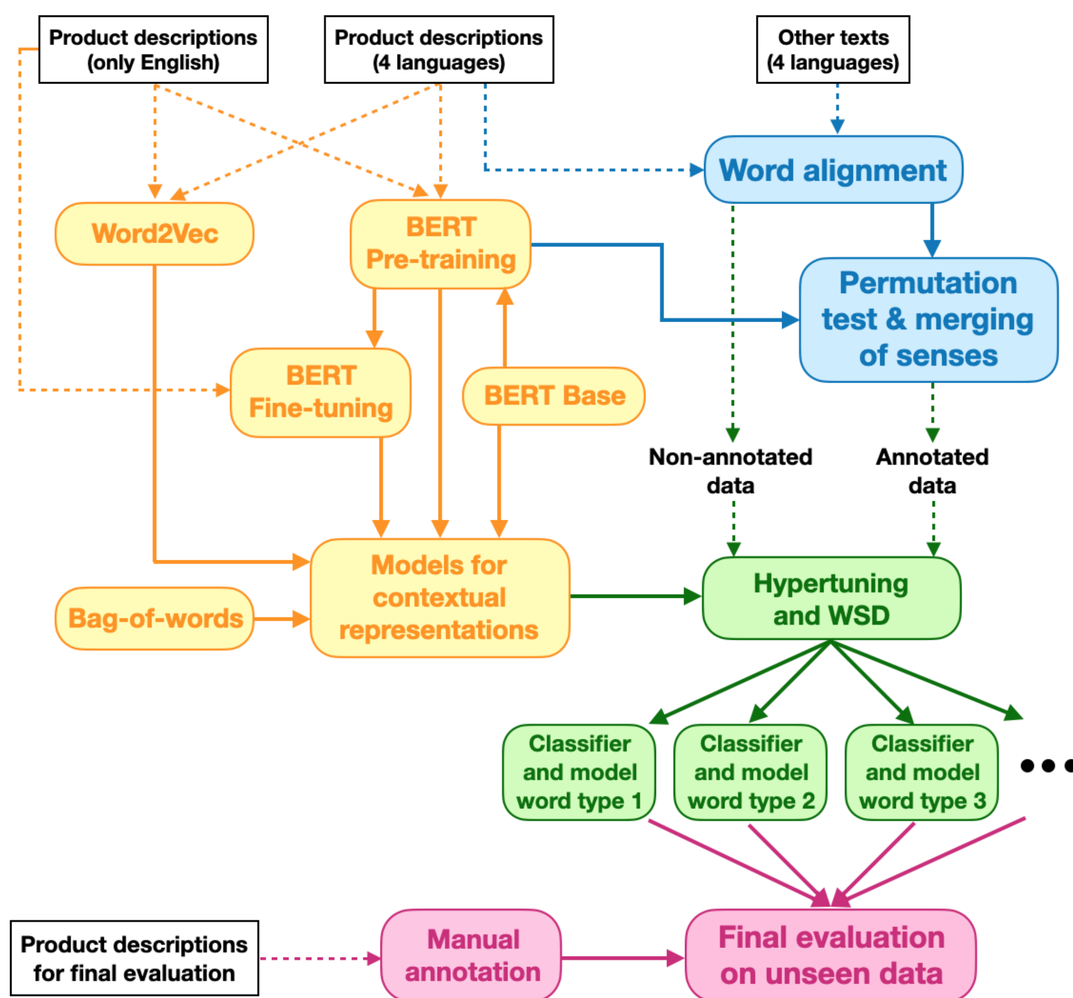
In [28] they present the following closed form for the expected improvement

$$EI(x : x_{1:k}) = \max(\Delta_k(x), 0) + \sigma_k(x)\varphi\left(\frac{\Delta_k(x)}{\sigma_k(x)}\right) - |\Delta_k(x)|\Phi\left(\frac{\Delta_k(x)}{\sigma_k(x)}\right)$$

where $\Delta_k(x) = \mu_k(x) - D^*(x_{1:k})$, φ and Φ are the probability- and cumulative distribution functions of a standard normal, respectively. Unlike the function D, the expected improvement is inexpensive to evaluate and is therefore easier to optimize.

3
Methods

The different components of the project are displayed in the figure below. As explained in Section 3.1 we used different kinds of data, but mainly product descriptions from different fashion sites that were obtained by scraping. In the first step (blue), senses were defined and data annotated automatically (see Section 3.2) using word alignment, a permutation test and a merging algorithm for senses. The second step (yellow) was then to prepare the different contextual models (see Section 3.3). The third step (green) was to train (see Section 3.4) one classifier performing word-sense disambiguation for each word type. In the final step (purple), unseen data was manually annotated followed by a final evaluation (see Section 3.5) on the obtained models in the previous step.



3.1 Datasets

In Table 3.1 we see the different kinds of product descriptions that were used. The texts from *H&M* had already been scraped by Textual while the ones from Textual were generated with their software. The rest of the texts were scraped using standard scraping tools in Python. To deal with the maximum number of requests problem, a rotating pool of proxies[1] was used.

Table 3.1: *Summary of all product description texts. Around 82000 texts were used in the training phase while around 9600 texts were used in the final evaluation. This is excluding headers (titles).*

Website	**Languages**	**Nr. of texts**	**Pre-train. BERT & W2V**	**Fine-tuning BERT**	**Def. senses**	**Final eval.**
Adidas	en, sv, fr, es	26.4k	Yes	No	Yes	No
Esprit	en, sv, fr, es	3.0k	Yes	No	Yes	No
H&M	en, sv, fr, es	19.5k	Yes	No	Yes	No
Lindex	en	4.9k	Yes	No	No	No
Next	en	20.0k	Yes	Yes	No	No
Textual	en, sv, fr, es	1.5k	Yes	No	Yes	No
Uniqlo	en	2.7k	Yes	No	No	No
Zara	en, sv, fr, es	4.0	Yes	No	Yes	No
Total		**82.0k**				
Gina Tricot	en, sv	1.0k	No	No	No	Yes
Kappahl	en, sv	1.8k	No	No	No	Yes
Man of a kind	en, sv	3.6k	No	No	No	Yes
Nike	en, sv, fr, es	3.2k	No	No	No	Yes
Total		**9.6k**				

As can be seen in Table 3.1, a majority (82.0k) of the texts were used in the training phase and the rest (9.6k) were used in a final evaluation. In that way, the latter had not been seen by the models before evaluating. The multilingual texts (*Adidas, Esprit, H&M, Textual* and *Zara*) were used for word alignment. All texts in the training phase were used in pre-training of BERT and Word2Vec, while only the texts from *Next* were used for fine-tuning. Note that the numbers above do not include headers (titles), which were used in the word alignment and pre-training phases. Including headers nearly doubles the number of texts.

Furthermore, additional data was used for word alignment due to unsatisfactory performance. The texts included movie subtitles from *OpenSubtitles* [29] and *Textual*'s vocabulary dictionary in the required languages.

[1]Using the `proxy-requests` library in Python

3.2 Defining Word Senses

Word senses of a word type were primarily defined by the translations in four languages, namely English, Swedish, French and Spanish. The word type was always considered to be the English word, meaning that the English translation would always be the same (for all senses of a word type). In order to find all translations, we used a procedure called word alignment which is described in Section 3.2.1. One sense then corresponds to a quadruple in the format (en_w, sv_w, fr_w, es_w). In a later step, contextually alike senses were merged using a permutation test (see Section 3.2.2) and an algorithm we call SenseMerge (see Section 3.2.3). Similarity between senses was measured using a custom measure that considers the Euclidean distance between the contextual representations of the ambiguous word. These vector representations were from the pre-trained BERT model (see Section 3.3.3).

The reason for choosing the pre-trained BERT model over the other models is that there are good reasons to believe that this model captures the context of product description texts in the best way. It is likely that this is the case compared to BERT Base, since the same pre-training procedure is continued but on product description texts. It is uncertain whether or not fine-tuning on a downstream task will increase the contextual knowledge of the words, which means that the pre-trained BERT model is probably a better choice than the fine-tuned BERT model. In addition, it is likely that BERT is a better choice than both Bag-of-Words and Word2Vec since it performs better on many context-dependent downstream tasks [19].

3.2.1 Initial Definition of Word Senses Using Word Alignment

Manual annotation of training data is usually a cumbersome and time-consuming process. To circumvent this we define the sense of an English word en_w as a quadruple (en_w, sv_w, fr_w, es_w) where sv_w, fr_w, es_w are the Swedish, French and Spanish words, respectively, that the English word en_w aligns to. The word alignment between the English texts and the Swedish, French and Spanish texts is done using `GIZA++`, which implements the IBM models (see Section 2.1).

1. Lemmatization: Since we don't want to have multiple senses of a word type, with the only difference being the conjugation or singular and plural version of the word, we lemmatized each of the four words (en_w, sv_w, fr_w, es_w) using the sentences they occurred in. The lemmatization is done using the open-source Natural Language Processing library `spaCy`. However, we observed that the lemmatizer from `spaCy` fails to lemmatize many fashion words. To solve this we extended the lemmatizer by adding the words it fails to lemmatize with the corresponding correct lemma.

2. Bidirectional alignment:

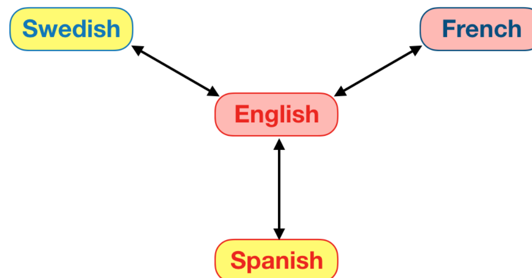


Figure 3.1: *Bidirectional word alignment between the four languages.*

As explained in Section 2.1.2, the alignment is from the source language to the target language and every word in the source language can align to at most one word in the target language. The disadvantage with having a unidirectional alignment is that when swapping the source and target language the alignment will not necessarily be the same. For example, if the source language is English, the target language is Swedish and the word *jacket* aligns to *kavaj* in an English-Swedish sentence pair, it doesn't mean that when swapping the source- and target language the word *kavaj* will align to *jacket*. To solve this we first align English to all other languages, resulting in the quadruple (en_w, sv_w, fr_w, es_w), and then we align in the opposite direction and check if sv_w, fr_w and es_w aligns to en_w. If this is not the case then the data point is ignored. This method of swapping the source and target language results in a bidirectional alignment and implies a bijection between (en_w, sv_w), (en_w, fr_w) and (en_w, es_w).

3. POS-tagging: Apart from the bijection between the words it also makes sense that the words in the quadruple should have the same POS-tag[2] (Part-of-Speech tag). However, imposing the constraint that the POS-tags obtained from `spaCy` should be the same for all words results in extensive loss of data. Instead we used the milder condition that the majority of the words in the quadruple should have the same POS-tag. If this is not the case then the data point is ignored. Note that this criteria is acceptable since the POS-tags from `spaCy` are not 100% accurate.

4. Splitting of Swedish compund words: For the word *jacket* we observed that many of the quadruples are the same except for in the Swedish word with the difference being hypernyms and hyponyms. For example, *jacket* can align to *jacka*, *bomullsjacka* and *läderjacka* where *jacka* is a hypernym to *bomullsjacka* and *läderjacka*, meaning that *bomullsjacka* and *läderjacka* are special cases of *jacka*. Keeping the hyponyms resulted in many senses for each word, making the classification much harder.

This problem was dealt with by splitting certain Swedish compound words into multiple parts. Splitting is motivated by the fact that many Swedish hyponyms were aligned to two (or more) words in English (*bomullsjacka* was aligned to *cotton* and *jacket* for example). The splits for our examples above would then be *bomullsjacka* → *bomull jacka* and *läderjacka* → *läder jacka*. Splitting was done manually for more than 3000 Swedish compound words, selected in terms of their respective number of occurrences in the data and relevance to the obtained ambiguous words.

[2]noun, verb, adjective, etc.

5. Including other texts: In addition to the product description texts we also included subtitle texts in English, Swedish, French and Spanish as well as Textual's vocabulary lexicon. Even though the subtitle texts won't contain so many fashion words they still improve the word alignment by producing better alignments for other more common words. Using single word pairs from Textual's lexicon in the word alignment can be thought about as including prior information about what the fashion words are supposed to align to. How strong this prior is depends on how many times we include the lexicon. Including it too many times puts a too strong prior on the fashion words and prevents them from aligning to words not present in the lexicon. Similarly, including too many subtitle texts could worsen the alignments between the product description texts. For this reason, the number of times we include Textual's lexicon and the number of subtitle texts we include are parameters to be tuned.

As shown in Section 2.2, word alignment is an important component in statistical machine translation, so it makes sense that the machine translation is good when the alignment is good and vice versa. This is not entirely true and evaluating word alignment using machine translation is only an indirect approach. A correct measure of the performance of the word alignment requires gold standard alignments, something that is very hard and time-consuming to obtain. Given this limitation, evaluation through machine translation is good enough for our purpose. The evaluation is done as follows: For every pair of (L, S) in a grid, where L is the number of times we include Textual's lexicon and S is the number of subtitle texts we include, we perform word alignment from the source languages Swedish, French and Spanish to English and build a machine translation model using the Moses Decoder[3] from these alignments. A test set of 20% of all product description texts, not included in the training of the word alignment, is used to calculate the BLEU score, see Section 2.2.2, when translating from the source languages to the target language English. For every source language, the pair (L^*, S^*) with the highest BLEU score (see Table 4.1) is selected and a final word alignment is performed with these parameters.

6. Removing outliers: Both in the subtitle texts and the product description texts there is sometimes noise or errors in the translations. Using sentence pairs that are not translations of each other deteriorates the word alignment. Some of these bad sentence pairs can be found by looking at the difference in sentence length between texts in different languages. It is not correct to remove those sentence pairs whose difference in length is much greater than 0 since the sentences in one language could naturally be longer or shorter than the sentences in another language. For this reason we used Tukey's method for detecting outliers [30] on the difference in length between sentences from different languages.

Even after including the steps 1, 2, 3, 4, 5, 6 above we saw that there were many misalignments and therefore also a large number of different quadruples for a word. Since the quadruples that are due to misalignments occur very infrequently, we

[3] https://github.com/moses-smt/mosesdecoder

solved this by removing the quadruples that occur less than 5% of the total number of quadruples for a given word. The data that we fail to annotate are saved in a separate file and later reused in self-training.

3.2.2 Permutation Test

The quadruples for a word type found from the word alignment doesn't necessarily correspond to different senses for an English word en_w. Some of them could be the same in the sense that they are used in similar contexts. To get good performance on the classifier it is important that the senses for a word are well-defined and distinct. To this end, every word type that has more than one quadruple undergo a permutation test [31] using the BERT vectors. Every BERT vector is found by feeding the English text that the ambiguous word occurs in into the pre-trained BERT model (see Section 3.3.3) and then taking the mean of the output from the last four encoders for the ambiguous word. Suppose that the quadruples for en_w are $Q_1, Q_2, \ldots Q_C$. For every pair of quadruples (Q_i, Q_j) we test the hypothesis

$$H_0 : Q_i = Q_j$$
$$H_A : Q_i \neq Q_j$$

where the equality $Q_i = Q_j$ means that Q_i and Q_j are used in similar contexts. The test statistic, a measure of dissimilarity between Q_i and Q_j, is given by

$$T\left(s(Q_i), s(Q_j)\right) = \frac{2 \cdot d_{0.5}(s(Q_i), s(Q_j))}{d(s(Q_i), s(Q_i)) + d(s(Q_j), s(Q_j))} \tag{3.1}$$

where

$$d\left(s(Q_i), s(Q_j)\right) = \frac{1}{|s(Q_i)||s(Q_j)|} \sum_{\substack{x \in s(Q_i) \\ y \in s(Q_j)}} ||x - y||^2 \tag{3.2}$$

and $s(Q_i)$ are the BERT vectors for the given ambiguous English word en_w with sense Q_i, and similarly for $s(Q_j)$. The function $d_{0.5}(s(Q_i), s(Q_j))$ is the same as $d(s(Q_i), s(Q_j))$ with the difference that instead of taking the average distance between $s(Q_i)$ and $s(Q_j)$ we only look at the 50% shortest distances between $s(Q_i)$ and $s(Q_j)$ and take their mean. Doing this the test statistic will increase the similarity between quadruples that have some degree of overlap between the BERT vectors. The denominator in T captures the compactness of $s(Q_i)$ and $s(Q_j)$.

The intuition of the permutation test is the following: Under H_0 the two quadruples are the same so a data point $x \in s(Q_i)$ might as well have belonged to $s(Q_j)$ instead of $s(Q_i)$. This means that under H_0 we can generate new samples by randomly permuting the labels of the data points and for each new sample calculate the test statistic given by (3.1). Suppose T^* is the statistic for the unpermuted sample and $T_1, T_2, \ldots, T_K$ the test statistics for the permuted samples. The p-value is then given by

$$p = \frac{|\{i \in \{1, 2, \ldots, K\} : T_i \geq T^*\}| + 1}{K + 1}.$$

The null hypothesis H_0 is rejected in favor of H_A if $p < \alpha$ where α is the significance level. In this project we chose $\alpha = 0.01$.

3.2.3 Merging of Senses

Ideally we would like all of the permutation tests to reject H_0 meaning that all the quadruples/senses are distinct. If the permutation test does not reject H_0 for two quadruples Q_i and Q_j it suggests that $Q_i = Q_j$ and then we would like to merge Q_i and Q_j into one single quadruple. As in the previous section, we suppose that the quadruples are $Q_1, Q_2 \ldots, Q_C$ and the associated p-values obtained from the permutation tests are $p_{i,j}$ for $i, j = 1, 2, \ldots, C$ and $i \neq j$. We can create a graph $G = G(V, E)$ where the set of nodes are $V = \{Q_1, \ldots, Q_C\}$ and there is an edge between Q_i and Q_j if and only if $p_{i,j} \geq \alpha$, i.e the null hypothesis $H_0 : Q_i = Q_j$ is not rejected. The existence of an edge between two quadruples Q_i and Q_j should be interpreted as they are the same. The goal is to merge the nodes in this graph until there are no edges left, meaning that the remaining nodes correspond to contextually distinct senses.

Suppose the quadruples are Q_1, Q_2, Q_3 and there is an edge between Q_1 and Q_2 and Q_1 and Q_3. If we merge Q_1 and Q_2 then a new permutation test has to be done to see whether or not there should be an edge between Q_3 and (Q_1, Q_2). Since the permutation test is very computationally expensive this is not feasible for a graph with many nodes. We have developed an algorithm, described below, that reduces the graph until there are no edges left without doing any new permutation tests.

```
 1: Input: Graph constructed from p-values
 2: Output: Graph where the nodes correspond to contextually distinct senses
 3: procedure SenseMerge(graph)
 4:     if #edges(graph) = 0 then
 5:         return graph                                              ▷ done
 6:     for node_1, node_2 ∈ graph do
 7:         if node_1 ~ node_2 then
 8:             if |node_1| < |node_2| then
 9:                 merge node_1 into node_2
10:             else
11:                 merge node_2 into node_1
12:             return SenseMerge(graph)
13:     find node ∈ graph with most edges
14:     neighs ← neighbours to node
15:     for point ∈ node do
16:         kNP ← k-nearest points of point taken from neighs
17:         label(point) ← majority vote of kNP over their labels
18:     remove node from graph
19:     return SenseMerge(graph)
```

Comments: The equivalence $\sim$ is defined as follows: $node_1 \sim node_2$ if and only if: 1) there is an edge between $node_1$ and $node_2$ and 2) for every other $node_3$ in the graph there is an edge between $node_1$ and $node_3$ if and only if there is an edge between $node_2$ and $node_3$. On line 8-11 the node with the least number of data

points is merged into the other larger node. On line 13, if there are many nodes with the same number of edges then the node with the least number of data points is selected. The statement label(*point*) $\leftarrow$ on line 14 means that *point* is assigned to a new label (node). A *point* refers to the BERT vector of an occurrence of the given ambiguous word.

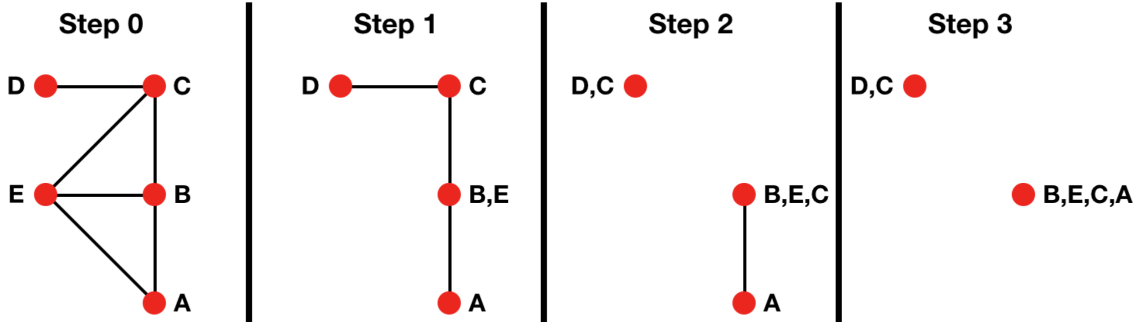


Figure 3.2: *An example showing how the SenseMerge algorithm reduces the graph until there are no edges left.*

The example in Figure 3.2 illustrates the SenseMerge algorithm for a graph with 5 nodes and 6 edges. In step 0, $B \sim E$ so they are merged together creating the node (B, E). In step 1, C is the node having the highest number of edges and least number of data points (not shown in the figure), so its points are transferred to node (B, E) and D depending on the k-nearest neighbours of every point in C. In step 2, $(B, E, C) \sim A$ so they are merged together creating the node (B, E, C, A). In step 3, we have reached a graph with no edges and we are done. For more examples that illustrate the permutation test and SenseMerge algorithm see Section A.3.

3.3 Preparation of Contextual Models

Five different models for contextual representations were used, namely Bag-of-Words, Word2Vec, BERT Base, pre-trained BERT and fine-tuned BERT. How the Bag-of-Words and Word2vec models are prepared and trained are explained in Section 3.3.1 and Section 3.3.2, respectively. BERT Base was directly taken from Google and was used as the initial model for pre-training (see Section 3.3.3). An additional training task was added on top of the pre-trained model called fine-tuning (see Section 3.3.4).

3.3.1 Bag-of-Words

In the Bag-of-Words model we construct a window of words around the ambiguous word. The size of the window and vocabulary are parameters that are tuned. Before finding the vocabulary and window of words, the text is lemmatized and stopwords[4] are removed.

[4]the, is, and, a, an, for, in, at etc

3.3.2 Word2Vec

As explained in Section 2.4.2, the Word2Vec model has to be trained on a large corpus of texts. Since the model will only be used on product description texts in fashion, these are also the texts it is trained on. To reduce the sparseness of the data and get more training samples for every word, the text is first lemmatized. The training of the embeddings is done with the following parameters:

- Dimension: 300
- Minimum word count: 25
- Window size: 10
- Downsampling: 1e-3

How to obtain a contextual representation for a word from the Word2Vec model is described in subsection 2.4.2.2. Similarly to the Bag-of-Words model, the size of the window of words is a parameter that is tuned and before finding the window of words the text is first lemmatized and stopwords are removed. When calculating the inverse document frequencies (idfs) used in the weighted sum in Equation 2.23, we only look at the window of words around the ambiguous word that we are interested in. A document thus corresponds to the window of words around a word.

3.3.3 Pre-training BERT

The BERT models provided by Google[5] have already been pre-trained on a vast and general corpus. The purpose of continuing this pre-training procedure is for BERT to learn the jargon and nuances for the types of texts that we use, namely product description texts in fashion. We used the BERT Base model as a starting point. The training procedure, consisting of masked language modelling (Masked LM) and next sentence prediction (NSP), is described in Section 2.4.3.5. An implementation of the pre-training procedure is provided in a Python script by the authors of BERT.

Pre-training was done using the following parameters suggested by the BERT authors:

- Learning rate: 2e-5
- Batch size: 32
- Number of warm-up steps: 100
- Max. sequence length: 128
- Masking rate: 15%
- Duplication factor: 5

Pre-training was done on 80% of the data with 150k training steps and tested every 10k steps on the other 20% of the data. Five measures of the performance are given, namely the combined loss, Masked LM Loss, Masked LM Accuracy, NSP Loss and NSP Accuracy. Which one to use depends on the task. For example, for

[5] `https://github.com/google-research/bert`

word tagging it is sometimes preferred to use Masked LM Accuracy/Loss while NSP Accuracy/Loss would be preferred for question answering. In this project, the model resulting from the number of training steps that yields the lowest combined loss on the test data is chosen.

3.3.4 Fine-tuning BERT

Fine-tuning means that the BERT model is trained together with the attached neural network on the downstream task at hand. In our case, this would mean connecting a classifier on top of BERT and training both of them at the same time to perform word-sense disambiguation on each ambiguous word. However, this would imply one BERT model ($\sim$ 0.5 GB) for each ambiguous word, which is not feasible due to resource limitations. Instead, fine-tuning was done on another downstream task, resulting in one BERT model. We call this fine-tuning but in fact it should be seen as a third pre-training task, since this is not the final word-sense disambiguation task.

The fine-tuning task was to classify the category of the text, which is an indicator of what the text is about. Examples of categories are *Joggers*, *T-Shirts*, *Sandals*, *Jackets*, *Jumpers* and *Tops*. The idea was that if BERT can successfully classify the category of the text, it also has a better understanding of the text and will therefore perform better in word-sense disambiguation, which has much to do with understanding the context of the text.

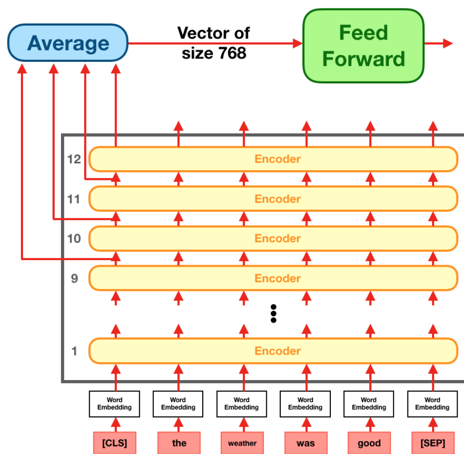


Figure 3.3: *The fine-tuning structure.*

For fine-tuning we only used the texts from *Next* (see Table 3.1), which included a category label for each text. The reason for not using more texts is that different companies use different kinds of categories. We chose *Next* since we had access to a relatively large amount of texts from them and the category label was easily accessible when scraping. All duplicates and texts not belonging to the 60 most frequent categories were removed. The reason for the latter was that there were only a few texts (around 1-5) for the least frequent categories, which is not enough for training a neural network. Duplicates were also removed. One problem was that the category name often appeared in the text itself, which would make it too easy to classify the correct category. To deal with this we replaced the word with the token [MASK] 50% of the time. This hides the word and forces BERT to rely more on the context. The reason we didn't mask the word 100% of the time is that we wanted some signal to pass through to improve learning.

We used the best BERT model obtained from pre-training together with a feed forward neural network initialized at random. The average of the last four hidden states of the [CLS]-token was then used as the input to the network. The architecture of the network consisted of an input layer with 768 neurons, three hidden layers with sizes 512, 256 and 128 and finally an output layer with 60 neurons. The ReLU activation function was used in all hidden layers, which was then followed by batch normalization. A softmax activation function was used on the output layer in order to produce probabilities. The output neuron with the highest probability was then chosen as the predicted category. The general structure of the fine-tuning procedure is displayed in Figure 3.3.

The `huggingface` implementation of BERT was used and, as explained above, a feed forward neural network was attached on top of it. When training, the weights of both BERT and the feed forward neural network were updated using stochastic gradient descent on the cross-entropy loss, see Equation 2.19. We used the same maximum sequence length as in the pre-training, namely 128. This was large enough to cover the vast majority of our texts, while limiting the memory usage. We used a batch size of 32, the largest possible without experiencing lack of memory issues. The learning rate was initialized at 2e-5 and then decreased linearly, without any warmup steps. In the BERT paper, this is the recommended learning rate for fine-tuning.

3.4 Word-sense Disambiguation

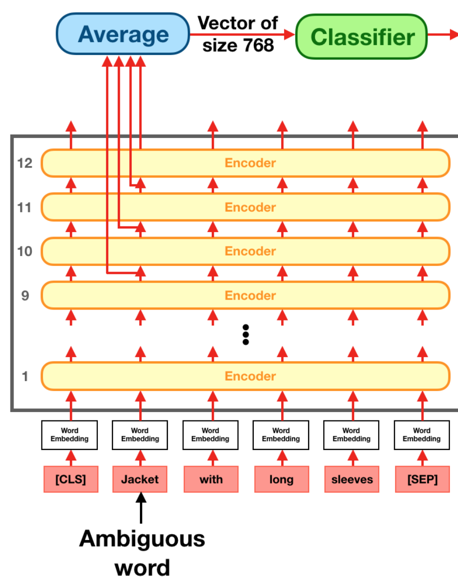


Figure 3.4: *Structure of the word-sense disambiguation procedure for the BERT models.*

Since the ambiguous words are labeled with a quadruple from Section 3.2.3, the word-sense disambiguation task can now be solved in a purely supervised manner. The models we examined to generate contextual representations, feature vectors, are presented in Section 3.3. For every model and word we performed hyperparameter optimization using the Python package `hyperopt` [32]. Usually one tunes the parameters for an already chosen classifier but it is also possible to regard the type of classifier as a hyperparameter that can be tuned. With this in mind, the parameters we tuned are the type of classifier, the hyperparameters for this classifier and also whether or not to do self-training. The non-annotated data used when doing self-training is the data that we were not able to annotate in the word alignment procedure. As classifiers we tried Neural Networks, Random Forests, Support Vector Machines, Gaussian Naive Bayes and k-Nearest Neighbors. In the end, for every word

we chose the model and classifier that performed the best. The performance is evaluated by taking the mean accuracy from a stratified 4-fold cross-validation over all the training data.

For the different BERT models, the average of the last four hidden states corresponding to the ambiguous word is fed to the classifier, see Figure 3.4. When training the classifier the weights in the BERT model are not updated due to the resource limitations explained in Section 3.3.4.

3.5 Evaluating the Performance on New Fashion Sites

It is not uncommon that product description texts differ by just a few words. This means that when doing cross-validation some of the texts in the test data will be very similar to texts in the training data, resulting in an accuracy that might not be completely accurate. Also, it could be the case that the classifier does not learn general features that constitute the sense of a word, but rather memorizes the texts in the training data and uses these when predicting the texts in the test data. This can be compared to when the same set of data is used for both training and testing, i.e. the training and test set is the same. To confirm that the classifiers have learned features and characteristics that determine the sense of a word, we evaluated the program on manually annotated data from completely new fashion sites not used in the training of the classifiers nor in the pre-training of BERT and Word2Vec. The program was mainly evaluated on words that we believe have truly different senses but also on some words where the difference is more subtle.

4 Results

In this chapter we start by showing an example of the bidirectional word alignment for a product description text as well as the performance of the entire word alignment procedure. Next, two plots are shown where the first one illustrates the optimal number of training steps for pre-training and the second one shows the progression of the loss and accuracy when fine-tuning the pre-trained BERT model. This is followed by an example of the permutation test and SenseMerge algorithm for the word *fastener*. After this, the results for word-sense disambiguation are shown. Finally, we end the section by giving some examples illustrating when the senses for a word are truly different senses or only apparent different senses.

4.1 Word Alignment

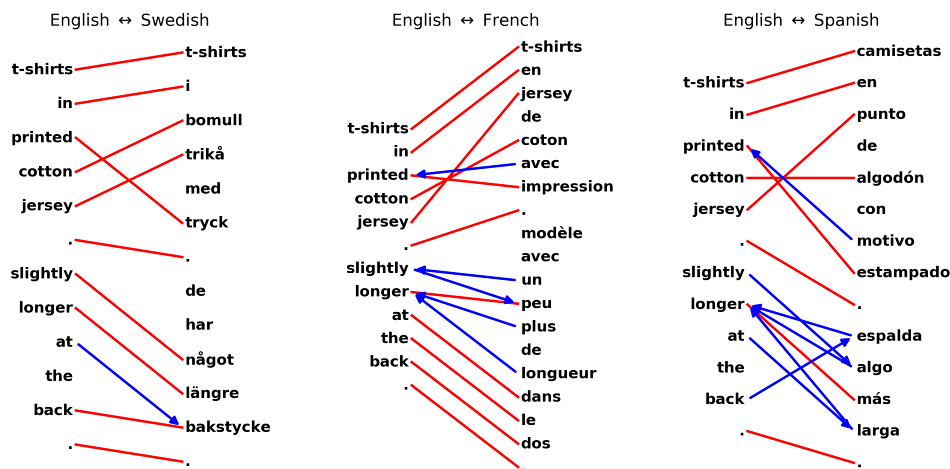


Figure 4.1: *Word alignment for one text in English to and from the corresponding texts in Swedish, French and Spanish. Red lines are double alignments, while blue arrows are single alignments.*

In Figure 4.1, we can see an example of a text that is aligned in two directions. The alignment is done from English to Swedish, French and Spanish and the other way around. Double alignments are displayed with red lines while single alignments are displayed with blue lines. We see, for example, that the English word *t-shirts* is doubly aligned to *t-shirts*, *t-shirts* and *camisetas*, respectively. Ideally we would

obtain double alignments for all words in the text, but this is only possible if the number of words in the texts is the same.

Obtaining a general result of the performance of the word alignment procedure is not straightforward. Intuitively, the more data that is annotated, the better. We observed that 51% of the occurrences of the chosen ambiguous words (see Table A.1) were annotated. Another measure of the performance is the BLEU score from machine translation. The table below shows the BLEU score when translating from Swedish, French and Spanish to English. We can see that it was beneficial to include Textual's lexicon one time and add subtitle texts but not too many. We can also see that translating from Spanish to English is harder than from Swedish and French to English. This indicates that the word alignment between Spanish and English is worse than between English and the other languages.

Table 4.1: *BLEU score of a machine translation model that uses word alignment. Used as an indirect evaluation of the word alignment. Lexicons refers to the number of duplicates of Textual's lexicon and Subtitles refers to the number of subtitle texts.*

Lexicons	Subtitles	Swedish	French	Spanish
0	0	0.680	0.652	0.580
1	0	0.684	0.655	0.584
5	0	0.684	0.652	0.584
20	0	0.683	0.647	0.575
60	0	0.679	0.638	0.563
0	50000	0.683	0.654	0.588
1	50000	**0.686**	**0.660**	0.590
5	50000	0.684	0.657	0.589
20	50000	0.683	0.652	0.584
60	50000	0.679	0.643	0.574
0	250000	0.682	0.656	0.589
1	250000	0.681	0.657	**0.596**
5	250000	0.684	0.657	0.595
20	250000	0.683	0.652	0.584
60	250000	0.676	0.645	0.576
0	500000	0.679	0.657	0.590
1	500000	0.684	0.658	0.594
5	500000	0.681	0.658	0.594
20	500000	0.679	0.652	0.586
60	500000	0.680	0.647	0.577
0	1000000	0.679	0.656	0.592
1	1000000	0.684	0.659	0.594
5	1000000	0.682	0.654	0.594
20	1000000	0.684	0.650	0.586
60	1000000	0.679	0.644	0.587

4.2 Training BERT

Training BERT consisted of two tasks, pre-training and fine-tuning resulting in two models, the pre-trained BERT model and the fine-tuned BERT model. Progress from pre-training is displayed to the left in Figure 4.2. We see that the combined loss first decreases, but then starts to increase while the masked LM and next sentence prediction accuracy increased or stayed stable during the entire training procedure. The lowest combined loss was obtained after 60k training steps, and this is the model that was later used in the other parts.

The second task, fine-tuning, was to predict the category of product description texts. This was only done on the texts belonging to the 60 most occurring categories. The training loss and the validation accuracy from fine-tuning are displayed to the right in Figure 4.2. We can see that the model stopped improving after around 45 epochs.

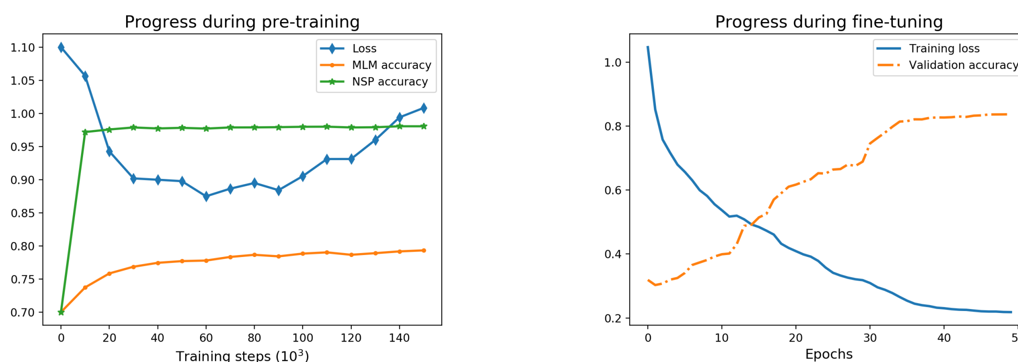


Figure 4.2: *Loss, masked LM accuracy, and next sentence prediction accuracy on test data during pre-training is shown in the left figure. The loss has been moved down for increased visibility. Training loss and validation accuracy during fine-tuning on category classification is shown in the right figure. The loss has been moved up for increased visibility.*

4.3 Merging of Senses

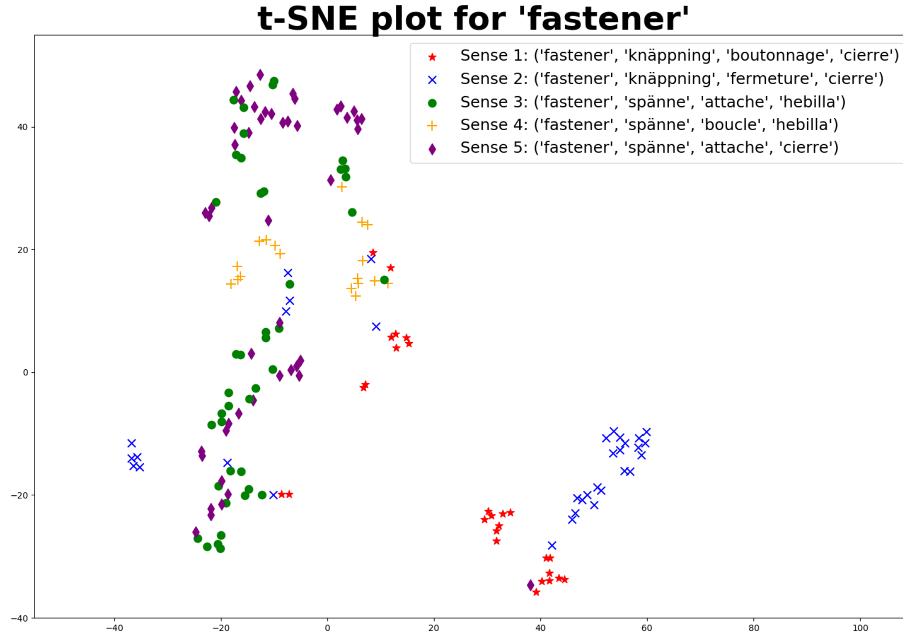


Figure 4.3: *t-SNE plot for the word fastener before SenseMerge. Every point represents the BERT vector for the word fastener in the sentence it is used in. The BERT model used is the pre-trained BERT model.*

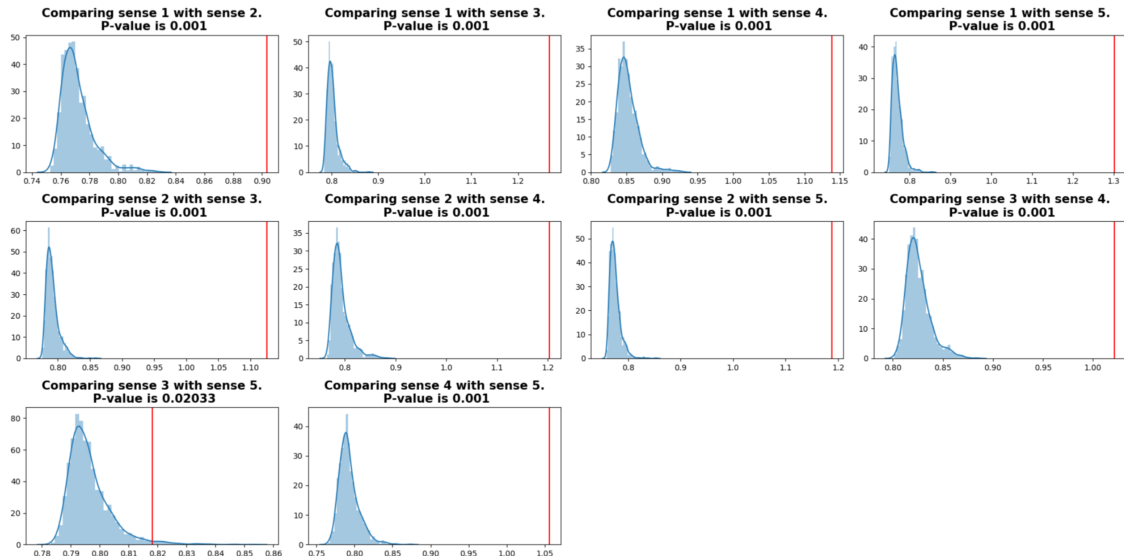


Figure 4.4: *Histograms showing the distribution of the statistic* (3.1) *under* H_0 *when comparing the five quadruples. The red line is the value of the statistic before permuting the labels.*

In Figure 4.3 we see that sense 3 and 5 seem to overlap, which is confirmed by the p-value of 0.02033 (see Figure 4.4). Since the p-value is greater than the significance level $\alpha = 0.01$, the null hypothesis H_0 is not rejected. This indicates that the two senses are contextually the same. In Figure 4.5 we can see the most common window of words around *fastener* for sense 3 and 5. Since the window of words for the two senses are very similar, the merging is justified. The actual difference between sense 3 and 5 is in the Spanish words *hebilla* and *cierre*. These words are not synonyms but similar in some contexts, which also motivates the merging. For all other comparisons H_0 is rejected since the p-value is almost 0. Furthermore, we see that the remaining four senses (after SenseMerge) will be slightly separated.

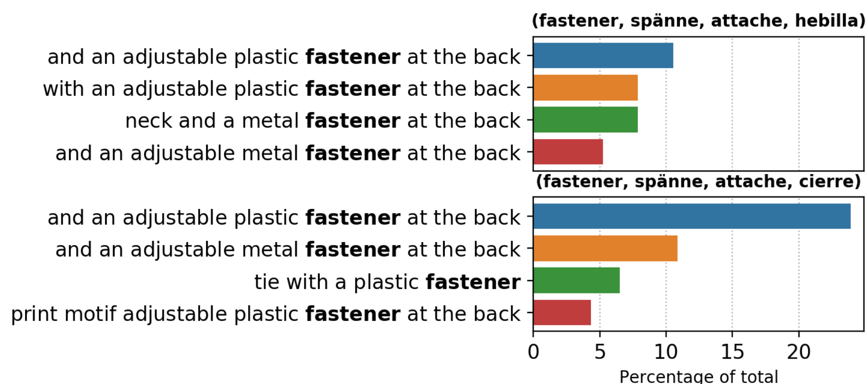


Figure 4.5: *The most common window of words for the two senses of the word fastener that are merged, see Figure 4.3.*

4.4 Word-sense Disambiguation

Table 4.2: *Obtained accuracy on 69 different words using the following contextual models: Baseline classifier (BL), Bag-of-Words (BoW), Word2Vec (W2V), BERT Base (BB), BERT Pre-trained (BP), BERT Pre-trained and Fine-tuned (BP&F) and the best of these models (Best).*

Word	BL	BoW	W2V	BB	BP	BP&F	Best
airy	0.6300	**0.7985**	0.7799	0.7830	0.7773	0.7831	0.7985
ankle	0.4932	**0.9544**	0.9453	0.9362	0.9407	0.9362	0.9544
back	0.2568	0.7534	0.7284	0.7692	**0.7927**	0.7844	0.7927
bag	0.3351	0.7707	0.7698	0.8449	**0.8823**	0.8397	0.8823
belt	0.9352	0.9489	0.9625	0.9694	**0.9727**	0.9658	0.9727
bottom	0.6234	0.9804	0.9872	**1.0000**	1.0000	0.9936	1.0000
button	0.7866	0.9538	0.9488	**0.9787**	0.9760	0.9678	0.9787
cap	0.4630	0.9034	0.8838	0.9228	0.9224	**0.9263**	0.9263
celebrate	0.5625	0.8594	0.7812	**0.8750**	0.8438	0.8438	0.8750
compartment	0.8088	0.8530	0.8723	0.8673	**0.8922**	0.8825	0.8922
cuff	0.4896	0.8574	0.8145	0.8976	**0.9006**	0.8785	0.9006

continued . . .

...continued

Word	BL	BoW	W2V	BB	BP	BP&F	Best
cup	0.7540	0.9497	0.9073	0.9656	**0.9736**	0.9470	0.9736
cut	0.9299	0.9872	**0.9936**	0.9915	0.9894	0.9894	0.9936
day	0.6503	0.9084	0.8038	**0.9331**	0.9268	0.8964	0.9331
denim	0.8761	1.0000	0.9956	**1.0000**	1.0000	0.9778	1.0000
durable	0.5673	0.7780	0.7080	**0.7828**	0.7485	0.7540	0.7828
elastic	0.7227	0.9378	0.9037	0.9365	**0.9495**	0.9295	0.9495
fabric	0.3302	0.7739	0.7505	0.7868	**0.8163**	0.7661	0.8163
fastener	0.5217	0.8572	0.8151	**0.9132**	0.9072	0.8888	0.9132
front	0.2836	0.6669	0.6444	0.7667	**0.7747**	0.7470	0.7747
gold	0.5076	0.9770	0.9770	0.9770	**0.9924**	0.9770	0.9924
hat	0.5172	0.9896	0.9554	0.9762	**1.0000**	0.9554	1.0000
heel	0.5485	**0.9909**	0.9849	0.9817	0.9878	0.9787	0.9909
hem	0.5523	0.8215	0.7860	**0.8580**	0.8570	0.8299	0.8580
high	0.6174	0.9523	0.9279	0.9365	**0.9584**	0.9536	0.9584
hoodie	0.5252	0.6973	0.6856	**0.6976**	0.6950	0.6767	0.6976
jacket	0.7478	0.9472	0.9471	0.9405	**0.9504**	0.9438	0.9504
jeans	0.6484	0.9816	0.9769	0.9816	**0.9907**	0.9816	0.9907
jersey	0.7729	0.9683	0.9411	0.9828	**0.9855**	0.9841	0.9855
lace	0.7818	**0.9638**	0.9501	0.9626	0.9626	0.9601	0.9638
large	0.4635	0.9850	0.9793	**1.0000**	1.0000	1.0000	1.0000
leather	0.7148	0.9821	0.9735	0.9778	**0.9838**	0.9735	0.9838
leggings	0.5911	0.9264	0.9296	0.9393	**0.9647**	0.9487	0.9647
light	0.6231	1.0000	0.9848	**1.0000**	1.0000	1.0000	1.0000
logo	0.5653	0.9914	0.9880	**0.9931**	0.9863	0.9897	0.9931
loose	0.3875	0.9599	0.9332	**0.9621**	0.9554	0.9621	0.9621
modern	0.5000	0.8884	0.8839	0.9018	**0.9152**	0.8795	0.9152
narrow	0.6421	0.8236	0.7856	**0.8306**	0.8264	0.8156	0.8306
neck	0.7387	0.9990	0.9960	0.9990	**1.0000**	0.9990	1.0000
neckline	0.3538	0.6359	0.6149	0.6596	**0.6774**	0.6629	0.6774
perfect	0.8125	0.9911	0.9376	0.9909	**1.0000**	0.9776	1.0000
print	0.3987	0.8146	0.7442	0.8625	**0.8706**	0.8146	0.8706
relaxed	0.2714	**0.7757**	0.7667	0.7707	0.7374	0.7527	0.7757
ring	0.6600	0.8810	0.8601	**0.9196**	0.8452	0.8780	0.9196
seam	0.5109	0.8946	0.8759	0.9186	**0.9218**	0.9207	0.9218
shirt	0.9091	0.9798	0.9819	0.9838	0.9859	**0.9879**	0.9879
side	0.4096	0.7875	0.7661	0.8587	**0.8683**	0.8561	0.8683
size	0.5748	0.9527	0.9189	0.9896	**0.9922**	0.9791	0.9922
slightly	0.5535	0.9391	0.9041	0.9686	**0.9705**	0.9391	0.9705
slim	0.2228	0.9381	0.8712	0.9207	0.9554	**0.9651**	0.9651
slit	0.5955	0.7886	0.7804	0.8028	**0.8250**	0.8172	0.8250
smooth	0.9178	0.9976	1.0000	**1.0000**	1.0000	0.9977	1.0000
sock	0.4980	0.8166	0.7740	0.7967	**0.8360**	0.7922	0.8360
soft	0.6330	0.8740	0.8610	**0.8849**	0.8828	0.8691	0.8849
strap	0.8396	0.9668	0.9636	0.9765	**0.9829**	0.9808	0.9829

continued ...

. . . continued

Word	BL	BoW	W2V	BB	BP	BP&F	Best
sweatshirt	0.4527	0.9886	0.9849	0.9861	**0.9912**	0.9798	0.9912
swimsuit	0.4646	0.9296	0.8600	0.8986	**0.9453**	0.9223	0.9453
t-shirt	0.5686	0.8445	0.8125	0.9002	**0.9055**	0.8933	0.9055
text	0.6536	0.9804	0.9609	0.9671	**0.9868**	0.9479	0.9868
tie	0.4859	0.6759	0.6549	0.7817	**0.7818**	0.7542	0.7818
tights	0.4250	**0.7566**	0.6573	0.6926	0.7536	0.7108	0.7566
top	0.4447	0.8444	0.8152	0.8665	**0.8815**	0.8771	0.8815
tracksuit	0.4259	0.8146	0.7692	0.8984	**0.9348**	0.8977	0.9348
vest	0.4870	0.8968	0.9051	0.9128	0.9045	**0.9563**	0.9563
waistband	0.5166	0.8423	0.8318	0.8410	**0.8555**	0.8450	0.8555
washed	0.8786	1.0000	1.0000	**1.0000**	1.0000	0.9938	1.0000
wide	0.6519	0.8449	0.8177	0.9193	**0.9238**	0.9156	0.9238
wool	0.7743	0.9823	0.9599	0.9604	**0.9825**	0.9604	0.9825
zip	0.4082	0.8941	0.8512	0.9248	**0.9296**	0.9142	0.9296
Average	**0.5835**	**0.8937**	**0.8708**	**0.9083**	**0.9149**	**0.9024**	**0.9197**

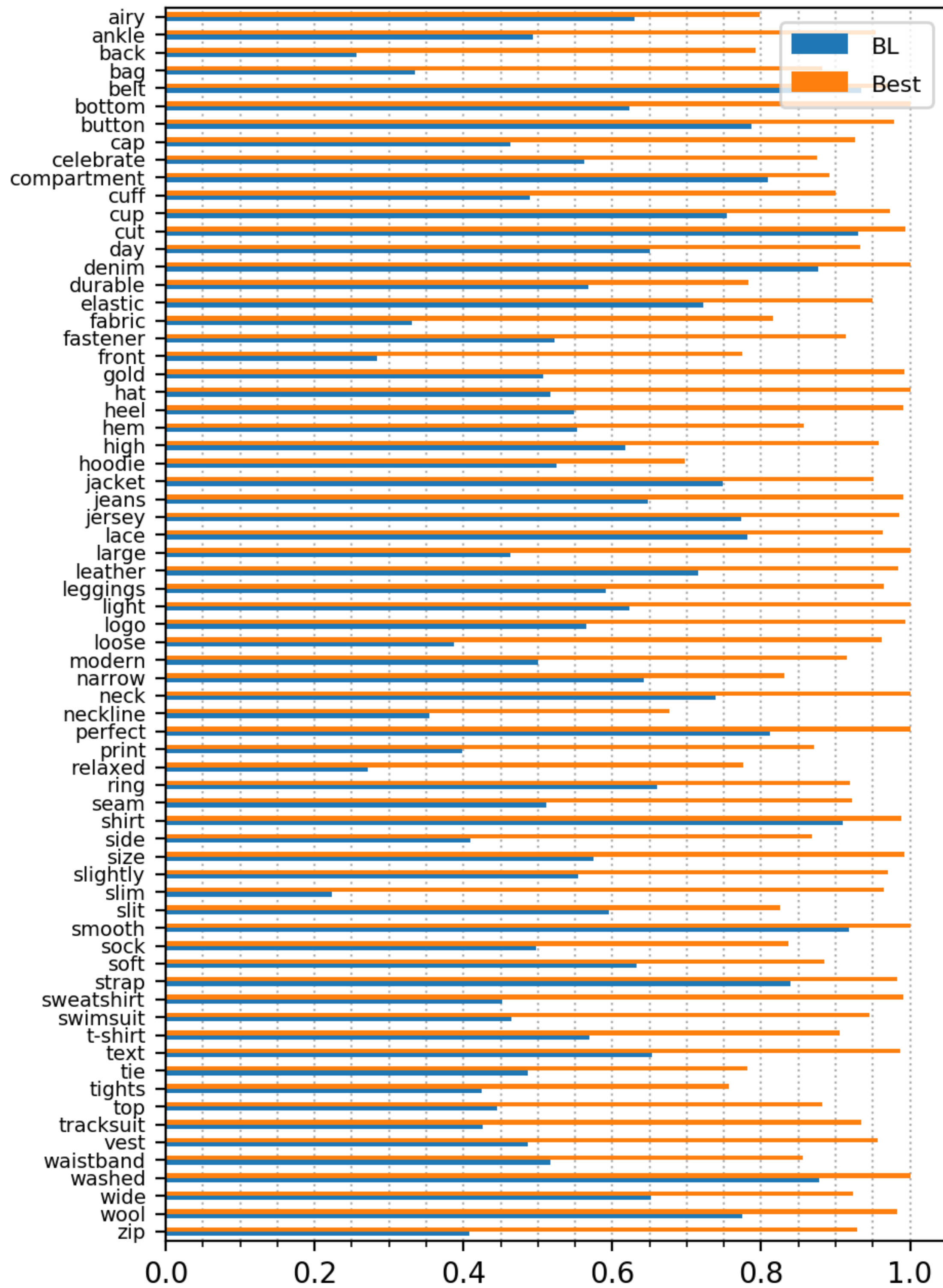


Figure 4.6: *Accuracy of the best model (see Table 4.2) and baseline (BL) for all words. The baseline is the classifier that predicts every sense to be the most frequently occurring sense (see Table A.1). It is clear that the baseline is beaten for all words.*

Table A.1 shows the sense definitions for 69 chosen ambiguous words. These words were manually chosen under three criteria; the word alignment is not obviously incorrect, there is at least 2 different senses and that the number of training points is not too small. The obtained accuracies for these words using each contextual model are shown in Table 4.2. The baseline is the classifier that predicts every sense to be the most frequently occurring sense (see Table A.1), while *Best* always uses the model (among the others) with the best performance. By looking at the average accuracy for these models, the last row in Table 4.2, we can see that it was beneficial to further pre-train the BERT model in order to capture the jargon of the texts. However, fine-tuning the pre-trained BERT model did not improve the average performance and the likely reason for this is a phenomenon known as *catastrophic forgetting* [33]. Among the five investigated models, the Word2Vec model performed the worst followed by the Bag-of-Words model. Nevertheless, their performance is impressive considering their simplicity compared to the BERT models. Note that all models contribute to the *Best* model since they have the best performance for at least one word. As a final remark we note that the baseline classifier is beaten for all the models, indicating that there is some difference between the senses that the classifiers have learned. For more detailed results of the performance of these models see Section A.1.

Table 4.3: *The gain from including the self-training procedure as a tunable parameter amounts to 0.1% for the best model.*

Word	With self-training	Only supervised	Gain from self-training
BB	0.9083	0.9063	0.0020
BP	0.9149	0.9127	0.0022
BP&F	0.9024	0.9021	0.0003
BoW	0.8937	0.8934	0.0003
W2V	0.8708	0.8666	0.0042
Best	0.9197	0.9187	0.0010

One parameter that can be tuned is whether to use self-training or not, see Section 2.6. Table 4.3 shows the average accuracy when including the self-training procedure as a parameter and the the average accuracy when not including it. Note that the parameter space when not including self-training is a subset of the parameter space when including the self-training procedure as a tunable parameter. This means that, for a sufficient number of iterations in the hyperparameter optimization, the accuracy when including the self-training option will always be larger than the accuracy when not including it.

Table 4.4 shows the performance of the program on 8 words when evaluating them on manually annotated data from the completely new and previously unseen fashion sites: *KappAhl, Nike, Gina Tricot* and *Man of a kind*. For those fashion sites that only have texts in English and Swedish, see Table 3.1, we regarded the senses that have the same Swedish translation as one single sense. For example, from *KappAhl* we can only determine if the sense of *jacket* is *jacka* or *kavaj* and not the finer distinctions of *jacka*. So if the sense is labeled as *jacka*, then the classifier makes a correct prediction if it predicts either *chaqueta* or *cazador*. As explained in Section 3.5, the results from Table 4.2 does not necessarily prove that the program can distinguish between the senses of a word when the text is from a new fashion site. However, from the results above we see that for all words the accuracy is significantly better than the accuracy of the baseline classifier, which is the classifier that predicts every sense to be the most frequently occurring sense found from the training data. This indicates strongly that the classifiers have truly learned characteristics that constitute the sense of a word in a product description text.

Table 4.4: *The total number of manually annotated data points, the baseline accuracy and the obtained accuracy when evaluating the program for eight ambiguous words on the new fashion sites; KappAhl, Nike, Gina Tricot and Man of a kind.*

Word	Total	Bl. Acc.	Acc.
jacket	160	0.8344	0.9490
slightly	178	0.1813	0.7778
neck	48	0.4762	0.8810
heel	74	0.8649	0.9865
top	150	0.7333	0.9933
bottom	74	0.2297	0.6041
elastic	152	0.5764	0.8681
wide	200	0.6550	0.7550
Average	**130**	**0.5689**	**0.8519**

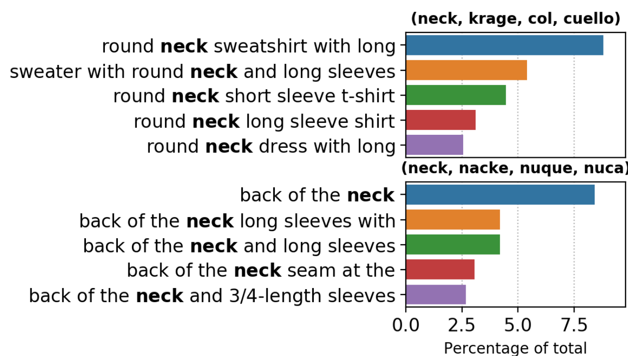


Figure 4.7: *Histogram over a small window of words around neck for its two senses.*

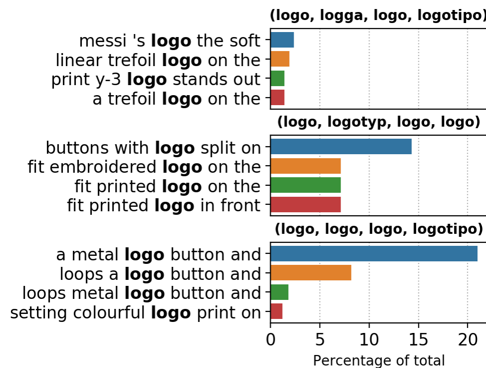


Figure 4.8: *Histogram over a small window of words around logo for its three senses.*

The difference between the two senses for *neck* is in the Swedish words *krage* and *nacke*, which are truly different senses and not apparent different senses. This is also confirmed by looking at the window of words for the two senses of *neck*, see Figure 4.7. From the quadruples for *logo* we see that the difference, if any, between them is very small. For the first quadruple, 99% of the annotated data with this sense comes from *Adidas*, for the second one 100% of the data with this sense comes

from *Textual* and for the third 99% of the data comes from *Esprit*. It is therefore possible that the difference between the three senses is merely a stylistic difference of how the fashion sites use the word. Nevertheless, it is clear from Figure 4.8 that the window of words around *logo* for its senses are different and that explains why the accuracy for this word is very high for all the models. From this we conclude that the window of words around an ambiguous word could be different not only for ambiguous words with truly different senses, but also for ambiguous words with apparent different senses.

Table 4.5: *The percentage of texts that contains the keywords: premium quality, suede, boots or black for the two senses of leather:* $leather_1$ *= ('leather', 'läder', 'cuir', 'piel') and* $leather_2$ *= ('leather', 'skinn', 'cuir', 'piel')*

	leather$_1$	**leather**$_2$
premium quality	9.2%	0%
suede	16.8%	2.1%
boots	4.9%	22.5%
black	0.4%	18%

There are more words than *neck* whose senses correspond to truly different senses and not apparent different senses. For example, for the word *wool*, the second to last entry in Table A.1, the difference between the two senses is the Swedish words *ull* and *ylle*. The word *ull* is the material or fibre while *ylle* is a fabric that is made of *ull*. Another example is the word *leather* where the difference is also in the Swedish words *läder* and *skinn*. From what we have seen, *läder* is usually of higher quality than *skinn*. Table 4.5 shows the percentage of texts that contain the keywords *premium quality, suede, boots* or *black* for the two senses of *leather*. We can see that when *premium quality* or *suede* occurs in the text then the sense is most likely *läder* but when *boots* or *black* occurs in the text then the sense is most likely *skinn*.

Table 4.6 lists some words that are important when classifying the sense of the word *jacket* in a product description text. For example, *button* and *single-breasted* are markers for $jacket_3$ (kavaj in Swedish), both of which are present in the text in Figure 4.9c).

Table 4.6: *The percentage of texts that contains the keywords: out, warm, button, pocket, single-breasted or long sleeve for the three senses of jacket:* $jacket_1$ *= ('jacket', 'jacka', 'veste', 'chaqueta'),* $jacket_2$ *= ('jacket', 'jacka', 'veste', 'cazador') and* $jacket_3$ *= ('jacket', 'kavaj', 'blazer', 'americano').*

	jacket$_1$	**jacket**$_2$	**jacket**$_3$
out	19%	0%	0%
warm	10.6%	0%	0%
button	6%	56.2%	72.9%
pocket	33.3%	93.8%	92.9%
single-breasted	0%	0%	34.1%
long sleeve	6.2%	60.4%	5.9%

(a) *This jacket is primed for outdoor adventures. Made of durable nylon, this quilted puffer coat is lightweight and insulated to help you stay warm when hitting the streets.*

(b) *Lapel collar jacket with long sleeves with buttoned cuffs. Chest patch pockets and welt pockets at hip. Interior pocket. Adjustable hem at sides. Front snap button closure*

(c) *Single-breasted jacket in woven fabric with narrow notch lapels, a chest pocket, flap front pockets and three inner pockets, one with a button. Decorative buttons at the cuffs and a single back vent.*

Figure 4.9: *Illustrating the difference between the senses ('jacket', 'jacka', 'veste', 'chaqueta') shown in* ***(a)****, ('jacket', 'jacka', 'veste', 'cazador') shown in* ***(b)*** *and ('jacket', 'kavaj', 'blazer', 'americana') shown in* ***(c)****.*

(a) *Hat in braided paper straw with sequined embroidery and a braided band. width of brim 2.5 cm.*

(b) *Hat in a soft rib knit with a faux fur pompom on the top and sewn - in turn - up with an appliqué.*

(c) *Sun hat in an airy cotton weave with broderie anglaise. Flounced brim with a decorative bow at the back, and ties under the chin. Lined. Width of brim approx. 4.5 cm.*

Figure 4.10: *Illustrating the difference between the senses ('hat', 'hatt', 'chapeau', 'sombrero') shown in* ***(a)****, ('hat', 'mössa', 'bonnet', 'gorro'), shown in* ***(b)*** *and ('hat', 'hatt', 'chapeau', 'gorro') shown in* ***(c)****.*

5 Discussion

5.1 Defining Senses

A main problem in this project was defining the senses and annotating the data. Instead of defining the senses of a word as the translations in Swedish, French and Spanish it is also possible to use *WordNet* [34], which is a dictionary that carefully describes the senses for ambiguous words. The distinctions between the senses in WordNet are usually clear but in general WordNet is criticized for being too fine-grained. For product description texts the the opposite holds, namely that WordNet is too coarse grained. This can be seen by considering two of the senses for *jacket* we found in Table A.1:

1. "It was cold outside so I wore a *jacket*."
2. "I wore a *jacket* at the party."

Although the distinction between these two words is not obvious, one can infer that in the second example the word *jacket* refers to a suit jacket, rather than an everyday jacket as in the first example. These two senses for *jacket* can not be found in WordNet. Another problem with using the senses in WordNet is that it would require us to manually annotate the ambiguous words. By defining the senses according to the translations in different languages, both of these problems are solved simultaneously. For example, the sense for *jacket* used in the second example above is *kavaj* in Swedish and the sense in the first example is *jacka* in Swedish. This means that by defining the senses of a word as the translations in different languages we do not require a predefined set of ambiguous words and senses, as would be the case if WordNet was used. In addition, we do not have to annotate the data manually, since this is handled by the word alignment.

Unfortunately, defining senses according to the translations in different languages also has some drawbacks. First of all, it is highly dependent on the performance of the word alignment. If the word alignment is of poor quality then we obtain ill-defined senses and the number of annotated words will be small. For example, from our word alignment we were only able to label roughly half of the occurrences of the ambiguous words. The performance of the word alignment is in turn dependent on the languages used. Since a sense is defined as a quadruple of translations, a bad translation in one language affects the entire quadruple. As can be seen in Table 4.1, the word alignment between English and Spanish is much worse than for the other

languages. The reason for this could lie in the structural differences between the languages. For example, it makes sense that it is easier to do word alignment between languages that are structurally similar to each other than between languages that are structurally dissimilar. From this, one could argue that the performance of the overall program would increase if Spanish was replaced by another language that is structurally more similar to English.

Another drawback is that the quadruples might not necessarily be contextually different. For example, if the difference between two quadruples is in the Swedish language and the two Swedish words are synonyms, then the two quadruples should also be the same. In this project these kinds of quadruples are merged using the SenseMerge algorithm. The reason for this is twofold. Firstly, training a classifier on a supervised problem where the labels are not different from each other might make it difficult for the classifier to learn what features that determine the label. Secondly, merging classes that are the same increases the number of training points for that class and consequently also the performance.

5.2 Comparing and Analyzing the Models

As expected, the pre-trained BERT model performs better than BERT Base (+0.66% on average). This indicates strongly that the model has learned the jargon and nuances of the types of texts that were used and hence confirms that the pre-training framework is successful.

Somewhat surprisingly, the fine-tuned BERT model performs worse than both BERT Base (-0.59%) and the pre-trained BERT model (-1.25%). This is likely due to the problem of catastrophic forgetting, where the model forgets what it has previously learned (in pre-training) in order to learn the new task (in fine-tuning). To deal with this problem, one might consider alternating between pre-training and fine-tuning in the sense that the model is first pre-trained for a few epochs, then fine-tuned, then pre-trained, and so on. This would ensure that the model does not forget the first task when training the second. However, this framework has the potential drawback that the model might not learn any of the tasks particularly well and would hence not get an improved understanding of the texts.

Another possible reason for the relatively poor performance of the fine-tuned BERT model is that it was trained on the *Next* texts, which were not used in word-sense disambiguation. If these texts differ significantly compared to the other texts, the performance is likely to drop. One way to test this hypothesis would be to fine-tune the model on texts from another company, for example *Adidas* that has even more texts than *Next*. Another, and possibly the preferable way, would be to find a way to combine the category labels of the texts from all companies and fine-tune using this data. However, the risk is that combining these category labels might lead to only a few generic categories. Learning such labels could be extremely easy and might not lead to much improvements in the later task word-sense disambiguation.

Although the fine-tuned BERT model doesn't have an overall performance exceeding the ones by the other BERT models, it still leads to an improved performance in the final program. This is due to the fact that it is the best model for five word types. One example where this is extra apparent is for the word *vest*, where the performance is improved by 4.35% compared to the second best model.

As seen in Table 4.2, all three BERT models performs better than Word2Vec and Bag-of-Words. One reason for this is that BERT is a much more complex model and can capture the context of a word in a better way. Another reason is that Bag-of-Words and Word2Vec do not take the word order into account and only look at a window of words around the ambiguous word and not the full text. If the order of the words is important for the context, and not only which words that co-occur with the ambiguous word, it makes sense that BERT performs better than Word2Vec and Bag-of-Words.

We find it surprising that Bag-of-Words outperforms Word2Vec. This is probably due to the fact that aggregating the word vectors by a weighted mean is not the best way to represent the context. A better approach could be to input the word vectors around the ambiguous word to a bidirectional RNN, since this maintains the word order and causes no loss of information that is due to summing the vectors. This is also something that could be experimented with for the BERT models.

5.3 Using Non-annotated Data

From Table 4.3 we see that the gain from including the self-training procedure as a tunable parameter only amounts to 0.1% for the best model, which we find to be disappointing. The non-annotated data accounts for about half of all the data so it is likely that efficient usage of it could increase the performance. However, as shown in [35], self-training (and also a few other semi-supervised learning algorithms) does not in general improve the performance compared to the classifier learned only from the labeled data. Nevertheless, further investigation could be done on efficient usage of the non-annotated data, by for example trying other algorithms such as co-training, tri-training and co-EM.

5.4 Conclusion

As explained in Section 3.5, the results from Table 4.2 do not necessarily prove that the program can distinguish between the senses of a word when the text is from a fashion site not used in the training process. However, from the results in Table 4.4 we see that for all words the accuracy is significantly better than the accuracy of the baseline classifier, the classifier that predicts every sense to be the most frequent occurring sense. This indicates strongly that the classifiers have truly learned characteristics that constitute the sense of a word in a product description text.

Bibliography

[1] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993.

[2] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[3] Tohru Nitta. *Complex-Valued Neural Networks: Utilizing High-Dimensional Parameters: Utilizing High-Dimensional Parameters*. IGI Global, 2009.

[4] C. F. Jeff Wu. On the convergence properties of the em algorithm. *Ann. Statist.*, 11(1):95–103, 03 1983.

[5] Ye-Yi Wang and Alex Waibel. Decoding algorithm in statistical machine translation. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, pages 366–372. Association for Computational Linguistics, 1997.

[6] Ulrich Germann, Michael Jahr, Kevin Knight, Daniel Marcu, and Kenji Yamada. Fast and optimal decoding for machine translation. *Artificial Intelligence*, 154(1-2):127–143, 2004.

[7] Xiao Cui and Hao Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[8] Christopher D Manning, Christopher D Manning, and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

[9] Franz Josef Och, Nicola Ueffing, and Hermann Ney. An efficient a* search algorithm for statistical machine translation. In *Proceedings of the workshop on Data-driven methods in machine translation-Volume 14*, pages 1–8. Association for Computational Linguistics, 2001.

[10] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[11] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[12] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger,

editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[13] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[14] Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[15] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proc. of NAACL*, 2018.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[17] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, pages 5754–5764, 2019.

[18] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[20] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[21] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[22] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[23] Yuh-Jye Lee, Yi-Ren Yeh, and Hsing-Kuo Pao. An introduction to support vector machines. *National Taiwan University of Science and Technology*, 2010.

[24] Alexandru Niculescu-Mizil and Rich Caruana. Predicting good probabilities with supervised learning. In *Proceedings of the 22nd international conference on Machine learning*, pages 625–632, 2005.

[25] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27, 2011.

[26] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[27] Jafar Tanha, Maarten van Someren, and Hamideh Afsarmanesh. Semi-supervised self-training for decision tree classifiers. *International Journal of Machine Learning and Cybernetics*, 8(1):355–370, 2017.

[28] Peter I. Frazier. A Tutorial on Bayesian Optimization. *arXiv e-prints*, page arXiv:1807.02811, Jul 2018.

[29] Pierre Lison and Jörg Tiedemann. Opensubtitles2016: Extracting large parallel corpora from movie and tv subtitles. 2016.

[30] Aylin KOLBAŞI and Aydın ÜNSAL. A comparison of the outlier detecting methods: An application on turkish foreign trade data.

[31] Markus Ojala and Gemma C Garriga. Permutation tests for studying classifier performance. *Journal of Machine Learning Research*, 11(Jun):1833–1863, 2010.

[32] James Bergstra, Daniel Yamins, and David Daniel Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.

[33] Anthony Robins. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146, 1995.

[34] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[35] Yuanyuan Guo, Xiaoda Niu, and Harry Zhang. An extensive empirical study on semi-supervised learning. In *2010 IEEE International Conference on Data Mining*, pages 186–195. IEEE, 2010.

A

Appendix

A.1 Sense Definitions

Table A.1: *Definition of the senses used to produce the results in Table 4.2. The count and proportion columns refer to the number of texts that the particular sense appears in (in the annotated data).*

Word	Senses (en, sv, fr, es)	Count	Proportion
airy	airy, luftig, aérien, vaporoso	121	0.370
	airy, skir, vaporeux, vaporoso	206	0.630
ankle	ankle, ankel, cheville, tobillero	108	0.493
	ankle, ankel, cheville, tobillo	75	0.342
	ankle, vrist, cheville, tobillo	36	0.164
back	back, bak, arrière, detrás	237	0.110
	back, bak, arrière, espalda	171	0.079
	back, bak, dos, detrás	494	0.229
	back, bak, dos, espalda	555	0.257
	back, bak, dos, trasero	367	0.170
	back, rygg, dos, espalda	337	0.156
bag	bag, bag, sac, bolsa	22	0.119
	bag, bag, sac, bolso	17	0.092
	bag, bag, sac, riñonera	27	0.146
	bag, väska, sac, bolsa	19	0.103
	bag, väska, sac, bolso	62	0.335
	bag, väska, sac, riñonera	38	0.205
belt	belt, bälte, ceinture, cinturón	18	0.065
	belt, skärp, ceinture, cinturón	274	0.935
bottom	bottom, byxa, pantalon, pantalón	19	0.130
	bottom, nedtill, base, inferior	38	0.247
	bottom, trosa, bas, braga	96	0.623
button	button, knapp, bouton, botón	1736	0.787
	button, knäppning, boutonnage, botón	471	0.213
cap	cap, cap, casquette, gorra	62	0.241
	cap, holkärm, mancheron, japonés	76	0.296
	cap, keps, casquette, gorra	119	0.463

continued ...

. . . continued

Word	Senses (en, sv, fr, es)	Count	Proportion
celebrate	celebrate, hylla, célébrer, homenaje	36	0.562
	celebrate, hylla, célébrer, rendir	28	0.438
compartment	compartment, fack, compartiment, compartimento	165	0.809
	compartment, fack, poche, compartimento	39	0.191
cuff	cuff, mudd, poignet, puño	93	0.138
	cuff, slut, bas, puño	330	0.490
	cuff, slut, manche, puño	134	0.199
	cuff, slut, terminé, puño	117	0.174
cup	cup, bygel, bonnet, copa	93	0.246
	cup, kupa, bonnet, copa	285	0.754
cut	cut, modell, coupe, corte	438	0.930
	cut, passform, coupe, corte	33	0.070
day	day, dag, jour, día	57	0.350
	day, dag, journée, día	106	0.650
denim	denim, denim, denim, vaquero	198	0.876
	denim, denim, jean, denim	28	0.124
durable	durable, slitstark, durable, duradero	16	0.094
	durable, slitstark, résistant, resistente	58	0.339
	durable, tålig, résistant, resistente	97	0.567
elastic	elastic, elastisk, élastique, elástico	177	0.208
	elastic, resår, élastique, elástico	615	0.723
	elastic, resår, élastique, goma	59	0.069
fabric	fabric, material, matière, tejido	312	0.270
	fabric, material, tissu, tejido	361	0.313
	fabric, textil, textile, tela	100	0.087
	fabric, tyg, tissu, tejido	381	0.330
fastener	fastener, knäppning, boutonnage, cierre	24	0.155
	fastener, knäppning, fermeture, cierre	34	0.211
	fastener, spänne, attache, cierre	84	0.522
	fastener, spänne, boucle, hebilla	18	0.112
front	front, fram, avant, delantero	319	0.136
	front, fram, avant, frontal	285	0.121
	front, fram, devant, delante	467	0.199
	front, fram, devant, delantero	666	0.284
	front, fram, devant, frontal	611	0.260
gold	gold, guld, doré, dorado	65	0.492
	gold, guld, or, oro	67	0.508
hat	hat, hatt, chapeau, gorro	17	0.195
	hat, hatt, chapeau, sombrero	25	0.287
	hat, mössa, bonnet, gorro	45	0.517
heel	heel, häl, talon, talón	181	0.548
	heel, klack, talon, tacón	149	0.452
hem	hem, benslut, jambe, bajo	314	0.155

continued . . .

...continued

Word	Senses (en, sv, fr, es)	Count	Proportion
	hem, nederkant, base, bajo	1120	0.552
	hem, nederkant, ourlet, bajo	144	0.071
	hem, nedtill, base, bajo	450	0.222
high	high, hög, haut, alto	505	0.617
	high, hög, montant, alto	87	0.106
	high, hög, montant, subido	226	0.276
hoodie	hoodie, hoodie, capuche, capucha	177	0.525
	hoodie, hoodie, capuche, chaqueta	24	0.071
	hoodie, hoodie, hoodie, capucha	35	0.104
	hoodie, hoodie, hoodie, sudadera	18	0.053
	hoodie, huvtröja, capuche, chaqueta	83	0.246
jacket	jacket, jacka, veste, cazador	144	0.159
	jacket, jacka, veste, chaqueta	679	0.748
	jacket, kavaj, blazer, americano	85	0.094
jeans	jeans, jeans, jean, jeans	142	0.648
	jeans, jeans, jean, vaquero	77	0.352
jersey	jersey, jersey, maillot, camiseta	171	0.078
	jersey, trikå, jersey, punto	1705	0.773
	jersey, tröja, maillot, camiseta	330	0.150
lace	lace, snöre, lacet, cordón	86	0.107
	lace, snörning, lacet, cordón	89	0.111
	lace, spets, dentelle, encaje	627	0.782
large	large, stor, grand, grande	89	0.464
	large, stor, maximiser, realzar	50	0.260
	large, vid, grand, amplio	53	0.276
leather	leather, läder, cuir, piel	836	0.715
	leather, skinn, cuir, piel	334	0.285
leggings	leggings, leggings, legging, legging	185	0.591
	leggings, leggings, legging, malla	23	0.073
	leggings, tights, tight, malla	105	0.335
light	light, lätt, léger, ligero	81	0.623
	light, tunna, léger, ligero	49	0.377
logo	logo, logga, logo, logotipo	211	0.363
	logo, logo, logo, logotipo	329	0.565
	logo, logotyp, logo, logo	41	0.072
loose	loose, figurvänlig, ample, holgado	146	0.325
	loose, ledig, ample, holgado	129	0.287
	loose, lös, ample, holgado	174	0.388
modern	modern, modern, moderne, actual	112	0.500
	modern, modern, moderne, moderno	112	0.500
narrow	narrow, smal, fin, estrecho	115	0.157
	narrow, smal, fin, fino	470	0.642
	narrow, smal, mince, fino	81	0.111

continued ...

...continued

Word	Senses (en, sv, fr, es)	Count	Proportion
	narrow, smal, étroit, estrecho	66	0.090
neck	neck, krage, col, cuello	738	0.739
	neck, nacke, nuque, nuca	261	0.261
neckline	neckline, halsringning, encolure, cuello	340	0.354
	neckline, halsringning, encolure, escote	199	0.207
	neckline, ringning, encolure, cuello	184	0.191
	neckline, ringning, encolure, escote	238	0.248
perfect	perfect, perfekt, idéal, ideal	182	0.812
	perfect, perfekt, parfait, perfecto	42	0.188
print	print, mönster, imprimé, estampado	313	0.283
	print, mönstrad, imprimé, estampado	86	0.078
	print, tryck, impression, estampado	122	0.110
	print, tryck, imprimé, estampado	441	0.399
	print, tryckt, imprimé, estampado	144	0.130
relaxed	relaxed, avslappnad, décontracté, holgado	49	0.233
	relaxed, avslappnad, décontracté, relajado	47	0.224
	relaxed, ledig, décontracté, desenfadado	56	0.271
	relaxed, ledig, décontracté, holgado	56	0.271
ring	ring, ring, anneau, anilla	33	0.660
	ring, ring, anneau, aro	17	0.340
seam	seam, avskur, découpe, costura	417	0.435
	seam, avskur, découpe, entallada	52	0.054
	seam, söm, couture, costura	490	0.511
shirt	shirt, skjorta, chemise, camisa	450	0.909
	shirt, tröja, t-shirt, camiseta	45	0.091
side	side, sida, côté, lado	186	0.084
	side, sida, côté, lateral	902	0.410
	side, sida, latéral, bies	704	0.320
	side, sida, latéral, lateral	410	0.186
size	size, storlek, dimension, medida	219	0.575
	size, storlek, taille, talla	106	0.278
	size, storlek, taille, tamaño	56	0.147
slightly	slightly, lätt, légèrement, ligeramente	300	0.554
	slightly, något, légèrement, ligeramente	242	0.446
slim	slim, slim, slim, ajustado	86	0.213
	slim, smal, fin, ajustado	90	0.223
	slim, smal, slim, ajustado	71	0.176
	slim, smal, slim, entallado	88	0.218
	slim, smal, étroit, ajustado	69	0.171
slit	slit, instick, italien, abiertos	81	0.165
	slit, slit, fente, abertura	118	0.240
	slit, sprund, fente, abertura	293	0.596
smooth	smooth, slät, lisse, liso	391	0.918

continued ...

...continued

Word	Senses (en, sv, fr, es)	Count	Proportion
	smooth, smidig, doux, suave	35	0.082
sock	sock, socka, chaussette, calcetín	22	0.086
	sock, socka, socquette, calcetín	17	0.067
	sock, strumpa, chaussette, calcetín	127	0.498
	sock, strumpa, chaussette, media	57	0.224
	sock, strumpa, socquette, calcetín	32	0.125
soft	soft, mjuk, doux, suave	1798	0.633
	soft, mjuk, souple, suave	1043	0.367
strap	strap, band, bretelle, tirante	785	0.840
	strap, hängsle, bretelle, tirante	150	0.160
sweatshirt	sweatshirt, sweatshirt, molleton, felpa	101	0.127
	sweatshirt, sweatshirt, molleton, sudadera	333	0.420
	sweatshirt, sweatshirt, sweat, sudadera	359	0.453
swimsuit	swimsuit, baddräkt, bain, bañador	47	0.370
	swimsuit, baddräkt, bain, baño	59	0.465
	swimsuit, swimsuit, bain, bañador	21	0.165
t-shirt	t-shirt, t-shirt, t-shirt, camiseta	746	0.569
	t-shirt, tee, t-shirt, camiseta	424	0.323
	t-shirt, tröja, t-shirt, camiseta	142	0.108
text	text, text, inscription, texto	100	0.654
	text, text, texte, texto	53	0.346
tie	tie, knyta, nouer, anudar	71	0.250
	tie, knytband, lien, lazada	41	0.144
	tie, knytband, lien, tira	138	0.486
	tie, knytband, nouer, anudar	34	0.120
tights	tights, strumpbyxa, collant, media	16	0.057
	tights, tights, collant, malla	68	0.243
	tights, tights, cuissard, malla	35	0.125
	tights, tights, legging, malla	119	0.425
	tights, tights, tight, malla	42	0.150
top	top, topp, top, camiseta	192	0.170
	top, topp, top, top	436	0.385
	top, upptill, haut, superior	503	0.445
tracksuit	tracksuit, pants, pantalon, pantalón	46	0.426
	tracksuit, track, survêtement, chándal	22	0.204
	tracksuit, track, survêtement, pantalón	40	0.370
vest	vest, linne, débardeur, tirante	29	0.252
	vest, tanktop, débardeur, manga	56	0.487
	vest, väst, gilet, chaleco	30	0.261
waistband	waistband, linning, ceinture, cintura	68	0.090
	waistband, linning, taille, cintura	390	0.517
	waistband, linning, taille, cinturilla	235	0.313
	waistband, midja, taille, cintura	61	0.081

continued ...

. . . continued

Word	Senses (en, sv, fr, es)	Count	Proportion
washed	washed, blek, délavé, lavado	59	0.121
	washed, tvättad, laver, lavado	427	0.879
wide	wide, bred, large, ancho	719	0.652
	wide, vid, ample, amplio	156	0.141
	wide, vid, large, amplio	228	0.207
wool	wool, ull, laine, lana	175	0.774
	wool, ylle, laine, lana	51	0.226
zip	zip, dragkedja, glissière, cremallera	362	0.192
	zip, dragkedja, zip, cremallera	771	0.408
	zip, dragkedja, zippé, cremallera	756	0.400

A.2 Results for Each Model

Table A.2: *The results from WSD with the Bag-of-Words model for 69 ambiguous words after hypertuning. Each word uses different classifiers (with different parameters that are not displayed here) and are trained in either a supervised manner or using self-training. Note that it was advantageous to use the non-annotated data (self-training) for 39% (27/69) of the words. The total column refers to the total number of annotated data points. The baseline classifier is simply taken to be the classifier that always predicts the most frequent occurring sense, see Table A.1. Its performance is given in the column BL acc.*

Word	Classifier	Trained	Tot.	Senses	BL acc.	Acc.
airy	RFC	Supervised	327	2	0.6300	0.7985
ankle	Neural net	Supervised	219	3	0.4932	0.9544
back	Neural net	Supervised	2161	6	0.2568	0.7534
bag	RFC	Self-training	185	6	0.3351	0.7707
belt	RFC	Supervised	293	2	0.9352	0.9489
bottom	SVC	Self-training	154	3	0.6234	0.9804
button	Neural net	Self-training	2207	2	0.7866	0.9538
cap	RFC	Self-training	257	3	0.4630	0.9034
celebrate	SVC	Supervised	64	2	0.5625	0.8594
compartment	RFC	Supervised	204	2	0.8088	0.8530
cuff	Neural net	Self-training	674	4	0.4896	0.8574
cup	kNN	Self-training	378	2	0.7540	0.9497
cut	GaussianNB	Self-training	471	2	0.9299	0.9872
day	kNN	Supervised	163	2	0.6503	0.9084
denim	GaussianNB	Supervised	226	2	0.8761	1.0000
durable	RFC	Supervised	171	3	0.5673	0.7780
elastic	Neural net	Supervised	851	3	0.7227	0.9378
fabric	Neural net	Supervised	1154	4	0.3302	0.7739
fastener	RFC	Supervised	161	4	0.5217	0.8572
front	Neural net	Supervised	2348	5	0.2836	0.6669
gold	RFC	Self-training	132	2	0.5076	0.9770
hat	Neural net	Supervised	87	3	0.5172	0.9896
heel	SVC	Self-training	330	2	0.5485	0.9909
hem	Neural net	Self-training	2028	4	0.5523	0.8215
high	Neural net	Self-training	818	3	0.6174	0.9523
hoodie	Neural net	Supervised	337	5	0.5252	0.6973
jacket	Neural net	Supervised	908	3	0.7478	0.9472
jeans	RFC	Supervised	219	2	0.6484	0.9816
jersey	Neural net	Self-training	2206	3	0.7729	0.9683
lace	Neural net	Supervised	802	3	0.7818	0.9638
large	Neural net	Supervised	192	3	0.4635	0.9850
leather	Neural net	Supervised	1171	2	0.7148	0.9821

continued . . .

. . . continued

Word	Classifier	Trained	Tot.	Senses	BL acc.	Acc.
leggings	Neural net	Supervised	313	3	0.5911	0.9264
light	Neural net	Self-training	130	2	0.6231	1.0000
logo	Neural net	Supervised	582	3	0.5653	0.9914
loose	RFC	Self-training	449	3	0.3875	0.9599
modern	RFC	Supervised	224	2	0.5000	0.8884
narrow	Neural net	Self-training	732	4	0.6421	0.8236
neck	SVC	Supervised	999	2	0.7387	0.9990
neckline	RFC	Supervised	961	4	0.3538	0.6359
perfect	Neural net	Supervised	224	2	0.8125	0.9911
print	Neural net	Self-training	1106	5	0.3987	0.8146
relaxed	GaussianNB	Supervised	210	4	0.2714	0.7757
ring	Neural net	Supervised	50	2	0.6600	0.8810
seam	RFC	Supervised	959	3	0.5109	0.8946
shirt	Neural net	Supervised	495	2	0.9091	0.9798
side	Neural net	Supervised	2202	4	0.4096	0.7875
size	RFC	Supervised	381	3	0.5748	0.9527
slightly	SVC	Self-training	542	2	0.5535	0.9391
slim	Neural net	Self-training	404	5	0.2228	0.9381
slit	Neural net	Self-training	492	3	0.5955	0.7886
smooth	Neural net	Supervised	426	2	0.9178	0.9976
sock	RFC	Self-training	255	5	0.4980	0.8166
soft	Neural net	Supervised	2842	2	0.6330	0.8740
strap	Neural net	Self-training	935	2	0.8396	0.9668
sweatshirt	Neural net	Supervised	793	3	0.4527	0.9886
swimsuit	RFC	Supervised	127	3	0.4646	0.9296
t-shirt	RFC	Supervised	1312	3	0.5686	0.8445
text	Neural net	Supervised	153	2	0.6536	0.9804
tie	Neural net	Self-training	284	4	0.4859	0.6759
tights	RFC	Self-training	280	5	0.4250	0.7566
top	Neural net	Self-training	1131	3	0.4447	0.8444
tracksuit	RFC	Self-training	108	3	0.4259	0.8146
vest	RFC	Self-training	115	3	0.4870	0.8968
waistband	Neural net	Supervised	755	4	0.5166	0.8423
washed	kNN	Self-training	486	2	0.8786	1.0000
wide	Neural net	Supervised	1103	3	0.6519	0.8449
wool	Neural net	Supervised	226	2	0.7743	0.9823
zip	SVC	Supervised	1889	3	0.4082	0.8941
Average	**-**	**-**	**675**	**3.0**	**0.5835**	**0.8937**

Table A.3: *The results from WSD with the Word2Vec model for 69 ambiguous words after hypertuning. Each word uses different classifiers (with different parameters that are not displayed here) and are trained in either a supervised manner or using self-training. Note that it was advantageous to use the non-annotated data (self-training) for 48% (33/69) of the words. The total column refers to the total number of annotated data points. The baseline classifier is simply taken to be the classifier that always predicts the most frequent occurring sense, see Table A.1. Its performance is given in the column BL acc.*

Word	Classifier	Trained	Tot.	Senses	BL acc.	Acc.
airy	RFC	Self-training	327	2	0.6300	0.7799
ankle	kNN	Self-training	219	3	0.4932	0.9453
back	SVC	Self-training	2161	6	0.2568	0.7284
bag	SVC	Self-training	185	6	0.3351	0.7698
belt	RFC	Supervised	293	2	0.9352	0.9625
bottom	kNN	Supervised	154	3	0.6234	0.9872
button	SVC	Self-training	2207	2	0.7866	0.9488
cap	Neural net	Self-training	257	3	0.4630	0.8838
celebrate	GaussianNB	Supervised	64	2	0.5625	0.7812
compartment	RFC	Supervised	204	2	0.8088	0.8723
cuff	kNN	Supervised	674	4	0.4896	0.8145
cup	kNN	Self-training	378	2	0.7540	0.9073
cut	GaussianNB	Supervised	471	2	0.9299	0.9936
day	RFC	Self-training	163	2	0.6503	0.8038
denim	SVC	Self-training	226	2	0.8761	0.9956
durable	RFC	Self-training	171	3	0.5673	0.7080
elastic	kNN	Self-training	851	3	0.7227	0.9037
fabric	SVC	Self-training	1154	4	0.3302	0.7505
fastener	kNN	Supervised	161	4	0.5217	0.8151
front	RFC	Self-training	2348	5	0.2836	0.6444
gold	GaussianNB	Self-training	132	2	0.5076	0.9770
hat	SVC	Supervised	87	3	0.5172	0.9554
heel	SVC	Supervised	330	2	0.5485	0.9849
hem	SVC	Self-training	2028	4	0.5523	0.7860
high	RFC	Supervised	818	3	0.6174	0.9279
hoodie	SVC	Supervised	337	5	0.5252	0.6856
jacket	Neural net	Supervised	908	3	0.7478	0.9471
jeans	SVC	Supervised	219	2	0.6484	0.9769
jersey	RFC	Supervised	2206	3	0.7729	0.9411
lace	SVC	Self-training	802	3	0.7818	0.9501
large	RFC	Self-training	192	3	0.4635	0.9793
leather	kNN	Self-training	1171	2	0.7148	0.9735
leggings	Neural net	Self-training	313	3	0.5911	0.9296
light	Neural net	Self-training	130	2	0.6231	0.9848
logo	SVC	Supervised	582	3	0.5653	0.9880

continued . . .

...continued

Word	Classifier	Trained	Tot.	Senses	BL acc.	Acc.
loose	Neural net	Self-training	449	3	0.3875	0.9332
modern	RFC	Self-training	224	2	0.5000	0.8839
narrow	kNN	Supervised	732	4	0.6421	0.7856
neck	kNN	Self-training	999	2	0.7387	0.9960
neckline	RFC	Supervised	961	4	0.3538	0.6149
perfect	kNN	Supervised	224	2	0.8125	0.9376
print	Neural net	Supervised	1106	5	0.3987	0.7442
relaxed	GaussianNB	Supervised	210	4	0.2714	0.7667
ring	RFC	Self-training	50	2	0.6600	0.8601
seam	Neural net	Supervised	959	3	0.5109	0.8759
shirt	SVC	Supervised	495	2	0.9091	0.9819
side	RFC	Self-training	2202	4	0.4096	0.7661
size	RFC	Supervised	381	3	0.5748	0.9189
slightly	SVC	Supervised	542	2	0.5535	0.9041
slim	SVC	Self-training	404	5	0.2228	0.8712
slit	SVC	Supervised	492	3	0.5955	0.7804
smooth	kNN	Supervised	426	2	0.9178	1.0000
sock	SVC	Supervised	255	5	0.4980	0.7740
soft	RFC	Supervised	2842	2	0.6330	0.8610
strap	SVC	Self-training	935	2	0.8396	0.9636
sweatshirt	Neural net	Self-training	793	3	0.4527	0.9849
swimsuit	SVC	Supervised	127	3	0.4646	0.8600
t-shirt	RFC	Self-training	1312	3	0.5686	0.8125
text	RFC	Supervised	153	2	0.6536	0.9609
tie	RFC	Supervised	284	4	0.4859	0.6549
tights	RFC	Supervised	280	5	0.4250	0.6573
top	RFC	Supervised	1131	3	0.4447	0.8152
tracksuit	RFC	Supervised	108	3	0.4259	0.7692
vest	kNN	Self-training	115	3	0.4870	0.9051
waistband	SVC	Supervised	755	4	0.5166	0.8318
washed	Neural net	Self-training	486	2	0.8786	1.0000
wide	RFC	Self-training	1103	3	0.6519	0.8177
wool	kNN	Self-training	226	2	0.7743	0.9599
zip	SVC	Supervised	1889	3	0.4082	0.8512
Average	**-**	**-**	**675**	**3.0**	**0.5835**	**0.8708**

Table A.4: *The results from WSD with the BERT Base model for 69 ambiguous words after hypertuning. Each word uses different classifiers (with different parameters that are not displayed here) and are trained in either a supervised manner or using self-training. Note that it was advantageous to use the non-annotated data (self-training) for 45% (31/69) of the words. The total column refers to the total number of annotated data points. The baseline classifier is simply taken to be the classifier that always predicts the most frequent occurring sense, see Table A.1. Its performance is given in the column BL acc.*

Word	Classifier	Trained	Tot.	Senses	BL acc.	Acc.
airy	Neural net	Supervised	327	2	0.6300	0.7830
ankle	RFC	Supervised	219	3	0.4932	0.9362
back	SVC	Self-training	2161	6	0.2568	0.7692
bag	SVC	Supervised	185	6	0.3351	0.8449
belt	SVC	Supervised	293	2	0.9352	0.9694
bottom	RFC	Supervised	154	3	0.6234	1.0000
button	Neural net	Supervised	2207	2	0.7866	0.9787
cap	RFC	Self-training	257	3	0.4630	0.9228
celebrate	kNN	Supervised	64	2	0.5625	0.8750
compartment	kNN	Supervised	204	2	0.8088	0.8673
cuff	SVC	Self-training	674	4	0.4896	0.8976
cup	SVC	Supervised	378	2	0.7540	0.9656
cut	kNN	Supervised	471	2	0.9299	0.9915
day	SVC	Self-training	163	2	0.6503	0.9331
denim	SVC	Supervised	226	2	0.8761	1.0000
durable	GaussianNB	Supervised	171	3	0.5673	0.7828
elastic	SVC	Self-training	851	3	0.7227	0.9365
fabric	kNN	Supervised	1154	4	0.3302	0.7868
fastener	Neural net	Supervised	161	4	0.5217	0.9132
front	SVC	Supervised	2348	5	0.2836	0.7667
gold	Neural net	Supervised	132	2	0.5076	0.9770
hat	SVC	Supervised	87	3	0.5172	0.9762
heel	kNN	Self-training	330	2	0.5485	0.9817
hem	Neural net	Supervised	2028	4	0.5523	0.8580
high	Neural net	Self-training	818	3	0.6174	0.9365
hoodie	Neural net	Self-training	337	5	0.5252	0.6976
jacket	SVC	Self-training	908	3	0.7478	0.9405
jeans	SVC	Self-training	219	2	0.6484	0.9816
jersey	Neural net	Self-training	2206	3	0.7729	0.9828
lace	kNN	Supervised	802	3	0.7818	0.9626
large	RFC	Supervised	192	3	0.4635	1.0000
leather	Neural net	Supervised	1171	2	0.7148	0.9778
leggings	Neural net	Self-training	313	3	0.5911	0.9393
light	kNN	Supervised	130	2	0.6231	1.0000
logo	SVC	Supervised	582	3	0.5653	0.9931

continued . . .

...continued

Word	Classifier	Trained	Tot.	Senses	BL acc.	Acc.
loose	SVC	Self-training	449	3	0.3875	0.9621
modern	SVC	Supervised	224	2	0.5000	0.9018
narrow	kNN	Supervised	732	4	0.6421	0.8306
neck	kNN	Supervised	999	2	0.7387	0.9990
neckline	SVC	Supervised	961	4	0.3538	0.6596
perfect	SVC	Self-training	224	2	0.8125	0.9909
print	SVC	Self-training	1106	5	0.3987	0.8625
relaxed	kNN	Supervised	210	4	0.2714	0.7707
ring	Neural net	Supervised	50	2	0.6600	0.9196
seam	Neural net	Self-training	959	3	0.5109	0.9186
shirt	Neural net	Supervised	495	2	0.9091	0.9838
side	SVC	Supervised	2202	4	0.4096	0.8587
size	SVC	Supervised	381	3	0.5748	0.9896
slightly	SVC	Self-training	542	2	0.5535	0.9686
slim	SVC	Self-training	404	5	0.2228	0.9207
slit	RFC	Self-training	492	3	0.5955	0.8028
smooth	SVC	Self-training	426	2	0.9178	1.0000
sock	kNN	Self-training	255	5	0.4980	0.7967
soft	SVC	Self-training	2842	2	0.6330	0.8849
strap	SVC	Self-training	935	2	0.8396	0.9765
sweatshirt	GaussianNB	Supervised	793	3	0.4527	0.9861
swimsuit	SVC	Supervised	127	3	0.4646	0.8986
t-shirt	SVC	Self-training	1312	3	0.5686	0.9002
text	SVC	Self-training	153	2	0.6536	0.9671
tie	RFC	Self-training	284	4	0.4859	0.7817
tights	RFC	Self-training	280	5	0.4250	0.6926
top	RFC	Supervised	1131	3	0.4447	0.8665
tracksuit	Neural net	Supervised	108	3	0.4259	0.8984
vest	RFC	Self-training	115	3	0.4870	0.9128
waistband	kNN	Self-training	755	4	0.5166	0.8410
washed	Neural net	Supervised	486	2	0.8786	1.0000
wide	SVC	Supervised	1103	3	0.6519	0.9193
wool	SVC	Self-training	226	2	0.7743	0.9604
zip	SVC	Self-training	1889	3	0.4082	0.9248
Average	**-**	**-**	**675**	**3.0**	**0.5835**	**0.9083**

Table A.5: *The results from WSD with the pre-trained BERT model for 69 ambiguous words after hypertuning. Each word uses different classifiers (with different parameters that are not displayed here) and are trained in either a supervised manner or using self-training. Note that it was advantageous to use the non-annotated data (self-training) for 52% (36/69) of the words. The total column refers to the total number of annotated data points. The baseline classifier is simply taken to be the classifier that always predicts the most frequent occurring sense, see Table A.1. Its performance is given in the column BL acc.*

Word	**Classifier**	**Trained**	**Tot.**	**Senses**	**BL acc.**	**Acc.**
airy	RFC	Supervised	327	2	0.6300	0.7773
ankle	RFC	Self-training	219	3	0.4932	0.9407
back	RFC	Supervised	2161	6	0.2568	0.7927
bag	Neural net	Supervised	185	6	0.3351	0.8823
belt	SVC	Self-training	293	2	0.9352	0.9727
bottom	kNN	Self-training	154	3	0.6234	1.0000
button	SVC	Self-training	2207	2	0.7866	0.9760
cap	Neural net	Self-training	257	3	0.4630	0.9224
celebrate	GaussianNB	Supervised	64	2	0.5625	0.8438
compartment	SVC	Self-training	204	2	0.8088	0.8922
cuff	Neural net	Self-training	674	4	0.4896	0.9006
cup	Neural net	Supervised	378	2	0.7540	0.9736
cut	Neural net	Supervised	471	2	0.9299	0.9894
day	kNN	Self-training	163	2	0.6503	0.9268
denim	SVC	Supervised	226	2	0.8761	1.0000
durable	GaussianNB	Supervised	171	3	0.5673	0.7485
elastic	SVC	Self-training	851	3	0.7227	0.9495
fabric	SVC	Self-training	1154	4	0.3302	0.8163
fastener	SVC	Supervised	161	4	0.5217	0.9072
front	SVC	Self-training	2348	5	0.2836	0.7747
gold	kNN	Supervised	132	2	0.5076	0.9924
hat	SVC	Supervised	87	3	0.5172	1.0000
heel	SVC	Self-training	330	2	0.5485	0.9878
hem	RFC	Supervised	2028	4	0.5523	0.8570
high	Neural net	Self-training	818	3	0.6174	0.9584
hoodie	kNN	Supervised	337	5	0.5252	0.6950
jacket	kNN	Supervised	908	3	0.7478	0.9504
jeans	SVC	Supervised	219	2	0.6484	0.9907
jersey	kNN	Self-training	2206	3	0.7729	0.9855
lace	kNN	Self-training	802	3	0.7818	0.9626
large	Neural net	Self-training	192	3	0.4635	1.0000
leather	SVC	Self-training	1171	2	0.7148	0.9838
leggings	kNN	Self-training	313	3	0.5911	0.9647
light	Neural net	Supervised	130	2	0.6231	1.0000
logo	Neural net	Supervised	582	3	0.5653	0.9863

continued . . .

. . . continued

Word	**Classifier**	**Trained**	**Tot.**	**Senses**	**BL acc.**	**Acc.**
loose	RFC	Supervised	449	3	0.3875	0.9554
modern	kNN	Self-training	224	2	0.5000	0.9152
narrow	RFC	Supervised	732	4	0.6421	0.8264
neck	Neural net	Self-training	999	2	0.7387	1.0000
neckline	RFC	Supervised	961	4	0.3538	0.6774
perfect	SVC	Self-training	224	2	0.8125	1.0000
print	SVC	Self-training	1106	5	0.3987	0.8706
relaxed	kNN	Self-training	210	4	0.2714	0.7374
ring	Neural net	Supervised	50	2	0.6600	0.8452
seam	Neural net	Supervised	959	3	0.5109	0.9218
shirt	SVC	Supervised	495	2	0.9091	0.9859
side	RFC	Self-training	2202	4	0.4096	0.8683
size	SVC	Self-training	381	3	0.5748	0.9922
slightly	kNN	Supervised	542	2	0.5535	0.9705
slim	SVC	Self-training	404	5	0.2228	0.9554
slit	kNN	Self-training	492	3	0.5955	0.8250
smooth	kNN	Supervised	426	2	0.9178	1.0000
sock	SVC	Self-training	255	5	0.4980	0.8360
soft	Neural net	Supervised	2842	2	0.6330	0.8828
strap	Neural net	Supervised	935	2	0.8396	0.9829
sweatshirt	Neural net	Self-training	793	3	0.4527	0.9912
swimsuit	kNN	Supervised	127	3	0.4646	0.9453
t-shirt	kNN	Self-training	1312	3	0.5686	0.9055
text	RFC	Self-training	153	2	0.6536	0.9868
tie	RFC	Self-training	284	4	0.4859	0.7818
tights	SVC	Self-training	280	5	0.4250	0.7536
top	RFC	Supervised	1131	3	0.4447	0.8815
tracksuit	Neural net	Self-training	108	3	0.4259	0.9348
vest	GaussianNB	Supervised	115	3	0.4870	0.9045
waistband	kNN	Self-training	755	4	0.5166	0.8555
washed	SVC	Supervised	486	2	0.8786	1.0000
wide	kNN	Self-training	1103	3	0.6519	0.9238
wool	RFC	Supervised	226	2	0.7743	0.9825
zip	SVC	Supervised	1889	3	0.4082	0.9296
Average	**-**	**-**	**675**	**3.0**	**0.5835**	**0.9149**

Table A.6: *The results from WSD with the pre-trained and fine-tuned BERT model for 69 ambiguous words after hypertuning. Each word uses different classifiers (with different parameters that are not displayed here) and are trained in either a supervised manner or using self-training. Note that it was advantageous to use the non-annotated data (self-training) for 41% (28/69) of the words. The total column refers to the total number of annotated data points. The baseline classifier is simply taken to be the classifier that always predicts the most frequent occurring sense, see Table A.1. Its performance is given in the column BL acc.*

Word	**Classifier**	**Trained**	**Tot.**	**Senses**	**BL acc.**	**Acc.**
airy	SVC	Supervised	327	2	0.6300	0.7831
ankle	SVC	Self-training	219	3	0.4932	0.9362
back	Neural net	Self-training	2161	6	0.2568	0.7844
bag	Neural net	Supervised	185	6	0.3351	0.8397
belt	SVC	Supervised	293	2	0.9352	0.9658
bottom	Neural net	Supervised	154	3	0.6234	0.9936
button	Neural net	Supervised	2207	2	0.7866	0.9678
cap	kNN	Supervised	257	3	0.4630	0.9263
celebrate	GaussianNB	Supervised	64	2	0.5625	0.8438
compartment	SVC	Self-training	204	2	0.8088	0.8825
cuff	Neural net	Self-training	674	4	0.4896	0.8785
cup	Neural net	Supervised	378	2	0.7540	0.9470
cut	GaussianNB	Supervised	471	2	0.9299	0.9894
day	Neural net	Self-training	163	2	0.6503	0.8964
denim	RFC	Supervised	226	2	0.8761	0.9778
durable	SVC	Self-training	171	3	0.5673	0.7540
elastic	SVC	Supervised	851	3	0.7227	0.9295
fabric	SVC	Supervised	1154	4	0.3302	0.7661
fastener	Neural net	Self-training	161	4	0.5217	0.8888
front	SVC	Supervised	2348	5	0.2836	0.7470
gold	Neural net	Supervised	132	2	0.5076	0.9770
hat	SVC	Supervised	87	3	0.5172	0.9554
heel	RFC	Self-training	330	2	0.5485	0.9787
hem	Neural net	Supervised	2028	4	0.5523	0.8299
high	Neural net	Self-training	818	3	0.6174	0.9536
hoodie	RFC	Self-training	337	5	0.5252	0.6767
jacket	Neural net	Self-training	908	3	0.7478	0.9438
jeans	kNN	Self-training	219	2	0.6484	0.9816
jersey	SVC	Self-training	2206	3	0.7729	0.9841
lace	SVC	Supervised	802	3	0.7818	0.9601
large	SVC	Supervised	192	3	0.4635	1.0000
leather	Neural net	Supervised	1171	2	0.7148	0.9735
leggings	SVC	Supervised	313	3	0.5911	0.9487
light	SVC	Self-training	130	2	0.6231	1.0000
logo	SVC	Self-training	582	3	0.5653	0.9897

continued . . .

...continued

Word	Classifier	Trained	Tot.	Senses	BL acc.	Acc.
loose	Neural net	Supervised	449	3	0.3875	0.9621
modern	RFC	Supervised	224	2	0.5000	0.8795
narrow	SVC	Self-training	732	4	0.6421	0.8156
neck	Neural net	Supervised	999	2	0.7387	0.9990
neckline	Neural net	Supervised	961	4	0.3538	0.6629
perfect	Neural net	Self-training	224	2	0.8125	0.9776
print	SVC	Self-training	1106	5	0.3987	0.8146
relaxed	Neural net	Supervised	210	4	0.2714	0.7527
ring	SVC	Supervised	50	2	0.6600	0.8780
seam	Neural net	Supervised	959	3	0.5109	0.9207
shirt	SVC	Supervised	495	2	0.9091	0.9879
side	Neural net	Supervised	2202	4	0.4096	0.8561
size	SVC	Supervised	381	3	0.5748	0.9791
slightly	SVC	Self-training	542	2	0.5535	0.9391
slim	Neural net	Supervised	404	5	0.2228	0.9651
slit	RFC	Self-training	492	3	0.5955	0.8172
smooth	Neural net	Supervised	426	2	0.9178	0.9977
sock	RFC	Self-training	255	5	0.4980	0.7922
soft	SVC	Supervised	2842	2	0.6330	0.8691
strap	Neural net	Supervised	935	2	0.8396	0.9808
sweatshirt	Neural net	Self-training	793	3	0.4527	0.9798
swimsuit	SVC	Supervised	127	3	0.4646	0.9223
t-shirt	Neural net	Self-training	1312	3	0.5686	0.8933
text	Neural net	Self-training	153	2	0.6536	0.9479
tie	Neural net	Self-training	284	4	0.4859	0.7542
tights	RFC	Supervised	280	5	0.4250	0.7108
top	Neural net	Supervised	1131	3	0.4447	0.8771
tracksuit	Neural net	Supervised	108	3	0.4259	0.8977
vest	Neural net	Supervised	115	3	0.4870	0.9563
waistband	SVC	Self-training	755	4	0.5166	0.8450
washed	Neural net	Self-training	486	2	0.8786	0.9938
wide	SVC	Supervised	1103	3	0.6519	0.9156
wool	Neural net	Self-training	226	2	0.7743	0.9604
zip	Neural net	Supervised	1889	3	0.4082	0.9142
Average	**-**	**-**	**675**	**3.0**	**0.5835**	**0.9024**

A.3 Examples of the Permutation Test

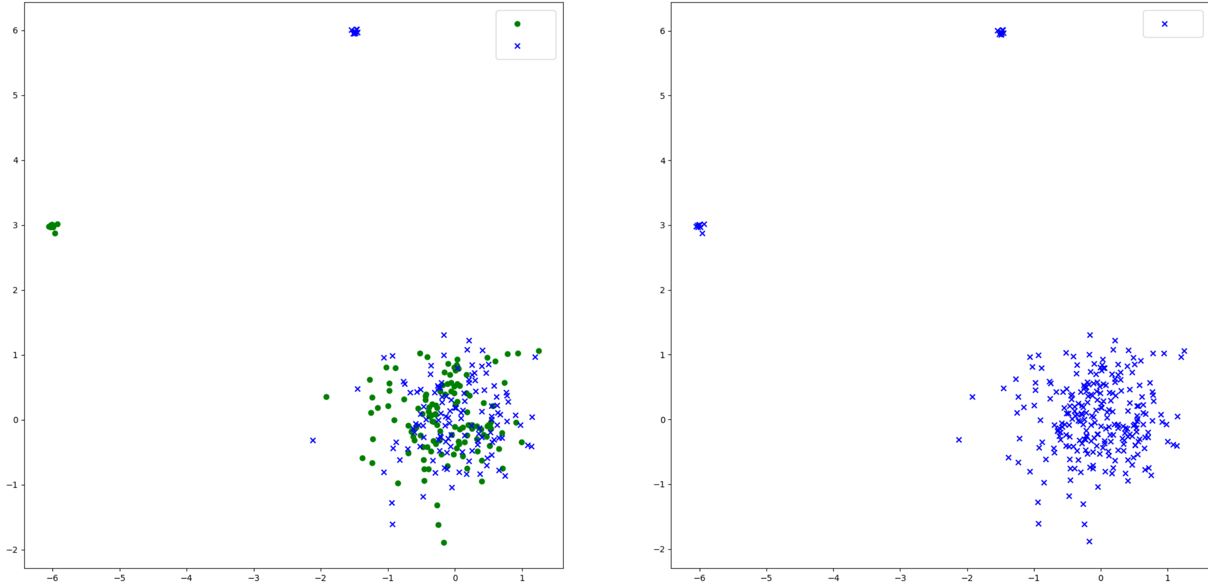


Figure A.1: *Left picture shows two classes before the permutation test and SenseMerge and right picture after.*

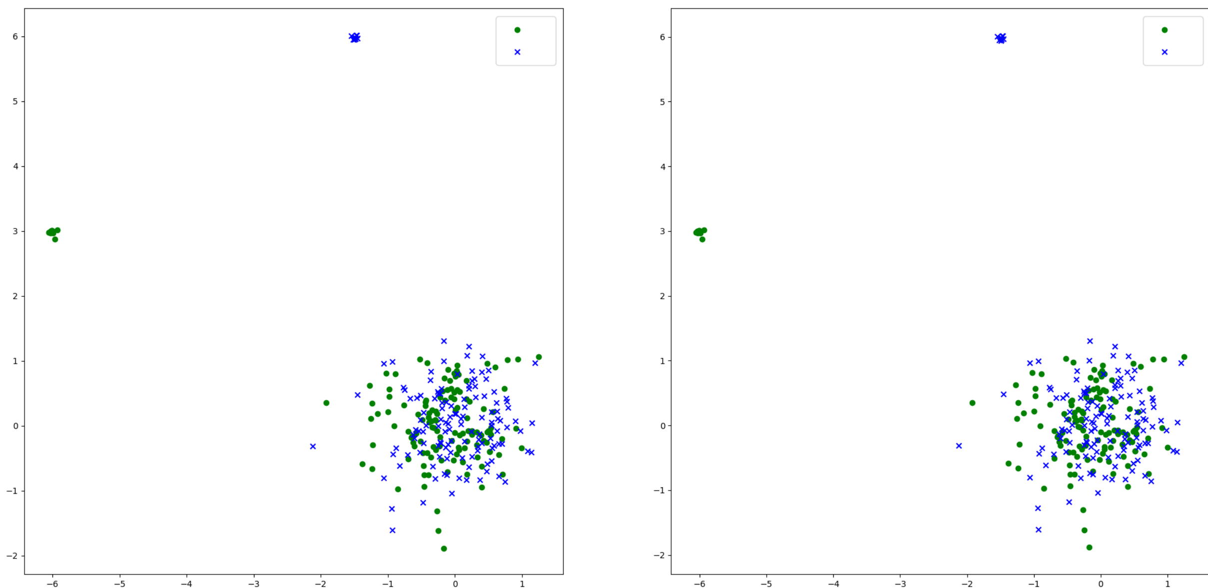


Figure A.2: *Same as the picture above but here we use $d(s(Q_i), s(Q_j))$ in the numerator instead of $d_{0.5}(s(Q_i), s(Q_j))$ in Equation 3.2. Note that the two classes are not merged which is not what we want.*

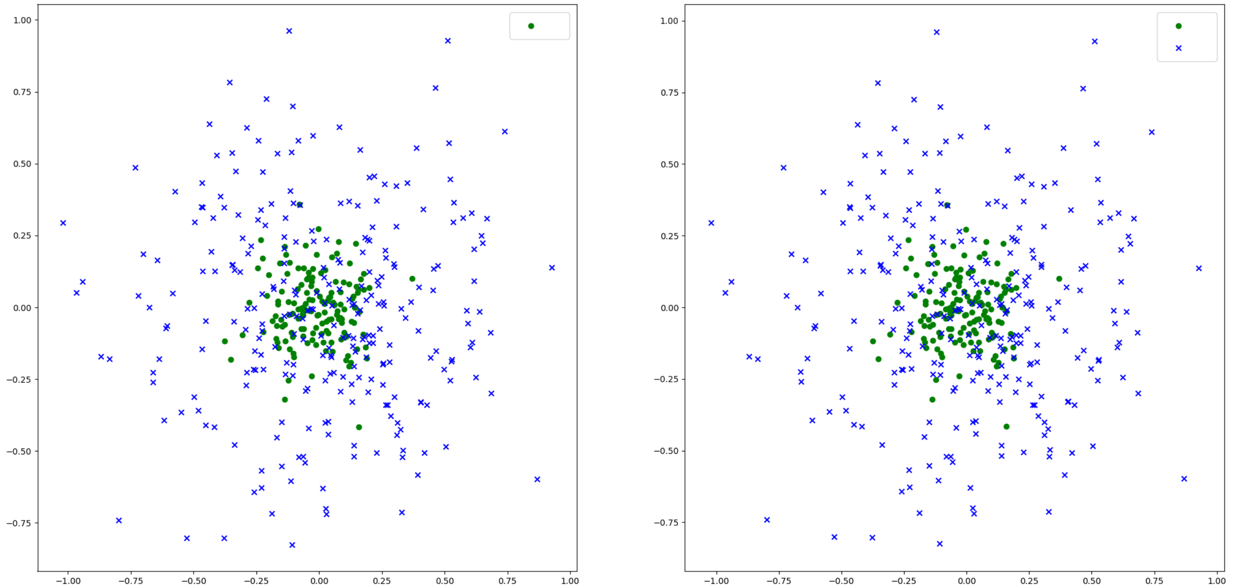


Figure A.3: *Left picture shows two classes before the permutation test and SenseMerge and right picture after. Green class is* $\sim N(0, 10^{-2}I)$. *Blue is* $\sim N(0, 10^{-1}I)$ *and within the disc* $x^2 + y^2 \leq 0.25$ *points are removed with probability 0.5.*

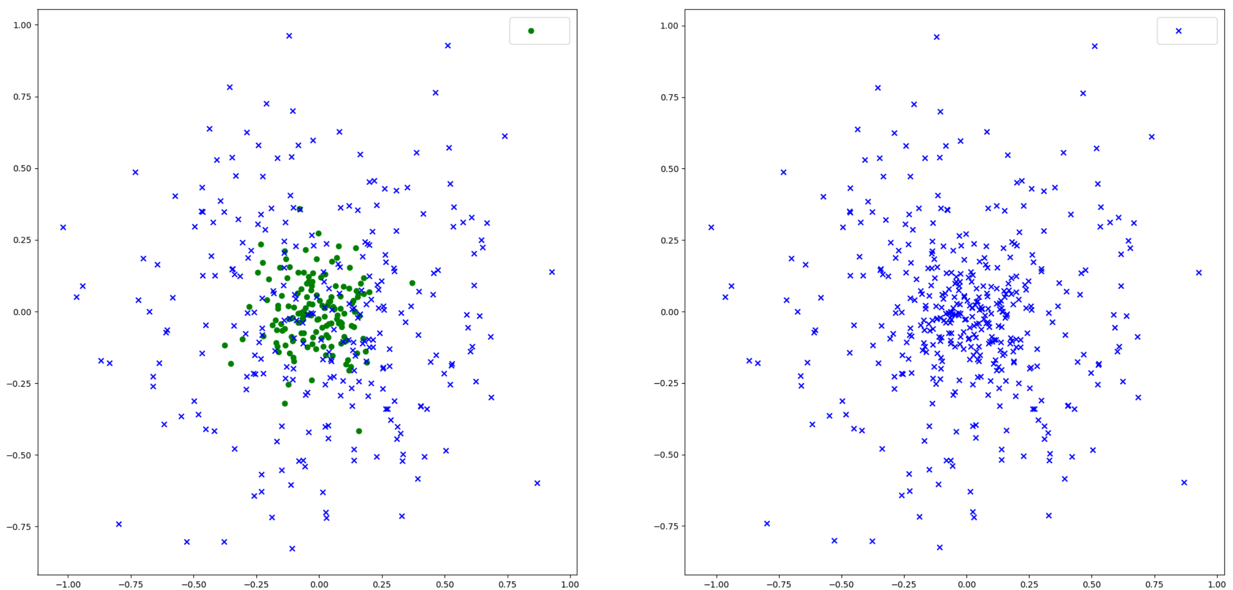


Figure A.4: *Same as the picture above but here we use* $T = d_{0.5}(s(Q_i), s(Q_j))$ *as statistic in Equation 3.2, i.e. without the denominator. Note that the two classes are merged which is not what we want*

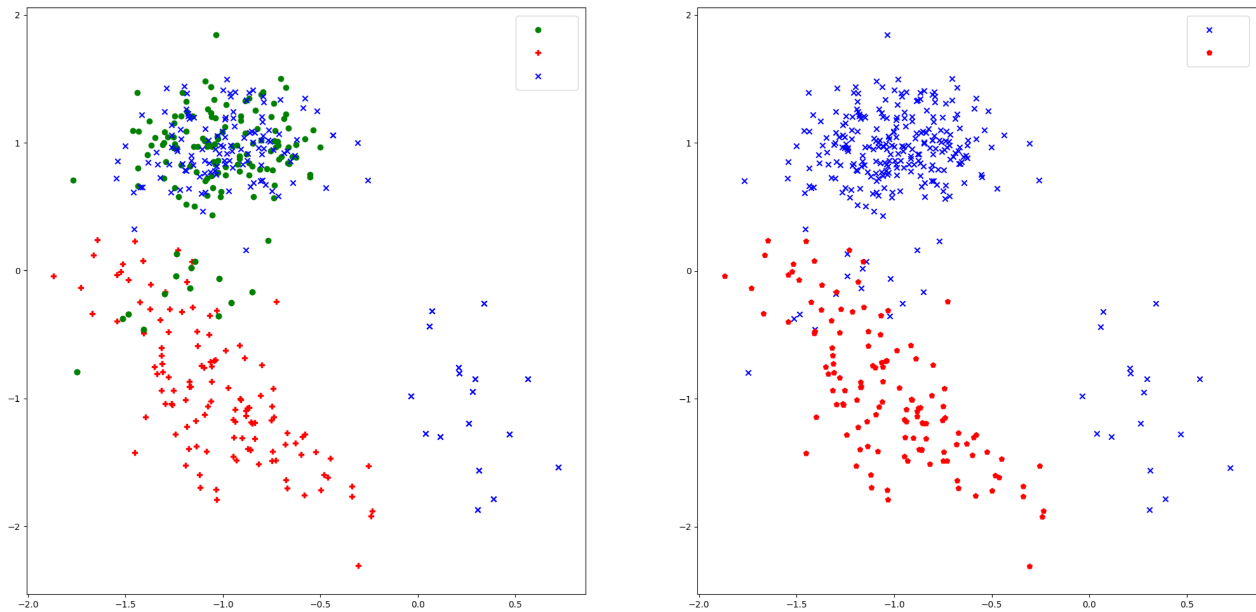


Figure A.5: *Left picture shows three classes before the permutation test and Sense-Merge and right picture after.*

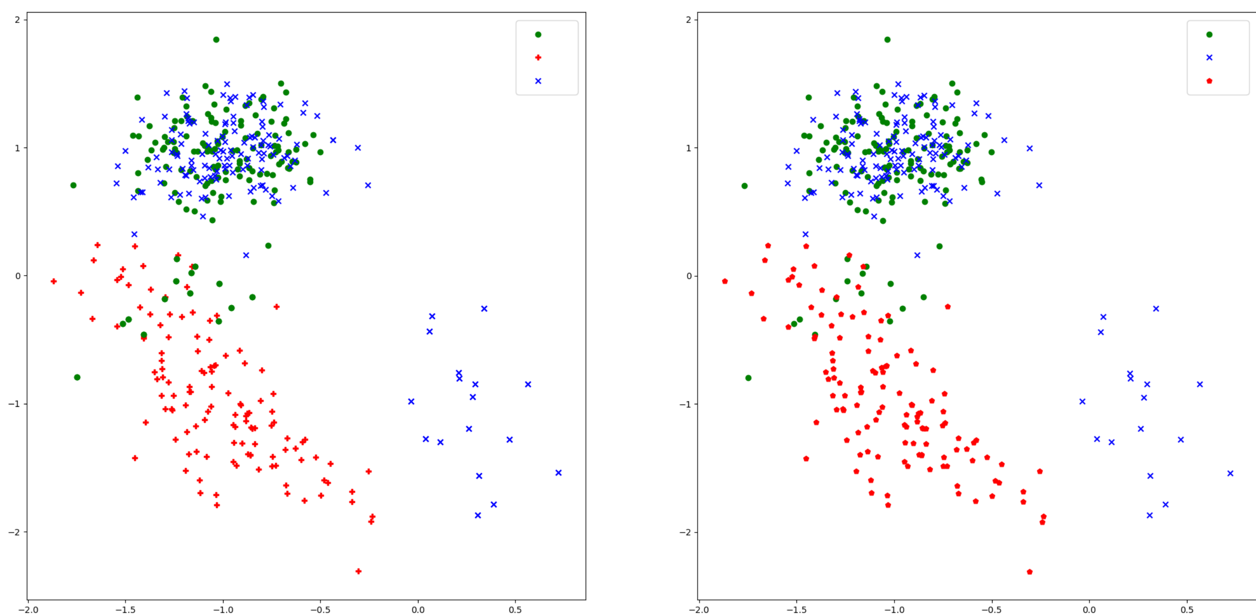


Figure A.6: *Same as the picture above but here we use $d(s(Q_i), s(Q_j))$ in the numerator instead of $d_{0.5}(s(Q_i), s(Q_j))$ in Equation 3.2. Note that the green and blue class are not merged which is not what we want.*

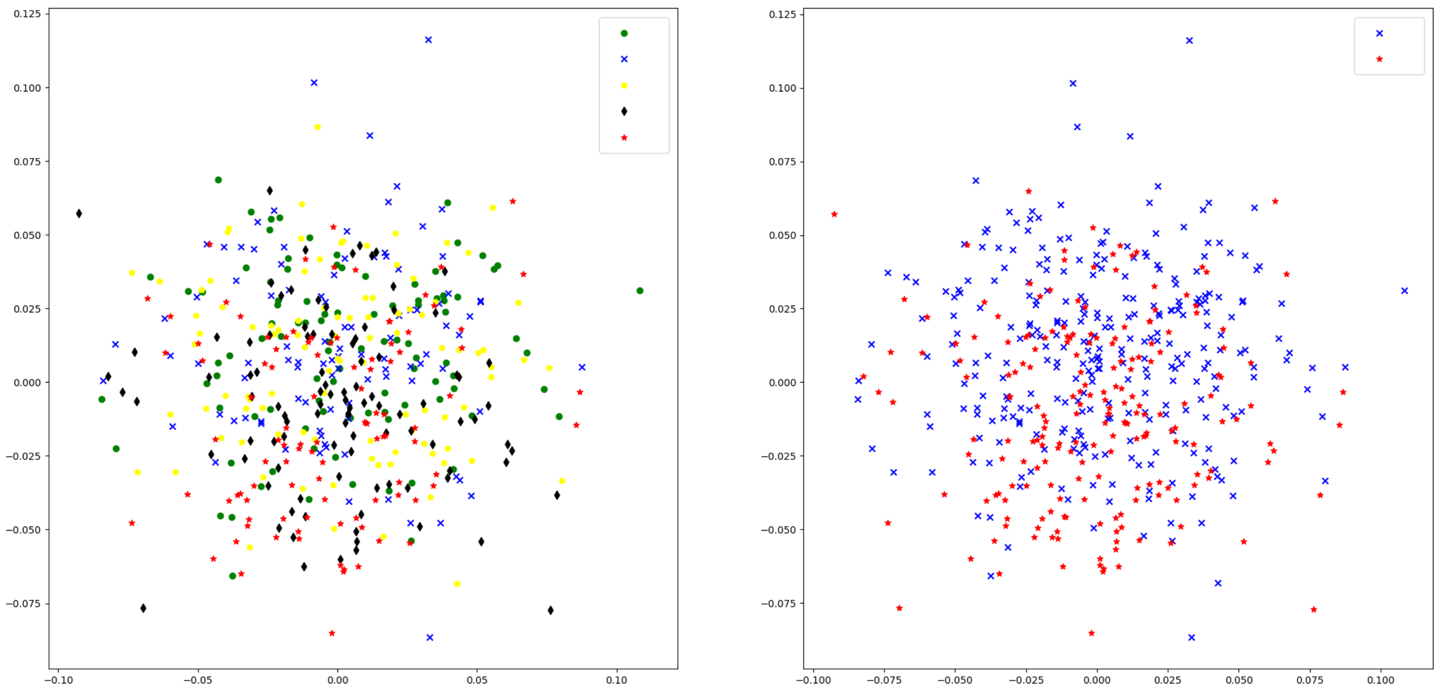


Figure A.7: *Left picture shows five classes before the permutation test and SenseMerge and right picture after. Green is* $\sim N([0, 0.015]^T, 10^{-3}I)$, *blue is* $\sim N([0, 0.015]^T, 10^{-3}I)$, *yellow is* $\sim N(0, 10^{-3}I)$, *black is* $\sim N([0, -0.015]^T, 10^{-3}I)$ *and red is* $\sim N([0, -0.015]^T, 10^{-3}I)$.

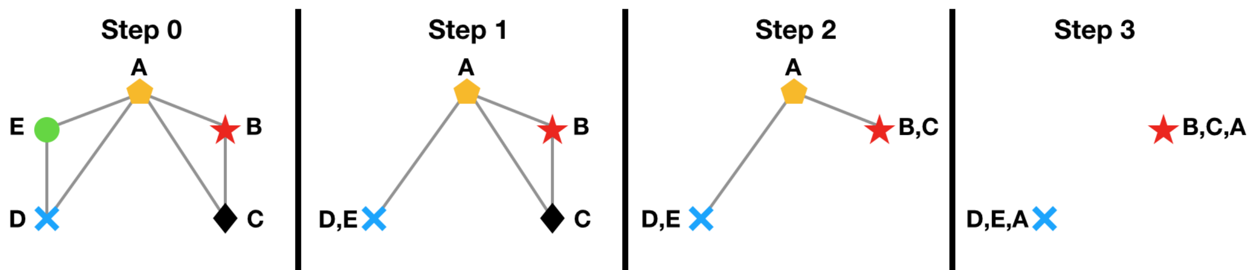


Figure A.8: *Ilustrating the steps in the SenseMerge algorithm for the scenario in the figure above.*

A.4 Example Usages of the Final Program

```
1  >>> import predictors
2  >>>
3  >>> # Example 1
4  >>> text = "Slim fit suit jacket from Filippa K made from 100% wool.
   ↪ Featuring four pockets, two rear vents, buttoned cuffs , corozo
   ↪ buttons and front buttoning closure. Single breasted and fully
   ↪ lined. Slim tailored fit. Combine this Rick suit jacket with Liam
   ↪ suit trousers for a complete suit."
5  >>>
6  >>> (senses, text) = predictors.disambiguate(text)
7  >>> for key, val in senses.items():
8  ...     print('{} {}'.format(key, val))
9  ...
10 index 0:  ('slim', 'slim', 'slim', 'ajustado')
11 index 3:  ('jacket', 'kavaj', 'blazer', 'americano')
12 index 10:  ('wool', 'ull', 'laine', 'lana')
13 index 17:  ('button', 'knapp', 'bouton', 'botón')
14 index 18:  ('cuff', 'slut', 'manche', 'puño')
15 index 21:  ('button', 'knapp', 'bouton', 'botón')
16 index 23:  ('front', 'fram', 'devant', 'frontal')
17 index 31:  ('slim', 'slim', 'slim', 'ajustado')
18 index 38:  ('jacket', 'jacka', 'veste', 'chaqueta')
19 >>>
20 >>>
21 >>> # Example 2
22 >>> text = "A perfect layering piece, this simple top is made in a
   ↪ soft technical jersey fabric. It has adjustable straps, and looks
   ↪ elegant with a pair of tailored trousers or worn underneath a
   ↪ smart blazer. The fabric is knit in Portugal."
23 >>>
24 >>> (senses, text) = predictors.disambiguate(text)
25 >>> for key, val in senses.items():
26 ...     print('{} {}'.format(key, val))
27 ...
28 index 1:  ('perfect', 'perfekt', 'parfait', 'perfecto')
29 index 6:  ('top', 'topp', 'top', 'top')
30 index 11:  ('soft', 'mjuk', 'doux', 'suave')
31 index 13:  ('jersey', 'trikå', 'jersey', 'punto')
32 index 14:  ('fabric', 'tyg', 'tissu', 'tejido')
33 index 18:  ('strap', 'band', 'bretelle', 'tirante')
34 index 35:  ('fabric', 'tyg', 'tissu', 'tejido')
35 >>>
```