Sigrid Lofthus Bjørndal

Exploring pretrained word embeddings for multi-class text classification in Norwegian

Master's thesis in Datateknologi
Supervisor: Jon Atle Gulla
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

DNB

Sigrid Lofthus Bjørndal

Exploring pretrained word embeddings for multi-class text classification in Norwegian

Master's thesis in Datateknologi
Supervisor: Jon Atle Gulla
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

Several pretrained word embeddings have recently been published, and recent development in the field of NLP is the use of contextualized word embeddings. This thesis explores the use of a pretrained, multilingual version of the BERT model, as well as pretrained word2vec embeddings for Norwegian. The BERT embeddings are combined with a simple feed-forward neural network (FFNN), and the word2vec embeddings with both an FFNN and an LSTM model. Naïve Bayes is used as a baseline model.

The task on which they are evaluated is hierarchical text classification of short Norwegian texts, specifically messages from the customer support chat of a Nordic bank. Additionally, the multilingual aspect of BERT is tested by training an FFNN model on exclusively Norwegian data, and subsequently testing the model on similar English and Finnish texts.

The main findings are that the BERT embeddings performs slightly better than the word2vec embeddings for the task, and the performance of the latter is highly dependent on model choice and dimensionality of the embeddings. BERT was also able to correctly classify some English examples, but made close to none correct predictions on Finnish examples.

Sammendrag

Det har nylig blitt publisert flere ferdigtrente word embeddings, og en ny utvikling innen fagfeltet er bruken av kontesktualiserte word embeddings. Denne oppgaven utforsker en ferdigtrent, flerspråklig versjon av BERT-modellen, i tillegg til ferdigtrente word2vec embeddings for norsk. BERT embeddings blir kombinert med et feed-forward neuralt nett (FFNN), og word2vec embeddings blir kombinert med en FFNN-modell og en LSTM-modell. Naïve Bayes brukes som en baseline-modell.

Oppgaven som embeddingene vurderes på er hierarkisk tekstklassifisering av korte norske tekster, som består av meldinger fra kundestøtte-chatten til en bank. I tillegg testes det flerspråklige aspektet av BERT ved at en FFNN-modell trenes utelukkende på norske data, og deretter testes på tilsvarende tekster på engelsk og finsk.

De viktigste resultatene er at BERT embeddings er litt bedre for denne oppgaven enn word2vec embeddings, og ytelsen til sistnevnte avhenger av modellvalg og embeddingenes dimensjonalitet. BERT har noen svake overføringsevner i den flerspråklige testen når den testes på engelske data, men nesten ingen når den testes på finske.

Preface

This thesis was written as a master thesis at the Norwegian University of Technology and Science (NTNU), Department of Computer Science (IDI). The thesis was a part of a collaboration project between NTNU and the Norwegian bank DNB.

I would like to thank my supervisor Professor Jon Atle Gulla for facilitating the collaboration and for supervising my thesis. I would also like to thank my co-supervisors Özlem Özgöbek and Cristina Marco for their guidance and assistance.
Next, I would like to thank Aria Rahmati at DNB for the opportunity to take part in the collaboration project, and Jan Thomas Lerstein, Abhilash Nair and the team at DNB IT Emerging Technologies for their support and assistance. I would also like to thank Anne Katrine Teige at DNB for explaining and helping me understand the current chatbot solution of DNB.

I would finally like to thank my family and friends for proofreading my thesis, and for their support during the entire course of my studies.

Table of Contents

List of Tables

List of Figures

Abbreviations

Symbol	=	definition
ANN	=	Artificial Neural Network
BERT	=	Bidirectional Encoder Representations from Transformers
CBOW	=	continuous bag-of-words
DAG	=	Directed Acyclic Graph
FFNN	=	Feed-Forward Neural Net
GDPR	=	EU General Data Protection Regulation
HC	=	Hierarchical Classification
HTC	=	Hierarchical Text Classification
LCL	=	local classifier per level
LCN	=	local classifier per node
LCPN	=	local classifier per parent node
LSTM	=	Long Short-Term Memory
MLM	=	Masked Language Model
NLP	=	Natural Language Processing
OOV	=	Out Of Vocabulary

Chapter 1

Introduction

This chapter provides an introduction to important aspects of the thesis, namely the motivation, project goal and approach, as well as a short summary of the results.

1.1 Motivation

The development of the Internet has changed how companies interact with their customers, and banks are no exception. A prime example of this is the ongoing development of online customer support chat at DNB, Norway's largest bank[1]. Although it is still possible to contact their customer support by phone, many of these customer interactions are now done through chat on their website. A substantial amount of resources is needed to ensure relatively quick answering of these messages, but not every question asked by a customer is so complex or unusual that it needs a custom human response, which is where automation comes into play in the form of a chatbot. The implementation and operation is a cooperation between DNB and a third-party vendor, Boost.ai[2], but much of the chatbot's operation all intents and purposes a black-box AI, as the classification model it uses is a trade secret of the vendor. Currently 51% of interactions through the chat interface are automated.

The chatbot is already in production, but there are continous developments in the field of NLP that may be beneficial. there is still room for improvement, which is where recent developments in NLP may be beneficial. There are potential application benefits of improving the chatbot, which could potentially automate even more of the current chat interactions with human customer support agents, and could give the customer a quicker and more accurate response to their questions.
It is also of interest to see how new developments in NLP perform on quite domain-specific data in a relatively low-resource language. There have been some promising results lately

[1] https://www.dnb.no/en/about-us/about-the-group.html
[2] https://www.boost.ai/

on pre-trained models. However, much of the work is done on English, a high-resource language, although there has been some research done on other languages as well. Of particular interest is BERT, a new pre-trained model released for over 100 languages that has shown good results for some tasks[5, 4].

1.2 Project goal and research questions

The goal of this thesis is to explore how different pretrained embeddings for Norwegian perform in a text classification problem, where the examples are very domain-specific and there are a lot of classes.
The research topics of interest for this are

1. What are the most efficient ways of classifying text into DNB's class hierarchy?
2. How can word embeddings improve the classification?
3. How do different pretrained embeddings compare to each other?
4. To what extent are trained classification models using BERT embeddings transferable to other languages?

1.3 Approach

The goal of the experiments in this thesis is to classify text messages, called examples, into one of the many pre-defined classes. The main approach for this classification task is to predict all classes at once, without taking the provided class hierarchy into account, but one experiment also considers utilizing the hierarchy. Different classifications methods are used, namely Naïve Bayes, feed-forward neural net (FFNN), Long Short-Term Memory (LSTM) and a most-common default classifier. The neural network architectures, i.e. FFNN and LSTM, use embeddings of the examples from either pretrained BERT or pretrained Norwegian word2vec models. Each classification model is evaluated using 10-fold cross-validation. One experiment also examines the performance of a trained classification model using BERT embeddings when classifying data from another language than Norwegian.

These experiments are all performed on the training data from the chatbot, but another example trains the models on the training data and tests them on the data which the chatbot uses for testing itself. This data is quite different from the training data.
The final experiment uses the local approach to classification, which incorporates the class hierarchy in the classification, instead of the flat approach of the other experiments. The model used for this experiment is Naïve Bayes.

1.4 Results

The results of the cross-validation is that the FFNN model using BERT embeddings has the best performance for classifying the Norwegian text messages, closely followed by LSTM

using word2vec embeddings. The baseline of a most-common predictor has terrible results with close to 0% accuracy, but Naïve Bayes, the other baseline model, shows surprisingly good results despite the model's simplicity. FFNN with word2vec embeddings has better results than the baseline models, but is worse than the two top models.

The BERT embeddings combined with an FFNN model managed to correctly classify a few English examples when trained on exclusively Norwegian texts, but made close to none correct predictions on Finnish examples.

The relative differences in performance between models when tested on the chatbot's original test data were consistent with the results of cross-validation on the training data, but every model performed much worse on the former. However, there are several differences between the test data and the training data of the chatbot, which likely explains the different results.

The experiment with using the local approach to hierarchical classification, i.e. utilizing the hierarchy, showed a slight improvement in prediction quality for the Naïve Bayes classifier. However, a flat approach together with models using embeddings had much better results than both experiments with Naïve Bayes.

1.5 Thesis outline

Chapter 1 - Introduction The current chapter gives an introduction to the thesis, including its motivation, approach and a short summary of the results.

Chapter 2 - Background This chapter provides a theoretical introduction to the techniques used in the experiments, such as the different word embeddings and the different classification models used.

Chapter 3 - Related work An overview of published work related to hierarchical text classification and different forms of pretrained word embeddings is presented in this chapter.

Chapter 4 - Data A description of the data and its source is presented in this chapter, including the classes and their hierarchy as well as the text examples themselves. The differences between the chatbot's training and test data are also presented here.

Chapter 5 - Method This chapter presents the methods used, both in the implementation with regards to preprocessing the data and using the models, and evaluating the results.

Chapter 6 - Experiments This chapter shows the results from all experiments performed in this thesis, and a discussion and comparison of their results.

Chapter 7 - Conclusion The final chapter provides a conclusion of the discussion and a summary of the project, as well as suggestions for further work.

Chapter 2

Background

This chapter discusses the theory of various techniques used in the experiments and some general theory relating to the task itself. First an overview of document representation is presented, followed by how the specific representation of embeddings may be generated. Subsequently the theory of some problem characteristics and how they affect what approaches are available is discussed. Finally the classification models used and their theory are presented.

2.1 Document Representation

An essential part of any language-related task, or indeed any machine learning problem, is how the data is represented. The choice of representation, and how the raw data is transformed into that representation, can have a large impact on the final results, as well as which methods can be applied.

Some restrictions on the representation are imposed by which algorithms are chosen, with different algorithms requiring different types of input. Some algorithms, such as decision trees, allow almost any representation form, such as categorical data, continuous values or discrete values. Others, such as neural networks, require that the input is vectors of values, sometimes also requiring that the values be within a specific range such as $v_i \in [-1, 1]$. The choice of algorithms used in this thesis, namely Naïve Bayes and different forms of neural networks, means that different forms of vectors with values will be used, and the discussion below is therefore limited to this representation form.

Assume for the following discussion that the documents have already been transformed from raw text to a list of tokens, with one list per document and each token corresponding to a word. Please see section 5.3 for details of this process.

2.1.1 Bag-of-words

Document representation using bag-of-words is a way of looking at and treating a document, and as the name suggests disregards the position of the words and treats them as a ”bag” [11, p. 65]. The document is represented as simply an unordered set of words, along with the frequency of each word in the original document.
Some model types, such as decision trees, might be able to handle data in this form, but most require it to be in the form of a vector of values. If the vocabulary is known and of size $|V|$, the document may be represented as a sparse vector of length $|V|$, where each position represents a word in the vocabulary and each word in the document is represented in the vector with its frequency. A vocabulary is in this context the set of all words that are present in any document in the dataset, and a sparse vector is a vector where most of the entries are zero. This way of representing the documents may for instance be used together with Naïve Bayes, which is described below in section 2.4.1.

If the document for example consisted of the sentence *Information about mortgages*, the bag-of-words approach transforms it into $\{about : 1, Information : 1, mortgages : 1\}$. This may be vectorized into $[1, 0, 0, 1, 1]$ if the vocabulary consisted of the words $\{about, how, I, Information, mortgages\}$. This is a toy example, as a vocabulary is typically much larger.

2.1.2 One-hot encoding

One-hot encoding is arguably the simplest type of document representation, as it represents each word as a vector of length $|V|$, where the dimension corresponding to the word index is the only element set to 1, the rest of the elements of the vector being 0 [11, p.147]. Word index is in this case the index of the word in the vocabulary, not its index in the document. The representation of the entire document, using this method, is thus a sparse matrix of dimensions $|d| \times |V|$, with $|d|$ being the number of words in the document and $|V|$ the size of the vocabulary. This representation form is quite useful for despite its simplicity, such as used for input to neural networks, for example those generating embeddings.

One of the disadvantages of one-hot encoding, which embeddings attempts to alleviate, is that each word vector is orthogonal to every other word vector in the vector space made up of these vectors.This means that using a metric to measure similarity, such as the cosines similarity $cos(\vec{x}, \vec{y})$, any word is equally similar, or rather dissimilar as $\forall \vec{x}, \vec{y} : cos(\vec{x}, \vec{y}) = 0$, to every other word in the vocabulary with respect to their representation. This is regardless of the actual meanings of the words, for example the word *cat* is equally similar to the word *dog* as it is to the word *anomaly*. Additionally, the document matrix might become very large if the vocabulary is large.

The example from the previous section, i.e. the document $Informationaboutmortgages$, would be transformed into the encoding

$$\begin{vmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{vmatrix}$$

when using the same vocabulary as before.

2.1.3 Embeddings

Both bag-of-words and one-hot encoding treats the words of a document as symbols, without the meaning of the words included in the representation itself. Additionally, the resulting vectors are both sparse and high-dimensional, in particular when the vocabulary size $|V|$ grows large. Word embeddings, also called dense word vectors, on the other hand, represent each word as a dense vector in a n-dimensional space[7], where typically $n \ll |V|$. The shorter vectors offer a computational advantage, and might also be better at generalization[7].

There are different models that may be used to achieve this, two of which are described below, but the general principle is that words that are similar in meaning should have similar representations. To continue the example above, this entails that the words *cat* and *dog* should have embeddings that are more similar than the words *cat* and *anomaly*. It is also possible to have a single embedding that represents a sentence or even a document [12], but this thesis will focus mainly on word embeddings.

Representing the document in the example above using word embeddings would result in one vector of values per word, that could be aggregated in some way or used as-is. An example of a representation of the words is the matrix

$$\begin{vmatrix} 0.1836 & 0.7487 & 0.1364 & -0.9831 \\ 0.9385 & 0.3452 & -0.3465 & -0.1275 \\ 0.7435 & -0.1627 & -0.4675 & 0.5268 \end{vmatrix}$$

or $[0.6219, 0.3104, -0.6776, -0.5838]$ if aggregated using the mean when using toy embeddings of length 4. The shortest embeddings used in this thesis is of length 100.

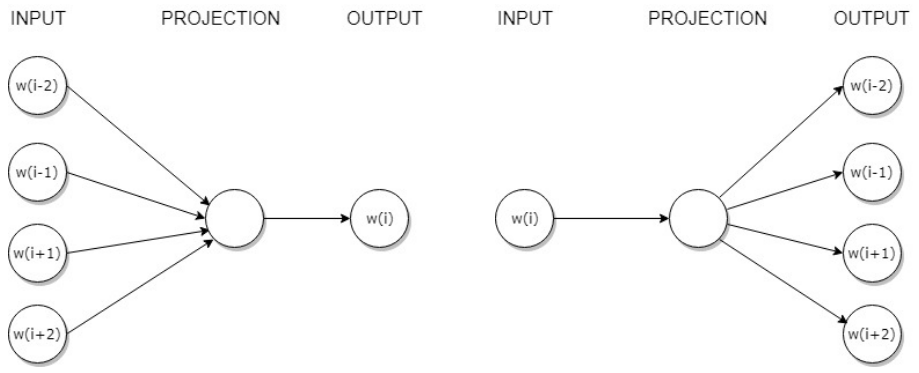
2.2 Generating Word Embeddings

As discussed briefly above in section 2.1.3, embeddings attempt to represent the meaning of the words rather than simply the words themselves. The meaning in question should be interpreted not as an absolute meaning of the word, but rather its relative meaning in relation to the other words in the vocabulary.

The famous linguist John R. Firth wrote

> ”You shall know a word by the company it keeps.”[6],

and both models discussed below embody this thought, as they use the surrounding words of the word in question to generate embeddings for it. While it is certainly possible to train these models from scratch, they may also be used as pretrained models. The approach of using pretrained models is equivalent to the principle of using transfer learning, i.e. training a model on general data and fine-tuning it afterwards on task specific data. This



(a) CBOW **(b)** Skip-gram

Figure 2.1: Word2vec architecture used for training vector representation of words. The examples here correspond to a window size of 2.

has been used with great success for image recognition [20], and has gained traction for NLP as well [5, 9, 17].
A basic introduction to artificial neural networks (ANNs) is presented in section 2.4.2, as both word2vec and BERT belong to this algorithm type.

2.2.1 Word2vec

Word2vec is two very similar models proposed by Mikolov et al. [13], used for computing vector representations of words. They require very large collections of texts to train, but as word2vec models uses unsupervised learning when training the data sets do not have to be annotated. Large text collections such as Wikipedia articles or digitized book collections can therefore be used without human annotation. The original paper uses a data set of 1.6 billion English words to train the models, but nevertheless reported a training time of only less than a day for some configurations.

The two versions of word2vec, called skip-gram and continuous bag-of-words (CBOW), both use a shallow neural network architecture for learning the vector representations of the words in the dataset, i.e. the embeddings. Skip-gram is trained by taking a word w_0 from a text as input, and trying to predict the n next and the n previous words. n is called the window size, and is typically small. CBOW, on the other hand, takes the words in the window as input and tries to predict the word w_0 from those. The words are one-hot encoded, a representation form described in section 2.1.2, i.e. as sparse vectors of length $|V|$, where $|V|$ is the size of the vocabulary. The two architectures are shown in figure 2.1.

The simplicity of the architecture means that by training the model on predicting these words, the projection layer is forced to find a good representation for the words in the training data. The layer is represented by a weight matrix W of dimensionality $|V| \times D$, where $|V|$ is the size of the vocabulary as before, and D is the desired vector dimension-

ality of the embeddings. When the architecture has been trained, this matrix becomes essentially an embedding lookup table, because the one-hot encoding representation of the input means that multiplying the representation of a word in the vocabulary, $\vec{w}$, with the weight matrix W gives the embedding for that word. Notice how this entails that any word not found in the vocabulary of the training data, typically called an out-of-vocabulary word (OOV), cannot get an embedding from this model. In applications this is handled by either ignoring the word, or assigning it the zero-vector $\vec{0}$. It should be noted here that the embedding for a specific word is the same regardless of context after training has been finished, in contrast to *contextualized* word embeddings such as BERT.

It has been shown, by [13] and others, that simple algebraic operations can be performed on the embeddings to answer the question "What is the word that is similar to word z in the same sense as word x is similar to word y?". For example, it is possible to use addition and subtraction to perform the following reasoning:

$$vector(\text{cat}) - vector(\text{feline}) + vector(\text{canine}) = vector(\text{dog}). \tag{2.1}$$

As semantic relationships between words are captured by the embeddings, they are used in many different NLP applications, such as machine translation and information retrieval, and their dense, numerical nature make them ideal for use with algorithms such as artificial neutral networks.

2.2.2 BERT

BERT is an abbreviation for Bidirectional Encoder Representations from Transformers, and is a recent language representation model proposed by Devlin et al. [5] that builds upon the Transformer architecture proposed by Vaswani et al. [24].

The Transformer architecture

The original Transformer architecture is a form of artificial neural network built upon an encoder-decoder structure. In such structures the encoder maps the input data to a different representation, typically some form of continuous representation, and the decoder converts these representations to output of the same form as the input. The model is autoregressive, which means that it consumes the output from the previous step as input, but it should be noted that it is trained exclusively left-to-right, i.e. uni-directionally.
The model architecture of the Transformer is shown in figure 2.2. There might be several encoder and decoder blocks in the model, the original paper uses 6 identical layers of each.

An important aspect that distinguish the Transformer from recurrent architectures, which are discussed in section 2.4.2, is the use of attention. Recurrent architectures look at just the previous state when predicting the output, which means that the model needs to encode both long-distance and short-distance dependencies into its state, represented by one or more fixed-length vectors. Attention mechanisms, on the other hand, looks at the entire input sequence at once, and "attends" to different parts of the sequence. What to

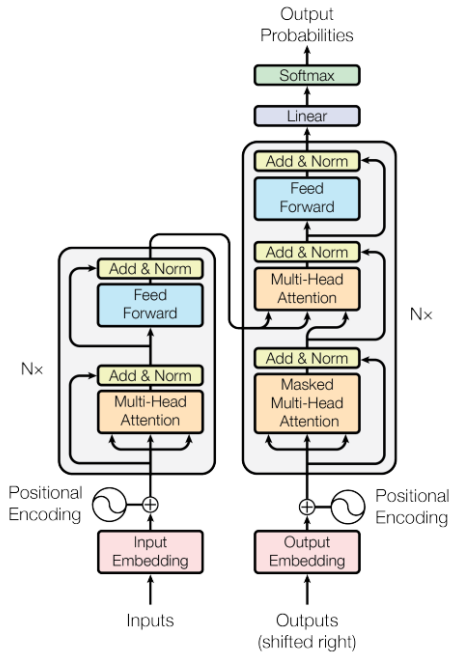


Figure 2.2: The Transformer architecture from the original paper [24]. The right half of the model is the encoder and the left half is the decoder. Image source: [24]

attend to is one of the things the model learns during training. Where the output used to depend on the last state, it now depends on a weighted combination of all input states, and the weights can be learned just like all other weights in the model.

BERT's architecture

The model architecture of BERT consists of several layers of Transformer encoders, and is bi-directional, in contrast to the original Transformer model that is uni-directional. There exists models that are bi-directional by combining independently trained left-to-right and right-to-left uni-directional models, such as ELMo [16] that uses bi-directional LSTMs, but BERT is truly bi-directional in all layers. A condensed figure of BERT's architecture is shown in figure 2.3, but the actual model is much larger.

BERT requires the input to the model to be in a very specific format, which requires that text be tokenized using WordPiece [25], as well as using embeddings for segment and position. The input representation is shown in figure 2.4. Segment embeddings are embeddings that allow for a pair of sentences to be given as input in a single sequence, while position embeddings denote the position of the word or token in the sequence. WordPiece tokenization start with a pre-determined vocabulary of tokens, and every word in the text

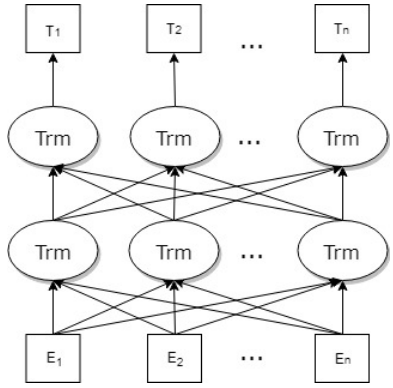


Figure 2.3: BERT uses a bidirectional Transformer encoder, with more layers than are shown here. Trm denotes Transformer blocks, E_i are input embeddings and T_i is the output of the model.

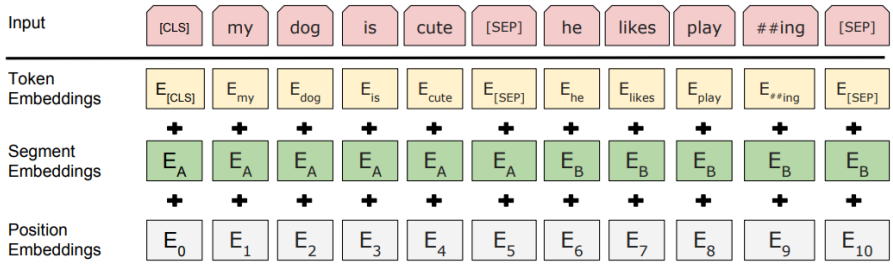


Figure 2.4: The input representation used by BERT. The segment embeddings are just E_A if only one sentence is passed to the model. Image source: [5]

is either matched with a single existing token, or split into parts until it does. Split word pieces are denoted with ##. There are thus no out-of-vocabulary words, an advantage if a language has many compound words, such as German and Norwegian. The resulting token sequence is also expanded with two special tokens, $[CLS]$ and $[SEP]$, that are used for classification and separating sentences respectively.

Training and generating embeddings

BERT is trained using two different tasks and a lot of data, as there are many weights in the model that need to be trained. As it is bi-directional it cannot use the traditional training task of predicting the next word in a sequence, because it uses the words both proceeding and following a word in its operation. The first task is therefore a modification of the traditional approach, called masked LM (MLM). 15% of the WordPiece tokens are chosen at random in this task, and most of them are replaced in the sequence by a $[MASK]$-token, and the model tries to predict the original word given the entire sequence. As this special token is never seen outside of this particular task, only 80% of the words chosen are re-

placed. Of the rest, half are replaced by a random word the other half is not replaced by anything at all, but allowed to retain the original token. As only 15% of the tokens are predicted instead of all of them in the traditional training task, the model may converge slightly slower, but the authors assert that MLM is empirically better.

The second training task is called Next Sequence Prediction, and consists of trying to predict whether sentence B is the sentence that follows sentence A in the dataset, i.e. *IsNext* or *NotNext*. This forces the model to look at the relationship between sentences, not only between words. Half of the examples provided to the model have the *IsNext* relationship, and the other half are sentences chosen at random. The decision of which relationship a specific pair of sentences have is taken by using the special $[CLS]$-tag.

As previously mentioned, word2vec generate word embeddings that can be viewed as a word vector lookup table after training, which makes the embedding independent of the current context of the word. BERT, on the other hand, generates *contextualized* word embeddings, which is dependent on not only the historical context of each word, but also its current one. This is particularly useful with words that may have very different meanings depending on context, for example the word *bark*, which might mean the sound of a dog or the outermost layer of a tree. A disadvantage however, is that it takes much longer to generate embeddings for an example by passing it through a large BERT model versus simply using a lookup table of word2vec embeddings.

Using BERT

There are two ways of using BERT, namely to either use it to generate embeddings or directly for classification, but in either case the entire sequence is passed as input at once. As discussed above BERT is trained on a binary classification task as well as a word prediction task, and the first task of the two used a special tag $[CLS]$ for classification. When using BERT it is possible to build a shallow neural network on top of the BERT model, and use this tag for classification.

There are many alternative ways of extracting embeddings from BERT. There are several layers in the architecture, and the output of any layer for any part of the input may be taken as an embedding, and used in some other model. The embedding for a single token from a single layer is a vector with dimensionality equal to the size H of the layer, and so the embedding for the entire sequence from that layer is a matrix of dimensionality $|d| \times H$, where $|d|$ is the length of the sequence. It is possible to take embeddings from different layers and combine them in some way, but this thesis only uses embeddings from the last layer.

2.3 Characteristics of classification tasks

The choice of document representation is certainly one of the more important choices that need to be made in the process of designing a classification solution, but of equal importance is the choice of algorithm for the classification itself. There are a lot of different

algorithms of varying complexity within the field of Machine Learning as a whole, however the choice is narrowed down by three central characteristics of the problem at hand.

The first is the question of whether the task is considered as unsupervised or supervised learning. *Supervised learning* is when the algorithm learns from examples where each example has both features and labels, and the goal of the task is to predict the labels based on the features [8, p. 101-104]. Depending on the problem there might be one or several labels per example, and the labels might have continuous or discrete values. *Unsupervised learning* also learns from examples that consists of features, but the difference is that no labels are provided, and the task is instead to learn useful characteristics of the data [8, p. 102]. There is also a third and more unusual option called *semisupervised learning*, where some examples include labels and some do not.
This thesis concerns itself almost exclusively with supervised learning, although it should be noted that training embeddings is a somewhat special case, as the algorithms used for this generate the labels themselves rather than being provided them.

Predicting the classes of input data, which is the main problem of this thesis, is supervised learning, which leads to the second characteristic that limits model choice: if it is a *classification* or *regression* problem [19, p. 696]. Both are types of supervised learning, and the distinction is based by what type of values the labels have. A task is regarded as classification if the goal is to predict what class an example belongs to, where the number of classes are finite. Regression, on the other hand, seeks to predict a numerical value.
This project is exclusively a classification problem, as the label to be predicted has a finite, albeit large, set of potential classes.

A third characteristic of the problem, following from the fact that it is a classification task, is precisely what type of classification task it is. There are three major types: binary, multi-class and multi-label classification [11, p. 55]. Binary and multi-class refer to a task where each example belongs to a single class, where the number of potential classes is two in the case of binary classification, and more than two in the case of multi-class classification. Multi-label, on the other hand, allows for an example to belong to multiple classes at once.
The precise nature of the classification problem described in this thesis is surprisingly hard to determine, as it depends on how one takes the class hierarchy into account. This is described in further detail below in section 2.3, but the approach used here is that of a multi-class classification task.

Hierarchical Classification

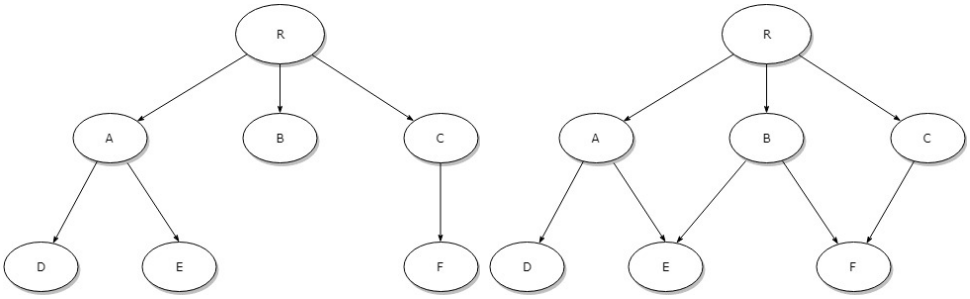
The classes of the data used in this thesis are organized in a hierarchy, described in section 4.3, and so an introduction to hierarchical classification (HC) is presented here. It should be noted that most experiments in this thesis ignore the hierarchy, which is a valid approach in hierarchical classification, but the hierarchy is nevertheless an inseparable part of the relationship between the classes.

There are three characteristics of a task of this classification type that need to be taken

into consideration: the type of hierarchy, the objective of the classification, and how the hierarchy is used by the approach [23].

Types of Hierarchies

There are typically only two options for the type of class hierarchy, namely either a tree or a directed acyclic graph (DAG), although the two types are quite similar [21]. Examples of both are shown in figure 2.5.



(a) Example of a tree hierarchy. **(b)** Example of a DAG hierarchy.

Figure 2.5: The most common types of hierarchies are trees and directed acyclic graphs (DAGs). Note that the root node R is typically implicit in applications, as no example is usually only a member of the root node.

The main different between a tree and a DAG hierarchy is that a node in a tree may only have a single parent, while a node may have one or more parents in a DAG. Most current literature focus on tree hierarchies, as it is typically a simpler problem, and this thesis is no exception. The hierarchy may be either provided at the outset or derived in some other way, such as through clustering of the classes, or through a combination of the two possibilities. The hierarchy in this project is a tree that is exclusively designed by humans, and described in section 4.3.

Classification Objective

The same types of classification are present here as in non-hierarchical problems, with the exception of binary classification, as that would be a rather trivial hierarchy. To summarize the earlier discussion, multi-class classification is when an example may be classified as a single class, but the number of potential classes are larger than two. Multi-label classification, on the other hand, allow for an example to be classified as multiple classes at once. One may in fact consider any hierarchical classification to be multi-label, as any example classified as a member of a non-root node may be considered a member of all the parents of that node as well. Therefore, in the case of hierarchical classification, multi-label is typically defined as a problem where an example may be classified as a member of more than one of the classes at the same level of the hierarchy, and the problem is considered of

the multi-class type if that is not the case [21].

Another aspect of the problem that needs to be taken into consideration, which is unique to HC, is whether the examples must be classified to a leaf-node, or if the classifier may terminate on a non-leaf node. The abbreviations used in the literature is typically MLNP for mandatory leaf-node prediction, and NMLNP for non-mandatory leaf-node prediction. In the case of the example hierarchy shown in figure 2.5a, the former category would only predict membership of the classes $\{D, E, B, F\}$, i.e. the leaf nodes, while the latter would have all nodes in the hierarchy as potential classes for the classification. It is possible to transform an NMLNP problem into a problem of type MLNP by adding so-called virtual nodes as children of all non-leaf nodes, with only examples that terminate at the virtual node's parent being classified as terminating at this new node.

The classification objective in this thesis is multi-class when using the definition described above, and allows for the classification to terminate at any node, regardless of its status as leaf- or non-leaf-node, i.e. NMLNP.

Utilizing the Hierarchy

There are three main approaches for using the hierarchy, characterized by how the hierarchy is utilized, if at all, during the classification. The three categories are called flat, global and local approaches, and the category of local has additional subcategories. This thesis mainly uses the flat approach, but briefly experiments with a local approach, namely using a local classifier per parent node.

Flat classification approach is the simplest of the three, in that it does not consider the hierarchy at all, but rather simply treats the problem as if it was a traditional multi-class or multi-label classification problem. This means that it considers all classes at once, and the number of classes might be very large which makes the problem rather challenging. The flat approach typically works better when the classification is only allowed to terminate at leaf-nodes, which makes the problem purely multi-class, rather than having to distinguish between non-terminating and terminating classification of a non-leaf class node. In the latter case it is possible to treat the problem as either multi-class or multi-label.
The advantage of using the flat classification approach is that is comparatively fast at both training and classifying, as well as allowing traditional classification methods to be used as-is. However, as it ignores the hierarchy it also ignores the information present therein.

Global classification approach learns a single model just as the flat approach, but incorporates the hierarchy. It might therefore be considered as a compromise between the flat approach and the local approach, which is described below. There exists algorithms that modify traditional classification algorithms so that they are able to take the hierarchy into account, but as this thesis will only look at flat and local approaches they will not be discussed here, and the interested reader should instead consider [21] and the models therein.

Local classification approaches are different approaches that all have in common that

they use local information in some way, but differ in exactly how they achieve this. The standard ways are to either have a local classifier per node (LCN), a local classifier per parent node (LCPN) or a local classifier per level (LCL). The LCN approach is essentially a binary classifier at each node, that determines whether or not an example is a member of that node, while LCPN uses a multi-class classifier instead to determine which child of the parent node the example should be classified as. The final type of approach, local classifier per level, is the method closest to a flat approach of the three, as it uses a multi-class classifier per level of the tree, and considers the entire level at once. LCN and LCPN are both common approaches, while LCL is quite uncommon.

2.4 Classification Models

There are a lot of different classification algorithms available, each with different advantages and disadvantages, and they vary wildly in complexity. The algorithms used for classification during in this thesis are Naïve Bayes, feed-forward neural network and LSTM, and all of them are presented here.

2.4.1 Naïve Bayes

Naïve Bayes is a rather simple algorithm that makes some significant simplifications in its approach, which makes it rather suitable for use as a baseline model. As the name suggests it is a Bayesian classifier, which means that it is based on the idea of Bayesian inference which uses Bayes' rule to calculate probabilities. It is called naïve because of how it simplifies the relationship between the features [11, p.65-68].

The approach of Naïve Bayes is that of a probabilistic classifier, i.e. a classifier that calculates the likelihood of an example d belonging to class $c \in C$ based on its features X. As any non-trivial classification problem has more than one possible class, the example is classified as the class that has the highest probability given the features, i.e.

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(c|X). \tag{2.2}$$

The probability $P(c|X)$ is calculated by using Bayes' rule

$$P(c|X) = \frac{P(X|c)P(c)}{P(X)}, \tag{2.3}$$

and since the probability for an example $P(X)$ does not change for each class the expression in eq. 2.2 simplifies to

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \frac{P(X|c)P(c)}{P(X)} = \underset{c \in C}{\operatorname{argmax}} P(X|c)P(c). \tag{2.4}$$

The simplifying assumption made by Naïve Bayes is that all features $x \in X$ are *conditionally independent* of each other given the class, although it is quite unrealistic, especially

in natural languages. This assumption means that equation 2.4 further simplifies into the final equation of the classifier

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(c) \prod_{x \in X} P(x|c). \tag{2.5}$$

The probabilities are calculated from frequencies in the data through the equations

$$\hat{P}(c) = \frac{|\{d \in c\}|}{|D|}, \quad \hat{P}(x|c) = \frac{count(x,c)+1}{\sum_{x \in V}(count(x,c)+1)}, \tag{2.6}$$

where D is the set of all examples, and V contains all possible values of x, i.e. the vocabulary in the case of text classification. The equations here use add-one smoothing, also called Laplace smoothing, which is used to avoid zero-probabilities in the case that $count(x,c) = 0$, which otherwise would result in the probability for the class being 0.

In text classification the documents are represented using the bag-of-words approach described in section 2.1.1, which ignores the position of the words within the texts.

2.4.2 Artificial Neural Networks

Artificial neural networks (ANN) are a group of machine learning algorithms built upon the artificial neuron. There exists a lot of different variants, from the simple feed-forward neural network to the amazingly complex LSTM and BERT. The three mentioned variants are all used during the course of this thesis. ANNs are only capable of accepting vectors of numerical values, and so in the case of language and text the input has to first be transformed into this representation. For a discussion of representation forms see section 2.1.

The artificial neuron

The basic building block of ANNs is the artificial neuron, also known as the perceptron, which is inspired after biological neurons in the brain. A diagram of a perceptron is shown in figure 2.6.

An ANN is typically built up of several connected layers of artificial neurons, with different architectures depending on what type of ANN it is, but the neurons are typically the same regardless of type. It is provided by a vector of input $\vec{x}$, and the output of neuron j is described by

$$a_j = \sigma(z_j), \quad z_j = \sum_{i=0}^{n} w_{j,i} \cdot x_{j,i}. \tag{2.7}$$

The output a may be passed onward to other neurons, and equivalently the input $\vec{x}$ may come from other neurons. $\sigma(\cdot)$ is called the activation function, as it controls the "activation" or output of the neuron, which is comparable to how biological neurons only activate if their stimulus exceed a limit. The activation function is fixed in the initialization of

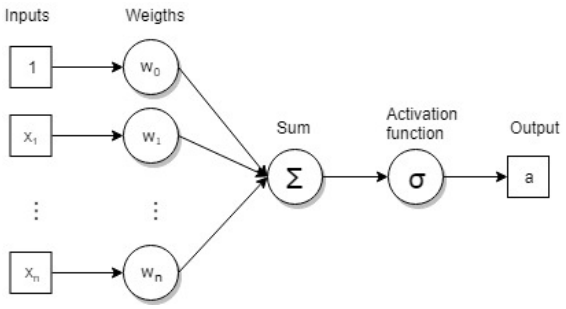


Figure 2.6: A schematic of an artificial neuron, also called a perceptron.

the ANN, but may be of different types for neurons in different layers. Learning is done by updating the weights w_i through a process called backpropagation, which is discussed further below.

Activation Functions

There are several different types of activation functions, although they have in common that they are typically derivable and monotonically non-decreasing. A very popular activation function is the rectified linear unit, which is defined by the function

$$\sigma(z) = max(0, z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases} \tag{2.8}$$

This function has the advantage that its derivate is equal to 1 if $x \geq 0$, and 0 otherwise.

Another activation function is the softmax function, which is typically used in the last layer when using the ANN for multi-class classification. This function is different than most other activation functions, as it depends on the all other neurons in the same layer as the neuron where it is applied. Its value at neuron k is

$$\sigma(\vec{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{H} e^{z_j}}, \tag{2.9}$$

where H is the total number of the neurons in the layer and z_j is defined in equation 2.7. An interesting property is that the sum of the output of all neurons in the layer sum to 1, which means that the output of a single neuron in the final layer can be viewed as the probability of the example belonging to the class corresponding to that node. The neuron with the highest value is therefore the likeliest class.

Backpropagation

The output for an example is calculated using an ANN by transforming the example to the input representation used, passing the resulting vector or vectors through the network, and taking the output. The value of the output depends on among other things the weights

of all the neurons, which are typically the only changeable parts of the architecture, and learning is done by changing these weights during training.

This process is called backpropagation, as the errors of wrongly classified examples, in the case of classification, or wrong values in the case of regression, are propagated backwards through the layers of neurons. The weights are changed accordingly to how much they affected the final result. As there are typically a lot of weights that affect the result each weight only has a very small impact on the result, and is therefore just changed slightly.
With the final output $\hat{y}$, where the correct output would have been y, the error is quantified by some loss function $L(y, \hat{y})$. How much each weight is updated, and if it should be increased or decreased, is determined by the gradient $\frac{\partial}{\partial w_{j,i}} L(y, \hat{y})$.

Training and regularization

ANNs are trained by changing the weights of the neurons through backpropagation of errors, as described above. More neurons in the architecture naturally leads to more weights that need to be trained, and therefore training larger architectures requires both more time and more examples. The entire dataset used for training is typically used several times, and each iteration where the dataset is used exactly once is called an epoch.

A problem when using ANNs, or indeed any machine learning algorithm, is how to avoid the problem known as *overfitting*. This refers to the fact that an algorithm might at some point become too good at predicting the training data, and in the process get worse at generalizing, i.e. predicting previously unseen examples. The opposite problem of *underfitting*, i.e. not being good enough on the training data, is remedied by training more, while overfitting may be remedied by training less. The balance between these two is therefore to some degree a Goldilocks problem with respect to how much training should be performed. However, there are a collection of techniques known as regularization that are intended to increase its generalization capabilities without at the same time increasing the errors it makes on the training set.
One of these techniques is called batching, and refers to strategy of not updating the weights after every example, but rather after a small group of examples. The errors from this group of examples are combined and used for training through backpropagation.

Another regularization method is called dropout, and affects the architecture of the ANN. The layers are usually fully connected, i.e. every neuron in a layer is connected to every neuron in the previous and proceeding layers, but when using dropout this is no longer true. This technique entails that some percentage of connections are removed when training, but not during testing, to make the model less dependent on each weight. As random connections are removed the model is forced to learn using more neurons, resulting in a reduction in the generalization error.

Recurrent neural networks

Recurrent neural networks (RNNs) are a comparatively complex subcategory of ANNs that are designed for dealing with data in the form of sequences, such as time series or text in the form of sequences of words. As the name suggests, these networks utilize a technique called recurrence, which attempts to capture dependencies across time or sequence position. A general overview of the structure of an RNN is shown in figure 2.7.

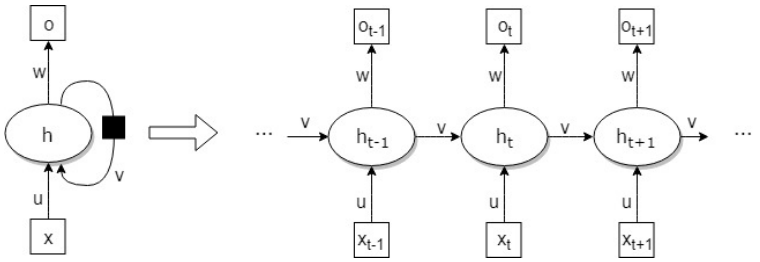


Figure 2.7: The basic structure of a neural network. The black square denotes a connection across a timestep.

The state of a recurrent network at time t depends on the state of the network at time $t-1$ as well as the input x_t, as shown in the figure. The weights, denoted w, u and v, are shared across the states. Note that the state is only dependent on the state at time $t-1$, and given that state is conditionally independent of all states before $t-1$. This means that any long-term dependencies needs to be encoded in the state, but also allows the system to be represented by just

$$h_t = f(h_{t-1}, x_t, \theta), \tag{2.10}$$

where θ is a set of parameters, instead of the cumbersome representation without recurrence as

$$h_t = g_t(x_t, x_{t-1}, ..., x_1). \tag{2.11}$$

Training an RNN is done through backpropagation, with the modification that the network is unrolled, like on the left half of figure 2.7, and since the output at every timestep has an effect on the loss they need to be taken into account. This is known as backpropagation through time. As the unrolled network of a standard RNN is in practice a very deep neural network if the sequence length is high, it suffers from the same problem of gradients tending to vanish in the lower layers of deep networks. Modifications on the standard RNN using gates, such as with LSTM described below, attempt to counteract this problem.

LSTM

The structure of an LSTM cell is shown in figure 2.8. Like other types of RNNs it uses recurrence to capture and generalize dependencies across time, and as the state only depends on the previous state in addition to the current input, any long-term dependencies

need to be encoded in the state in some way. This encoding of information acts in essence as a memory, and one of the features of an LSTM cell is that it can both forget information as well as add more to it.

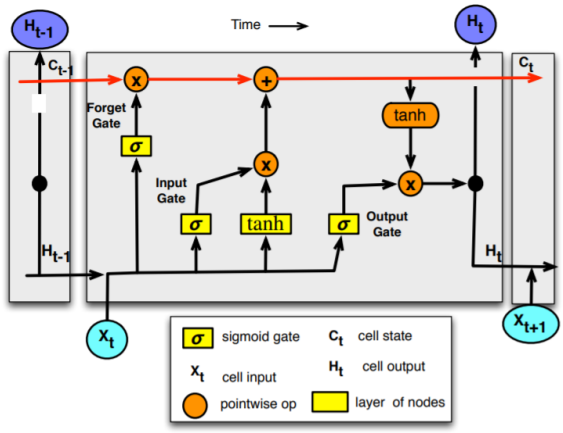


Figure 2.8: The structure of an LSTM cell at time t. Image source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-chain.png`

Information from the previous state is transferred in two ways, namely through the output at time $t-1$ as well as through the cell state c_{t-1}. The cell state is modified at each time step by passing the output from the previous step as well as the current timestep to a forget gate, one of the three gates of an LSTM cell, and this gate may remove some information. Current information is later added to it.
The other two gates are the input and output gates, which modify the information added to the cell state and what is passed along as output respectively.

The LSTM cell described above and shown in figure 2.8 can be used alone, but is typically used as part of a greater architecture. There may be several layers of LSTM cells and several cells in each level, as well as regular fully connected layers of standard neurons. Additionally, while the cell here is shown depending on the previous timestep and passing information along to the next, it is also possible to reverse the sequences and potentially combine both unreversed and reversed to make a bidirectional architecture. However, the standard configuration is uni-directional.

Chapter 3

Related Work

Both Natural Language Processing (NLP) in general and text classification in particular are areas where a lot of research is being done, and the volume of papers published is astounding. This chapter should therefore not be taken as an exhaustive guide to the field, but rather aims to provide an introduction to the fields of study that underpin this thesis.

3.1 Hierarchical text classification

Hierarchical text classification (HTC) is perhaps one of the less popular areas of research, but here has been some work done during the last years. The Large Scale HTC challenge [14] was organized between 2009 and 2014, and included several large-scale datasets that had class hierarchies, with some having a very large number of classes. Unfortunately, the data is provided in a quite heavily preprocessed form, with lemmatization and stop-word removal already performed and the documents are represented using bag-of-words. A total of 150 teams participated in the challenge, using a variety of different models, but an ensemble of multinomial Naïve Bayes models were among the top ten contenders.

[23] used corpora provided by Reuters, consisting of news articles with topic labels attached, where the topics are organized in a hierarchy. They tested different combinations of classification models and models for training and generating word embeddings, using both flat and local HTC approaches. Their results show that the local approach was better than the flat one, but the size of the difference was dependent on classification model. Additionally, they found that the embeddings from word2vec were slightly better than from GloVe.

[2] performed text classification on biomedical papers, using the skip-gram version of word2vec together with SVMs with linear kernels. They used different methods of aggregating the word embeddings for each word in a document into a single embedding, namely taking the maximum, minimum or average of the embeddings. They show that these aggregated representations perform better on shorter documents, and embeddings

with higher dimensionality consistently perform better than those with lower, regardless of document length. They also show that averaging has consistently the best results out of the three methods, but that concatenating the result of the three aggregating methods works better if the training dataset is sufficiently large. Although the classes are hierarchical they only use the flat approach.

3.2 Pretrained embedding models

There are many different models designed for training word embeddings, and several have been published quite recently. A very popular model of the comparatively older ones is word2vec, both in the skip-gram and CBOW version, which has a single, static word embedding for each word in the vocabulary after training has been finished[13]. Another model with the same type of word embeddings is GloVe [15], which uses an approach based on frequencies rather than the predictive approach of word2vec. Also notable is fastText [10], which is very similar to CBOW word2vec but also incorporates N-grams when training.

A new development in the field is the use of pretrained *contextualized* word embeddings. With this type of embedding a word has an embedding that is generated from itself and its current context, instead of being a static vector once training has finished as in the case of the previously mentioned models.
The break-through in the use of attention-based models for NLP came with the Transformer [24], and several other models build upon this work in some way. Although there are several different model architectures being explored at the moment, some of which are mentioned below, they all have in common that they use pre-training on more general datasets.

BERT [5] is one of models that utilizes the Transformer [24], and is the model used in this thesis for generating contextualized word embeddings. This model combines aspects of the Transformer architecture with bi-directionality in every layer, and has shown very good results on several benchmark tasks. Code and pre-trained versions of BERT has been published by the authors[1], among others two models in different sizes for English only, and a multilingual model that is pretrained for 100 languages, including Norwegian Bokmål and Nynorsk. The latter model is the particular version used in this thesis, and is equivalent in size to the smaller of the two English models. No larger version of the multilingual model has been published at the time of writing.
Two models recently published by OpenAI [17, 18] are also based on the Transformer architecture.

Another model that uses pre-training to train contextualized embeddings is ELMO [16], which uses bi-directional LSTMs instead of Transformers. Also of some interest is the ULMFiT model [9]. It does not provide pretrained word embeddings per se, nor does is it based on a Transformer model. However, it uses the same principle of pre-training as the

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

other models mentioned here to trained a language model based on an LSTM architecture, which is then later fine-tuned on target task data.

3.3 Pretrained embeddings for Norwegian

Norwegian is a rather low-resource language, especially compared to English, which is quite evident if one considers that Norway has around 5.3 million inhabitants[2], compared with English being used as a lingua franca by most of the world. It is therefore not unexpected that the amount of research published on NLP in general and word embeddings in particular that concerns itself mostly with English is much larger that the amount of research being done on Norwegian. However, there has been some recent developments of interest that are relevant for application in Norwegian.

As already mentioned, the authors of [5] have published code and pretrained models, and the multilingual version supports both Norwegian Bokmål and Nynorsk, the two versions of written Norwegian. However, no results of experiments with this model on Norwegian data has been published.

There has also been several pretrained word embedding models for purely Norwegian published recently [22], using different model types such as word2vec, fastText and GloVe[3]. The datasets used for pre-training are different combinations of three Norwegian corpora, consisting of news articles (NNC), downloaded web sites (NoWaC) and digitalized books (NBDigital) respectively. The evaluation of the word embeddings, which were tested on recognizing analogies and extracting synonyms, showed that no single model was best, but that fastText yielded best results on analogies and word2vec CBOW was best at synonym extraction.
[3] used these embeddings together with a convolutional network in order to perform sentiment analysis on Norwegian documents.

[2] `https://www.ssb.no/befolkning/statistikker/folkemengde`
[3] `http://vectors.nlpl.eu/repository/`

Chapter 4

Data

The data used in this thesis is all from the current chatbot solution used for customer support by the Norwegian bank DNB. The entire dataset, both the examples and the classes, where gathered from the chatbot solution in February and March of 2019. Work is continuously being done on the original data, so all characteristics of the dataset and ultimately the results of the experiments themselves may be different if completely up-to-date data is used to replicate the work of this thesis. No messages from real customers are included in the dataset, just examples that are constructed to be similar to such messages.

The first section in this chapter presents the chatbot solution and its origin. The second section provides a more in-depth presentation of the text messages that make up the dataset, while the third provides the same for the classes and their hierarchy. The fourth section presents the dataset used for testing by the chatbot, and how it differs from the training data. The final section provides a summary of this chapter.

4.1 Data source

The current chatbot solution in use, from which the data used in this thesis originate, is a product from the company Boost.ai[1], with modifications done by employees of DNB. The purpose of this chatbot is to act as a first-line responder for DNB's customer support chat, answering many of the common questions that are asked, and leaving the more complex or uncommon questions to human staff. The chatbot, called Aino, has automated 51% of the chat traffic on the bank's website[2]. It should be noted that the current state of the chatbot is a result of much work done by DNB, such as modification of the solution architecture and the data used for training and testing. The employees doing this work are known as AI trainers, and all have a background from customer facing roles in the company.

[1] https://www.boost.ai

[2] https://www.boost.ai/articles/2019/2/14/how-scandinavias-biggest-bank-automated-51-of-their-chat-traffic-with-ai

The product from Boost.ai is a package with everything needed to run a chatbot in production for customer support, such as data to train and test the chatbot, responses to different questions, a hierarchy for question types, and more. At the core of the chatbot is a model that classifies an incoming message as question type, called an *intent*, which has a predefined answer, or as a question it cannot answer. Templates for both the intents and the corresponding answers are a part of the provided package, and information is filled out by DNB's employees, who also add and remove intents as needed. The model itself is somewhat of a black box, as little information about it has been made public. However, it is known that is uses a synonym wordlist and performs spell-checking as part of the preprocessing.

Each intent has training and test data associated with it, which consists of examples of questions which should be answered with its corresponding answer. The chatbot is a single-turn system, which means it only considers the current message when deciding on a response, in contrast to systems that may be consider earlier messages in the conversation as well. The original language of the chatbot model is English, but the main language of the chatbot in production is Norwegian.

4.2 Text examples and their characteristics

The data itself consists of text messages, called examples, in different languages though mostly Norwegian, and they emulate questions from customers about different topics. Each example is already labeled with both class and language. It should be noted that the data does not contain any messages from real customers, as all examples have been constructed manually by either DNB's or Boost.ai's employees, so there are no applicable restrictions from GDPR or similar directives. An example is a single message, which typically consists of a single sentence.

There are a total of approximately $267,000$ examples in the dataset, and the language distribution is shown in figure 4.1. The main focus is on the Norwegian messages, which make up 72.5% of the dataset, i.e. approximately $193,000$ in total. Apart from Finnish and English messages, which are used in one experiment, there also exist a few messages in other languages such as Swedish and Danish.

Typical messages about for example information relating to mortgages may be

- Can you tell me about house loans?
- Can you give me information about mortgages?
- House loan
- Tell me about mortgages

The length of the messages varies a lot, and they do not necessarily consist of full, grammatically correct sentences. Some statistical measures for the length of the Norwegian examples in the dataset are shown in table 4.1. The length is the number of words in the

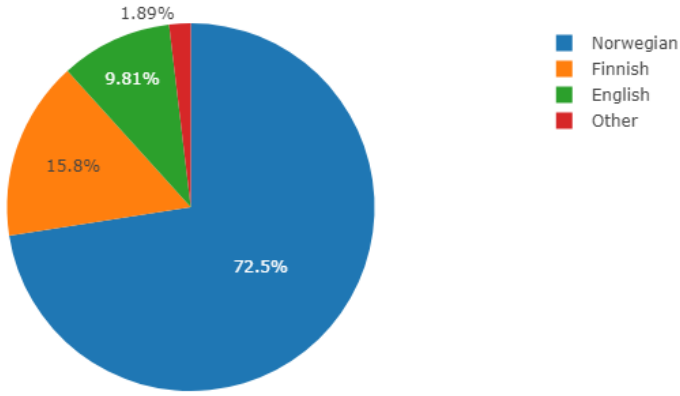


Figure 4.1: The language distribution of the messages. *Other* refers to languages which are not considered in this thesis, such as Swedish and Danish.

message, where a word is a token from the regular tokenization method described in section 5.3.1. As seen here a message may consist of just a single word, or up to over 50 word, but the average number of words is 9.6. Just 21% of the examples consist of more than 12 words, and only 1% consist of over 20 words.

Statistical measure	Value
Average	9.6
Standard deviation	4.1
Minimum	1
25th percentile	7
50th percentile	9
75th percentile	12
Maximum	51

Table 4.1: The length of the Norwegian messages in the dataset, measured in number of words.

The vocabulary of the dataset is not quite as varied as the message length. There are a total of 8790 distinct words in the Norwegian part of the dataset, although that includes different

grammatical forms of the same word, such as the past and present tense of verbs. There exists techniques for reducing a word to its base form, namely stemming and lemmatization, but they are not used in this thesis.
The frequencies of each word varies greatly, with the most common word being the Norwegian word for I, "jeg", with frequency $82,679$. The average frequency for a word is 210, but the standard deviation is $2,060$, which illustrates the great variance. Additionally, 65% of the words in the vocabulary have a frequency of less than 10, and half of these have a frequency of 1, which has a challenging implication that will be discussed in chapter 6.

4.3 Classes and their hierarchy

The classes are called *intents* by the current chatbot solution, but the subsequent discussion will use the term *class* instead to conform with standard terminology in the literature.

4.3.1 Classes

As mentioned briefly in section 4.1, each message from a customer is classified as a question type or intent, where all question types that can be found in the chatbot solution make up the set of classes C. There is no class for the question types that are not supported, so in the real world there will always exist question types that are not in C. However, all data used in this thesis belong to one of the known classes.

A question type has a quite narrow definition in this context. It refers to a generalized form of a question, which allows for different formulations of questions as long as they are approximately equal in meaning. For example, of the three questions

1. Can you tell me about house loans?
2. Can you give me information about house loans?
3. Is it possible to open a savings account?

only 1. and 2. ask for the same thing. They are therefore the same question type or intent, and should be classified as the same class, although the exact wording of the two is different. Question 3. asks about something different, and therefore belongs to a different class.

The classes are designed by humans, and each corresponds to an answer the chatbot may make, as mentioned in the first section of this chapter. In the case of question 1. and 2. above, the class may be *House loan*, and the response some general information about the house loans the bank provides. The class for the third question may be *Opening a savings account*, and the answer either information about this or a link to the website where the customer may open such an account.

There is a total of 1716 classes in the dataset used, which are organized in a hierarchy that is described below. All of these classes have training data in Norwegian, and some of them also have data in either English or Finnish, or both. The percentage of classes that

have data in the different languages is shown in figure 4.2. The main focus of the thesis will be on the Norwegian data, but the Finnish and English data will also be used briefly when evaluating the performance of BERT.

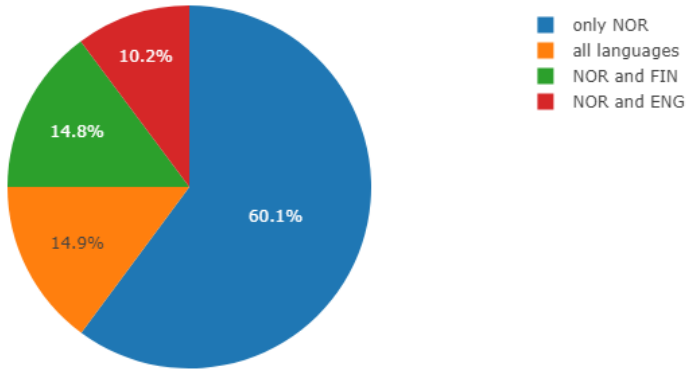


Figure 4.2: The percentage of the 1716 classes that have data in the different languages. NOR refers to Norwegian, FIN to Finnish and ENG to English.

4.3.2 Class hierarchy

There are a lot of classes in the dataset, but fortunately there is some additional structure to the classes in the form of a class hierarchy. This hierarchy is a tree, the theory of which is explained in section 2.3 and illustrated in figure 2.5a. The actual hierarchy is much larger than in that simple example, far too large to be shown in its entirety as there are over 1700 classes, so a textual description shall have to suffice.

There are only 22 classes at the top level of the hierarchy, with all other classes being descendants of these. Around 80% of the classes are leaf nodes, i.e. they have no descendants, so the hierarchy is very wide but quite shallow. In fact, the maximum depth of the hierarchy tree is 6, and the average level of the tree at which a class is located is 3.44. The number of classes located at each tree level is shown in figure 4.3.

The top-level classes mostly corresponds to product categories of the bank, such as insurance and loans, with some other more service-related categories such as general questions and questions regarding log-in. The different categories are not equally represented, as

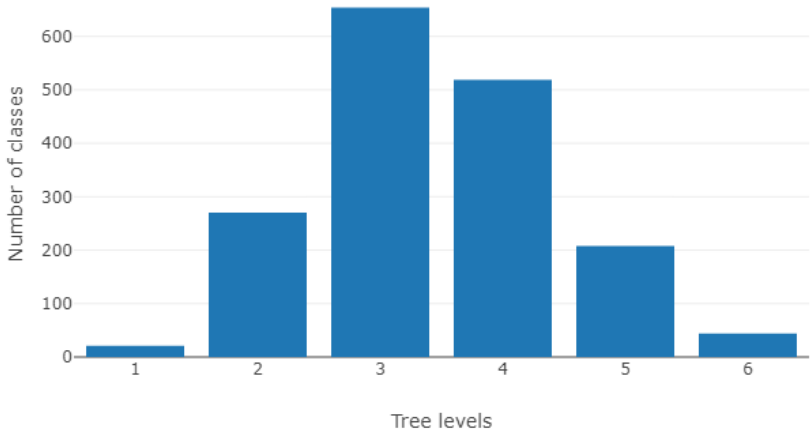


Figure 4.3: The number of classes which are located at each level in the class hierarchy tree.

shown by figure 4.4, which show the number of descendants per top-level class. Out of the 22 classes only 8 have over 50 descendants.

As mentioned above around 80% of the classes are leaf nodes as they have no descendants, but all classes have associated training data, and an example may be classified as any class in the hierarchy when testing. The classes represent more specific or detailed question types the further down in the hierarchy they are located, but as the customer is not restricted to only asking about the more specific topics, a text message may be classified as one of the more general categories although it is not a leaf node. This makes the classification objective of the type non-mandatory leaf-node prediction, the theory of which is described in section 2.3.

4.3.3 Class balance

There is a large variance in the size of the sub-trees beneath each top-level class in the hierarchy, as shown in figure 4.4 and discussed above, which means that some high-level topics are over-represented in the set of classes. However, that does not say much about the class balance in the dataset itself. Figure 4.5 shows the total number of examples that belong to either a top-level class or one of its descendant classes, along with the average number of examples per class in this category. The average for the whole dataset is 113 examples per class, but the figure shows how some categories have much higher or lower averages. Additionally, whether a category has a high or low average per class cannot

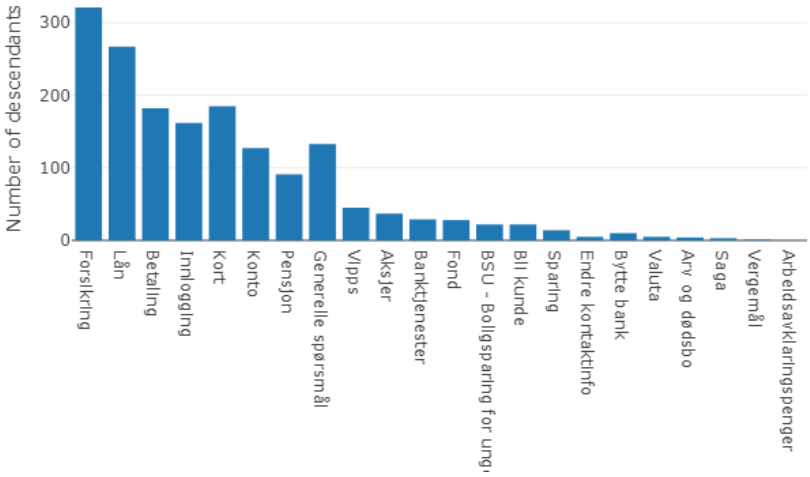


Figure 4.4: The number of descendant classes for each top level class, sorted according to the total number of examples in the category. The class names are from the original chatbot solution, and therefore in Norwegian. A translation of the names can be found in Appendix B.

necessarily be seen from the total number of examples of that category, or the number of classes for that category.

While it is evident from this that some categories of classes are more common than others, it is also of interest to look at the class balance irrespective of the class category. The total number of examples for each class is shown in figure 4.6. The example count is not quite equal for the different classes, in fact the maximum count is 682 and the minimum is 10. However, the most common class only makes up 0.4% of the dataset, so no single class can be said to dominate the dataset completely.

4.4 The chatbot's test data

The previous sections in this chapter have only discussed the data which the chatbot solution used for training. However, it also has a separate dataset which is used for testing the performance of the implemented chatbot, which consists of approximately $18,000$ examples. Note that this thesis mostly performs the experiments on the training data using 10-fold cross-validation, but the performance of classification on the test data is also reported.

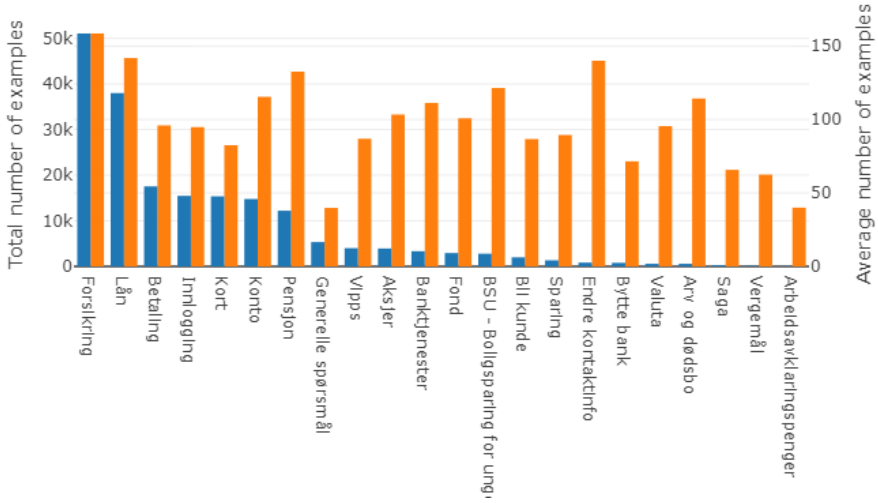


Figure 4.5: The total number of examples that are classified as either a top-level class or one of its descendants, as well as the average number of examples per class in that subtree.

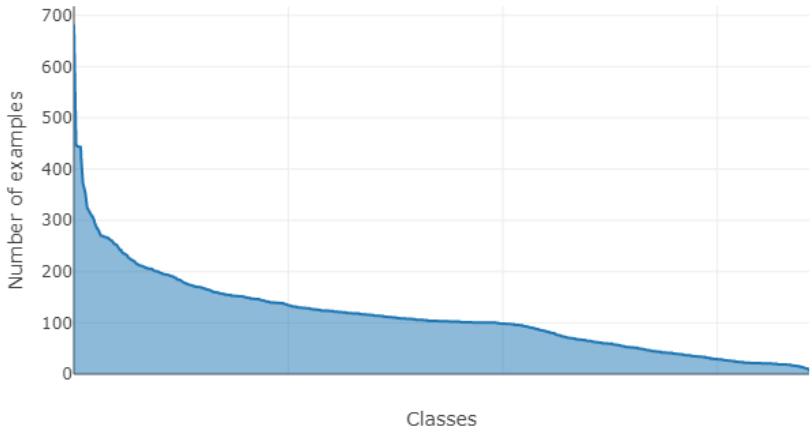


Figure 4.6: The total number of examples per class.

4.4.1 Differences in the examples

The training data, which this thesis focuses on, assumes ideal operating conditions, where the text examples are written completely in Norwegian without any spelling errors. The test examples however are not constructed under this assumption, and are potentially closer to representing the messages written by actual customers. They include both spelling errors and a few English, Swedish and Danish words. The vocabularies of the two datasets are therefore naturally not identical.

The difference between the two vocabularies lies both in their size and their content. The vocabulary of the training data V_{train} has 8788 words, while the vocabulary of the test data V_{test} has 4487 words. Note however that the number of training examples is much larger than the number of test examples, namely approximately $193,000$ Norwegian examples and $18,000$ examples respectively, so some difference is to be expected here.
Although V_{train} is nearly twice as large as V_{test}, the latter still contains a lot of words that are not present in the former. To be precise 22% of its words are not found in V_{train}. However, only 7% of the test examples contain a word not found in V_{train}. Some of those words are misspellings of Norwegian words, either unintentional or written as they are spoken, and some are words from other languages such as English or Swedish. Some of the words however are correctly spelled Norwegian words that are simply not found in the training data.

Another difference between the training and test examples are in their length, as measured by the number of tokens in each message. The training examples have an average length of 9.6, while the test messages are 8.5, a reduction in length of 11%. Other statistical measures of their length is shown in table 4.2.

Statistical measure	Training data	Test data
Average	9.6	8.5
Standard deviation	4.1	4.1
Minimum	1	1
25th percentile	7	5
50th percentile	9	8
75th percentile	12	11
Maximum	51	30

Table 4.2: The difference of the length of the Norwegian messages in the training and test datasets, measured in number of tokens.

4.4.2 Difference in class balance

There is some difference in the class balance between the training and test datasets, in the form of how many examples each class has. The classes and their hierarchy are identical for the two datasets, it is only the amount of data that differs. The class balance of the

training set is discussed in some detail in section 4.3.3.
In the experiments using training data the testing is performed using cross-validation, i.e.

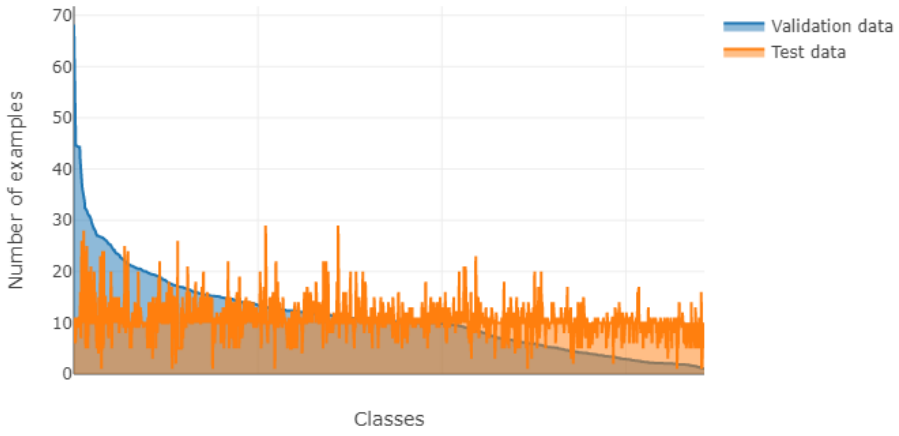


Figure 4.7: The average number of examples per class in a validation group, and the total number of examples for the same class in the test data.

tested on a subset of the data. The splitting into groups take into account the classes, so each class should have the same number of examples in each subset. Figure 4.7 shows the difference in the class balance between the test data and the average of a validation subset. The average count for a class in the test data is 10.6, while the average for the same in a validation group is 11.2. The test data has a better class balance, as its standard deviation is 3.0 and the standard deviation for a validation group is 7.8. However, as the figure shows any given class that has more or less examples than the others in one dataset does not necessarily have the same in the other dataset.

4.5 Summary of data

This chapter has provided an overview of the dataset used in this thesis.
The dataset originates in the current chatbot solution in use by the Norwegian bank DNB for its customer support. Both the classes and their hierarchy, as well as the text messages themselves, are gathered from its data.
The data itself consists of approximately $267,000$ text messages in mainly Norwegian, English and Finnish, although the main focus of this thesis is on the Norwegian data. 72.5% of the messages are in Norwegian, 15.8% in Finnish and 9.81% in English. The examples have varying length, but are generally quite short.

There are a lot classes, 1716 in total, all of which are organized in a wide but shallow tree hierarchy. An example may be categorized as any category in the hierarchy, not only leaf-nodes. There are 22 top-level categories into which the classes are organized, and these categories are unequally represented in the dataset. There is some imbalance in the number of examples per class as well, but no single class dominates.
The chatbot solution uses a separate dataset for testing, with $18,000$ examples in total. This data is different from the training data, as it contains spelling errors and words that are not found in the training data. The class balance is also different.

Chapter 5

Method

There are many different components that are combined in order to perform the experiments, and the configuration of each is presented in this chapter. First an overview of the different libraries used for the implementation is presented, followed by an overview of how the data was gathered. Next the tokenization process performed on the text documents is discussed, followed by the main focus of this thesis, namely the embedding models and their configuration. The different classification models are discussed next, followed by a description of the evaluation metrics. A summary of the methods used is presented in the final section.

5.1 Tools and libraries

Several Python libraries have been used for the experiments in this thesis, and they are all listed below, apart from those that are a part of Python itself.

Numpy[1] This library is used for representing and manipulating multi-dimensional arrays.

Pandas[2] Pandas contains objects for storing one- and two-dimensional data tables, as well as analyzing and manipulating their content. Both the object types DataFrame and Series are used extensively in the experimentation.

Scikit-learn[3] Scikit-learn, also called sklearn, is a library for Machine Learning, with implementation of different algorithms, as well as functionality for preprocessing and evaluation of the result. The experiments with Naïve Bayes uses the implementation from this library.

[1] `https://www.numpy.org`
[2] `https://pandas.pydata.org`
[3] `https://scikit-learn.org/stable/index.html`

Keras[4] This library is an API for other libraries that contain implementations for neural networks of various types. The library used for the actual implementation in this thesis is *TensorFlow*, which *Keras* is an interface to.

TensorFlow[5] TensorFlow contains efficient implementations of neural networks, capable of running on GPU or purely on CPU. It is used by Keras for the actual implementation of the neural network classifiers.

Pytorch[6] Pytorch is similar to TensorFlow, in that it contains implementation of neural networks. The *pretrained BERT* library is implemented using this library, and so it is used for generating embeddings from BERT.

Pytorch pretrained BERT[7] This library contains both reimplementation and pre-trained models for several pretrained Transformer-based models, such as BERT and OpenAI's GPT model, which is built upon the *Pytorch* library. The pretrained BERT model provided is the original model that was pretrained by the authors of BERT. This model is kept static throughout this thesis, i.e. no further training of this model is performed, and is used for generating embeddings.

Gensim[8] Gensim is a library that contains various functionality for Natural Language Processing (NLP). The word embedding models provided by [22], included the word2vec model used here, is implemented and provided using this library.

5.2 Gathering the data

This section presents an overview of how the data was gathered. The data itself and its source are described in detail in chapter 4.

Both the data itself and the class hierarchy was gathered by an automated web scraping script operating on an internal administration tool for the current chatbot solution. This tool gives access to all training and test data, where every example is labeled with its language and organized according to its class. The class hierarchy itself is also available through this tool.

The gathered data was first split into a training set that consisted of 90% of the examples, and a development test set with the remaining 10% of the data. The results reported in this thesis is on the basis of k-fold cross validation, where the whole dataset is first shuffled and then split into k parts with $k = 10$. The split is stratified, meaning that a class has the same number of examples in each part, so that the problem of classes being present in

[4] https://keras.io
[5] https://www.tensorflow.org
[6] https://pytorch.org
[7] https://github.com/huggingface/pytorch-pretrained-BERT
[8] https://radimrehurek.com/gensim/

the training set but not the test set and vice versa is avoided. Cross-validation entails that a model is trained on $k-1$ parts of the data, and tested on the single remaining part. This is repeated k times, so that a model is tested on any given data subset exactly once. The final result of the evaluation metrics is the average of the score for all k tests.

5.3 Tokenization

The first step of a natural language processing (NLP) task is typically tokenization. Tokenization is the process of grouping characters together into tokens, where a token typically corresponds to a distinct word or a punctuation mark, although that is not always the case. To a machine the raw text of a document is just a string of characters, numbers, punctuations and whitespaces, with no additional structure between the character and document level. This is assuming that the document is on a simple text format, without any additional structural information such as html tags, which is the case in this thesis. As a result of tokenization each document is transformed from a string to a list of tokens. Any further preprocessing depends on the classification model used, such as transforming the token list into the vector representation used by Naïve Bayes. The relevant steps for each model are therefore discussed in the sections covering it.

5.3.1 Regular tokenization

The tokenization used as part of the preprocessing before classifying with Naïve Bayes or generating word2vec embeddings is a very simple method; iterating over a text using the regular expression `((\w)+)`. In Regex for Python 3 the character `\w` means any character that is a Unicode letter, an ideogram, a digit or an underscore, and the + sign is a quantifier that means one or more. Each continuous group of these characters is therefore returned as a single token by the tokenizer, and each text is transformed into a list of such tokens. Additionally all characters are lowercased as part of the tokenization process.

5.3.2 Tokenization for BERT

The BERT model is rather particular in what tokens it will allow, and requires a particular tokenization method, namely using WordPiece Tokenization. This process is as follows:

1. Accent removal
2. Punctuation splitting
3. Whitespace tokenization
4. WordPiece tokenization

The accent removal step may also include lower-casing if required by the version of BERT used, but that is not the case for the model used in this thesis. The punctuation splitting adds whitespace around all punctuation characters, which are defined as any character that is not a letter, number or space character, i.e. characters such as ´, . and #.
At this point the original text sequence is still a single sequence of characters, but with

application of whitespace tokenization every subsequence of characters that is bounded by whitespace is designated as a single token.
WordPiece tokenization takes these tokens and transforms them into tokens that can be found in the provided vocabulary of WordPiece tokens. This algorithm takes each token and searches the vocabulary for an exact match. If no match is found the algorithm searches the vocabulary for the longest match, splits the matching substring from the token, and repeats the process with the remaining substring. Any token that is not the beginning of a word is denoted by `##`. For example the word `unaffable` may be split into the tokens {`un, ##aff, ##able`}.

The tokenization for the input passed to BERT is implemented in the *Pytorch pretrained BERT* library introduced in above in section 5.1, using the library's *BertTokenizer* class with the vocabulary for the multilingual BERT model. The WordPiece vocabulary used by this tokenizer consists of around $110,000$ WordPiece tokens, and contains tokens for all languages the model supports, i.e. tokens for languages such as Thai and Chinese in addition to Indo-European languages such as English and Norwegian.

5.4 Generating embeddings

The theoretical foundations and architectures of word2vec and BERT are discussed in sections 2.2.1 and 2.2.2 respectively, and is not repeated here. However, the precise details of how these models are used for generating embeddings as discussed below. Both models are already trained, and are kept unchanged throughout the experiments, although it would have technically been possible to fine-tune the BERT embeddings. However, as the word2vec model provided by [22] is in a format that does not easily allow for further training the two are both kept static to ensure a fair comparison. Also, the BERT model is quite large and would therefore require both time and a lot of computational power to finetune, so finetuning BERT itself is outside the scope of this thesis.

5.4.1 word2vec

The word2vec model used is the CBOW type, introduced in section 2.2.1, and the model itself and its training is described in section 3.3. Two versions are used, one where the embeddings have vector length 300, and another with length 100. The model with vector length 300 is only trained on the Norwegian Newspaper corpus, as that is the only one that has been made available for the model type. The one with vector length 100 is trained on all three corpora, and therefore has a vocabulary that is more than twice as large.

The model itself is provided using the *gensim* library, which makes it very easy to get the embedding for any word found in the vocabulary, as the model is in essence a dictionary with words as the key and their embedding as the value. However, the model cannot return an embedding for a word not found in the vocabulary, and the embedding for these OOV words are therefore set to $\vec{0}$.

The final embeddings for an example is dependent on what classification model it is passed

as input to. In the case of feed-forward NN, the word embeddings are averaged to generate a single embedding for the entire text sequence, which has the same vector length as the original word embeddings. In the case of LSTM, however, the word embeddings are kept unchanged and stored as a sequence of embeddings that match the sequence of tokens. As the LSTM model expects the sequences to be of an uniform length, they are padded at the beginning of the sequence with zero vectors.

5.4.2 BERT

As mentioned above in section 5.1, the actual BERT model used in this thesis is provided through the library *Pytorch pretrained BERT*, and is the *bert-base-multilingual-cased* version, which is the most recent and recommended multilingual model at the time of writing[4]. This model is pretrained on the Wikipedia[9] articles of 104 languages, and support Norwegian, Danish, Finnish, English, and many other languages. The model is cased, which means no lowercasing is performed on the words, and the size is the smaller of the two sizes mentioned in [5].

The embedding for an example is generated by first passing the raw text through the tokenization process described in section 5.3. The result is a list of WordPiece tokens, which is transformed into one-hot encoding, where the vocabulary for the encoding is the same as the WordPiece vocabulary, and passed to the BERT model. This encoding corresponds to *token embeddings* in figure 2.4. The segment embeddings in that figure can be ignored here, as there is only a single segment and they therefore have no impact on the result. The model expects the input to be of an uniform length, and the input sequence is therefore either padded to reach the length, or reduced by removing the last tokens. Along with the token embeddings a vector called an *input mask* is also passed as input. This vector is of the same length as the token sequence, and contains a 0 on the indices that have been added through padding, and 1 otherwise. The model ignores any element whose value in the input mask is 0.

The embeddings are generated by passing this input through BERT and taking the values for each token from the final layer of the model. This results in a contextualized embedding for each word, with a vector length of $h = 768$. For a given sequence with k WordPiece tokens, including the two $[CLS]$ and $[SEP]$ tags, the word embeddings together make a matrix with dimensionality $k \times h$. This is the same as $(d + 2) \times h$ if the number of tokens is the same as the number of words. The final embedding for an example is generated by taking the average of all word embeddings, so that the final result is a vector of length h.

5.5 Models

The main focus of this thesis on the embeddings, i.e. on BERT and the Norwegian word2vecmodel, rather than the models used for the actual classification. Still, to evaluate the performance of the embeddings on the classification task at hands requires some

[9] `www.wikipedia.org`

models to be implemented, and those are described below.

The most simple baseline used is not a Machine Learning model, but rather a simple algorithm, namely to just predict the most common class regardless of input. There is another baseline implemented, however, which also acts as a baseline against the use of embeddings, and that is Naïve Bayes.

5.5.1 Naïve Bayes

The theory of Naïve Bayes is discussed in section 2.4.1, and is therefore not repeated here. The input data is tokenized using the regular tokenization method described in section 5.3.1, which means that the input is transformed from a text sequence into a list of tokens. This list is transformed again using the bag-of-words approach described in section 2.1.1, with the result being one vector per each example which containts the word frequencies. The classification model used is multinomial Naïve Bayes, which is suitable for multi-class classification, and uses the implementation *MultinomialNB* from the library *scikit-learn*.

5.5.2 Feed-forward NN

The most simple neural network architecture used is a small feed-forward neural network, which is provided embeddings as input, either from word2vec or BERT. The generation of word embeddings for each example is discussed above in section 5.4. The configuration of the network is kept identical regardless of where the embeddings originated and of their vector length, in order to facilitate a comparison between the performance of different embeddings.
There are three layers in the architecture, which are shown in figure 5.1, namely the input layer, the hidden layer and finally the output layer.
The input layer has the same size as the length of the embeddings, which varies in the

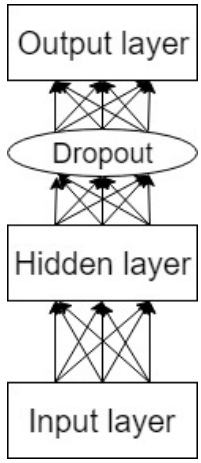


Figure 5.1: The structure of the feed-forward NN used in the experiments.

experiments. The hidden layer has size 256 and uses the identity function $f(x) = x$ as its activation function, while the output layer has the same size as the number of classes in the training data and uses the softmax activation function described in equation 2.9. Dropout is used between the hidden and output layer so that 20% of the connections between the two are dropped at random during training and no connections are dropped when testing.

The optimizer used is Adam[8, p.301,413], and the loss function is categorical cross entropy, which is defined as

$$L(y, \hat{y}) = -\sum_{i \in |C|} y_i log(\hat{y}_i). \tag{5.1}$$

$|C|$ is here the number of classes, y is a one-hot encoding of the true class, and $\hat{y}$ is a vector of probabilities for the classes, where $\hat{y}_i$ is the predicted probability of class i. Batch size is set to 64, and the number of epochs is 10 for the experiments using word2vec embeddings, and 6 for the experiments using BERT embeddings.
Note that no finetuning of these hyperparameters or the architecture itself has been performed, which is one of the tasks suggested for future work in section 7.1.

5.5.3 LSTM

The implementation of LSTM is quite similar to the feed-forward neural network (FFNN). They both use the embeddings from either word2vec or BERT as input, but while the FFNN just receives one single embedding per example the LSTM model instead receives a sequence of embeddings. This sequence mirrors the sequence of tokens that was passed to the embedding model. As the LSTM classifier requires the sequence to be of a uniform, predefined length, the sequence is either padded with zero-vectors if too short, or cut off if too long. Any padding is added at the beginning of a sequence, since the LSTM model used is uni-directional and receives each entry in the sequence in left-to-right order.

The architecture of the model consists of an input layer, followed by a layer of LSTM cells, and finally the output layer. Recurrent dropout is applied to the LSTM layer, which means that dropout is applied at the recurrance connections, i.e. connections between the architecture at different time steps.
The output layer is identical to the output layer of the FFNN model, i.e. it has the same size as the number of classes. The layer uses the softmax function as its activation function, described in 2.9. The loss function is categorical cross entropy as defined in equation 5.1, and the optimizer used is Adam.

5.6 Utilizing the class hierarchy

Almost all of the experiments in this thesis use the flat approach to hierarchical classification, as described in section 2.3. However, a secondary experiment explores the hypothesis that incorporating the class hierarchy in the classification process may improve the quality of the predictions. The approach chosen is local classifier per parent node (LCPN), which is one of the local approaches, together with the Naïve Bayes model described above. The

classification proceeds in a top-down manner through the tree that represents the class hierarchy, and one classifier is trained for each non-leaf node to determine which of its children the example should be classified as. The algorithm terminates when either a leaf node is reached, or the example is classified as a member of a virtual node as described in section 2.3. All other details, such as the preprocessing and the configuration of the model, are kept exactly the same as in the experiment with the flat approach using Naïve Bayes, in order to ensure a fair comparison.

5.7 Evaluation metrics

There are several ways of evaluating the classifying ability of a model, and different metrics typically give a slightly different view of the model's performance. This thesis therefore uses four different metrics, namely precision, recall, F_1 and accuracy, each of which is described below.

It is useful to establish a common terminology before the metrics are described, and a good starting point is the contingency table [1, p.325-326], the structure of which is shown in table 5.1. There are four group of examples for a given class c_i after the classification has been performed. True positives, denoted *tp*, is the group of examples that are classified as belonging to class c_i, and they actually belong to that class. If for instance the task is to classify pictures as being of one of the animals {`cat`, `dog`, `horse`}, and the class c_i is the cat pictures, then true positives *tp* are the cat pictures that are classified as such. False positives *tn* are the examples that are classified as belonging to the class c_i although they are actually not, i.e. pictures of either dogs or horses that are classified as being of cats. Equivalently, true negatives *tn* are examples correctly classified as not belonging to the class, while false negatives *fn* are incorrectly classified as not belonging to the class although they actually do.

	$y \in c_i$	$y \notin c_i$
$\hat{y} \in c_i$	true positive *tp*	false positive *fp*
$\hat{y} \notin c_i$	false negative *fn*	true negative *tn*

Table 5.1: Contingency table for class c_i. y is the true class of the examples and $\hat{y}$ is the predicted class.

5.7.1 Accuracy

Accuracy measures the fraction of correct classifications, either for a single class or for the entire dataset [1, p.326]. It is calculated by

$$A(c_i) = \frac{tp + tn}{tp + fp + tn + fp} \tag{5.2}$$

when measuring accuracy for a single class c_i. Sometimes the error is used instead, which is the fraction of incorrectly classified examples, i.e. $1 - A(\cdot)$, but only accuracy is used here.

Accuracy is a common metric, but should be used with caution when there is a class imbalance, i.e. some classes are much more common than others. If for example a dataset of pictures has the class distribution $\{$`cat`: 80, `dog`:10, `horse`:10$\}$, a classifier could get 80% accuracy by simply classifying a picture as `cat`, regardless of content. Another classifier with the same accuracy, but that also classified some examples as the other two classes, could be considered a better model although the metric does not reflect this. It is therefore to combine the use of accuracy with other metrics, such as the ones discussed below.

5.7.2 Precision and recall

Two metrics commonly used in tandem are precision and recall, which are quite similar to the metrics of the same names that are used in information retrieval [1, p.327]. They measure the quality of the classification for a single class c_i using the equations

$$P(c_i) = \frac{tp}{tp + fp} \tag{5.3}$$

$$R(c_i) = \frac{tp}{tp + fn} \tag{5.4}$$

where P is precision and R is recall.
Precision measures what percentage of examples classified as a class actually belong to that class, while recall measures the percentage of members of a class that were correctly classified as belonging to that class. The two metrics thus balance each other, and are more robust in the face of potential class imbalance than accuracy.

5.7.3 F_1

The F_1 metric, or F_β in its more general form, combines recall and precision into a single metric [11, p.74]. The general formula is

$$F_\beta = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}, \tag{5.5}$$

where β is a constant that governs the balance between precision and recall in their impact on the metric. The typical value is $\beta = 1$, which gives both equal weight, and the resulting F_1 metric is thus

$$F_1 = \frac{2PR}{P + R}. \tag{5.6}$$

The metric is derived from the harmonic mean of precision and recall, which is a more conservative metric than the common arithmetic mean. This entails that the lower number of the two has a heavier impact on the result.

5.7.4 Macro, micro and weighted averages

In the case of more than two classes, i.e. when the classification problem is not binary, it is necessary to combine the results for each class into a single metric [11, p.75]. This is especially true when the number of classes grows large, as it does in this project, when using one metric per class results in a quite unwieldy collection of metrics. There are two main ways of combining these values, namely macro-averaging and micro-averaging. For a given metric $m(\cdot)$, macro-averaging computes the metric for each class first, and then takes the average of the results. Micro-averaging, on the other hand, computes the metric just once for the entire classification.

The difference between the two averaging methods when comparing results is that micro-average gives the same importance to each *example*, while macro-average gives the same importance to each *class*. The distinction is unimportant if the classes are well balanced, i.e. the sets of examples belonging to each class are approximately equal in size, but should be considered closer if there is a class imbalance.

There standard way of calculating the macro-average is to take the arithmetic mean of the metric result for each class, which means that each class has the same impact on the average, regardless of how many examples they contain. Another way is to take the weighted macro-average. This entails calculating the metric for each class, and weighing their average according to the number of examples they have in the dataset. The weighted macro-average and the micro-average are therefore quite similar in what they measure, but are calculated differently.
The averaging methods that will be used are macro average and weighted macro-average.

5.7.5 Confusion matrix

A confusion matrix is not quite a metric but simply a count of errors, but is nevertheless useful for examining a classification result in depth. A confusion matrix contains a count of the number of times one thing was confused with another, with the thing being a class or a class category. For example, say the classes are cat, dog and $horse$, then the resulting confusion matrix may be like the one shown in table 5.2. The rows are the predicted class and the columns are the true class.

	cat	dog	horse
cat	60	5	2
dog	10	2	1
horse	10	3	7

Table 5.2: An example of a confusion matrix. The predicted class is represented by the rows, and the true class by the columns.

5.8 Summary of methods

All data used was gathered from the current chatbot solution, already labeled with their language and class. The data is split into 10 groups using stratified splitting so that it may be used for k-fold cross validation, where $k = 10$.
The tokenization performed on the data depends on what embedding model, if any, is used. For word2vec embeddings or when Naïve Bayes is the current classification model the tokenization is performed using simple regular tokenization, while when BERT embeddings are used the tokenization is performed using WordPiece.
There are two different models used for generating embeddings, namely word2vec and BERT. These embeddings are passed to either a feed-forward neural network, or an LSTM-based architecture. Naïve Bayes is used as a baseline model, which does not use embeddings but rather count vectorization.
The results of the experiments are evaluated using the metrics accuracy, precision, recall and F_1.

Chapter 6

Results and discussion

This chapter presents the results of the experiments performed, as well as a discussion of each result and a comparison of the different models. The theory behind both the classification models and embedding models used is discussed in chapter 2, and the methodology of the experiments can be found in chapter 5. The evaluation metrics are also presented in the latter chapter, although how they relate to the task at hand is discussed below. The different data used in the experiments are discussed in chapter 4.

All experiments described in this chapter are evaluated using 10-fold cross-validation unless stated otherwise. This technique splits the data into 10 groups, and tests a model on one group after it has been trained on the remaining nine groups. This is repeated so that the model is tested on each group exactly once, and the reported evaluation scores are an average of the 10 results.
All experiments except the last one uses the flat approach to classification, which is explained in section 2.3.
The classes in the dataset all have Norwegian names, but a translation into English can be found in Appendix B for the top-level classes, which are also defined as the main class categories due to their position at the first level of the hierarchy.

6.1 Observations on the evaluation metrics

There are several metrics that are used for evaluating the results of the different experiments in this thesis, namely accuracy, precision, recall and F_1, and different ways of calculating their average are used. The metrics and how they are calculated are discussed in section 5.7. These metrics are all widely used in the literature, not only for hierarchical text classification which is the task in this thesis. The different characteristics of this type of task are described in section 2.3. It is useful to discuss how the evaluation metrics are affected by the characteristics of the task before the results of the experiments are presented.

Of particular relevance to how the evaluation metrics should be interpreted is the presence of a class hierarchy. The metrics do not take this hierarchy into account, although some classes are more closely related to others as represented by the hierarchy. One might consider a misclassification as a class that is wrong, but close to the true class in the hierarchy, can be considered less wrong than if the predicted class further away from the true class. However, the metrics do not take this into account as each misclassification is regarded as equally wrong, so no partial credit is given for an answer that is almost but not exactly correct. The judgement of classification quality inherent in the metrics is therefore quite harsh. They nevertheless give a good indication of each model's performance, and as the same metrics are used regardless of experiment the comparisons made should also be valid.

The main discussion uses the proper evaluation metrics, i.e. accuracy, precision, recall and F_1, but confusion matrices are also used for some of the main experiments. Keep in mind that the confusion matrices show classifications according to main class category, and there are several classes in each category. This means that only misclassifications as classes in the wrong category are shown as such, while examples predicted as an incorrect class which is in the same category as the correct class are counted as the same as a correct classifications. However, one might consider the cross-category misclassifications as more severe mistakes, as they are further away from the correct class in the hierarchy.

Keep in mind for the following discussion that accuracy is calculated from the total number of true and false predictions made by each model, while precision, recall and F_1 are calculated first for each class, and then averaged using the average calculations described in section 5.7.4.

6.2 Baseline models

When making any observation about the quality of a classification model it is useful to have something to compare it to, in particular as it is difficult to say whether a given model is good or not without knowing how a bad model would perform. Good and bad are relative terms, but in general a good model makes more correct predictions, and is able to learn to do good classifications for many classes. It is also often desirable to choose the simpler alternative if two models are equal in performance but unequal in complexity, and a comparison between a simple and a complex model should give some indication as to whether its complexity is needed.
Two baseline models are used in this thesis, namely Naïve Bayes and a simple model that just predicts the most common class.

6.2.1 Predicting the most common class

This approach can hardly be called a machine learning model, as effectively no learning is being done. It is simply a counting algorithm, that counts the classes of the experiments in the training data, and always predicts the most common class of this dataset, regardless of

input at the time of testing. The evaluation scores of this approach is shown in table 6.1.

Evaluation metric	Average calculation	Score
Accuracy	-	0.004
Precision	Macro	0
	Weighted	0
Recall	Macro	0.001
	Weighted	0.004
F1	Macro	0
	Weighted	0

Table 6.1: The evaluation scores of always predicting the most common class regardless of input.

The class balance of the dataset is discussed in some detail in chapter 4, and it shows that while there is not a perfect balance between the the amount of data each class has in the dataset, there are nevertheless no single class that completely dominates the others. There are a lot of classes, and the most common class accounts for only 0.4% of the examples. It is therefore as expected that the approach of always predicting this class yields an identical accuracy as its frequency, namely 0.004 or 0.4%, and that the other metrics are extremely low. There is obviously much room for improvement with these scores.

6.2.2 Naïve Bayes

Naïve Bayes is perhaps a more sensible approach than the first baseline, as it actually attempts to learn from the data. A theoretical description of the model can be found in section 2.4.1. The evaluation scores from training and testing the model using 10-fold cross-validation are shown in table 6.2, as well as in figure 6.1.

Evaluation metric	Average calculation	Score
Accuracy	-	0.516
Precision	Macro	0.567
	Weighted	0.668
Recall	Macro	0.337
	Weighted	0.516
F1	Macro	0.396
	Weighted	0.507

Table 6.2: The evaluation scores of Naïve Bayes. Also shown in figure 6.1.

As seen from these results, the model is surprisingly competent despite its simplicity, with an accuracy of 51.6%, much higher than the previous baseline accuracy of 0.3%. Although every example has only one correct class, and there are 1715 other classes it can be classified as that are incorrect, the model still predicted the correct class about half the time.

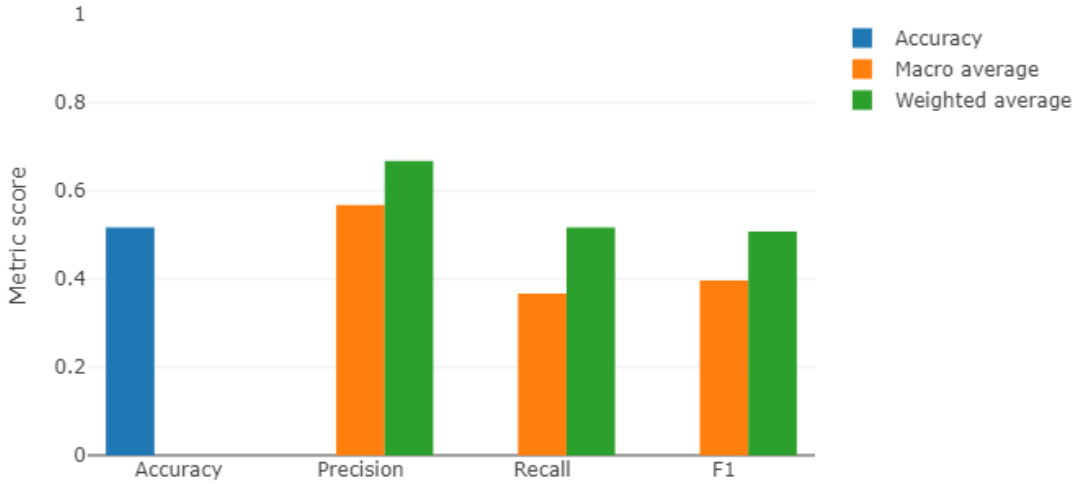


Figure 6.1: The evaluation scores of Naïve Bayes, also shown in table 6.2. The accuracy score is calculated once for the entire classification, so unlike with the other metrics no average calculation is applied.

The possible reasons for this result are discussed below.

The precision and recall metrics show a more detailed picture. The precision is higher than recall, regardless of how the average is computed. This means that on average the percentage of examples that belong to a given class and is classified as it, is lower than the percentage of examples classified as that class which truly belong to it.

Note how the weighted average of all metrics are higher than the macro average. When using the macro average each class has equal impact on the score, regardless of support in the data set, i.e. its number of examples in the data. The weighted average, on the other hand, weights the metric score of each class according to its support, and so a class with a higher number of examples has a larger effect on the result than one with a smaller amount. That the weighted averages are higher is not surprising, as classes with more training data available should be easier to later correctly classify. There is some class imbalance in the dataset, i.e. some classes have more data than others, which is described in section 4.3.3.

There are too many classes to make examining each individual class a viable approach,

and so instead the classes are organized into categories according to which of the 22 top-level class in the hierarchy they are a descendant of. The accuracy for each is shown in figure 6.2, and a confusion matrix is not shown here due to its size, but can be found in table A.1.

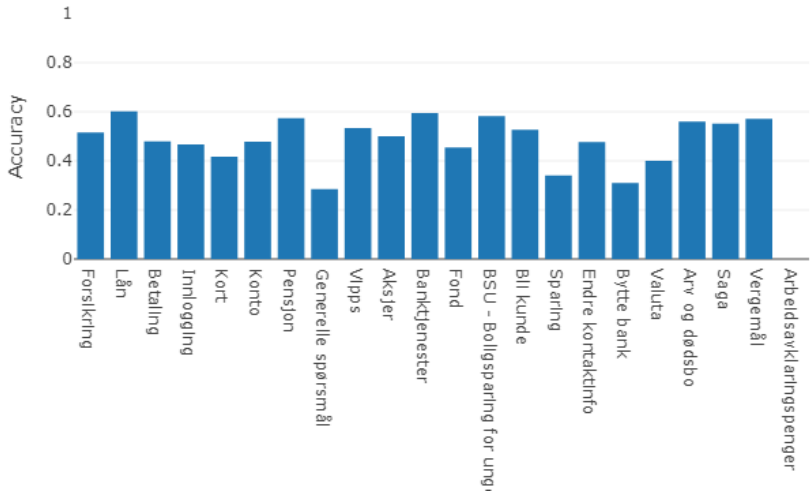


Figure 6.2: The accuracy scores calculated for each top-level class and its descendants, i.e. each main class category, when classifying using a Naïve Bayes model. The categories are sorted according to the total number of examples in each.

Figure 6.2 shows that the accuracy varies a lot between the different categories. There may be several reasons for this. One is that different categories contain very different numbers of classes, as shown in figure 4.4. Another is that some classes have more data than others, and the average number of examples per class varies quite a lot, as shown in figure 4.5. As mentioned above, the weighted averages are higher than the macro averages, which indicate that the model performs better on larger classes.
Note that accuracy is calculated from the total number of correct classifications, and does not take the size of the classes into account. The larger classes therefore have a larger impact on the accuracy, and the model is better at predicting those than the smaller ones.

There may be another underlying reason for the large differences, namely that some categories may be easier to correctly classify than others. This may again be the result of several factors.
There is a possibility that the classes of some categories are more well-defined and separated, i.e. more different from the other classes, than in other categories. The class hierarchy and the classes themselves have after all been constructed over a long time by

several different people.

Another factor may be that the examples of some classes are easier to recognize than others, due to for instance the presence or absence of certain keywords. Naïve Bayes can after all be considered as a sophisticated keyword classifier, as it only looks at the words present in an example when predicting its class. This is due to the central assumption of Naïve Bayes, namely that words are conditionally independent of each other given the class. If some words are highly indicative of a single class this class should be relatively easy to predict. It is important to note here that Norwegian, the language of the examples in most of the experiments, is a language with a lot of compound words, so it is likely that there are more single words in the data that are highly indicative of classes than what would be the case with English examples.
It is very likely that this is the case with some classes. If the example for instance contains the words *innvilge, huslån*, which means *grant* and *mortgage* respectively, it is almost certain that the example is about granting a mortgage. There might be only one class that corresponds to this topic or question, but if there are several classes that concerns this the model may be confused, and make a wrong prediction.

The confusion matrix in table A.1 shows some interesting behaviour with respect to what mistakes the model makes. For instance, more examples of the category *Forsikring* (insurance) are classified as the category *Generelle spørsmål* (general questions) than the total number of examples in the latter category. However, no example in the latter category was classified as another category. The *Generelle spørsmål* category nevertheless has quite low accuracy, which means that all the misclassifications were as classes that were a part of the same category, i.e. about a similar topic, but nevertheless were incorrect. It may be that some of the classes within the category are so similar that the model becomes confused and unable to distinguish properly between them. It is also worth noting that this category has a quite low number of examples compared to the number of classes, so the model has perhaps too little data to properly distinguish them.

In general, when examples are wrongly classified, the predicted class is most often in one of the categories with the most examples in the data set. It might be that the model favours the classes with the most data if there are too little information that distinguish two or more classes, so that the result of the classification is the majority vote of the possible classes, given the words in the example. This makes intuitively sense, as the larger classes have a higher probability of being the true class if no information is known, or if the words present are equally likely in the different classes, and so are more likely to be predicted. Equation 2.5 which defines the Naïve Bayes classifier supports this intuition.

6.3 Word2vec embeddings

There are two versions of the word2vec embeddings trained by Stadsnes [22] that are used in the experiments, namely one with vector length 100 and one with length 300. The exact details of how the embeddings are generated for an example are described in section 5.4.1. [22] contains a detailed evaluation and discussion of these embeddings, which will not be

repeated here.

6.3.1 Feed-forward Neural Network

The first experiment that uses the word2vec embeddings combines them with a feed-forward neural network (FFNN). Its architecture and other information about its configuration can be found in section 5.5.2. The evaluation scores for this experiment can be found in table 6.3, as well as in figure 6.3.
There are two models that have been trained, and they are identical to each other in all ways except that the first uses the embeddings of length $|v| = 100$, and the second uses the ones with length $|v| = 300$. Since they are very similar in both behaviour and performance they are presented together here.

Evaluation metric	Average calculation	Score for $\|v\|$ = 100	Score for $\|v\|$ = 300
Accuracy	-	0.673	0.703
Precision	Macro	0.701	0.721
	Weighted	0.711	0.742
Recall	Macro	0.660	0.678
	Weighted	0.673	0.703
F1	Macro	0.657	0.677
	Weighted	0.670	0.702

Table 6.3: The evaluation scores of FFNN using the different word2vec embeddings, also shown in figure 6.3. $|v|$ is the vector length of the embeddings used.

Both versions of the model show quite good results, with an accuracy of 67.3% and 70.3% for the one using the short and the long embeddings respectively, and the small difference between their accuracies illustrates how similar they are in performance. Just as with the experiment with a Naïve Bayes model the precision is higher than the recall, but the difference is quite small.

Interestingly, the weighted averages are almost equal to the macro averages for all metrics and regardless of embedding length. This indicates that the classes with more data in the dataset perform on average only slightly better than the ones with less data, although one might expect that a model is better at recognizing a class when it has had more opportunities to learn how to do so. It is also a fact that a class with more examples has a higher probability of being the true class than one with fewer if no other information is present, or the information available to the model does not distinguish between the classes. However, the amount of data per class does not appear to have much of an impact on the quality of the classification for that class, since the differences between the average types are so small. Still, the difference is not exactly zero, so one cannot conclude that it has no impact at all, just that the impact is likely very small.

Note that there is much information inherent in the word embeddings, so the model is given a head start when it comes to encoding information, and can concentrate on using

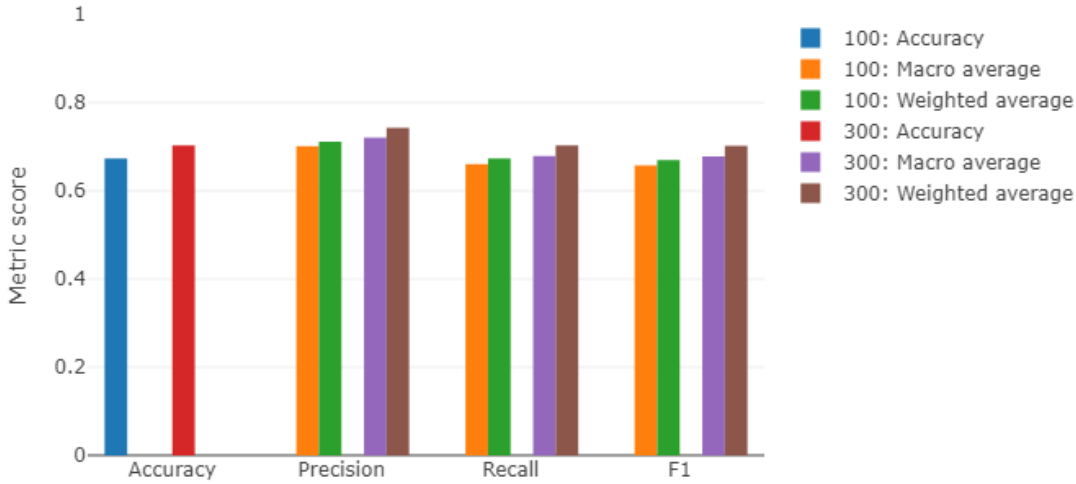


Figure 6.3: The evaluation scores of FFNN using the different word2vec embeddings, also shown in table 6.3. The numbers in the legend refers to the vector length $|v|$ of the embeddings. The accuracy score is calculated once for the entire classification, so unlike with the other metrics no average calculation is applied.

the difference between the examples to distinguish between classes. Also, since these embeddings are trained on quite general texts such as news articles and not on the current dataset, the information inherent in them is independent of the amount of data per class.

The longer embeddings have three times as much space available for encoding information, as the long embeddings have vector length $|v| = 300$ and the short have length $|v| = 100$. [22] found that the embeddings of length $|v| = 300$ were better at determining semantic and symantic relationships than those of length $|v| = 100$, although embeddings of length $|v| = 600$ were slightly worse overall than those of length $|v| = 300$.
One might therefore expect that the longer embeddings tested, which are of length 300, perform better than the shorter. To be fair they do perform better, but as shown in table 6.3 and figure 6.3 the improvement is very small.
Note that in this thesis, as described in section 5.4.1, the word embeddings are averaged when predicting using FFNN, so that each example is represented by a vector of length $|v|$.

There may be several reasons for why the improvement is so small. It is possible that

since the embeddings are not trained on domain-specific documents, such as chat logs or financial documents, the embeddings simply cannot perform better in the current domain, regardless of an increase in the vector size. It is also a factor here that Norwegian contains a lot of compound words, so not all words have available word embeddings. Those words are by the original configuration of the training of the word2vec embeddings either very infrequent or not present in the corpora used. It is reasonable to assume that these words are quite specific to the banking domain, and give a good indication of what class an example belongs to, so the removal of those words will impact the result negatively.

Another reason may be the Curse of Dimensionality [8, p.151], which indicates that if the dimensionality is increased, then the model needs more data for training. The dimensionality is increased by a factor of 3 when using the longer embeddings as opposed to the shorter ones, but the amount of training data available stays the same. Each class has an average of 113 examples, but the lowest number for a single class is 10, so there is perhaps too little data available for the model to improve on the results.

Figure 6.4 shows the accuracy for each main class category. The last category, *Arbeidsavklaringspenger*, is a special case as it contains only a single class and few examples, but otherwise the results here support the observations above. The longer embeddings perform only slightly better than the shorter in some categories, and in some they even perform worse. However, the differences are so small that no distinction between their performance can be made.

There is a large variance between the performance of the models when classifying examples of different categories. There are many possible explanations for this, so it is difficult to conclude why the differences between the accuracy scores are so large. However, some observations may be made as to what reasons there may be.
One possible answer lies in the dataset itself rather than the model. Some categories may be more difficult than others to predict, or some categories may contain only a few classes with high or low accuracy, while other contain a larger or more varied mix. As discussed in section 4.3.3 there is certainly a large difference between categories as to how many classes they contain.

Another possible reason lies with the embeddings themselves. It is possible that the embeddings are better for some words that are more often present in some category than others, and so make the classes of those categories easier to correctly predict. Both the categories *Vipps* and *BSU* refers to banking products, and are some of the categories with lowest scores. The lack of domain-specific training data for the training of the embeddings themselves is likely to have affected the embedding quality for these categories. On the other hand, *Saga* is also a banking product, but has one of the best results. However, there may be some cross-domain influence from other meanings of the word here, as the word *saga* may also mean a Norse legendary tale, and thus probably has a word embedding quite far from more banking-related words in the vector space of embeddings.

A confusion matrix for the different main categories is shown in table A.2. It shows that no category is only misclassified as another category, but the misclassifications have

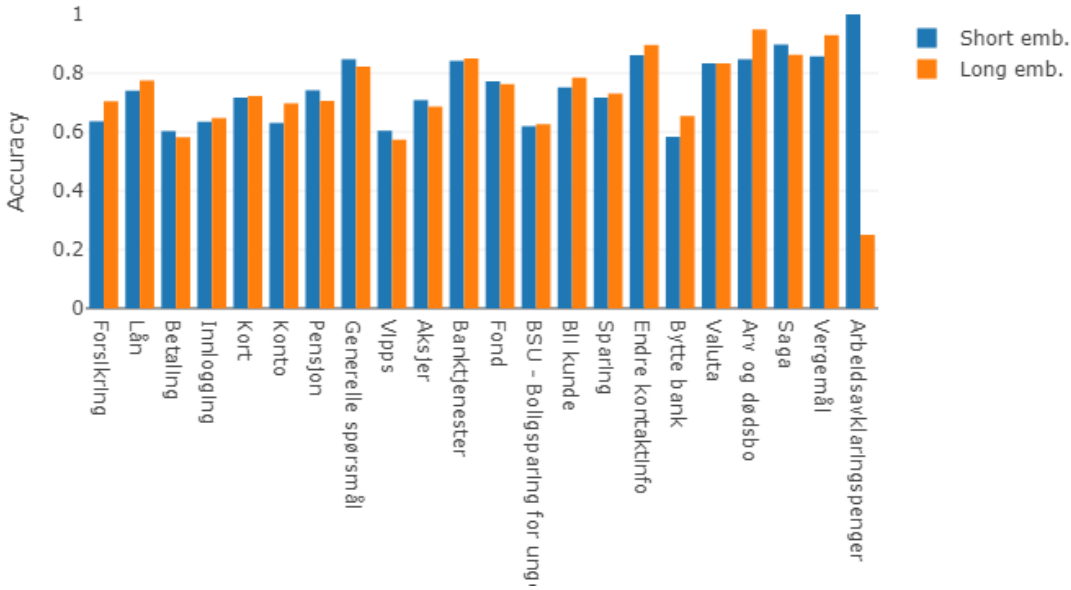


Figure 6.4: The accuracy scores calculated for each top-level class and its descendants, i.e. each main class category, when classifying using FFNN with word2vec embeddings. The long embeddings have vector length $|v| = 300$ and the short ones have $|v| = 100$. The categories are sorted according to the total number of examples in each. The last class, *Arbeidsavklaringspenger*, is a special case as it contains only a single class and few examples.

a tendency to be one of the more populous categories. This is as expected, because as previously stated any class with more examples is more likely to be true than one with fewer in the absence of information, or if the uncertainty as to which is correct is high.

6.3.2 LSTM

The second experiment that utilizes word embeddings trained by Stadsnes [22] uses the same embeddings as the previous example. However, this time the word embeddings are used to construct sequences that corresponds to the list of tokens for each example, as described in section 5.4.1. These sequences are used to train an LSTM neural network, and the configuration of this model is described in section 5.5.3. The results of the evaluation metrics for this model are shown in table 6.4, as well as in figure 6.5.

The accuracy of the LSTM model is quite high, in particular when it uses embeddings of length $|v| = 100$ as input, which results in an accuracy of 86.8%. Precision is still slightly

Evaluation metric	Average calculation	Score for $\|v\| = 100$	Score for $\|v\| = 300$
Accuracy	-	0.868	0.807
Precision	Macro	0.870	0.809
	Weighted	0.883	0.828
Recall	Macro	0.848	0.778
	Weighted	0.868	0.807
F1	Macro	0.849	0.781
	Weighted	0.868	0.808

Table 6.4: The evaluation scores of LSTM using the different word2vec embeddings, also shown in figure 6.5. $|v|$ is the vector length of the embeddings used.

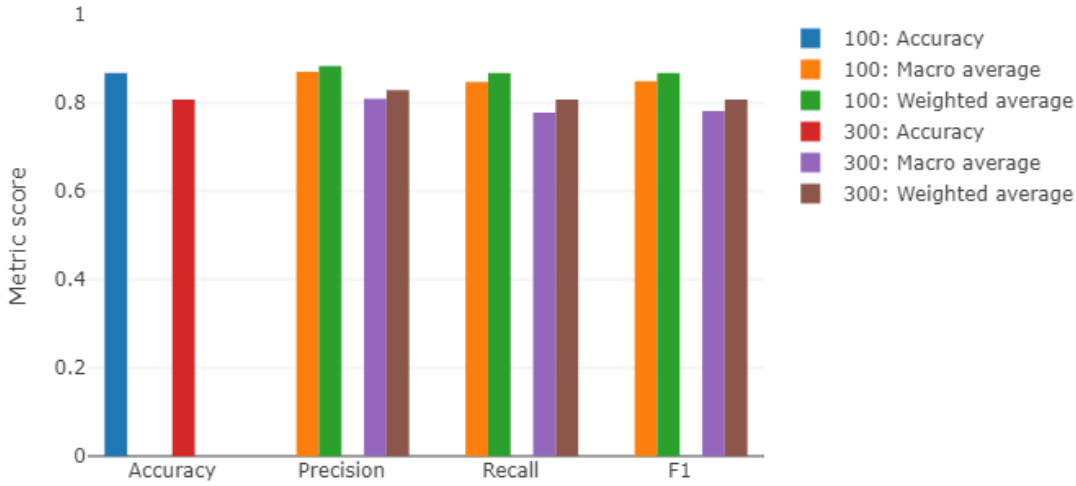


Figure 6.5: The evaluation scores of LSTM using the different word2vec embeddings, also shown in table 6.4. The numbers in the legend refers to the vector length $|v|$ of the embeddings. The accuracy score is calculated once for the entire classification, so unlike with the other metrics no average calculation is applied.

higher than recall, but all in all they are quite balanced for this model.

The weighted averages are again just slightly higher than the macro averages, which

echoes the relationship between the two average types in the previous experiment. This means that the classes with a greater number of examples still on average have a slightly better result than those with a lower number. The difference has not increased though, so the relative amount of data compared to the other classes is still likely to have a small impact on the results.

Surprisingly, all the metric scores are lower for the model using the longer embeddings, although the scores are by no means low when compared to the baseline models of Naïve Bayes and the most common class predictor. There may be several reasons for this, and the Curse of Dimensionality is a likely candidate. As mentioned in the previous section, it indicates that the model needs more data for training if the dimensionality is increased. In the previous experiment the dimensionality of the input for each example was $(|v|)$, where $|v|$ is the length of the embeddings. However, when using an LSTM-based architecture the input is transformed into sequences, and the dimensionality in this case is $(\textit{sequence length}, |v|)$. The sequence length is here set to 12, because around 80% of the examples consists of 12 words or less. This means that in the case of some examples not all words are included, but also that very short messages are not unnecessarily dwarfed by padding.
The dimensionality is therefore already quite high when the embeddings have length $|v| = 100$, and an increase in this length to $|v| = 300$ represents an increase in dimensionality by a factor of 3. This may explain why the increase in the length of the embeddings provides a small benefit in the case of a model based on FFNN, but a comparatively larger detriment in the case of an LSTM-based one.

Another factor that probably impacts the results is the relative small amount of training data per class, and LSTMs are a quite complex type of architecture. This means that it needs more resources for achieving good results, both in the form of training data and computing resources. Note that this does not necessarily mean that it needs more data than a simpler model to become better than it, only that it needs this to achieve the optimal results for the particular configuration of the model.

This complexity of the model also means that LSTMs are prone to overfitting, i.e. training until the training error decreases but the generalization error starts to increase. ANN-based architectures such as the FFNN model in the previous example may also overfit, which is why dropout is applied as a regularization technique, but LSTMs have a particular tendency to do so. It may be that the LSTM model of this experiment overfits to some degree on some classes, but perhaps not on others.

The average scores for each main class category are shown in figure 6.6. The categories all have quite high accuracy, which is expected when the overall accuracy of the model is so high, although there are still minor differences between the classes in the model's performance. The accuracy of the model using the longer embeddings are less than or equal to the accuracy for the model using the shorter ones, for all categories except one, but the difference between the accuracy of each varies greatly between categories.

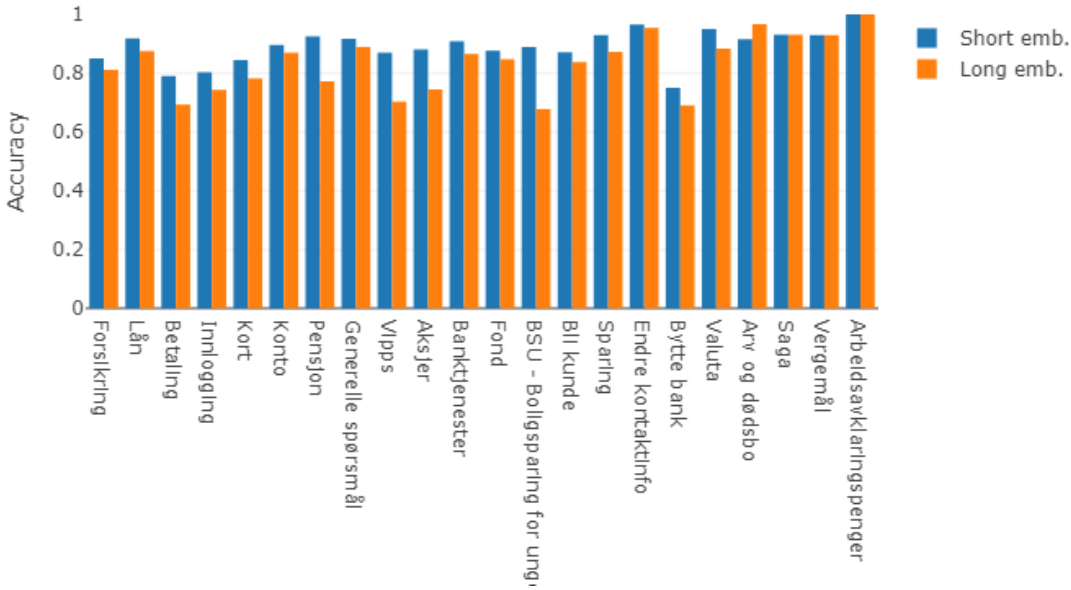


Figure 6.6: The accuracy scores calculated for each top-level class and its descendants, i.e. each main class category, when classifying using LSTM with word2vec embeddings. The long embeddings have vector length $|v| = 300$ and the short ones have $|v| = 100$. The categories are sorted according to the total number of examples in each. The last class, *Arbeidsavklaringspenger*, is a special case as it contains only a single class and few examples.

There may be several reasons for this, such as perhaps the embeddings for the words found only in some categories are better than others. As mentioned above, the model might overfit on some categories, while underfitting on others. However, it is also a possibility that some of the classes are easier to predict than others, and that only some of the difference in scores can be attributed to the model itself.

The confusion matrix for the model using embeddings of length $|v| = 100$ can be found in table A.3. Most of the entries here are quite similar to the confusion matrix, only lower in number of misclassification, but the category of *Bli kunde* has an interesting behaviour. It misclassifies a lot of examples as the category *Betaling*, but there are hardly any misclassifications the other way. This relationship between the two classes is not present in the confusion matrices of the previous experiments, so it may be just a quirk of the model, arising out of some combination of the embeddings and the model architecture used.

6.4 Norwegian BERT embeddings

All previous experiments have only concerned themselves with the Norwegian part of the dataset, and the word2vec embeddings used are only trained for Norwegian. The current experiment uses the same Norwegian data as the other experiments, although BERT itself is pretrained for around 100 languages at the time of writing. An experiment involving this multilingual aspect can be found in section 6.6.2.

There are many ways of extracting embeddings from BERT,described in section 2.2.2, such as using the $[CLS]$-tag, or extracting the output of some of the layers of the architecture and combining these in some way. This experiment uses the method described in section 5.4.2, i.e. extracting the output of the final layer and taking the average. The embeddings are therefore comparable to the word2vec embeddings used together with FFNN, an experiment described in section 6.3.1.
The results of training an FFNN architecture with BERT embeddings can be found in table 6.5, as well as in figure 6.7.

Evaluation metric	Average calculation	Score
Accuracy	-	0.911
Precision	Macro	0.906
	Weighted	0.921
Recall	Macro	0.887
	Weighted	0.911
F1	Macro	0.886
	Weighted	0.909

Table 6.5: The evaluation scores of FFNN using embeddings from BERT, also shown in figure 6.5.

The model has an impressively high accuracy of 91.1%, especially if one takes into accounts the quite low amount of training data available per class, and the high number of classes. There are 1715 incorrect classes and only one correct for the classification of any given example, and the model still manages to predict the correct class in more than nine out of ten cases.
The precision is just as before slightly higher than the recall, but the difference is quite small and both scores are quite good, as evidenced by the F_1 score.

The weighted averages are all somewhat higher than the macro averages, which indicates that this model, like the others, also has a tendency to perform better when predicting on classes with more training data than others. This is by no means surprising, and only confirms the well-established rule of thumb that more data tends to give better results, regardless of task or choice of model. This should of course be taken with a grain of salt, as the results also depends on the quality of the data, such as the presence of incorrect labels or a few examples that are very different from the others.
Note however that the differences between the averages are quite small, so there is no large

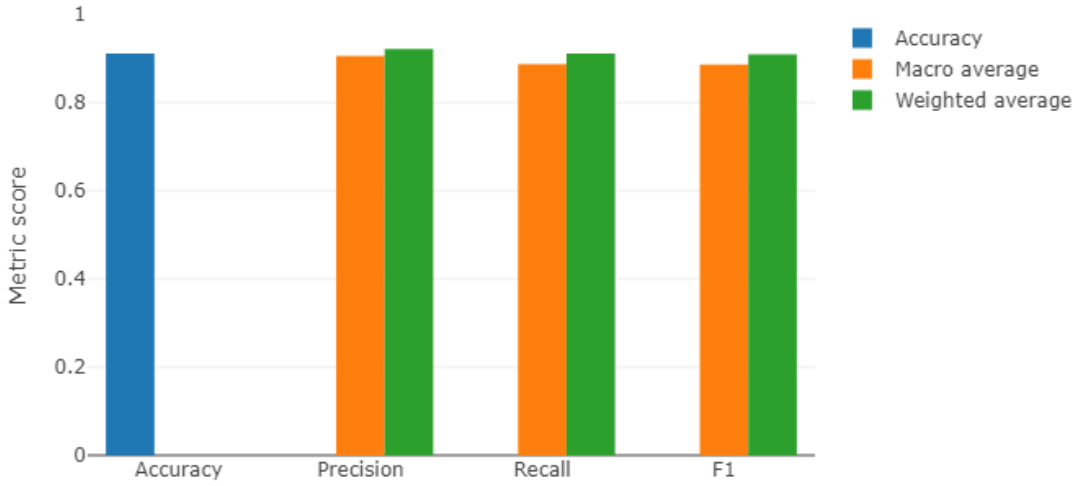


Figure 6.7: The evaluation scores of FFNN using embeddings from BERT, also shown in table 6.5. The accuracy score is calculated once for the entire classification, so unlike with the other metrics no average calculation is applied.

difference in the prediction quality of larger versus smaller classes.

The accuracy for each main class category is shown in figure 6.8. The scores are generally quite high, although a couple of the smaller categories have lower accuracy than others. However, there is no clear tendency of smaller categories having consistently lower scores than the larger. This observation, together with the fact that the weighted averages are only slightly higher than the macro averages, indicates that the models finds some categories more difficult than others, and the reason may not necessarily be the amount of data available for each.
The confusion matrix for the main categories can be found in table A.4. The categories all have quite high accuracy, which means that there are few misclassifications to be found in the matrix, but the mistakes that have been made are quite evenly distributed. No single category has a clear tendency to be misclassified as another specific category.

The BERT architecture, from which the embeddings are generated, is trained on quite general data, which is likely to have an impact on the current classification result. It is

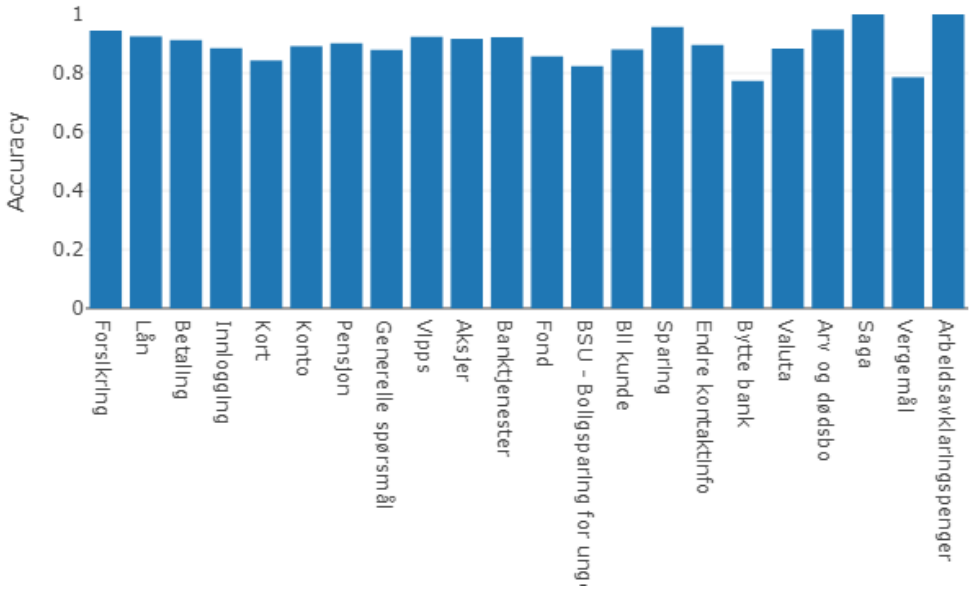


Figure 6.8: The accuracy scores calculated for each top-level class and its descendants, i.e. each main class category, when classifying using FFNN with BERT embeddings. The categories are sorted according to the total number of examples in each.

trained on all data available from Wikipedia[1] in the languages the model supports, which is the 100 languages with the largest Wikipedia database. The language used in the articles on that site is typically a quite formal style, but the examples in this dataset, i.e. from the chatbot solution of DNB, are of a more informal nature. This is highly evident in for instance the length of the examples, described in section 4.2, as the average length of an example is only 9.6 words. This means that the style of language in the examples used to train the embeddings is different from the style of the examples for which to generate embeddings. It also means that since the messages are so much shorter, there is less context from which to generate an embedding. This may impact the quality of the embeddings, and therefore the classification model, but the model nevertheless shows quite good results.

The only language in the dataset used in this experiment is Norwegian, but, as previously mentioned, BERT is trained on several other languages as well. The examples are all written in Norwegian bokmål, but BERT also has support for Norwegian nynorsk, Danish, Swedish, English and others. Norwegian bokmål and nynorsk are formally considered two different languages, but they are very similar, and Norwegian and Danish is very sim-

[1] `www.wikipedia.org`

ilar as well. The BERT model does not distinguish between languages at the input, and trains a single model for all languages it supports. It is highly likely that as it supports all these languages the quality of the Norwegian BERT embeddings is better than it would have been if the model was monolingual. However, there does not exist a pretrained model of BERT exclusively for Norwegian, and so a possible way of improving the results is to train such a model, alternatively improve the existing BERT model by finetuning using Norwegian data.

As mentioned earlier and described in section 5.4.2, the vector given as input to the classification model is an average of the word embeddings for an example, and are generated from BERT. The act of averaging the word embeddings may cause some information inherent in the embeddings to be lost, as they are compressed down into a single vector. This loss could negatively impact the result. However, it is also possible that the overall effect is positive, since the dimensionality is reduced and the Curse of Dimensionality, which is discussed in previous sections, may have less of an impact. Furthermore, the good results of BERT in the literature [5] are reported on architectures combining BERT with a simple FFNN architecture like the one in this experiment. It would however be an interesting avenue of research to combine word embeddings from BERT with a classification model that does not require aggregation of the embeddings, for example an uni-directional LSTM-based architecture as it showed quite good results with word2vec embeddings as described in section 6.3.2, or a bi-directional LSTM.

6.5 Comparison of the models

The previous sections have presented and discussed the results of the different models that have been tested, and the purpose of this section is to compare them to each other. Note that the experiments with word2vec embeddings of length $|v| = 300$ are not included here, only those with embedding length $|v| = 100$, but a discussion and comparison of both can be found in sections 6.3.1 and 6.3.2.

The evaluations of the different experiments can be found in table 6.6, and in figure 6.9. Note that everything except the models themselves and the choice of embeddings is identical for each experiment, i.e. they are all tested using 10-fold cross-validation on the same dataset.

It is quite clear from the results that BERT, which refers to a model using BERT embeddings together with an FFNN architecture, is the best model overall. It has better results across all evaluation metrics, although the LSTM model using word2vec embeddings is quite close in performance.

Naïve Bayes is the worst model in overall performance, especially when the performance is measured using the recall metric, although it has a weighted precision that is almost as high as the FFNN model with word2vec embeddings. It is not surprising that such a simple model as Naïve Bayes has the lowest results of the models tested. However, it is a bit surprising that its results is not worse than they are, in particular as there are so many

Evaluation metric	Average calculation	Most common class predictor	Naïve Bayes	word2vec FFN	word2vec LSTM	BERT
Accuracy	-	0.004	0.516	0.673	0.868	0.911
Precision	Macro	0	0.567	0.701	0.870	0.906
	Weighted	0	0.668	0.711	0.883	0.921
Recall	Macro	0.001	0.337	0.660	0.848	0.887
	Weighted	0.004	0.516	0.673	0.868	0.911
F1	Macro	0	0.396	0.657	0.849	0.886
	Weighted	0	0.507	0.670	0.868	0.909

Table 6.6: Comparison of the evaluation of the previous experiments. Only the evaluations of the models using the word2vec embeddings of length $|v| = 100$ are included here, not the models using those of length $|v| = 300$. The comparison is also shown in figure 6.9.

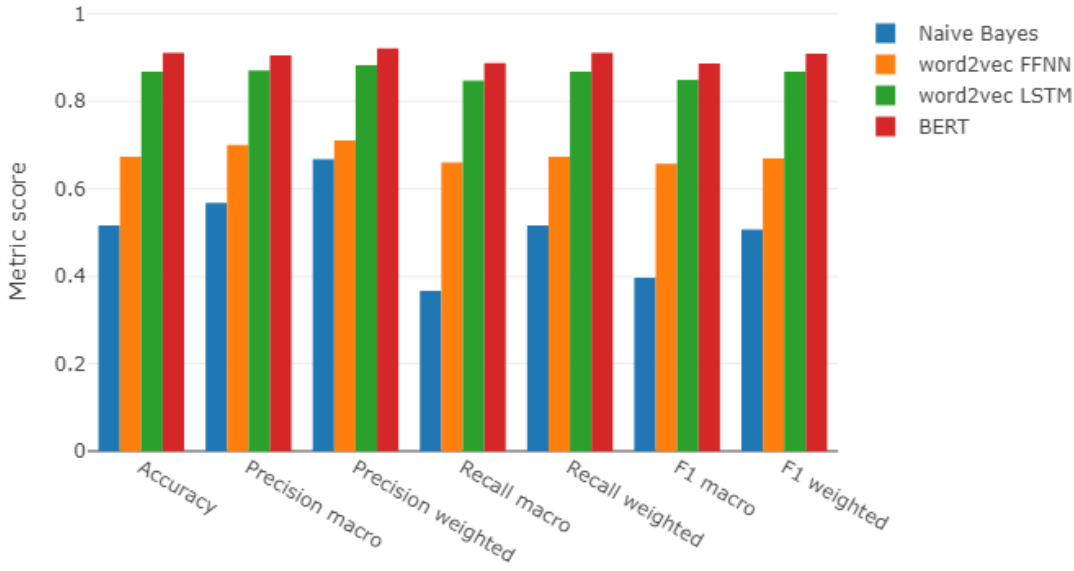


Figure 6.9: Comparison of the evaluation of the previous experiments. Only the evaluations of the models using the word2vec embeddings of length $|v| = 100$ are included here, not the models using those of length $|v|$. The comparison is also shown in table 6.6.

classes. Also, one should keep in mind that the comparison between Naïve Bayes and the embedding-based models is a bit unfair when it comes to the total amount of training data that has been available. After all, the embeddings are pretrained, while the Naïve Bayes

model is only trained using the data found in the chatbot dataset. The embeddings have in effect learned the language first through pretraining, and the classification model uses its general knowledge to make specific conclusions about the domain. However, Naïve Bayes has to start completely from scratch, without any knowledge of the language other than a list of the possible tokens it may encounter in examples.

That the results of Naïve Bayes is so much better than expected may therefore indicate that some examples are quite easy to classify, and the improvement the other models make are in classifying the more difficult examples. For instance, figure 6.10 shows that Naïve Bayes has very low accuracy scores in some categories but not in others. The other models are a particularly high improvement compared to Naïve Bayes in those categories, bringing the accuracy up to be approximately equal to the other categories for the given model. This is despite the fact that some of those categories have quite few training examples, and no modifications of the examples themselves have been performed.

Another interesting result is that the same word2vec embeddings give a much better result when combined with an LSTM model than with an FFNN model. The difference would have been slightly less pronounced if the longer embeddings were used, i.e. $|v| = 300$, since the LSTM model using those has slightly worse results than for the shorter embeddings, and the FFNN model with the longer embeddings has almost identical results to the current version. However, the LSTM model would still have been best, as the difference in performance between the two LSTM models is quite small.
Note that although the original word2vec embeddings are the same, the vector given as input to the FFNN model is an average of the word embeddings of an example, while the LSTM is given a sequence of all embeddings without any aggregation. It is likely that the act of compressing the embeddings down to a single vector causes some information to be lost. However, if that was the only reason for the difference in the results, one might expect the longer embeddings to be more resilient to information loss in the compression, as there is three times as much space for the information to be encoded. However, the results for the different lengths are nearly identical with the FFNN model, so it is unlikely that the whole difference in results between FFNN and LSTM models can be explained by this.

While there is a large difference between the LSTM and FFNN models using word2vec embeddings, there is only a small difference in performance between the LSTM model with word2vec embeddings and the FFNN model using BERT embeddings.
A possible explanation for this is that the LSTM augments the word embeddings with contextual connections, as it is recurrent in nature and trained on sequences, with one embedding per token in an example. The BERT embeddings are contextualized word embeddings, and it so it might be that the LSTM model learns the context already present in the BERT embeddings. However, the LSTM model used here is uni-directional while BERT is bi-directional. This means that the LSTM model may only learn from the words preceding the current word, instead of both the words preceding and succeeding the word as in the case of BERT.

Another factor that may have an impact on the relatively similar results of the two best

models, is that the BERT model that generates the embeddings is multilingual, while the word2vec model is only for Norwegian. Although the creators of BERT have not published a scientific paper about the multilingual version of their model, they have indicated that a monolingual model performs slightly better than a multilingual one for the same language [4]. The FFNN model using the BERT embeddings may therefore have performed slightly better if the BERT architecture was either pretrained exclusively on Norwegian, or finetuned using additional Norwegian texts. However, this is a process that demands both significant computing resources and a lot of training data, so is outside of the scope of this thesis.

A comparison of the accuracy for each main category is shown in figure 6.10. BERT is interestingly not always the best model for the categories, as the LSTM model with word2vec embeddings is better at classifying some of them. Also interesting is that different models struggles with different categories, so the difference in accuracy between the categories is evidently not only dependent on the amount of data for each.
There may be another reason for why a specific model struggles with a given category. Perhaps the quality of embeddings it uses is worse for those categories than the other embeddings, and the other way around, and so the model that uses the embeddings finds the category difficult.

Although the BERT embeddings with FFNN shows comparable results to word2vec with LSTM, it is worth noting that BERT is able to achieve good results with a very simple model, as its results when using an FFNN model is much better than when using word2vec embeddings with the same. It is difficult to continue improving the result of using word2vec embeddings when the best model is already quite complex, while the success of the simple model for BERT raises the possibility that its performance may be further improved by experimenting with differences in the classification model itself.

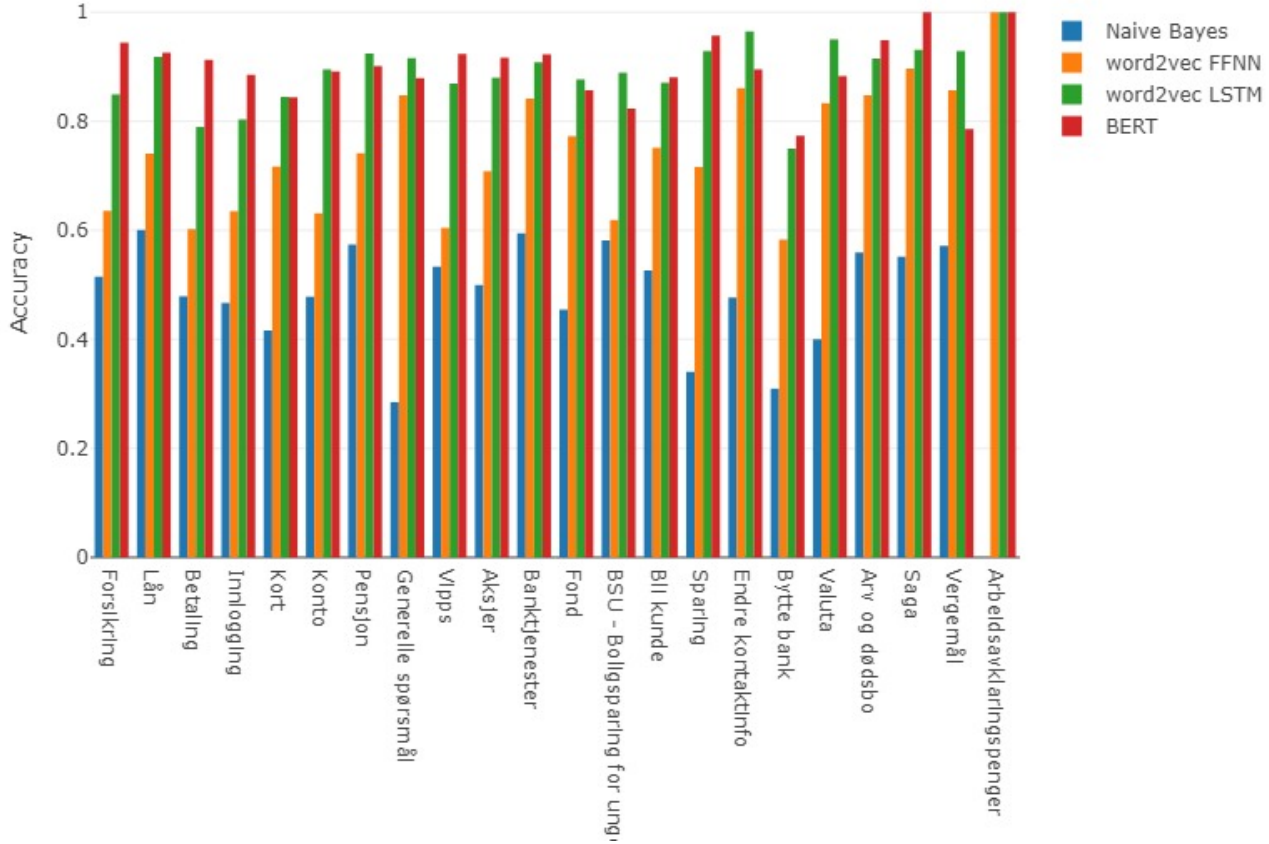


Figure 6.10: Comparison of the accuracy of each models classifications for the main class categories. The categories are sorted according to the total number of examples in each.

6.6 Secondary experiments

The previous experiments are all evaluated by running cross-validation on the chatbot's Norwegian training data, and all use the flat approach to hierarchical classification. This is the main focus of this thesis, but some additional experiments have also been run that do not conform to this description.

The first of these secondary experiments trains the model on the training data, but tests it on the original test data from the chatbot. The second experiment uses BERT embeddings together with an FFNN model to test the multilingual property of BERT, by training on exclusively Norwegian data, and testing on English and Finnish data. The final experiment tries the local approach to hierarchical classification, which incorporates the class hierarchy into the classification process, both at the time of training and testing. The model type used for this experiment is Naïve Bayes.

6.6.1 Testing on the chatbot's test data

The original chatbot solution has two datasets that are kept separate, one which is used for training and one for testing. The differences between the two datasets are described in section 4.4. The main models of the primary experiments, i.e. all models that are compared with each other in section 6.5, have been tested by training on the original dataset meant for training, and tested on the chatbot's test data. The results of this are presented in table 6.7, and in figure 6.11, together with the results from the previous experiments with the same models. Note that the models tested on the test data have been trained using the same amount of data as during the cross-validation in order to keep the comparison fair, as the amount of data available for training may have some impact on the result.

Model	Result of cross-validation	Result on test data
Naïve Bayes	0.516	0.302
word2vec FFNN	0.673	0.501
word2vec LSTM	0.868	0.674
BERT	0.911	0.691

Table 6.7: Accuracy of the different models when tested using cross-validation, versus when trained on the chatbot's training data and tested on its original test data. The results are also shown in figure 6.11.

These results show that every model is consistently much worse when tested on the original test data, independent of model type, than when tested using cross-validation on the training data. A likely reason for this is that the examples contained in the training and test data are quite different from each other. Table 4.2 shows that on average the examples in the training data are over a word longer than the examples in the test data, and all of the percentiles reported are higher as well. Additionally, over 20% of the words in the test data vocabulary are not present in the training data.

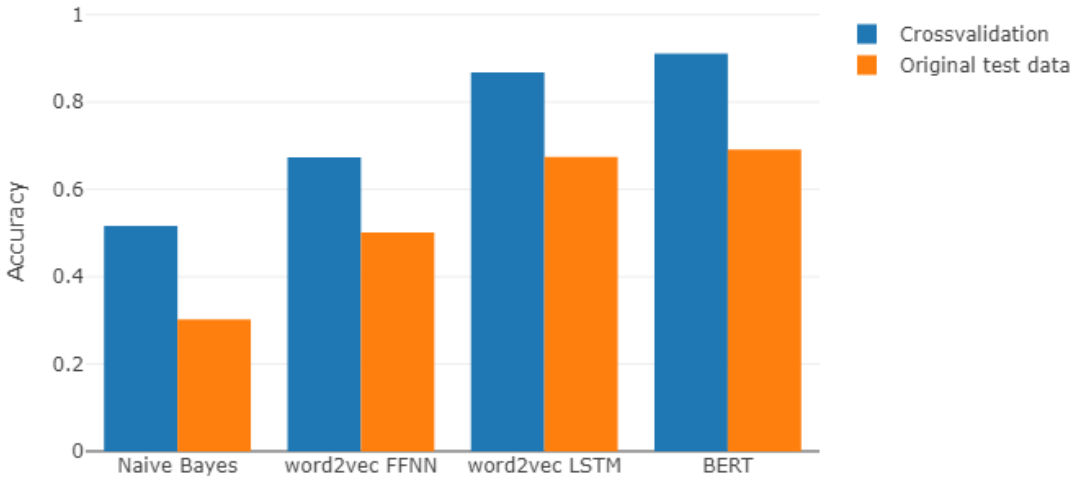


Figure 6.11: Comparison of the performance of the different models on the training data and the original test data. Only the evaluations of the models using the word2vec embeddings of length $|v| = 100$ are included here, not the models using those of length $|v|$. The comparison is also shown in table 6.7.

It is only to be expected that a model struggles to classify data that is different from the data it was trained on, including containing words it has never seen before, and in addition to this challenge it needs to determine class membership based on fewer words than what it was trained for.
Note that if a word is missing from the training vocabulary, it may nevertheless be present in the vocabulary of the word2vec embeddings, and BERT has no vocabulary restriction as it can generate embeddings for any word due to its WordPiece tokenization. However, some of the words in the test vocabulary are spelling mistakes or from another language than Norwegian, and those words are highly unlikely to be found in the word2vec vocabulary.

Another factor that affects the result is that the test data has a different class balance from the training data. The different number of examples for the datasets are shown in figure 4.7. The models have during the earlier experiments show a tendency to perform better on classes with more data, as the weighted averages are a little bit higher than the macro averages, but as the differences are only small the tendency is weak. That the different

number of examples per class in the test data is more uniform than in the training set therefore probably has some small effect on the result, but not enough to explain all of the difference.

It is important to note the purpose of the datasets in the original chatbot solution. There are multiple components in the chatbot, such as a spell-checker, a synonym wordlist and the classifier itself, and the training data is only meant for training the classifier. The purpose of the test data, however, is to test the entire system, not only the performance of the classifier. These results therefore indicate that the models trained during the course of this thesis cannot compete with the entire chatbot solution, as they either need additional preprocessing such as correction of spelling errors, or to be trained on data that is more similar to the test data. However, the good results of the models BERT with FFNN, and word2vec with LSTM, on the cross-validation, show that they may be able to handle real-world data if modifications are made or additional training performed.

6.6.2 BERT embeddings for other languages

The version of the BERT model used for generating embeddings is multilingual, which means it support several languages, including Norwegian, English and Finnish. Those three languages are the most frequent in the training data, although by far the most examples are in Norwegian.
This experiment looks at the multilingual aspect of BERT, and attempts to classify English and Finnish examples after training the model on only Norwegian examples. The results of this are shown in table 6.6.2, and in figure 6.12.

Evaluation metric	Average calculation	Norwegian	English	Finnish
Accuracy	-	0.911	0.094	0.020
Precision	Macro	0.906	0.066	0.023
	Weighted	0.921	0.228	0.047
Recall	Macro	0.887	0.033	0.014
	Weighted	0.911	0.094	0.020
F1	Macro	0.886	0.033	0.012
	Weighted	0.909	0.107	0.017

Table 6.8: The performance of FFNN with BERT embeddings when trained on Norwegian and tested on other languages. The result is also shown in figure 6.12.

The model performs very well on the Norwegian data, an experiment which is discussed in detail in section 6.4, but its performance on both English and Finnish is abysmal. Its performance on English is overall a little better than on Finnish, but still very far away from the Norwegian results.
It is probably a factor here that English and Norwegian have some words and grammar in common, as they are both considered Germanic languages, while Finnish belongs to a completely different language family.

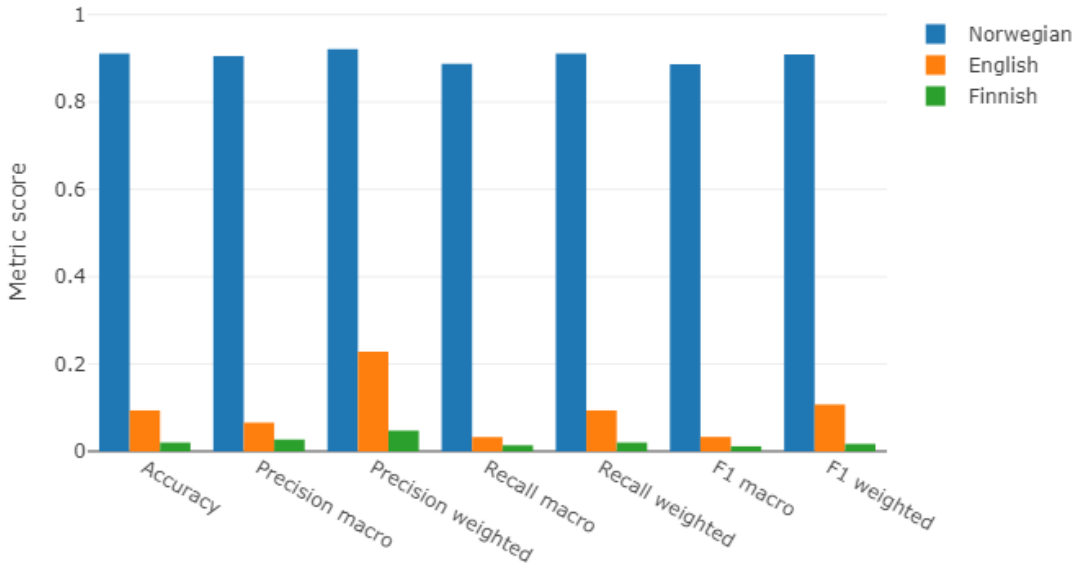


Figure 6.12: The performance of FFNN with BERT embeddings when trained on Norwegian and tested on other languages. The result is also shown in table 6.8.

Surprisingly, the weighted precision of the classification on English is much higher than the other metrics, including the macro precision. This might indicate that the model performs very well on some comparatively frequent classes, and perhaps those classes typically contains examples with words that are similar or identical in Norwegian and English.

There may be several underlying reasons for why there seems to be very little transferable learning from Norwegian to the other languages. Although BERT supports a lot of languages, it might be that the resulting embeddings are good for representing words within a single language, but that the relationship between the embeddings breaks down when there are several languages involved. However, it might be that the relationship between embeddings is something that can be learned, which requires that the languages are all present in the training data for the classification model.
It is also worth noting that BERT is trained on articles from Wikipedia, not on chatlogs or financial documents, so perhaps the result would have been different if it had been trained on documents that are closer related to the banking and customer support domain.

An important reason to consider here is the work that DNB has performed on its datasets.

They focus on improving the Norwegian version, as that is the main language of its customers that access the support chat, and so the Finnish and English examples are not necessarily similar to the Norwegian ones. This makes the task doubly difficult, as not only is the language itself different, the content of the examples themselves are also probably different. Note how much worse the FFNN model with BERT embeddings performs on the original test data from the chatbot, although the test language is only Norwegian. It might therefore be the case that it is not the BERT embeddings themselves that are at fault here, but rather the nature of the examples themselves. However, the results here are not sufficient to conclude that this is the only reason for the low metric scores of this experiment.

6.6.3 Utilizing the hierarchy

All the other experiments use the flat approach to classification, which means that they do not take the class hierarchy into account, and instead considers every class as a possibility when classifying an example. This experiment uses the local approach, which consists of training a classifier for every non-leaf node in the hierarchy that classifies which child node the experiment belongs to. A more detailed description of the approach can be found in section 2.3.
The experiment uses a Naïve Bayes model for each classifier, and its results, as well as a comparison to the flat approach using Naïve Bayes, can be found in table 6.9 and in figure 6.13.

Evaluation metric	Average calculation	Flat approach	Local aproach
Accuracy	-	0.516	0.579
Precision	Macro	0.567	0.592
	Weighted	0.668	0.645
Recall	Macro	0.337	0.477
	Weighted	0.516	0.579
F1	Macro	0.396	0.470
	Weighted	0.507	0.539

Table 6.9: The performance of Naïve Bayes when using the flat approach to hierarchical classification versus the local approach. The results are also shown in figure 6.13.

The results show overall a slight increase in all metrics except the weighted precision. The metric with the largest increase is the macro average of the recall, although the increase in the weighted average of recall does not increase as much. This means that classes with relatively few examples have a higher recall than before, but that the benefit to the other classes is not as pronounced. This is very likely the effect of the local approach on the amount of data available for each class at the time of training. The classifier at a node has to decide which of its child nodes the classification should continue through, or if the classification should terminate with the current class. The set of examples for each child node is expanded with all the examples for the child node's descendants as well, which

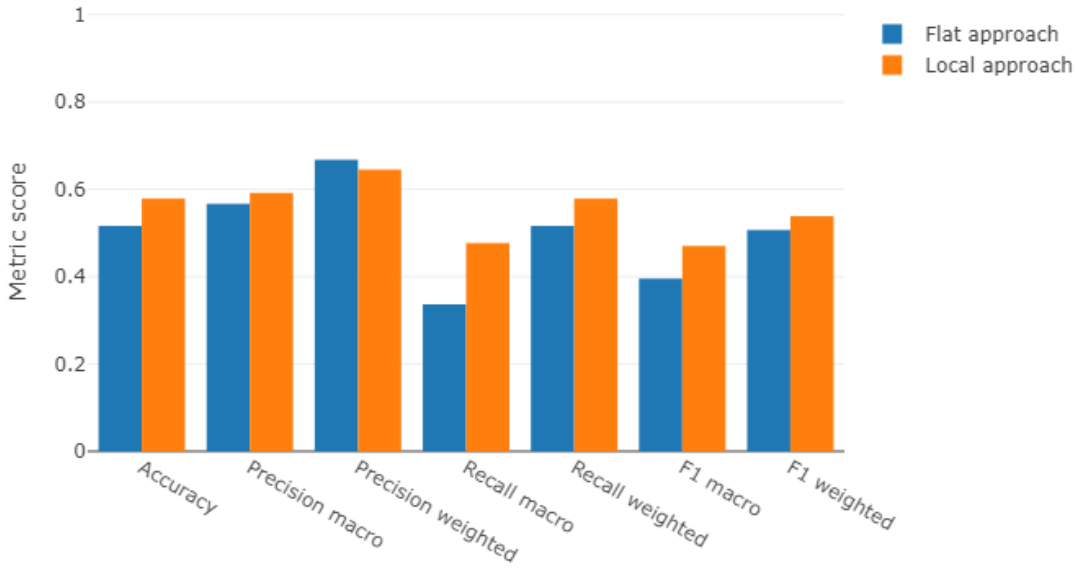


Figure 6.13: The performance of Naïve Bayes when using the flat approach to hierarchical classification versus the local approach. The results are also shown in table 6.9.

likely improves the result for classes that had too little data in the flat approach.
However, note that around 80% of the classes are leaf nodes, and a class is located on average at level 3 in the tree hierarchy. The shallow and wide class hierarchy means that the classifier at each node has to decide between a lot of different classes, and there are few steps in the classification before the it terminates, so the possible improvement by using the local approach instead of the flat approach is likely quite small.

Figure 6.14 shows the accuracy for each main category when classifying using either the flat or the local approach. Most categories show a small improvement when using the local instead of the flat approach, but there are some exceptions. The category *Generelle spørsmål* shows a particularly large improvement, and interestingly that category has a lot of descendants compared to its total number of examples, as shown in figures 4.4 and 4.4. It might be that these properties mean that the benefits of utilizing the hierarchy are particularly beneficial for this category, in this case the benefit of exploiting the information inherent in the hierarchy in the absence of sufficient examples for a flat approach.

Also worth noting is the fact that the categories with little to negative improvement in the accuracy are all those with very few examples in total and very few descendants. It is

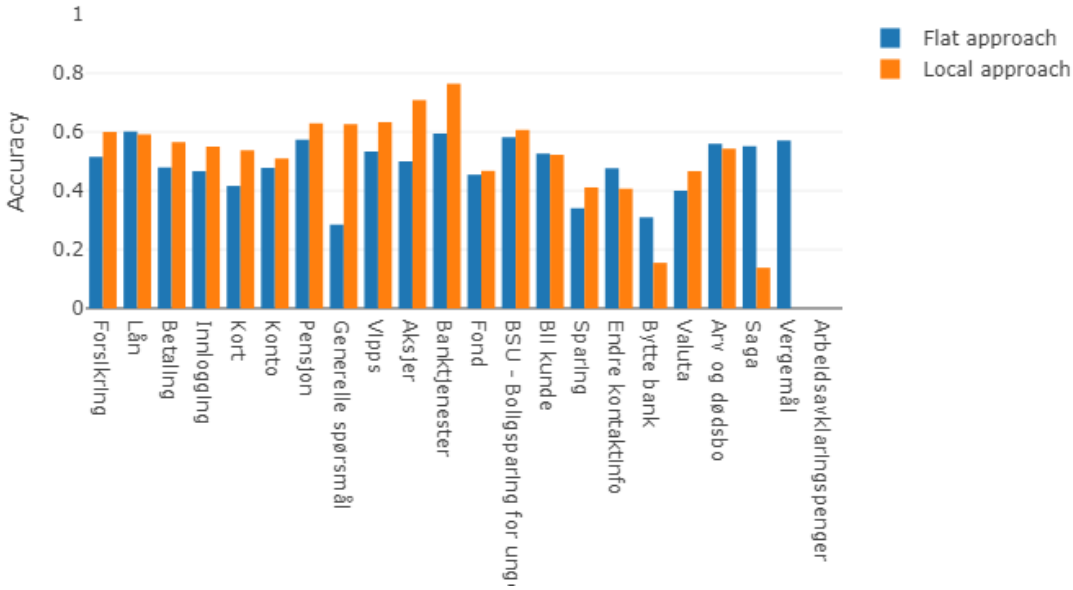


Figure 6.14: Comparison of the accuracy of Naïve Bayes for the main class categories using the flat approach to hierarchical classification versus the local approach. The categories are sorted according to the total number of examples in each.

probably the case that those categories are disadvantaged at the first classification node, i.e. the first level of the class hierarchy, as the larger categories have more data and are thus more likely choices for the classifier. There is in essence a very large class imbalance at the first level, which biases the final classification against those categories. It is therefore highly likely that the local approach would work better if the class hierarchy itself was better balanced, i.e. with more similar number of descendants and number of examples for the different branches of the tree that makes up the hierarchy.

The small improvement when using the local instead of the flat approach comes at a high cost. It needs more time for training, as there are several models that need to be trained, one for each non-leaf node. Additional time is also needed when the class of an example is predicted, as there are often several models that need to classify it before the algorithm terminates at the correct class in the hierarchy. Additional memory is also needed for storing all the models and the hierarchy structure that connects them, which also makes the implementation much more complex. It is therefore doubtful that the small improvement that results from using the local approach is sufficient to justify its use for the classification task in this thesis. It might however be different for a hierarchical classification problem

with a deeper hierarchy.

6.7 Final reflections

A more complex model generally means that more data is required to train it. However, a too simple model is likely to not be able to capture and learn all the nuances of the data well enough. The FFNN architecture itself, used in this experiment and the one in section 6.3.1, is very simple, with just one hidden layer and dropout applied on the connections between it and the output layer. A closer description of the entire model can be found in section 5.5.2. One might therefore consider the complexity of the model to be in the embeddings themselves, and consider those embeddings an integral part of the model.

BERT is clearly a more complex model than word2vec, as the former generates contextualized word embeddings for any input, and the latter just has static embedding for a word in its vocabulary once its training is finished, regardless of the context of the word. Increased complexity is not always beneficial, but it does allow the embeddings to potentially capture more information than it would otherwise have been able to.

BERT has some additional advantages over word2vec beyond its performance in the classification due to its properties. It has support for several languages and can generate an embedding from any word, while the word2vec embeddings used in this thesis are restricted to a single language and can only generate embeddings for words found in its vocabulary. Additionally, BERT can easily be finetuned to generate better embeddings for the domain or language if provided with enough related documents and computing power, while the pretrained word2vec embeddings are provided as a static model.

Chapter 7

Conclusion

This thesis has explored the use of pretrained word embeddings for classifying mainly short Norwegian texts, where both the classes and the data itself originate in the customer support chatbot solution of a bank. The main experiments compare different models and embeddings, namely the static word2vec embeddings of length $|v| = 100$ and $|v| = 300$, and embeddings extracted from the last layer of the BERT model. Neither the word2vec embeddings nor the BERT model have been trained beyond their pretrained state, only the classification models themselves.

An FFNN model with BERT embeddings as input performs the best of all the models in the experiments, although LSTM with the shorter word2vec embeddings comes a close second. However, the LSTM model with longer word2vec embeddings performs worse than the LSTM model with shorter embeddings. The FFNN model with word2vec model performs the worst of all the models using embeddings, but still yields a significant improvement over a Naïve Bayes model.

The results demonstrate that using the word2vec embeddings may at best perform on par with BERT embeddings, given good choices of model and embedding dimensionality. BERT embeddings performs well even when using a very simple classification model as opposed to the complexity of the LSTM. Furthermore, the BERT embeddings are more versatile and easier to continue training than the word2vec embeddings. One may therefore conclude that the BERT embeddings are superior, and that its results are more robust than the results of word2vec embeddings.

The multilingual aspect of BERT was tested in a separate experiment, where an FFNN model was trained with BERT embeddings exclusively on Norwegian data, and subsequently tested on English and Finnish data. The model was able to produce some correct predictions on the English examples, but almost close to none correct predictions on Finnish examples. This indicates that it is not sufficient to train BERT embeddings on one language only, if its purpose is to be used on other languages later on, although the

BERT model is able to generate embeddings for languages on which it has not priory been exposed to, apart from its pretraining. However, there are aspects of the data that complicate the situation, such as potential differences in the content and quality of the messages, which means that one cannot make a conclusive judgement on the multilingual ability of BERT.

A secondary experiment used the local approach to hierarchical classification, as it consisted of training a multi-class Naïve Bayes classifier for every non-leaf node in the class hierarchy. This resulted in a small overall improvement compared to the Naïve Bayes classifier used in the main experiment, which classified for all classes at once, i.e. a flat approach. The model performance was significantly improved for some of the class categories when using a local rather than a flat approach, but all the models combining word embeddings and a flat approach outperformed the local approach. Furthermore, the local approach have several drawbacks as it requires more time both to train the model and to generate predictions. It is more complex to implement and requires more memory to store the model.

Hence, the conclusion is that the flat approach using BERT embeddings with an FFNN model, alternatively word2vec embeddings with an LSTM model, performs the best of all the models tested in this thesis at correctly classifying short Norwegian texts.

7.1 Further work

There are several interesting avenues of research that arise from the results of this thesis, and some of those are listed below.

Train domain-specific embeddings The embeddings used in this thesis, both word2vec and BERT, are trained on quite general datasets. While the best results are very good, they may become even better with embeddings trained on domain-specific documents. This is typically done through unsupervised learning, which means that the documents do not need to be annotated, but requires large amounts of data.

Finetuning BERT A more accessible alternative to training domain-specific embeddings from scratch is to finetune an already trained model. The BERT model has been published by its authors in such a way that it is possible to finetune the model by continue training the pretrained model. This also requires potentially large amounts of data, but likely requires less than training from scratch before the results may start to improve.

Finetuning hyperparameters and model architecture The experiments in this thesis did not include finetuning of the hyperparameters or the model architectures, and so it is likely that the results may be improved at least somewhat by performing this.

Different types of BERT embeddings The BERT embeddings used in this thesis were simply an average of the word embeddings for each token in an example, where the word embeddings were extracted from the output of the model's final layer.

There may be that other ways of combining these, such as taking the maximum or minimum, or a mix of these as in [2], may improve the result. Embeddings from other layers or combinations of layers may also be of interest.

Further testing of the multilingual aspect of BERT The multilingual experiment in this thesis did not show particularly good results when testing on previously unseen languages, but it might be possible to make the classification model multilingual if it is trained on the languages as well. Such a model would be very versatile and useful, able to handle different languages in the input without first needing to determine the language as long as it supports it.

Embeddings with bi-directional LSTMs Most experiments in this thesis used a simple feed-forward neural network, but the best results for word2vec embeddings were with a uni-directional LSTM. It may be possible to improve the results of this experiment by using a bi-directional LSTM architecture instead, or combine BERT embeddings with either uni- or bi-directional LSTMs.

Bibliography

[1] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*. Pearson Education Limited, 2 edition, 2011.

[2] Georgios Balikas and Massih-Reza Amini. An empirical study on large scale text classification with skip-gram embeddings. *arXiv preprint arXiv:1606.06623*, 2016.

[3] Eivind Alexander Bergem. Document-level sentiment analysis for norwegian. Master's thesis, 2018.

[4] Jacob Devlin. Bert multilingual. `https://github.com/google-research/bert/blob/master/multilingual.md`. Accessed: 2019-06-01.

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

[6] John R Firth. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*, 1957.

[7] Yoav Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016.

[8] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. MIT press Cambridge, 2016.

[9] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

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

[11] Dan Jurafsky and James H Martin. Speech and language processing. `http://web.stanford.edu/~jurafsky/slp3/`, 2019. 3rd ed. draft, accesed 2019-02-01.

[12] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

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

[14] Ioannis Partalas, Aris Kosmopoulos, Nicolas Baskiotis, Thierry Artieres, George Paliouras, Eric Gaussier, Ion Androutsopoulos, Massih-Reza Amini, and Patrick Galinari. Lshtc: A benchmark for large-scale text classification. *arXiv preprint arXiv:1503.08581*, 2015.

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

[16] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[17] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/research-covers/languageunsupervised/language understanding paper. pdf*, 2018.

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

[19] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[20] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

[21] Carlos N Silla and Alex A Freitas. A survey of hierarchical classification across different application domains. *Data Mining and Knowledge Discovery*, 22(1-2):31–72, 2011.

[22] Cathrine Stadsnes. Evaluating semantic vectors for norwegian. Master's thesis, 2018.

[23] Roger Alan Stein, Patricia A Jaques, and João Francisco Valiati. An analysis of hierarchical text classification using word embeddings. *Information Sciences*, 471:216–232, 2019.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

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

Appendix A

Confusion matrices

This appendix includes the confusion matrices for some of the experiments presented and discussed in chapter 6. The matrices report the result of one iteration of the cross-validation, i.e. the result of training on nine out of ten parts of the dataset, and testing on the remaining part. The split of the dataset into these parts is a stratified split, which means that each class has as close as possible to an equal number of examples in each part.

	Forsikring	Lån	Betaling	Innlogging	Kort	Konto	Pensjon	Generelle spørsmål	Vipps	Aksjer	Bank-tjenester	Fond	BSU - Boligsparing for ungdom	BSU - Bli kunde	Sparing	Endre kontaktinfo	Bytte bank	Valuta	Arv og dødsbo	Saga	Vergemål	Arbeids-avklarings-penger
Forsikring	4941	117	27	4	9	78	36	0	1	2	0	0	35	5	0	2	0	0	0	0	0	0
Lån	382	3359	33	4	5	25	67	0	1	1	0	1	37	5	0	0	0	0	0	0	0	0
Betaling	354	89	1153	8	22	50	51	0	19	4	1	0	71	14	0	0	0	0	0	0	0	0
Innlogging	261	38	90	1020	18	66	34	0	32	2	3	1	35	6	0	3	1	0	2	0	0	0
Kort	258	100	43	18	1055	38	33	0	11	2	7	0	51	1	0	2	0	0	0	0	0	0
Konto	221	72	35	1	5	1033	63	0	3	3	0	1	89	8	0	0	0	0	3	0	0	0
Pensjon	94	55	6	2	0	13	1080	0	0	1	0	0	10	0	0	0	0	0	0	0	0	0
Generelle spørsmål	247	32	14	8	17	17	21	175	2	0	8	1	42	12	0	0	0	0	1	0	0	0
Vipps	51	14	13	2	3	7	24	0	294	0	0	0	11	1	0	0	0	0	0	0	0	0
Aksjer	77	27	15	8	0	13	8	0	0	246	0	0	14	0	0	0	0	0	0	0	0	0
Bank-tjenester	70	25	5	0	1	6	5	0	1	0	228	1	6	0	0	0	0	0	0	0	0	0
Fond	70	14	16	0	1	11	7	0	0	3	0	176	8	2	0	0	0	0	0	0	0	0
BSU - Boligsparing for ungdom	15	22	0	3	0	1	10	0	0	0	0	0	238	0	0	0	0	0	0	0	0	0
Bli kunde	44	9	1	0	1	22	4	0	0	2	0	0	8	118	0	0	0	0	0	0	0	0
Sparing	37	9	3	3	1	7	20	0	0	3	0	3	7	0	48	0	0	0	0	0	0	0
Endre kontaktinfo	16	4	2	1	2	1	12	0	1	1	2	0	3	0	0	41	0	0	0	0	0	0
Bytte bank	13	18	5	0	1	11	3	0	0	0	0	0	3	1	0	0	29	0	0	0	0	0
Valuta	15	4	0	0	0	1	3	0	0	5	0	0	3	0	0	0	0	29	0	0	0	0
Arv og dødsbo	13	4	2	0	0	1	2	0	0	0	3	0	1	0	0	0	0	0	33	0	0	0
Saga	5	2	0	0	1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	18	0	0
Vergemål	2	0	0	0	0	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	8	0
Arbeids-avklarings-penger	3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table A.1: Confusion matrix for classification using Naïve Bayes. The results of the experiment are described in section 6.2.2.

	Forsikring	Lån	Betaling	Innlogging	Kort	Konto	Pensjon	Generelle spørsmål	Vipps	Aksjer	Bank-tjenester	Fond	BSU - Boligsparing for ungdom	Bli kunde	Sparing	Endre kontaktinfo	Bytte bank	Valuta	Arv og dødsbo	Saga	Vergemål	Arbeids-avklarings-penger
Forsikring	4747	148	98	11	20	121	37	2	14	19	9	5	5	6	11	1	0	0	3	0	0	0
Lån	85	3605	61	11	12	70	21	4	4	9	8	1	13	2	10	1	2	0	1	0	0	0
Betaling	67	65	1354	23	27	167	29	4	21	22	10	2	8	2	30	0	1	3	1	0	0	0
Innlogging	21	31	59	1396	17	32	6	1	31	3	2	3	6	1	2	1	0	0	0	0	0	0
Kort	27	44	44	28	1365	56	14	4	7	3	4	4	5	4	6	1	1	2	0	0	0	0
Konto	31	58	61	7	13	1267	17	4	6	28	5	2	14	2	17	2	0	0	3	0	0	0
Pensjon	21	20	14	4	1	59	1094	3	3	14	1	3	18	1	5	0	0	0	0	0	0	0
Generelle spørsmål	2	4	14	4	5	5	7	543	1	2	7	0	0	0	2	0	0	1	0	0	0	0
Vipps	13	9	46	26	11	32	11	0	260	5	2	0	1	0	3	0	0	1	0	0	0	0
Aksjer	5	22	13	6	0	26	17	0	0	305	0	3	10	0	1	0	0	0	0	0	0	0
Banktjenester	5	6	6	5	1	3	2	1	0	1	315	0	1	1	0	0	0	1	0	0	0	0
Fond	4	5	6	3	0	6	7	2	2	6	1	259	1	1	4	0	0	0	1	0	0	0
BSU - Boligsparing for ungdom	6	11	6	11	5	32	8	1	1	4	1	1	194	0	8	0	0	0	0	0	0	0
Bli kunde	3	9	6	3	1	11	3	0	2	0	1	2	2	163	1	0	2	0	0	0	0	0
Sparing	2	12	3	1	2	8	4	2	0	0	0	1	0	0	106	0	0	0	0	0	0	0
Endre kontaktinfo	2	1	0	6	3	0	0	0	0	0	0	0	0	0	0	74	0	0	0	0	0	0
Bytte bank	3	4	2	1	1	4	0	0	0	0	0	0	2	1	0	0	66	0	0	0	0	0
Valuta	1	4	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	52	0	0	0	0
Arv og dødsbo	0	1	0	0	1	2	0	0	0	2	0	0	0	1	1	0	0	0	51	0	0	0
Saga	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	29	0	0
Vergemål	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	11	0
Arbeids-avklarings-penger	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4

Table A.2: Confusion matrix for classification using word2vec embeddings of length 100 together with FFNN. The results of the experiment are discussed in section 6.3.1

	Forsikring	Lån	Betaling	Innlogging	Kort	Konto	Pensjon	Generelle spørsmål	Vipps	Aksjer	Bank-tjenester	Fond	BSU - Boligsparing for ungdom	Bli kunde	Sparing	Endre kontaktinfo	Bytte bank	Valuta	Arv og dødsbo	Saga	Vergemål	Arbeids-avklarings-penger
Forsikring	5133	34	27	2	1	7	1	0	13	14	7	2	2	8	5	0	0	0	1	0	0	0
Lån	55	3769	16	6	7	13	7	5	9	12	1	3	2	9	5	0	0	0	0	0	1	0
Betaling	36	17	1598	8	12	13	2	8	10	16	4	2	8	99	0	0	3	0	0	0	0	0
Innlogging	5	10	5	1552	9	5	1	4	5	9	2	1	1	2	1	0	0	0	0	0	0	0
Kort	10	9	17	10	1523	11	7	2	4	5	2	3	1	8	3	3	1	0	0	0	0	0
Konto	10	6	11	2	11	1446	4	0	13	16	3	6	5	1	2	1	0	0	0	0	0	0
Pensjon	2	1	2	1	1	4	1241	0	2	0	0	2	2	1	1	1	0	0	0	0	0	0
Generelle spørsmål	1	0	7	3	1	1	1	571	1	6	1	1	1	2	0	0	0	0	0	0	0	0
Vipps	6	3	6	6	3	12	2	0	375	6	0	0	0	1	0	0	0	0	0	0	0	0
Aksjer	5	10	2	1	2	8	5	0	2	364	0	2	1	4	2	0	0	0	0	0	0	0
Banktjenester	1	4	8	0	2	1	0	2	1	3	323	1	0	0	0	0	2	0	0	0	0	0
Fond	1	3	2	2	1	3	0	0	1	1	0	291	1	0	1	1	0	0	0	0	0	0
BSU - Boligsparing for ungdom	1	4	2	2	3	7	4	0	0	1	1	0	264	0	0	0	0	0	0	0	0	0
Bli kunde	4	5	4	0	0	0	0	3	0	0	0	0	1	192	0	0	0	0	0	0	0	0
Sparing	1	3	0	0	0	3	2	0	0	0	0	0	0	0	132	0	0	0	0	0	0	0
Endre kontaktinfo	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	85	0	0	0	0	0	0
Bytte bank	1	0	1	0	0	1	0	0	0	1	0	0	0	0	0	1	79	0	0	0	0	0
Valuta	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	58	0	0	0	0
Arv og dødsbo	0	0	0	2	1	0	0	0	0	1	0	0	0	0	0	0	0	0	55	0	0	0
Saga	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	29	0	0
Vergemål	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	13	0
Arbeids-avklarings-penger	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4

Table A.3: Confusion matrix for classification using word2vec embeddings of length 100 together with LSTM. The results of the experiment are discussed in section 6.3.2

	Forsikring	Lån	Betaling	Innlogging	Kort	Konto	Pensjon	Generelle spørsmål	Vipps	Aksjer	Bank-tjenester	Fond	BSU - Boligsparing for ungdom	Bli kunde	Sparing	Endre kontaktinfo	Bytte bank	Valuta	Arv og dødsbo	Saga	Vergemål	Arbeids-avklarings-penger
Forsikring	5217	15	5	2	2	2	2	2	0	2	2	0	1	1	2	1	1	0	0	0	0	0
Lån	6	3886	6	1	2	2	5	0	0	0	5	3	1	1	2	0	0	0	0	0	0	0
Betaling	5	15	1774	13	6	5	7	1	0	0	2	0	2	0	3	1	0	0	0	1	0	1
Innlogging	2	9	10	1566	7	5	2	1	1	0	1	0	0	1	2	2	1	2	0	0	0	0
Kort	5	5	13	3	1571	6	5	3	1	1	1	0	0	0	1	4	0	0	0	0	0	0
Konto	2	8	13	7	6	1485	2	1	0	3	1	1	3	1	2	1	1	0	0	0	0	0
Pensjon	0	3	1	3	1	1	1250	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
Generelle spørsmål	4	7	6	5	2	2	2	559	0	0	2	0	0	4	1	0	0	0	0	1	0	0
Vipps	1	0	2	1	0	0	1	0	413	1	0	0	0	0	1	0	0	0	0	0	0	0
Aksjer	1	1	2	0	1	2	0	0	0	398	0	1	0	1	0	0	0	0	0	0	0	1
Bank-tjenester	1	2	1	1	1	0	0	4	0	2	332	0	0	2	1	1	0	0	0	0	0	0
Fond	1	1	1	1	1	1	3	0	0	0	0	298	0	0	1	0	0	0	0	0	0	0
BSU - Boligsparing for ungdom	2	1	0	0	0	1	0	0	1	0	0	0	284	0	0	0	0	0	0	0	0	0
Bli kunde	1	2	0	1	1	3	0	1	0	0	0	1	0	195	0	1	3	0	0	0	0	0
Sparing	0	1	0	0	0	0	2	0	0	0	0	0	0	0	138	0	0	0	0	0	0	0
Endre kontaktinfo	0	0	0	1	2	0	0	0	0	0	0	0	0	0	0	83	0	0	0	0	0	0
Bytte bank	0	0	0	0	0	1	0	0	0	0	2	0	0	0	1	1	78	1	0	0	0	0
Valuta	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	57	0	0	0	0
Arv og dødsbo	0	0	0	0	0	1	2	0	0	0	0	0	0	0	0	0	0	0	56	0	0	0
Saga	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	29	0	0
Vergemål	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	11	2
Arbeids-avklarings-penger	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4

Table A.4: Confusion matrix for classification using BERT embeddings together with FFNN. The results of the experiment are discussed in section 6.4

Appendix B

Translation of names of main class categories

Original name	English translation
Aksjer	Stocks
Arbeidsavklaringspenger	Work assessment allowance
Arv og dødsbo	Inheritance and estate
BSU - Boligsparing for ungdom	BSU - Home savings scheme for young people
Banktjenester	Bank services
Betaling	Payment
Bli kunde	Become a customer
Bytte bank	Change bank
Endre kontaktinfo	Change contact information
Fond	Fund
Forsikring	Insurance
Generelle spørsmal	General questions
Innlogging	Login
Konto	Account
Kort	Card
Lån	Loan
Pensjon	Pension
Saga	Saga (a product specific to DNB)
Sparing	Savings
Valuta	Currency
Vergemål	Guardianship
Vipps	Vipps (a mobile payment application)

Sigrid Lofthus Bjørndal

Exploring pretrained word embeddings for multi-class text classification in Norwegian

NTNU
Norwegian University of
Science and Technology

DNB