Neural methods in political classification

using the Talk of Norway dataset

Eivind Hestetun Thomassen

Thesis submitted for the degree of
Master in Language and Communication
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2019

Neural methods in political classification

using the Talk of Norway dataset

Eivind Hestetun Thomassen

Neural methods in political classification

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

This thesis examines the use of neural network methods for classifying parliamentary speeches by textual content. It introduces previous work in the field using traditional methods, and discusses aspects of the Talk of Norway (ToN) corpus, while covering the theoretical background behind neural methods in detail. Detailed analyses are given for various architectures applied to the task. We finally describe how we used an ensemble classifier to improve results.

Acknowledgements

I would like to thank my supervisors, Taraka Rama Kasicheyanula and Erik Velldal, for their guidance and patience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Document classification is a task within the field of natural language processing (NLP) which seeks to automatically identify and assign a label to a document. It is an instance of "supervised" machine learning. Supervised machine learning requires a training corpus of labeled documents. This thesis discusses document classification as it pertains to political speech. The documents are in our case speeches at the Norwegian parliament, and the labels we use are the political parties to which the speakers belong. Our goal is therefore to produce a model which is able to assign a political party label to a political speech.

The central question this thesis examines is whether political party affiliation can be determined based purely on text. To this end experiments are performed on the Talk of Norway (ToN) dataset, which contains 250,373 speeches made at the Norwegian parliament between 1998 and 2016. Norway has a multi-party system, meaning that assigning party labels becomes a multi-class classification task. The different parties naturally have different levels of electoral support and therefore representation in the legislature, meaning that the distribution of speeches into different classes is not uniform.

The ToN dataset also contains a great deal of meta-data, but for this thesis we will focus only on the textual content of individual speeches, without any contextual information. Previous work has been done on the ToN corpus using traditional machine classification methods, but no results have been reported using neural methods.

What can we expect a classifier trained on political speeches to have learned? As input we use the words that politicians use in their parliamentary speeches, and from this we try to output we the speaker's political party affiliation. At some level, the task may be viewed as merely an investigation into correlations between particular words and membership in a given political party. One may, however, hope to achieve something a bit deeper than this—a connection to the ideology which those words imply.

Classification of political speech is a problem area which has been examined to some extent before. Chapter 3 summarizes much of the work which has been done in this field. The chapter describes works that deal

with classification of political speeches in the legislatures of the United States, Canada, United Kingdom as well as the European parliament. One aspect which is common to all these approaches is that they use traditional machine learning methods, as opposed to methods based on neural networks. This thesis, as far as we are aware, marks the first large-scale attempt to apply neural machine learning methods to the field of parliamentary speech.

Chapter 2 introduces the ToN corpus in more detail. Section 2.1 discusses experiments which have previously been performed on this corpus using traditional machine learning methods.

In chapter 4 we discuss the use of neural networks in the realm of NLP. Section 4.1 presents the history as well as basic feed-forward models and simpler methods, while sections 4.2 and 4.3 introduce somewhat more advanced neural architectures, convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Section 4.5 discusses the representation of text in dense vectors using word embeddings, while 4.2 and 4.3 give a basic overview of two common architectural elements, namely CNNs and RNNs. Section 4.4 gives an overview of the many different functions common to neural networks.

Chapter 5 lays forth the groundwork that we performed to accommodate the experiments we ran on the ToN corpus. We describe in detail the method we used to split the corpus into training, development and test sets to ensure that each split would contain an equivalent distribution of speeches. This thesis has involved a great deal of large-scale experimentation and tuning of model hyperparameters. Section 5.4 introduces the Abel high-performance computing cluster, which provided the computational power to run all the experiments. Abel enabled testing of a wide variety of different neural network architectures, which were defined using the Keras neural network library. Keras is discussed in some detail in section 5.3.

Results for all the model architectures examined are presented in detail, with in-depth analyses of the effects given by using different configurations for individual hyperparameters. Experiments using simple models are detailed in chapter 6. Section 6.1 presents a feed-forward architecture using bag-of-words (BOW) representations. Here we discuss for instance the effect of vocabulary size for BOW representations, and try to reason about the low relative performance we see for the adaptive moment estimation (Adam) optimizer for this architecture. Section 6.2 introduces averaged word embeddings as input representations. Among other hyperparameters, we examine different ways of generating such representations in terms of the embeddings used, how to consider individual tokens and dimensionality.

Chapter 7 introduces more advanced architectures. Section 7.1 deals with CNNs and contains, inter alia, analyses of the effects of different window sizes and number of filters. Subsection 7.1.3 quantifies the effect of non-determinism in the experiment results.

Section 7.2 presents results using a RNN architecture, specifically the long short-term memory (LSTM) variant. and presents results for these, along with discussion about certain related pitfalls and considerations.

Here we discuss difficulties using recurrent layers with a high number of parameters, and we explore methods for generating fixed-size representations from recurrent layer outputs, including max pooling and self-attention.

The findings in brief show that the different classifying architectures all gave models with remarkably similar results, but that combining these into an ensemble classifier resulted in a classifier that significantly surpassed each of its constituent models.

Chapter 2

The Talk of Norway corpus

This chapter discusses the Talk of Norway (ToN) corpus (Lapponi et al. 2018), putting it in the context of the classification task. Later, section 2.1, discusses some experiments which have previously been performed on it. The ToN corpus consists of 250,373 speeches delivered at the Norwegian parliament (Stortinget) in sessions spanning from 1998 to 2016. The speeches are associated with meta-data including:

- information about the speaker:
  - name
  - party affiliation, as well as:
    - number of seats currently held by that party
    - whether that party is in position or opposition
  - constituency which the speaker represents
  - gender
  - membership of parliamentary committees
  - role, i.e., member of parliament, minister etc.
- debate under which the speech was held
- cabinet at the time

In addition, the texts of the speeches themselves have been pre-processed and annotated using the Oslo-Bergen Tagger (OBT) (Johannessen et al. 2012). The original version of OBT was released in 1996 as a rule-based tagger using the Constraint Grammar format. While it still uses rules primarily, in later years it has been enhanced with a hidden Markov model, that makes decisions that remove ambiguity left behind by the rule-based module. This processing segments speeches into sentences, which are then split into series of tokens. Each of these tokens is then annotated with the original word form, its lemma, part-of-speech tag and inferred morphological features.

Norway has a multi-party parliamentary system with representative representation. This leads to a multitude of parties being represented

Party/source	abbreviation	# speeches	# tokens
Other (mainly president)	-	72,693	2,590,448
Labour Party	Ap	43,483	16,008,420
Conservatives	H	32,945	11,481,762
Progress Party	Frp	30,217	9,729,435
Socialist Left Party	SV	19,941	7,218,136
Center Party	Sp	18,255	5,874,381
Christian Democrats	KrF	19,720	6,653,088
Liberal Party	V	11,579	3,830,095
Green Party	MDG	508	153,834
Coast Party	Kp	492	128,709
Non-partisan	-	409	97,001
Independent	-	131	38,284

Table 2.1: Statistics for the Talk of Norway corpus

in each parliamentary period, and no single party has held an outright majority since 1961. The cabinets reflect this and mostly consist of coalition governments which are stable for the entire four-year parliamentary period. In the period represented in the corpus, the one exception to this is the single-party majority government of Torbjørn Jagland, which replaced the preceding cabinet for the last year of that parliamentary period. Statistics for the corpus are presented in Table 2.1. The corpus may be accessed on-line.[1]

The speeches included in the corpus also vary greatly in terms of content. The amount of signal contained in a speech that can be connected to a political party is correspondingly variant. For a taste of what this entails, we reproduce a random speech from the training corpus, speech no. 17995:

> La meg si det på denne måten: Det statlig styring og eierskap i denne sammenheng innebærer, er en inngjerding av den teigen som private markedsinteresser skal få lov til å bevege seg fritt innenfor, og vi har da slått ned noen gjerdestolper som skal holde DnB og Kreditkassen fra hverandre, i Venstre. Venstres standpunkt behøver ikke å være det store problemet for denne sal. Snarere tvert om, vi har – om ikke som de eneste – i hvert fall gitt klart uttrykk for hva vi mener. Det jeg mener er problemet for denne sal, er: Hva vil egentlig Arbeiderpartiet? Frøiland forstår ikke Venstre. Jeg har store problemer med å forstå i hvilken retning Arbeiderpartiet vil gå. Det er det største problemet for denne sal, tror jeg.

This speech contains both direct reference to the party of the speaker as well as content describing the speaker's ideological standpoint. It should

1. https://github.com/ltgoslo/talk-of-norway

therefore be relatively easy to classify, assuming that the classifier is able to pick up on such indicators. Other speeches appear to be rather more generic, such as speech no. 164028:

> Jeg kjenner ikke dette eksempelet konkret, men håndteringen av hva man gjør når slike situasjoner oppstår, skjer lokalt i hvert enkelt tilfelle, og man skal selvfølgelig vurdere det ut fra å ta mest mulig hensyn til kandidaten. Jeg vet ikke hva som ligger bak hvordan akkurat dette skjedde, men det som er en mulighet lokalt, er at man sender den opprinnelige besvarelsen hvis den finnes hos skolen.

Standing by itself, this speech does not appear to contain any indication of political standpoint or ideology. The meta-data (which we will not be feeding to our classification) reveals that this is a cabinet reply to a question from the opposition in a question hour session. Without any information on the context in which this speech is made, one would be hard-pressed to find any indication of which party the speaker belongs to.

2.1 Previous Work on the ToN corpus

The paper which originally presented the ToN corpus included a preliminary experiment which involved training a support vector machine (SVM) classifier (Lapponi et al. 2018) for political party classification. Lapponi et al. (2018) set out to investigate to what extent a classifier trained on a representative sample of parliamentary speeches would be able to assign the correct party label to a speech drawn from the same population.

Lapponi et al. (2018) removed speeches which lacked an associated political party, speeches consisting of less than 200 tokens and those from parties which were not represented during all parliamentary periods. The abridged data were then split into six folds, each corresponding to a particular cabinet period, to facilitate cross-validation.

After removing stop words, speeches were then transformed into TF-IDF-weighted (term frequency inverse document frequency) vectors, making use of token *n*-grams, lemma *n*-grams and part-of-speech tags generated by OBT. For some experiments, the vectors generated from these linguistic features were further enhanced with auxiliary non-linguistic features such as speaker, gender of the speaker, county the speaker represented, type of debate, keywords describing the debate, committee name and type of case being debated.

The SVM classifier was trained on speeches including these meta-data, using the Linear SVM package in Scikit-learn (Pedregosa et al. 2011). The regularization parameter, which modifies the loss function of the classifier in order to strike balance between training accuracy and generalization ability, was tuned empirically using Scikit-learn's grid search functionality (the optimal value was consistently 1).

The results from their classifier are shown in table 2.2. Baseline refers to majority-class assignment, i.e., assuming all speeches belonged

Party/source	P	R	F_1	accuracy
Socialist Left Party	0.578	0.490	0.531	-
Labour Party	0.471	0.624	0.537	-
Center Party	0.618	0.527	0.569	-
Christian Democrats	0.578	0.433	0.495	-
Liberal Party	**0.637**	0.351	0.452	-
Conservatives	0.503	0.485	0.494	-
Progress Party	0.603	**0.665**	**0.632**	-
Baseline	0.035	0.142	0.056	0.248
Macro	0.570	0.511	0.538	0.539

Table 2.2: Results from Lapponi et al. (2018)

to the Labor Party (Ap) class. Classifier performance is evaluated using different metrics. The most obvious is accuracy, which describes the overall proportion of correctly labelled instances. Beyond this, precision is defined as the proportion of instances assigned to a class that indeed belong to that class, while recall refers to the proportion of instances belonging to a class which were correctly assigned to that class. For most purposes we care about both recall and precision, so we also have the F_1 score, which is the harmonic mean of these, introducing a penalty when the two diverge. For evaluating the overall performance of a multi-class classifier, it is considered most prudent to calculate the macro average across classes, which weighs each class equally independently of size.

Speeches in the Norwegian parliament are transcribed in one of the two written variants of the language: Nynorsk and Bokmål. A preference for one or the other variant can be partly motivated by political views, and the distribution of speeches in either variant is not equal across party lines. The only parties with more than one percent of speeches in Nynorsk are Socialist Left Party (SV), Christian Democrat Party (KrF) and Center Party (Sp), with respectively 18, 19 and 33 percent of the speeches. Lapponi et al. (2018) voiced concern that a classifier might be driven by this aspect rather than speech content, but the F_1 scores for these parties does not appear to be higher, as one might assume would then be the case.

The SVM classifier performed best on the far-right Progress Party (Frp), suggesting that parties with a pronounced political profile were easier to classify. At the same time, the classifier's accuracy on speeches by this party decreased dramatically for the period where it was in government and not in opposition. The authors note that performance was higher for opposition parties in general and that parties which were part of the ruling cabinet of any given period were more likely to be misclassified as Ap. This is both the most common class and the party which has defined politics for much of the post-war era, so the classifier may have recognized it as inhabiting a sort of ideological center ground. Lapponi et al. (2018) also mentioned patterns explored in Hirst, Riabinin, and Graham (2010), which argued that a political ideology classifier could be prone to pick up

	system	SV	Ap	Sp	KrF	V	H	Frp	Macro	Acc
	maj. class	-	-	-	-	-	-	-	0.05	0.24
	meta only	0.19	0.38	0.36	0.31	0.24	0.16	0.34	0.28	0.30
unigram	stem	0.57	0.58	0.58	0.51	0.51	0.58	0.62	0.57	-
	token	0.65	0.65	0.66	0.61	0.62	0.65	0.69	0.65	-
	lemma	0.64	0.65	0.65	0.62	0.62	0.64	0.68	0.64	-
	lemma/pos	0.66	0.66	0.67	0.64	0.64	0.66	0.70	0.66	-
	+meta	0.69	0.69	0.72	0.68	0.69	0.68	0.73	0.70	-
n-gram	stem	0.63	0.66	0.65	0.60	0.61	0.65	0.68	0.65	-
	token	0.67	0.69	0.69	0.66	0.66	0.67	0.71	0.68	-
	lemma	0.68	0.69	0.69	0.66	0.67	0.68	0.72	0.69	-
	lemma/pos	0.69	0.70	0.71	0.67	0.69	0.69	0.73	0.70	-
	+meta	0.71	0.72	0.73	0.71	0.72	0.70	0.75	0.72	-

Table 2.3: Results from Lapponi (2019)

language patterns emerging from the dynamics of opposition and position parties. Hirst, Riabinin, and Graham (2010) will be further discussed in the following chapter.

Lapponi et al. (2018) noted that the size of the class did not appear to have an effect on performance; aside from Liberal Party (V) there appears to be no correlation between the number of speeches as seen in Table 2.1 and the performance as measured by F_1 score.

The work in Lapponi et al. (2018) was expanded upon as part of Lapponi (2019), from which Table 2.3 is reproduced. Whereas Lapponi et al. (2018) used the various cabinets as folds, cross-validation in this experiment used a random separation into ten folds. It also reported scores across various feature combinations, giving an overview of the contributions of textual preprocessing and meta-data. Performance for the most part increased with input complexity, and the most complex feature combination, *n*-grams of part-of-speech (POS) tagged lemmas and contextual meta-data such as the county of the speaker, gave a macro F_1 performance of 0.72. The experiments that follow in this thesis make no use of contextual meta-data, and are based only on tokens.

Chapter 3

Previous work

In this chapter we will discuss some of the research which has been published dealing with the automatic classification of legislative speech by political party membership. An early effort is described in Yu, Kaufmann, and Diermeier (2008). The expressed goal of this study was to recognize political ideology, and party membership was further used to define the target classes. Given that ideology was the aspect that Yu, Kaufmann, and Diermeier (2008) wanted the classifier to recognize, other factors needed to be isolated. Yu, Kaufmann, and Diermeier (2008) identified three such factors of which a classifier should ideally be able to perform independently:

- Person: The classifier should be able to recognize a political belief across different speakers, rather than just pick up individual speech patterns.
- Time: It also needs to be able to perform well on speeches made during different parliamentary periods than the one it is trained on.
- Issue: Lastly, it needs to recognize the ideology underlying the speech regardless of the particular issue which is being debated.

Yu, Kaufmann, and Diermeier (2008) used data from the US legislature, with all speeches by one speaker in a single year combined into a single document. In other words, they classified speakers rather than speeches, as opposed to Lapponi et al. (2018) and Lapponi (2019). A consequence of this decision is that the data were reduced to fewer, but larger, data points for the classifier to train upon and distinguish. The effect of this is likely to be quite great; on the one hand each element of classification is very data-rich, but on the other hand the number of samples is heavily limited. The authors experimented with various support vector machine (SVM) classifiers and two Naive Bayes classifiers, all based on a bag-of-words (BOW) approach. They obtained the best results with an SVM classifier using term frequency–inverse document frequency (TFIDF) features, and used this method to train two classifiers: 1) using lower chamber speakers in the 2005 session, and 2) using upper chamber speakers from the same year. For the purpose of controlling for the "person" aspect described

above—ensuring the classifier is not simply distinguishing the speech patterns of a given speaker—they applied each classifier to speakers of the other chamber. The classifier which was trained on the lower chamber speakers performed well on upper chamber speakers, but the opposite was not the case. They suggested this may be due to the former chamber being more generally polarized and the opinions voiced thus being more readily distinguishable.

In order to determine whether the classifier was able to generalize across time periods (the second aspect mentioned above), Yu, Kaufmann, and Diermeier (2008) took the classifier which they had trained on lower chamber speeches from 2005 and applied this to upper chamber speeches from various years. In Yu, Kaufmann, and Diermeier (2008) there was also reference to an earlier study by Diermeier, which took seven years of upper chamber speech as training and used this to classify speech from the following year. This was taken as evidence of the classifier's ability to generalize across time periods. The accuracy of this newly trained classifier varied greatly depending on the time frame of the speeches they attempted to classify. Yu, Kaufmann, and Diermeier (2008) surmised that this was either because the issues being debated vary—so that this variance reflected an issue-dependency rather than a time-dependency—or, alternatively, because the ideological orientation of Congress had shifted.

Another issue which Yu, Kaufmann, and Diermeier (2008) noted was that the content of an individual speech tends to depend on what has been said by the preceding speaker. Therefore the probability of a speaker generating a speech is not simply dependent on the ideology of the speaker. While it could be interesting to attempt to build a classifier which takes into account this dependence, this aspect will not be explored in this thesis, although we in the introduction note a speech given as response to a question, exemplifying this issue.

Diermeier et al. (2012) appears to describe in further detail some experiments which were also mentioned in passing in Yu, Kaufmann, and Diermeier (2008). Diermeier et al. (2012) used as a data source upper chamber speech data from seven periods of the US legislature, spanning from 1989 to 2004. From each period they extracted speeches by the 25 most "extreme" senators at each end of the political spectrum, as measured by a metric called DW-NOMINATE scores. The paper experimented with SVM classifiers trained on boolean, normalized frequency- and TFIDF-weighted vectors. Each of these vector weighting methods was applied to six distinct feature sets, or types of document representations. All document representations used the BOW approach, but they were build from respectively either 1) full word forms, 2) word stems, or only those words which had been identified by a part-of-speech tagger as 3) nouns, 4) verbs, 5) adjectives or 6) adverbs. The combination of these three vector weighting methods and six feature sets yielded 18 distinct input representations, and on each of these an SVM classifier was trained. The worst results were, as one might expect, reported on the classifiers trained only on adverbs, while the best results were attained by using TFIDF

weighted vectors constructed from word forms. This study, like Yu, Kaufmann, and Diermeier (2008) before it, combined every speech from each of the senators in a given period into one document, leaving them with 350 training documents. For validation they used data from the next congressional period, giving them 50 test documents. They reported being able to correctly classify 46 of these. For the five senators in the test set who were not present in the training set, they reported an out-of-sample accuracy of four out of five.

Diermeier et al. (2012) did not really have as a goal to determine an accurate classification method for parliamentary speech in itself. The stated intention was to examine which the SVM classifier had learned, and therefore shed light on the content of ideologies. For this purpose this method was quite advantageous, since the way in which each dimension of a BOW document representation contributes to the SVM classifier's decision is entirely transparent. They reported that words such as "disabilities", "gay", "wealthiest" and "policing" were indicative of a "liberal" ideology, while words such as "surtax", "homosexual", "partial-birth" and "taxing" were indicative of a "conservative" ideology.

Hirst, Riabinin, and Graham (2010) formulated a direct response to Yu, Kaufmann, and Diermeier (2008), taking a critical eye to the reported results. Hirst, Riabinin, and Graham (2010) suggested that, in the context of SVM classifiers, whether a party is in government or opposition may be the main trait that is picked up by the classifier, rather than the ideological position.

Hirst, Riabinin, and Graham (2010) trained their own models using similar methods to Yu, Kaufmann, and Diermeier (2008), i.e., an SVM classifier trained on TFIDF weighted vectors. As input data, however, Hirst, Riabinin, and Graham (2010) used Canadian Parliament speeches. As in Yu, Kaufmann, and Diermeier (2008), speeches from a single speaker were combined into one document, giving a total of 200 training vectors. The authors stated that this helped avoid overlap in terms of speaker between the training and test data, controlling for the "person" aspect mentioned earlier and discussed in Yu, Kaufmann, and Diermeier (2008).

Hirst, Riabinin, and Graham (2010) was an attempt to create a classifier comparable to that which Yu, Kaufmann, and Diermeier (2008) had produced, but the data sources are somewhat different, owing to the different political landscape. Elections for the Canadian legislature do not employ proportional representation[1], yet there is a strong regional party (Bloc Québécois), a split centre-left and at various times independent representatives and representatives from other minor parties, which in combination lead to a more complicated terrain than that seen in the United States legislature. In order to then align itself with the objective of Yu, Kaufmann, and Diermeier (2008), classifying into two ideologies (liberal and conservative), Hirst, Riabinin, and Graham (2010) lumped multiple parties into each group. They ignored left-wing parties, which did not fit

1. in which it is attempted to make the proportion of parties in parliament similar to the proportion of votes nationally, as opposed to first-past-the-post systems

into either of these two blocs.

A finding which was highlighted in Hirst, Riabinin, and Graham (2010) was that the classifier gave particularly high results (97%) for speeches made during the oral question period. This is a parliamentary period mostly consisting of opposition politicians putting "hostile questions" to government ministers. The most distinguishing features included words such as "he", "we" and "why", leading the authors to suggest that the classifier may simply have learned to separate "questions from answers or attack from defence", rather than picking up on actual differences in ideology. Indeed, when the authors trained a new classifier on new data in which the blocs" positions had swapped, they found that several words had gone from indicating "liberal" to "conservative" and vice versa. Applying either classifier on data representing a different opposition/position constellation also, as they predicted, gave very low, sub-majority baseline results. Hirst, Riabinin, and Graham (2010) points out that this undermines the idea that the classifier was learning to distinguish ideology, and suggests that it was instead picking up on the party's status.

A more recent attempt at classifying parliamentary speeches was made in Høyland et al. (2014), using European Parliament speeches. In one major respect, the task therein described is closer to that which will be explored in this thesis than the two earlier efforts discussed. As we have seen, Yu, Kaufmann, and Diermeier (2008) and Hirst, Riabinin, and Graham (2010) described binary classification tasks, the former using speakers from a two-party system and the latter combining parties into blocs. Høyland et al. (2014) and Lapponi et al. (2018), on the other hand, attempted to classify individual speeches into one of seven parties. This constitutes multiclass classification, which is a different and in many ways more challenging task than binary classification. The data were, as with Yu, Kaufmann, and Diermeier (2008) and Hirst, Riabinin, and Graham (2010), combined into one vector for each speaker, with a total of 689 speakers. As training data they used the parliamentarians of the 5th European Parliament, and those of the 6th were held aside for testing.

A central question which Høyland et al. (2014) examined was whether the addition of linguistically informed features to simple BOW representations would lead to increased accuracy. In order to test this, they created two versions of the data: one using simple BOW representations (using either lemmas, stems or full word-forms) and another where this was enhanced with parts-of-speech (POSs) and dependency relation tags. Two otherwise identical classifiers were then trained, each using one of these variations. Høyland et al. (2014) reported that the classifier trained on data enhanced with linguistically informed features had slightly higher accuracy than the classifier which did not have these data. They therefore concluded that such features were indeed useful.

The European Parliament is very different from the Norwegian parliament in that the parties of the former can rather be considered as groupings of national parties. There may potentially be great ideological gaps between the national parties which constitute one European party, which

the authors suggest may have resulted in low observed performance on the European Liberal Democrat and Reform Party (ELDR) and Union for Europe of the Nations (UEN) parties. Moreover the delegates, being as they are from different countries, potentially also have very different ways of expressing themselves depending on their mother tongue; this could potentially lead to confusion across (European) party lines between delegates from the same country.

Peterson and Spirling (2018) describes an interesting usage case for political speech classification. Peterson and Spirling (2018) took all speeches in the British parliament in the timeframe 1935 to 2013, and produced one classifier for each of the 78 sessions. In an interesting choice, four different classification algorithms[2] were used, and for each parliamentary session that algorithm which gave the highest accuracy was chosen as representative. The corpus included three and a half million speeches in total. The (full corpus) vocabulary includes full word forms that occur in at least 200 speeches, giving 24,726 dimensions to their BOW representation; all speeches over 40 characters were kept, while those consisting of fewer than this were dropped.

The goal of this paper was, however, not strictly to develop methods for classifying political speech. Instead, the researchers sought to investigate whether the accuracy of a machine classification algorithm could be taken as an indicator of the degree of political polarization at any given time. The resulting classifiers showed low accuracy during and after the second world war, and high accuracy during the Thatcher era. Peterson and Spirling (2018) further demonstrate that these results are correlated with specific historical qualitative and quantitative evidence of political polarization. Given that ideological utterances are likely to be more common in periods of greater political polarization, this can be taken as evidence to strengthen the hypothesis that a political speech classifier is indeed able to pick up on indicators of political ideology.

2. a perceptron, a stochastic gradient descent (SGD) classifier, a "passive aggressive hinge-loss classifier" and logistic regression with L_2 penalty

Chapter 4

Methodological background

In this chapter we discuss the theory and methods behind neural networks. Sections 4.1 to 4.3 of this chapter review some common neural network architectures. Section 4.4 goes into some detail on the various functions which are used in neural networks. Section 4.5 introduces word embeddings, which are semantic representations of linguistic units.

Traditional machine-learning models were long favoured for natural language processing (NLP) tasks. This includes the linear support vector machine (SVM) approach used by the all the papers examined in section 3 as well as Lapponi et al. (2018) and Lapponi (2019) in section 2.1. There were attempts at using neural networks in NLP in the 1990s and even earlier, but at that point such methods were not able to give very good results. Owing to advances in computing power and an increase in the amount of data available, there has in recent years been a revival, which can perhaps be traced back to the seminal paper Collobert et al. (2011). This paper reported close to state-of-the art results for a variety of NLP tasks, including part-of-speech (POS) tagging, chunking, named-entity recognition and semantic role labelling, using a unified architecture.

The importance of considering the specific task that is to be accomplished when building a neural network has been noted, for instance by Goldberg (2017, p. 149). Different types of layers may be well-suited to different aspect of the task, and a neural network may be constructed using various combinations of such. In addition to the fully-connected layers described in section 4.1, many other types of building blocks have been found to function well for NLP, such as the convolutional and pooling layers described in section 4.2 and recurrent architectures described in 4.3.

4.1 Neural networks

Neural networks work by transforming an input through a series of layers of nodes, between each of which lie sets of weights. In a fully connected feed-forward neural network all the nodes in the input layer (each representing a dimension of the input data) are connected to all the nodes in the next, "hidden", layer through a matrix of weighted connections. The outputs of the input nodes are multiplied by these

weights, along with a bias term, which gives the input values for this layer. The sum of the inputs for each node in this layer is then individually put through a non-linear activation function, such as a sigmoid function, resulting in an output value. This process may be repeated through any number of hidden layers until the output layer is reached, the activations of which are usually calculated using a different function, such as softmax for estimating individual probabilities.

Neural networks with at least one hidden layer have been shown to be universal approximators, meaning they can, given the right set of parameters (i.e. weights and biases), approximate any function within some margin of error. While this is true in theory, finding these parameters is the tricky part, and the best way of doing this is a matter of empirical testing. The process which we use will not be described in detail here. In condensed terms, the method is as follows: Initialize the parameters to an empirically useful distribution, calculate a loss function, and then update the parameters by the product of a learning rate and the negative of the gradients of their activation functions chained with the loss function with respect to the parameters. This process is known as backpropagation, and is repeated either an empirically chosen number of times or until the error on a validation set starts to increase, indicating that the training algorithm has started to overfit—overly adapting to the training set, while losing the ability to generalize.

4.2 Convolutional neural networks

We would very much like for a network to able able to pick up on compound phrases such as "neural network" as well as syntactic combinations such as "not good", in which the order of the constituent words is significant. If we stick to the basic unigram bag-of-words (BOW) model, however, the ordering of words is ignored; the documents "good, not bad" and "bad, not good" become equivalent. An obvious measure to detect combinations of words is to encode the input into word n-grams and treating the input as a bag of n-grams. Unfortunately, this would not only greatly increase the dimensionality of the input, we would also be very unlikely to encounter sufficient instances of each combination during training for learning to take place. (Goldberg 2017, p. 151) Ideally we would like our network to understand that replacing "network" with "net" as in "neural net" leads to a very close meaning, but these would be two entirely distinct bigrams in the training corpus.

Convolutional neural networks (CNNs) are able to consider the local area around a feature in a smarter manner. First we define a window of k words. We slide this window over the words in the text, looking up d_{emb}-dimensional word embeddings (see section 4.5) for each word as we go. The embeddings in each window are then concatenated into a vector x_i of length $k \cdot d_{emb}$. We then apply one "filter", or l filters, to the window; this is done by multiplying x_i with a matrix U of l weight vectors. Commonly we add a bias vector b and apply a non-linear activation function g to the result.

This yields a vector p_i of dimensionality l, representing the i-th window, in which each dimension is a scalar containing the result of each filter.

We now have our l-dimensional vectors $p_{i:m}$, where m corresponds to the number of convolution windows. The value of m ultimately depends on whether padding is added to the start and end of the text sequence (namely a "wide convolution") or not ("narrow convolution"). These l-dimensional vectors are "pooled" into a single vector, also of dimensionality l, representing the entire sequence. The most common pooling operation is "max pooling", by which the highest scalar is picked in each of the l dimensions, in effect giving us the most pronounced features. As alternatives there are also "average pooling", taking an average of the scalars; k-max pooling, picking the top k scalars in each dimension and yielding a $k \times l$ matrix; and "dynamic pooling", using a combination of different pooling methods. This latter approach benefits from knowledge of the problem domain, (p. 157) but it can also be applied when tuning hyperparameters experimentally.

Regardless of which pooling method is used, the output from the pooling layer is fed into the downstream network. This network trains on whatever task it is training on, and the gradients from the loss are propagated back to the convolution layer, updating the parameters in U and b.

4.3 Recurrent neural networks

While CNNs can pick up on relations between features, this only applies to the local window surrounding that feature. Recurrent neural networks (RNNs), on the other hand, can also make connections between features farther apart, all the while allowing for encoding variable length sequences into a fixed length vector.

On a high level an RNN takes as input an arbitrarily n long sequence of in-dimensional vectors and outputs one out-dimensional vector, which is then used as input for another task.

In a recursive layer we have a recursive function R, which takes a state vector s_{i-1} and an input vector x_i, corresponding respectively to the state resulting from the previous recursive call and the input at the current point in the sequence. This outputs a new state vector, s_i. In the simple implementation of an RNN, also known as an Elman network, or a vanilla RNN, we map the last state, s_n to an output vector y_n using an activation function. This output is fed into the downstream network, either by itself in an "acceptor" application, or along with other information as an "encoder". The loss from the downstream network is propagated backwards to train the parameters θ. Variants may also use the intermediate outputs y_i as in a transducer, given that we have some way of calculating local loss signals for the intermediate outputs. One may also train a bidirectional RNN, in which one creates an additional, parallel, RNN which runs through the input sequence backwards, whereupon the two outputs are concatenated. In this manner each time step receives a

representation that is also conditioned on subsequent time steps, and not only preceding time steps.

While the simple vanilla RNN is a practical and intuitive method, it is difficult to train due to an issue known as "vanishing gradients". This simply posits that the deeper our network is, the farther the loss signal has to travel, and the smaller the gradients at the bottom of the network will be, leading to small updates. Since a single simple RNN layer once unrolled is as deep as the length of the input sequence, the problem that this poses becomes apparent, particularly when considering long input sequences. Long short-term memory (LSTM) and gated recurrent unit (GRU) layers are two types of architectures designed to work around this issue.

The LSTM, introduced by Hochreiter and Schmidhuber (1997), works by extending the state vector s_i into two parts. One part is designated as "memory" cells, that preserve information across multiple time steps and can consequently propagate error gradients across time. The other part is "working memory", which represents the state produced at the current time step. The model introduces three "gates", modeled on logical gates: input, forget and output. While logical gates are usually binary, the gates in an LSTM are real numbers, meaning that the functions run on them are differentiable and can pass gradients. The values of the gates are computed by calculating linear combinations of the current input x_i and the working component of the previous state h_{i-1} multiplied by a weight matrix that is shared between all steps. This is then put through a sigmoid function, yielding complementary vectors where most values are close to 0 and 1. An update candidate z_i is calculated in a similar manner, except with a hyperbolic tangent (TanH) activation function rather than sigmoid. The memory state c_i is updated by running z_i through the input gate and the previous memory c_{i-1} through the forget gate, and the working state is updated by TanH-activating c_i and running it through the output gate. In more precise mathematical terms, it may be defined as follows:

$$
\begin{aligned}
s_j = R_{\text{LSTM}}(s_{j-1}, x_j) &= [c_j; h_j] \\
c_j &= f \odot c_{j-1} + i \odot z \\
h_j &= o \odot \tanh(c_j) \\
i &= \sigma(x_j W^{xi} + b_i + h_{j-1} W^{hi}) \\
f &= \sigma(x_j W^{xf} + b_f + h_{j-1} W^{hf}) \\
o &= \sigma(x_j W^{xo} + b_o + h_{j-1} W^{ho}) \\
z &= \tanh(x_j W^{xz} + b_z + h_{j-1} W^{hz})
\end{aligned}
\tag{4.1}
$$

$$
O_{\text{LSTM}}(s_j) = h_j
$$

In the above, i stands for the input gate, f is the forget gate, o is the output gate, and z is the update candidate. In order to distinguish the input gate from the current time step, j is used in this formula instead of i. The formulae for activation functions can be found in 4.4.1.

While the LSTM architecture works very well, it is difficult to analyse

and expensive to compute due to its complexity. Cho et al. (2014) introduced the GRU, which is a simplification of LSTM that has nonetheless been shown to to have comparable performance. GRU forgoes the memory component, and uses only two gates: The reset gate controls access to the previous state to compute a proposed new state, while the update gate controls how the elements of the proposed state are to be combined with the previous state.

4.3.1 Fixed-length representations from intermediate time steps

The outputs from a recurrent layer at different time steps contain information local to these time steps. This may be used to create a new sequential representation (where each time step is conditioned on the rest of the time steps of the sequence), but in this thesis we examine three methods of flattening this representation into a single vector representing the entire sequence.

Conneau et al. (2017), while dealing with sentence representations rather than representations of longer documents, detailed experiments using mean/max pooling and a self-attentive encoder on RNN sequences. The first two are the same operations as are used in the the pooling layer of a CNN. When applied to a sequence, this operation has the result that for each dimension across the output RNN vectors from the various time steps, the value from the vector with the highest value is taken. Conneau et al. (2017) shows that this works well empirically. Mean pooling works in a similar way but, predictably, takes the mean of each dimension across the time steps.

Lin et al. (2017) suggested another method to combine RNN outputs to generate sentence representations, namely self-attention. This works by trying to determine the weight, or attention, that should be given to any given time step. The process is as follows:

$$\begin{aligned} \bar{h}_i &= tanh(Wh_i + b_w) \\ s_i &= \bar{h}_i u_w \\ \alpha_i &= \frac{e^{s_i}}{\sum_{i=1}^{T} e^{s_i}} \\ u &= \sum_{i=1}^{T} \alpha_i h_i \end{aligned}$$

In this formula $h_i, \ldots, h_T$ are the hidden states at each time step. Each of these is put through the RNN weight layer (W, b_w) to create an activated output for the time step with the outputs compressed nicely between -1 and 1 using TanH. These outputs are put through the learned context query vector u_w, which has an output dimension of one for each time step, giving us a weight score s_i for each time step. The weight scores are normalized using the softmax function, which is further described in 4.4, to generate the weight vector α with a weight scalar for each time step. Once we have the weight vector, we use it to generate the weighted

representation of the document u similarly to mean pooling, but with the weights used to determine the importance given to each time step. Through backpropagation the attention vector u_w is updated, and the network learns which parts of the input to pay more attention to.

4.4 Functions used by neural networks

There are many different types of functions that are used in neural networks, each serving a different role in training. For instance, in multi-class classification the softmax function is used to create an estimate of the probabilities of the different output classes:

$$softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_i e^{z_i}}$$

Here the scores are first exponentially increased, and then normalized by the sum of these scores. This in effect generates a probability distribution. As the name implies it is a kind of "soft" max function, but whereas a max function would simply select the class with the highest score, softmax will assign some probability to each class. This makes the loss function continuously differentiable. Since we have probabilities for all the other classes, we can use this to measure the distance from the correct classification and update the gradients accordingly. In this section we go on to discuss activation functions, loss functions, regularization and optimization functions.

4.4.1 Activation functions

Rectified linear unit (ReLU) is a commonly used activation function, favored due to its simplicity (and consequently faster calculation time) and empirically generally good results for many types of architectures. For values above zero it simply returns the value; otherwise it returns zero:

$$ReLU(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

The derivative of this is of course extremely trivial:

$$ReLU'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

The original activation function, as introduced in the seminal Rumelhart, Hinton, and Williams (1986) which brought forward the concept of backpropagation and the multi-layer perceptron (MLP), was the sigmoid function, which squeezes all values between one and zero:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

This formula and, crucially, its derivative, are more expensive to calculate than the ReLU function:

$$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$$

The gates in RNNs should produce values close to one and zero, as described in section 4.3. To this end, the sigmoid function would be very apt. However, due to the aforementioned computational cost, and since these gates are calculated more frequently than the output activation, one will commonly use a "hard" sigmoid, which is a segmented linear approximation that does not require calculating any exponents. The version of this which is implemented in Keras uses the following formula:

$$H\sigma(x) = \begin{cases} 0 & \text{for } x < -2.5 \\ 1 & \text{for } x > 2.5 \\ 0.2 \cdot x + 0.5 & \text{for } -2.5 \leq x \geq 2.5 \end{cases}$$

This has the following derivative, which is quite similar to that of ReLU:

$$H\sigma'(x) = \begin{cases} 0 & \text{for } 2.5 > x < -2.5 \\ 0.2 & \text{for } -2.5 \leq x \geq 2.5 \end{cases}$$

Another common activation function is TanH. This is a variation of the sigmoid function which stretches the resulting value so that it is centered at 0. In an RNN this is commonly used to activate the output rather than ReLU. TanH is defined thus:

$$TanH(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

Its derivative is:

$$TanH'(x) = 1 - TanH(x)^2$$

4.4.2 Loss functions

Loss functions are functions that calculate how much a prediction deviates from the ground truth. A loss function is used in combination with an optimization function to update the parameters in a neural network and move the parameters in the network toward an optimum in which the loss function is minimized. Depending on the type of output, different types of loss functions may be used. For the experiments in this thesis we will be predicting political parties, which are independent categories, and so we will need to use a categorical loss function. The most commonly used such loss function is categorical cross-entropy:

$$\mathcal{L}_{CE} = -\sum_{i=1}^{C} t_i \log(s_i)$$

In this formula, t_i is the ground truth for class i, and s_i is the score (for our purposes softmax) calculated for that class. The sum of these across all classes is the loss.

An alternative categorical loss function, which is examined and described in section 6.2, is Kullback–Leibler divergence:

$$\mathcal{L}_{KL} = -\sum_{i=1}^{C} t_i \log \frac{t_i}{s_i}$$

4.4.3 Optimizers

Based on the values returned by the loss function, the optimizer updates the parameters. The most basic optimizer we see used is standard gradient descent, which has the form

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta)$$

in which η is the learning rate (an adjustable hyperparameter), and $\mathcal{L}$ is the loss function applied to the parameters. Stochastic gradient descent (SGD), which is the same operation as gradient descent but applied batch-wise, often gives good results Ruder (2016). However, all the parameters are updated equally, which means that parameters that are activated more frequently are updated more frequently, while those parameters that are rarely used receive fewer updates. adaptive gradient algorithm (Adagrad) is an algorithm which attempts to improve upon this aspect. In Adagrad each parameter has its own learning rate, and the learning rate is reduced as training proceeds.

$$g_{t,i} = \nabla_\theta \mathcal{L}(\theta_{t,i})$$
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

In this formula, $g_{t,i}$ is the partial derivative for the ith parameter at time step t. Each parameter is updated according to its gradient, but the learning rate by which the gradient is multiplied for the update is modified by another term. G_t is a diagonal matrix for time step t where t, i contains the sum of the squares of the gradients w.r.t θ_i at time step t; ϵ is a smoothing term that prevents division by zero. Since the sum of the values in G increase as the network updates, the values by which the parameters are updated decrease constantly. This also leads to the weakness of Adagrad, since once sufficient time has passed, the modified learning rate will approach zero and the network will stop learning. Root mean square propagation (RMSProp) was developed to try to counter the issue of ever-decreasing learning rates:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + (0.1)g_t^2$$
$$\theta_{t+i} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Instead of storing all the squares of the previous gradients, as Adagrad does in its G matrix, RMSProp instead calculates a running average $E[g^2]$, which is calculated based on the current gradient and the existing average. The term $E[g^2]$ can therefore decrease if the gradients are small, meaning that the learning rate can recover.

Adaptive moment estimation (Adam) is another method for calculating individual learning rates. It is very popular, but rather more complex:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$\theta_{t+1} = \theta_1 - \frac{\eta}{\sqrt{v_t} + \epsilon} m_t$$

Here v_t is an exponentially decaying average of squared gradients similar to the $E[g^2]$ term in RMSProp, with β_2 used as a tuneable hyperparameter for the 0.9 specified for RMSProp. In addition to this term, Adam also uses the term m_t, which stores the exponentially decaying average of the un-squared gradients. This latter term acts as a kind of momentum, which is a type of operation that can both nudge parameters over small upwards slopes and towards local minima, like a rolling ball. The update is calculated similarly to RMSProp, but with the gradient average m_t substituting for the actual gradient at the time step.[1]

4.4.4 Regularization

Regularization is a method to prevent a neural network from overfitting by putting constraints on its parameters. The experiments in this thesis employ two methods of regularization, namely L_2 regularization and dropout regularization. L_2 regularization works by applying a new term to the optimization objective, and penalizes large parameter values by trying to keep the sum of the squares of the parameter values low. The L_2 term is calculated thus:

$$R_{L_2}(\Theta) = ||\Theta||_2^2 = \sum_{i,j} (\Theta_{[i,j]})^2$$

This is then added to the optimization objective with a hyperparameter λ, which is a low number that controls the strength of regularization:

$$\hat{\Theta} = argmin_\Theta L(\Theta) + \lambda R(\Theta)$$

Dropout regularization is different, in that it does not work directly on the optimization objective. Rather, for a given training sample, some of the parameters are dropped (set to 0), so that they do not contribute to the classification result. This is used in an effort to keep the network from relying too much on specific parameters but instead learn to generalize. It

1. While Ruder (2016) mentions intermediate steps that calculate $\hat{m}_t$ and $\hat{v}_t$ to counter certain biases, this does not appear to be used by Keras by default.

is common to employ this in a high number of parameters at once, such as half the parameters in a given weight matrix.

4.5 Word embeddings

An important question for NLP tasks is how to represent the text that is input to a system. For document classification a common approach is, as we have seen, to represent a document as a bag-of-words. To reiterate, this takes either the original word forms as they appear, their lemmas, or some combination of features, and lumps them together in a term frequency–inverse document frequency (TFIDF) weighted vector, in which each dimension represents the frequency of the feature in the document weighted down by the general frequency of the feature in the wider corpus. While this method of document representation has been used for a long time with good results, it has weaknesses, particularly in that this method relies on seeing the exact same feature representation (e.g. word form).

Much research has been done into attempting to instead encode words into representations that in some way capture their underlying meaning, or something close enough to this to suffice for a specific task. The question of what a word means is of course an open and subjective question, but a practical starting point may be found in Wittgenstein (1997), which held that "the meaning of a word is its use in the language". If two words are used in similar ways, one may then assume that their meanings are similar. Wittgenstein's contemporary. Firth (1957), formulated it thus: "You shall know a word by the company it keeps!" That is to say, similarity between words may be determined based on the contexts in which they occur.

A basic application of this idea, known as the distributional hypothesis, can be seen in sparse word embeddings as defined by a co-occurrence matrix. This method takes a vocabulary of $|V|$ words and generates a $|V| \times |V|$ matrix. For a given definition of context (generally document, sentence or window of L words on each side), one counts the number of times each word occurs in the same context as each other word. The resulting two-dimensional matrix then represents a vector space in which words which frequently co-occur are close to each other. When inputting a text to some task, one may then look up each word in this matrix and use the embedding vector instead of the word form or its derivative.

A co-occurrence matrix will generally produce meaningful semantic representations of words, but there are two main downsides. The first is that the vectors are very long and sparse, since each dimension represents a vocabulary word and most words will not co-occur with most other words. This length gives rise to a high computational complexity when attempting to use them as inputs to neural networks. The second is that, since each word in the co-occurrence matrix is represented as a distinct dimension, the vector model may be unable to represent similarities between different words that tend to co-occur with different synonyms standing for the same concept or thing.

4.5.1 Dense word vectors

Given the disadvantages inherent in the sparse representations of a co-occurrence matrix, it would be nice to have a method to generate dense embedding vectors for our words. One simple solution is to apply principal component analysis (PCA) or another dimensionality reduction method to sparse embeddings generated by an algorithm such as that based on a co-occurrence matrix above. Another is to generate them directly using a neural network. One such method is the family of models introduced by Mikolov et al. (2013a) and known as word2vec. Indeed, Mikolov, Yih, and Zweig (2013) demonstrated that word embeddings trained in this manner do indeed encode meaningful semantic and syntantic information. word2vec introduced two main models, the continuous bag-of-words (CBOW) model and the skip-gram model.

In the CBOW version we have a fully connected neural network with one hidden layer, and in the most basic case the training objective is to predict a focus word based on a context word. The input and output layers have one dimension for each vocabulary word, while the hidden layer has the dimensionality we wish to have for our embeddings. The input context word in a given pass through the network is represented as a one-hot vector, in which the value in the corresponding dimension is set to 1 while the others are all 0. This has the effect of copying the corresponding row from the weight matrix to the hidden layer. The hidden layer essentially now contains our current suggestion for an embedding. No activation function is applied to this layer; the layer is directly multiplied with the weight matrix connecting it to the output layer. The output layer is then activated through softmax, giving a probability distribution over all the vocabulary words. The true output vector is, as the input vector, also a one-hot vector representing the target word, meaning that calculating the loss and updating the weights through backpropagation is trivial. After enough iterations, the weights will have stabilized, and we simply take the the weight matrix which has been created between the input and hidden layers as our embedding matrix.

Of course, the CBOW nomenclature implies that the input can be a bag-of-words rather than a single word, or a bag containing only one word, as described in the paragraph above. The described algorithm can be simply modified to accommodate this by setting the value in each dimension representing a context word to $\frac{1}{C}$, where C is the number of context words. This has the effect of setting the hidden layer to the average of the embeddings of the context words.

Skip-gram can be described as the "mirror-image" of the CBOW model. The dimensions of the network layers and parameters are the same. Here, however, the input is the focus word (still represented as a one-hot vector) rather than the context word. As with the one-context-word version of CBOW above, the hidden layer is a projection of the embedding for the focus word. Where this version differs is in the calculation of the output. Here, probability distributions are generated for all of the C context words, and the loss of the network is then taken as the sum of these losses.

The algorithms as described above are highly computationally expensive, since update vectors have to be calculated for all the vectors in the vocabulary. In practice, optimization methods are employed, such as "negative sampling". When we use negative sampling, we still calculate the vectors for the positive samples (i.e. the focus word for CBOW and the context words for skip-gram), but rather than iterating over the entire rest of the vocabulary, only a "negative sample" is included, so as to move the word embeddings away from non-context words. This has the effect of regularizing the embeddings.

4.5.2 Considerations

When discussing the training of word embeddings there are several aspects to consider. For instance, we care about the source of the data upon which they are trained: the linguistic quality of the text, the domain the text belongs to etc. It is also worth considering what definition to use for context. Large context windows result in embeddings that reflect topical similarity, while smaller context windows tend to emphasize syntactical similarity (Goldberg 2016, p. 23). The dimensions desired for the resulting embeddings can also be considered, though Mikolov et al. (2013b) showed diminishing returns after 300 dimensions.

These algorithms are implemented in the word2vec software package (Mikolov et al. 2013a). A newer library which expands upon the algorithm with the added use of character *n*-grams is fastText (Bojanowski et al. 2017). This use of sub-word information enables the model to pick up on morphemes and compound words. This is potentially highly beneficial for languages which write compound words without spaces, such as Norwegian, and as such, this project is particularly likely to benefit from this method.

Pre-trained word embeddings are available for many languages. For Norwegian, there are embedding matrices trained on Wikipedia articles (Al-Rfou, Perozzi, and Skiena (2013), Bojanowski et al. (2017), Grave et al. (2018)), and on news corpora (Fares et al. 2017a). The NLPL project (Fares et al. 2017b) contains a wide variety of word embeddings, particularly rich in embeddings for Norwegian. Such pre-trained, publicly available word embeddings are trained on very large amounts of data and therefore have both large vocabularies and can be expected to have learned a lot of interesting similarities. Even so, the learned similarities represented in such word embeddings are not necessarily useful for all tasks; the way a word is used on Wikipedia or in a news article may differ from how it is used in, for instance, a parliamentary setting. The Talk of Norway (ToN) corpus contains 63 million tokens, and for the task at hand we show that this, or rather a training subset of this, is sufficient to create useful embeddings, as is demonstrated in section 6.2.

Chapter 5

Experimental setup

In this chapter we will discuss the computational environment as well as the basic experimental setup used for our experiments on classifying the Talk of Norway dataset. Sections 5.1 and 5.2 describe the initial steps taken to prepare the data by excluding certain speeches and splitting into training, development and test sets. Sections 5.3 and 5.4 describe respectively the Keras machine-learning library and the Abel high-performance computing cluster. Section 5.5 discuss some common setup steps and hyperparameters used for most or all of the experiments.

5.1 Replication experiments

Initially we considered it prudent to attempt to replicate the split into ten folds that was used in Lapponi (2019) in order to perform comparable experiments. In Lapponi (2019), all speeches of less than 100 tokens, as well as speeches not belonging to the seven major parties and speeches without a party label (such as speeches by the President of Parliament), were removed from consideration, yielding a reported 152,405 speeches. Each chronologically sorted speech was then successively binned into a fold so that each of the ten folds held a representative sample of speeches across time. Attempting to replicate this split turned out to be less than straightforward. The annotated version of the corpus, as available on Github (Lapponi and Søyland 2016), is already split into tokens. Filtering out speeches based on these token counts, however, yielded 155,904 speeches. A comparison with the original list of speeches as used in Lapponi (2019) suggests that a different tokenization algorithm was used to determine the token count in this instance:

- split texts into tokens on whitespace and hyphens
- remove non-alphabetic tokens (n-dash, percent sign)

The first step used by Lapponi (2019) appears to be have been to split the texts into tokens based on whitespace and hyphens. This is exemplified by speech no. 130,212, the one speech which appears in the list of 152,405 speeches but not in our list of 155,904. This speech has 99 tokens by the

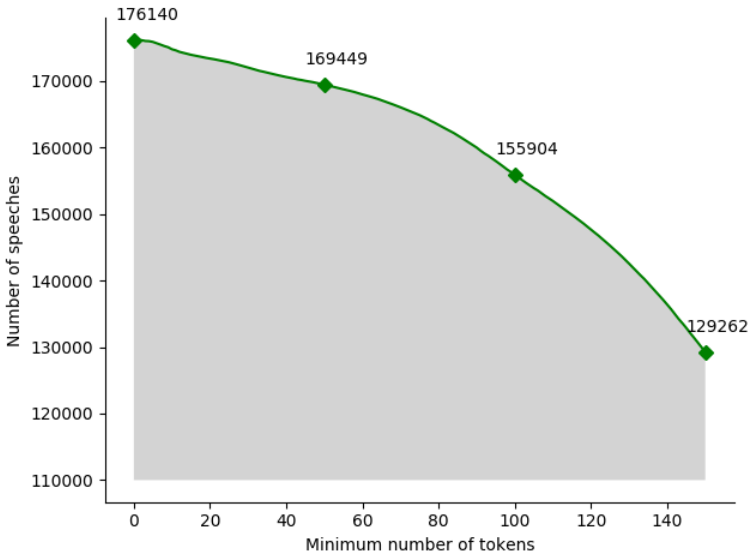


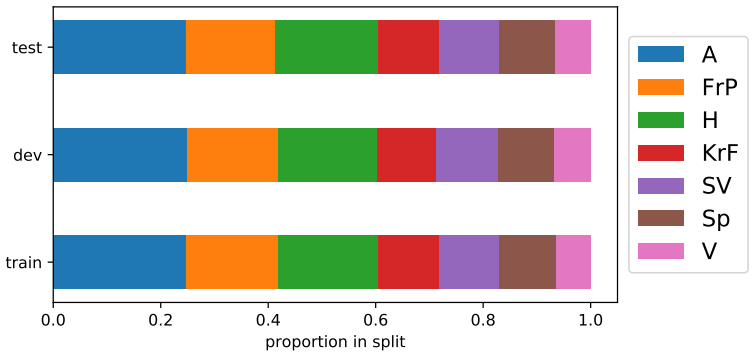
Figure 5.1: Number of speeches included at various token cutoffs

annotations count. Splitting on whitespace gives 96 tokens. However, if one in addition splits on hyphens it has 100 tokens, since the speech contains four hyphenated compound nouns. This does not account for all the missing speeches, however. In addition to punctuation which is joined to the preceding word, Norwegian also uses punctuation with whitespace on either side, mostly en-dashes which are used parenthetically and percent signs which, as a unit of measure, are also preceded by a whitespace. Such non-alphabetic tokens appear to also be excluded from the token count. Removing these gets us closer to the mark, but even after these steps had been taken 85 speeches were unaccounted for. These belonged to the seven major parties and were quite long, with an average token count of 497 tokens, so it is not obvious why these were excluded.

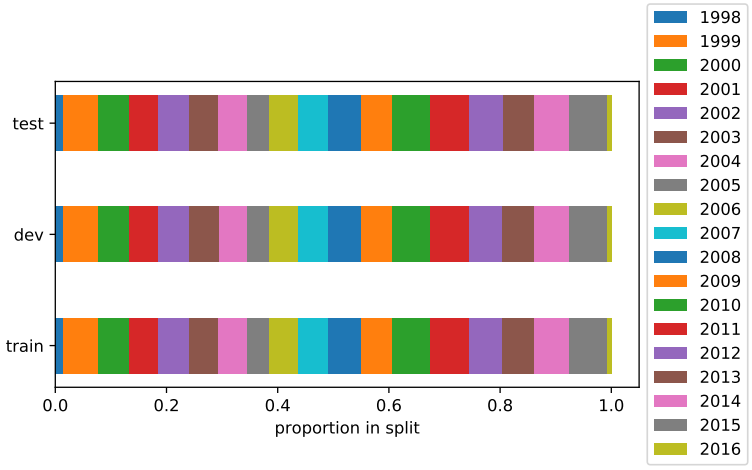
5.2 Splitting into training, development and test sets

Given the difficulties described above, it was judged infeasible to replicate the folds exactly, since any missing speeches would offset the positions at which the speeches were binned into separate folds. Moreover, we quickly discovered that the more complicated architectures we wished to evaluate required relatively long training times—sometimes more than a full day. Performing ten-fold cross-validation would naturally increase the amount of computational resources needed tenfold, and since we wished to perform as exhaustive hyperparameter searches as possible, the decision was made to instead create a held-out test set on which to perform final evaluation.

There was also an issue with setting the token cut-off at 100 tokens, since this excluded a lot of data, as can be seen in figure 5.1. By lowering this threshold, and including all speeches of at least 50 tokens (as



(a) Proportions of political parties in splits



(b) Distribution of years in the splits

Figure 5.2: The distribution of aspects of the speeches into splits

measured using the annotated corpus), the number of speeches is increased to 169,449. This gives us significantly more data to work with. With this new-found resource, we were able to set apart a tenth of the data—specifically the fifth fold as generated by the described algorithm—to act as the aforementioned test set.

This left 152,504 speeches for training and development. The development set was taken from these 152,504 speeches in the same way as the test set—by splitting the remaining into ten folds, each fold containing 9% of the total data, and taking the fifth fold. This resulted in a highly uniform distribution of party labels in the three sets, as can be seen in figure 5.2a, and the chronological distribution of speeches is similarly guaranteed by the selection through binning, ensuring that each set makes up a representational sample; this is demonstrated in figure 5.2b.

5.3 Keras

Keras is a high-level interface for defining and working with neural networks (Chollet et al. 2015). It has been used to train all the neural networks described in this thesis. Keras supports multiple back-end frameworks that deal with the actual training of networks, but in practice most use TensorFlow (Martín Abadi et al. 2015), which is also what is used in this thesis. Other than Keras and TensorFlow, there are many other libraries which are available and which could be chosen, such as PyTorch (Paszke et al. 2017) and AllenNLP (Gardner et al. 2017), which is based on the former and specializes on natural language processing (NLP) applications. PyTorch and TensorFlow are different in several aspects, but one which has been given much attention touches on how the two libraries define the graphs used in computation. In TensorFlow, the computational graph that defines a neural network is static, and once defined cannot be changed. PyTorch, being the newer of the two, supports dynamic computational graphs, meaning that the graphs can be modified on the fly, allowing for more flexibility in architecture design. For the models created in the experiments in this thesis static graphs have been sufficient, although it may have allowed for using variable length sequences as input to an recurrent neural network (RNN) with an attention layer, an issue described in section 7.2.

While Keras lacks support for dynamic graphs, it has other characteristics that make it attractive. It has several built-in modules which can serve directly as the building blocks of neural networks, and using Keras alleviates the need to manually define a great deal of elements such as individual cost functions and the like. The Keras functional API[1] includes pre-defined layer types, such as dense layers and various RNN layers, and allows for specifying most if not all the hyperparameters one may want to adjust. As the name "functional API" implies, the layers are defined as functions, and they are connected to previous layers through function calls in an intuitive manner. This significantly simplifies the job of defining neural network architectures compared to working directly in an underlying framework. This has the dual effects of greatly reducing the the scope of human error and allowing for quick prototyping of various architectures.

5.4 The Abel computer cluster

Training neural networks is computationally demanding. While training a single network can be manageable, in order to ascertain the most optimal hyperparameter configuration for a given neural architecture there is a need to train multiple networks. Therefore we need to use a computing cluster; this gives us access to a great deal of resources, and allows us to run multiple parallel training operations. The Abel computer cluster is a high-performance computing cluster hosted by the Department for Research

1. https://keras.io/getting-started/functional-api-guide/

Computing at USIT, the University of Oslo IT-department.[2] Abel uses the Slurm system[3], which manages the computer cluster and schedules jobs. It allocates resources based on users' quotas and ensures that users get access to the resources to which they are entitled while at the same time preventing abuse and overuse. There are multiple ways to submit jobs to this system, but in general jobs are submitted through enhanced shell scripts using the `sbatch` program. This program will parse special `#SBATCH` lines containing options that specify certain environment variables particular to the Slurm system. These are used to specify which resources are needed, e.g. number of CPU cores or amount of RAM, and how long the job should be allowed to run—once the runtime is expended, the job is immediately terminated.

When jobs are submitted using `sbatch`, they are not run immediately. The scheduler will calculate the resources requested, and depending on the resources, the user and the group to which the user belongs it may take quite a while before the job is allowed to run. Moreover, each group is assigned a finite amount of resources, and if this is all depleted then it could be impossible to continue computation until more resources are granted or a new allocation period starts. The Talk of Norway (ToN) dataset, while not particularly huge, is relatively large, and some of the architectures explored in this thesis are complex in terms of the computational resources they require. In addition to having to wait sometimes for quite a long time between starting an experiment and being able to analyse the result, this has also meant that the experiments described in this thesis have demanded a not insignificant piece of the resource budget—for instance, the main RNN experiment detailed in section 7.2 required over 20,000 core hours. This raised the challenge of needing to balance personal experimental needs against those of the greater fellowship of users, and some experiments that were later judged methodologically non-optimal were nevertheless not repeated in the interest of not wasting these shared resources. In 7.1.3 we discuss one such methodological shortcoming, namely the lack of accounting for non-determinism, and try to quantify the possible effect.

5.5 Common neural network hyperparameters

In addition to the common training and development sets that are described in section 5.2, there are a few more basic aspects which were common to nearly all the experiments, namely:

- early stopping after two epochs without improvement in F_1 score on the development set, with the weights from the best-performing epoch used for evaluation
- categorical cross-entropy as loss function

2. https://www.usit.uio.no/english/about/organisation/rc/
3. https://slurm.schedmd.com

Network	Accuracy	Precision	Recall	F_1 score	Train time (min)
Majority	24.97	3.57	14.29	5.71	
Rand-uni	14.50	12.38	13.81	12.55	
fastText	53.07	54.75	49.45	51.06	

Table 5.1: **Majority**: Majority classifier, classifying all as Ap; **Rand-uni**: Uniform random classifier; **fastText**: fastText classifier with default parameters (Joulin et al. 2017)

- default Keras hyperparameters for optimizers:
  - Adagrad: initial $\eta = 0.01$
  - Adam: initial $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$
  - RMSprop: initial $\eta = 0.001$, $\rho = 0.9$
- batch size of 32
- normalized (lower-cased) word forms as input representations, with punctuation as separate tokens

In certain instances we have have deviated from the hyperparameters described above, in which case this is explicitly noted.

For basic comparison, the table in 5.1 shows the results that simple, crude baseline methods yield, as well as the fastText classifier detailed in Joulin et al. (2017). A majority classifier will in this instance classify every speech as belonging to the Labor Party (Ap). A uniform random classifier will classify each speech completely randomly, with each class being equally likely, while the random distributionary classifier distributes randomly based on likelihood distribution generated from the training corpus.

For experiments using pre-trained embeddings, we used embeddings we trained ourselves using fastText in addition to publicly available embeddings. Our embeddings were trained using 10 epochs and a learning rate of 0.025, matching the settings used to train the equivalent publicly available embeddings. Apart from this the default settings were used.

5.6 Other considerations for splitting

As discussed and illustrated by figure 5.2, the approach taken in this thesis ensures an equal distribution of texts in terms of political party and time, into each of our three sets of data. In other words, this distribution training, development and test data does not take into account the aspects discussed in chapter 3 and as detailed in Yu, Kaufmann, and Diermeier (2008), namely person, time and issue—each of these aspects can be expected to be roughly equally distributed among the sets. When we then evaluate our classifiers using data stemming from the same persons, the same time periods and discussing the same issues, we ignore the dependencies between these

coinciding variables. To what extent the classifier is then picking up on ideology or policy positions, and to what extent it is simply recognizing, for instance, the elocution of particular members of parliament or the issues of which they usually speak, is therefore difficult to measure.

The splits used in thiis thesis are similar to but do not replicate those used in Lapponi (2019). The paper which preceded this (Lapponi et al. 2018) on the other hand, split the speeches into folds corresponding to different parliamentary periods, and therefore different government-opposition configurations and time periods. This method considers the aspects of time, person and issue that Yu, Kaufmann, and Diermeier (2008) raises and we discuss in chapter 3. The reason this is the case for the first of these three aspects is obvious, and to some extent it controls for the latter two as well since not all the parliamentarians are serving in all the periods, and the issues discussed will vary over time. The government-opposition aspect discussed by Hirst, Riabinin, and Graham (2010) is also accounted for, since the parties are variably either in government or in opposition in any given split.

Chapter 6

Preliminary experiments

In this chapter we will describe some experiments for our classification task using relatively simple architectures. It might have been interesting to review some of the more traditional classifier training methods such as Linear-SVM and Logistic Regression for comparison, since although Lapponi (2019) does present such experiments, the data we are working with in this thesis differs slightly in terms of document selection, as was established in section 5.2. The focus of this thesis is however to look into methods using neural networks, and the traditional methods will therefore not be explored.

The neural networks described in this chapter all consist of dense layers that only feed forward. Section 6.1 describes the simplest method, using bags-of-words (BOWs) as input, while section 6.2 demonstrates a minimalist input representation based on word embeddings. Section 6.3 explores a hybrid architecture combining the two input representations. For each architecture, we discuss the effects of different hyperparameter choices and reason around these. Table 6.1 shows the best-performing hyperparameter configuration for each type of network architecture examined and the resulting scores.

6.1 Feed-forward neural network on BOWs

The first experiment employed a feed-forward neural network architecture. As discussed in 5.1, speeches labelled as being from the president of parliament, KP, TF and MDG were removed from the corpus prior to training, and normalized word forms were used for the input representations.

6.1.1 Training and parameters

Six combinations of hyperparameter settings for generating BOWs were tested. Since including every single token discovered in the corpus in the vocabulary would give prohibitively long vectors, the input vector size was limited to 15k, 50k or 100k words. The second hyperparameter tested was the use of either un-weighted or TFIDF-weighted counts of occurrences in the sample vectors.

Network	Accuracy	Precision	Recall	F_1 score	Train time (min)
FF-BOW	67.77	68.41	66.56	67.38	96
FF-AE	54.88	56.21	53.59	54.64	6
FF-BOWAE	68.50	69.29	67.10	68.06	266

Table 6.1: **FF-BOW**: feed-forward network on bags-of-words, vectorization mode: TFIDF, vocabulary size: 100,000, optimizer: Adagrad, regularization: L_2, 0.0001, hidden layers: 1; **FF-AE**: feed-forward on averaged embeddings, activation: ReLU, optimizer: Adam, hidden layers: 3, loss: KL divergence, input: 600-dimensional ToN-trained embeddings; **FF-BOWAE**: parallel feed-forward network on both bags-of-words and averaged embeddings, with the respective hyperparameters corresponding to those of the two above network, with the exception of the optimizer (Adagrad)

- vocabulary size: 15, 50 or 100 thousand
- weighting: raw counts or TFIDF

For the model configuration, the following hyperparameters were tested:

- activation function: ReLU
- regularization: L_2 (0.001), dropout (0.2) or none
- optimizer: Adagrad or Adam
- layers: one or two hidden layers of 300 dimensions each

In this architecture, each speech is input as a sparse vector where each dimension corresponds to one word form as found in the vocabulary and contains either a raw or TFIDF-weighted count for that word form. While the architecture does not make explicit use of word embeddings, the first hidden layer in fact acts as a kind of encoding layer to encode the words in a continuous bag-of-words (CBOW) context. To see why this is the case, recall the definition of the word2vec method. That model takes as input a one-hot vector the length of the vocabulary, where only the target or context word is active; in the model described in this section, meanwhile, the input vector is still the length of the vocabulary, but every dimension for each token which occurs in the document is active. Since each row of weights in the first layer corresponds to one vocabulary word, and these weights are updated in accordance with how much they contribute to the result on the downstream task, it is natural to think that this layer contains useful representations of the vocabulary words. An attempt to use this weight matrix as an embedding matrix is described later in this chapter.

6.1.2 Evaluation

In this experiment, the optimizer turned out to be the hyperparameter that most influenced performance. This was a bit surprising, both since

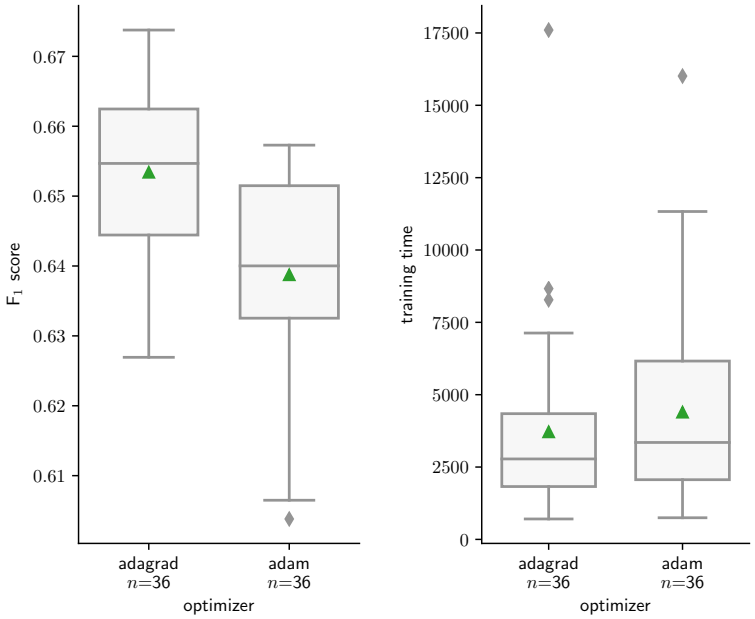


Figure 6.1: Effect of optimizer on F_1 score and training time of feed-forward network

Adam is generally taken to be a good all-round optimizer suitable for most situations (Kingma and Ba 2014), and because we might reasonably assume that any optimizer would be likely to arrive at the same local optimum given enough time. As we can see in figure 6.1 however, in this experiment Adagrad not only gave significantly higher results; it also trained much faster. The Adam models trained in average for 6 epochs, while the Adagrad models trained in on average 6.6 epochs, suggesting that the increased complexity of the gradient calculation had a significant effect on training time. Both use parameter-specific learning rates, with the main difference being that Adagrad employs a constantly decaying learning rate that makes the gradients vanishingly small after time, while in Adam they can recover. In addition, Adam employs momentum, which is meant to help escape from local optima. For both optimizers the Keras default parameters were used for both optimizers, in which the initial η is 0.01 for Adagrad and 0.001 for Adam.

Interestingly, increasing the batch size appears to lessen the gap. While we do not control for the effects of non-determinism and therefore cannot draw any absolute conclusion, a model trained using a batch size of 256 showed an F_1 score that was 1.7 higher than the same configuration trained with a batch size of 32. The effect seems to taper off as we increase to a batch size of 512, as seen in figure 6.2.

Since the data set is quite large, and the epochs correspondingly long, we figured it could be interesting to see the performance of the model between epochs, as opposed to to only at the epoch boundaries. This was tested with both Adam and Adagrad and a batch size of 64. The model using Adam showed more fluctuation, while the Adagrad model seems to

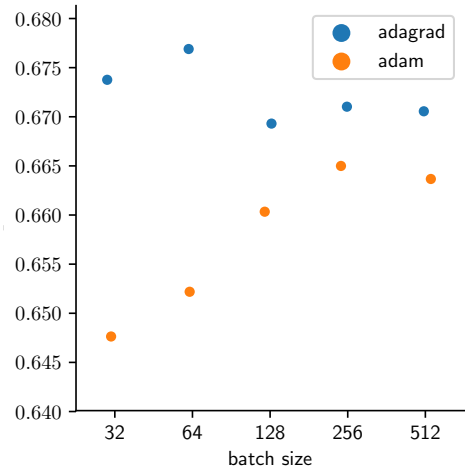


Figure 6.2: Best BOW-FF configuration run using Adam and Adagrad with various batch sizes

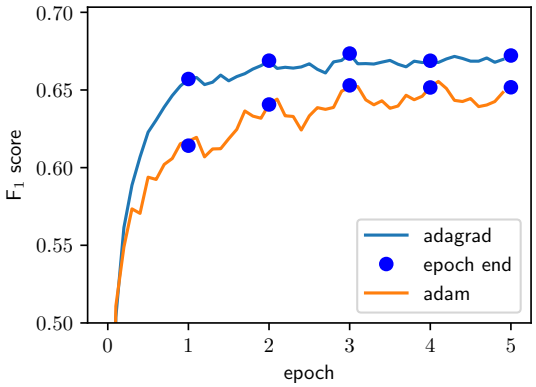


Figure 6.3: Evolution of F_1 score on development set within epochs, with the blue dots representing scores calculated between epochs

settle earlier. These fluctuations, which are graphed in 6.3, demonstrate that the F_1 scores calculated at epoch boundaries do not show the whole picture; a lot is in fact happening in the meantime. More to the point, we note that while the model training using Adam appears to be making large adjustments to the weights, as evidenced by the relatively large fluctuations in development F_1 score compared to the Adagrad model, it is failing to consistently improve and converge to a good optimum.

Vocabulary size appears to be directly correlated to a higher F_1 score, with the highest-performing model employing the largest vocabulary size, as can be seen in figure 6.4. At the same time, training time scaled roughly linearly with the input dimensionality, which is of course determined by the size of the vocabulary.

The models using TFIDF weighted input representations seem to have a bit of an edge on the models using raw counts; figure 6.5 shows that the

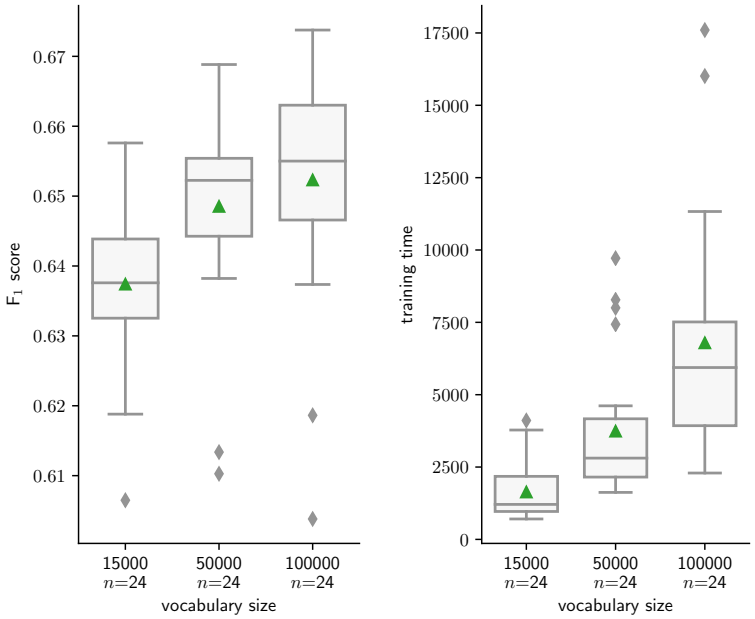


Figure 6.4: Effect of vocabulary size on F_1 score and training time of feed-forward network

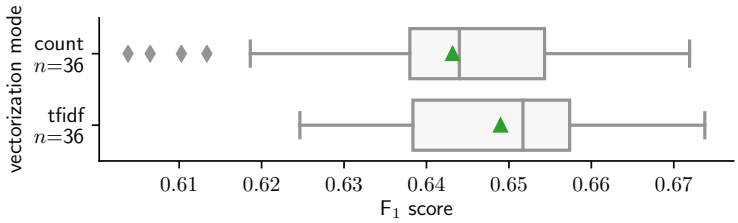


Figure 6.5: Effect of vectorization mode on F_1 score for feed-forward network

median is nearly a full point higher. With a neural network trained on BOWs, the weights from each input dimension should adapt in accordance with the importance of each vocabulary item. The TFIDF weighting in preprocessing is arguably likely to have a similar effect, in that rarer and potentially more indicative word forms are weighted more strongly. Performing the TFIDF weighting does however give the network a head start by adding information about corpus-wide frequencies, and it seems that the network is able to make use of this information to a certain extent.

Applying regularization appears to increase the span of resulting F_1 scores for the models, while performance is more stable in the models not employing any. The models using L_2 regularization show a much greater span of scores, with both the best-performing and worst-performing model utilizing this, as we see in figure 6.6. Dropout regularization seems to have failed in the main goal of preventing overfitting, as measured by the difference between training loss and validation loss.

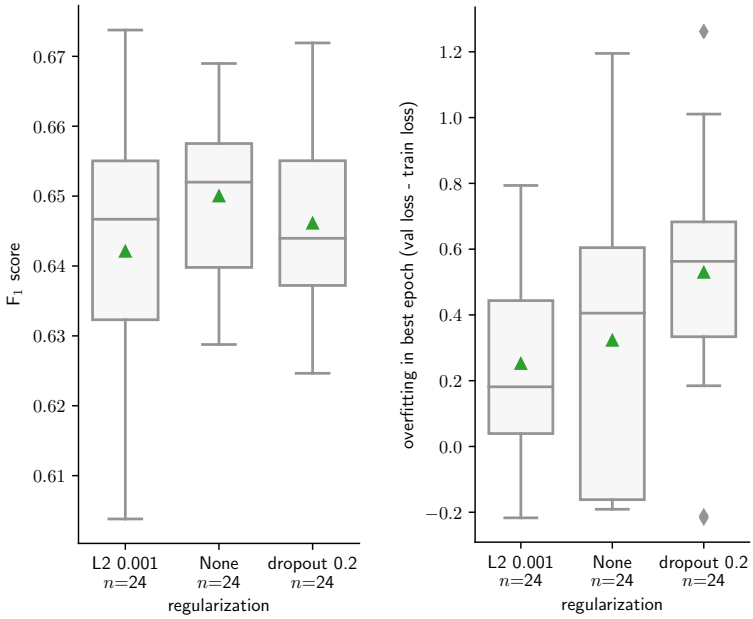


Figure 6.6: Effect of regularization on F_1 score and overfitting for feed-forward network

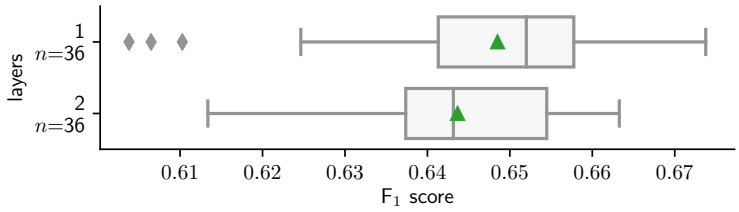


Figure 6.7: Effect of number of layers on F_1 score for feed-forward network

Increasing to two layers did not improve the score; rather, the simpler networks outperformed the more complex networks significantly, as figure 6.7 shows. In general networks with more layers should always be able to at least match the performance networks with fewer layers. That said, gradient flow may be hampered if the gradients have to travel farther through the graph. It is likely that this is what has happened in this case.

In conclusion, the model configuration which performed best on the development set used TFIDF-normalized input vectors with a vocabulary of 100,000 dimensions, one hidden layer of 300 dimensions, L_2 regularization with $\lambda = 0.001$, and Adagrad as the optimizer. The macro F_1 score for this model on the development set was was 67.38.

The results are not directly comparable to those reported in Lapponi (2019), since we are including those speeches with sizes between 50 and 100 tokens. Furthermore, Lapponi (2019) used ten-fold cross-validation, whereas we used a fixed train-dev split. With these caveats, we can only

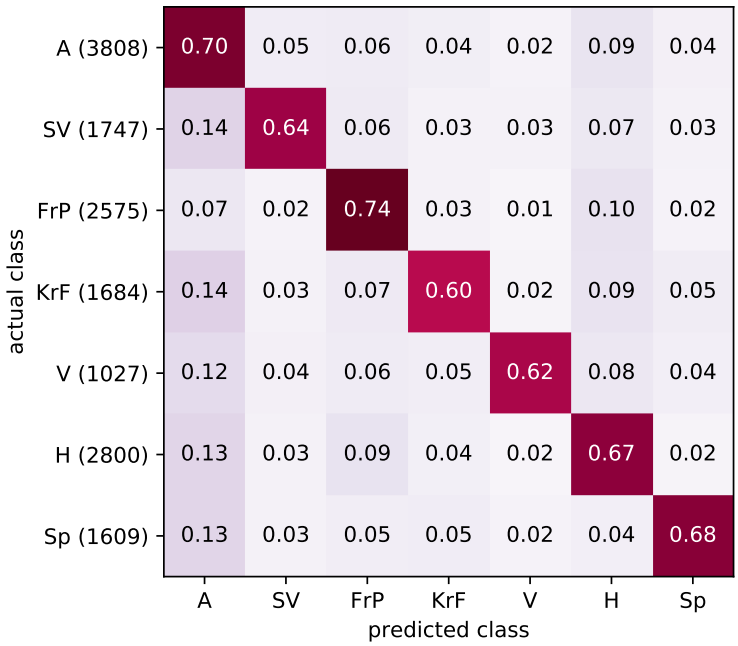


Figure 6.8: Confusion matrix of best feed-forward network, normalized by class support (shown in parentheses)

conclude the results are similar to Lapponi (2019), who reported a macro F_1 score of 65 when using a feature configuration corresponding to a BOW over word forms.

The confusion matrix in figure 6.8 shows a tendency to predict the majority class. Of all the classes, the classifier has the highest performance on the far-right Progress Party (Frp), with a recall rate of 0.74, indicating a distinct ideology. The party it is closest to ideologically, Høyre, is also the one with which it is most likely to be confused, something which goes both ways.

As touched upon in the introduction to this section, the first layer of weights in a neural network based on BOWs has similarities to an embedding layer. It is indeed possible to take the weights from this layer and pair it with the vocabulary and use them as task-trained embeddings. The weights from the best-performing network architecture described in this section were therefore extracted and put into the word2vec format for use with other network architectures.

A feed-forward architecture on BOWs is attractive in its simplicity, and the manner in which different input dimensions contribute to the classification decision is intuitive. Once we consider documents as continuous wholes, however, this input representation has certain shortcomings. Since the input only encodes the prevalence of a given term in a document, foregoing information about the order in which the terms occur, it becomes difficult to argue that the model is making inferences based on ideological position—for that to be the case, we would want the model to consider not only what topics are raised, but also what opinions or positions are

expressed with regards to said topics. In the worst case, the model may simply be making shallow inferences based on the lexical predilections of various political parties' representatives. It would be more interesting to try to create an input representation and model which is more semantically informed, i.e. could make some connection between the various words used.

6.2 Averages of pre-trained embeddings as input to a BoW model

The first experiment moving on from the simple feed-forward BOW model was done using a simple method employing word embeddings to generate document representations. In this method, the average of all the embeddings for the tokens in the document is computed, and this is then used as the document representation for classification. The term "semantic fingerprinting" was coined by Kutuzov et al. (2016), while Goldberg (2017, p. 93) uses the term CBOW to refer to this method. This method still does not take into account the order in which words occur—indeed, the input representation even obfuscates which words are actually used. On the other hand, it does make use of information gained through employing the distributional hypothesis, as discussed in 4.5, through the power of word embeddings.

Averaged embeddings were generated for all the speeches in the dataset. As a first step, we created a list of words by combining of the 100 most frequent words in the corpus as measured by inverse document frequency with a list of stop words provided by McDowall (2016). The words in this combined list were not used in the calculation of the average embeddings, since stop words and frequent words are liable to introduce noise in the averages. Eight versions of the dataset were generated. These used different fastText embeddings models. Some models were taken from the NPL repository (Fares et al. 2017b), and were trained on the Norwegian Newspaper Corpus and NoWaC with respectively 300 and 600 dimensions. We also trained our own embeddings on the training subset of the ToN corpus with the same dimensions. For each of these were created a version that counted unique tokens only once, and a version that used each occurrence of the token to calculate the embeddings.

These representations were then fed into various configurations of a feed-forward neural network. The following hyperparameters were tested:

- activation functions: ReLU, hyperbolic tangent (TanH)
- loss functions: categorical cross-entropy, Kullback-Leibler divergence
- optimizers: Adam, Adagrad
- layers: 1, 2, 3, 4 and 5 layers of 300 dimensions
- embeddings pre-trained on the Norwegian newspaper corpus and NoWaC using either 300 or 600 dimensions (identifiers 128 and 129

on the NLPL embeddings repository), as well as embeddings pre-trained on our training corpus

- using all occurrences of a token when calculating the average (counts) or only counting each unique token once (binary)

In conjunction with the eight different dataset configurations, this yielded a total of 320 different configurations. A grid search was run on all these combinations.

6.2.1 Evaluation

Training times were very short across the board, with the caveat that the averaged embeddings were pre-generated. The slowest model trained in nine minutes. However, the results were quite far off from those of the feed-forward BOW network. The best-performing model, which trained in six minutes, achieved an F_1 score of 54.64.

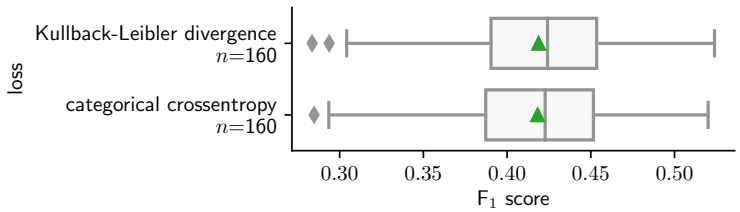


Figure 6.9: Effect of loss on F_1 score for averaged embeddings architecture

As figure 6.9 shows, it does not appear to be significant which loss function was used. Kullback-Leibler divergence and categorical cross-entropy are very similar formulae (see section 4.4.2), and as such this is not very surprising.

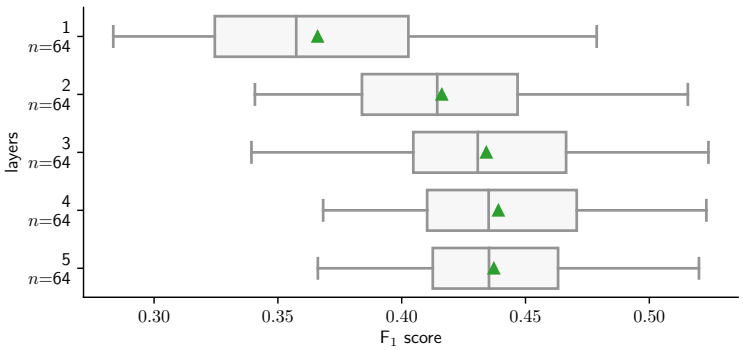


Figure 6.10: Effect of layers on F_1 score for averaged embeddings architecture

Increasing the number of layers appears to work very well to improve the performance of the network, at least to a point. The difference from

the feed-forward network trained on BOW representations in section 6.1 is stark. The BOW representations are sparse representations which in the first hidden layer are turned into dense representations, while the averaged embeddings which are used as input for the present architecture are already dense. It is possible that the already-dense representations generated here are more fit for recombination in subsequent layers.

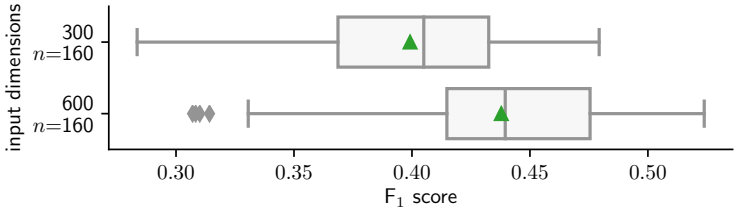


Figure 6.11: Effect of input dimensions on F_1 score for averaged embeddings architecture

Using embeddings of higher dimensionality proved a reliable way of increasing performance. Figure 6.11 shows an increase of several points by using embeddings of 600 rather than 300 dimensions. While the data upon which the embeddings were trained were the same, having more dimensions to indicate the relations between different words appears to be very helpful to the network. Adding dimensions to the hidden layers to match the increased size of the input could also be a way to utilize this added information further—as it is, all the networks used hidden layers of 300 dimensions.

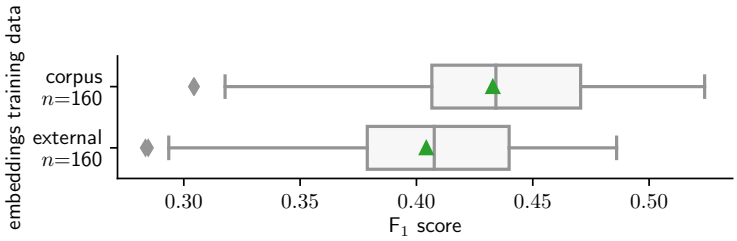


Figure 6.12: Effect of embeddings training data on F_1 score for averaged embeddings architecture

Using embeddings pre-trained on the ToN corpus gave consistently better results than using external pre-trained embeddings from the NLPL repository; figure 6.12 shows an increase in F_1 score very similar to that of using embeddings of higher dimensions. The corpus-trained embeddings are likely to lack embeddings for many of the words in the development set, being trained only on the training set; the external embeddings, on virtue of being trained on a much larger corpus, are likely to have fewer such holes. Any effect this may have had is obviously offset, however, by the increased relevance to the task of the in-corpus embeddings.

There was no real difference between the models trained on input

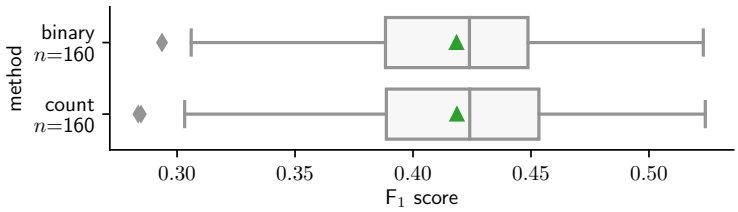


Figure 6.13: Effect of method on F_1 score for averaged embeddings architecture

representations generated by only looking at binary presence or absence of terms and those trained on input representations using every occurrence of a term, as shown in figure 6.13. Despite this evidence to the contrary, how the generation of input representations is handled is likely to be an important factor, as averaging a high number of distinct embeddings is likely to generate a great deal of noise. It is possible that neither of the examined methods is optimal, and that weighting the terms using TFIDF would make more sense, although Kutuzov et al. (2016) reports insignificant changes using this method.

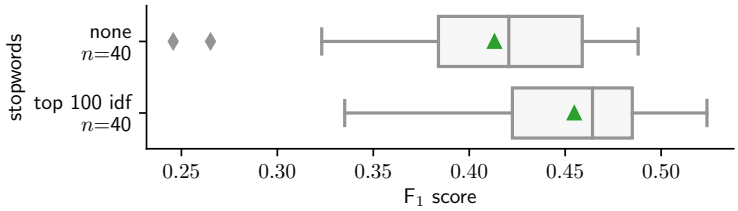


Figure 6.14: Effect of stop words on F_1 score for averaged embeddings architecture

The effect of not including common words and stop words was examined by generating a version of the best-performing input representation which did not exclude stop words. It seems clear that disregarding the most common words is very beneficial. This being a tunable hyperparameter, it would be interesting to examine what particular cut-off produces the most indicative document representations.

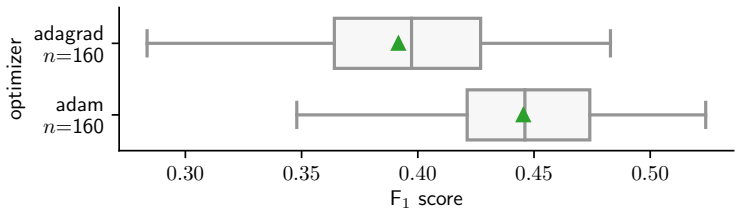


Figure 6.15: Effect of optimizer on F_1 score for averaged embeddings architecture

For this experiment, we obtained higher results using Adam than Adagrad. Compared to the experiments discussed in 6.1 and 7.1, the networks discussed in the present section were trained for many more epochs. The constantly decaying learning rate is the main weakness of Adagrad, and it is possible that this weakness, and the corresponding strength of Adam, did not have time to manifest themselves with fewer epochs. Moreover, the fact that many of the variants of the network were still learning at the end of training suggested that the number of epochs was insufficient.

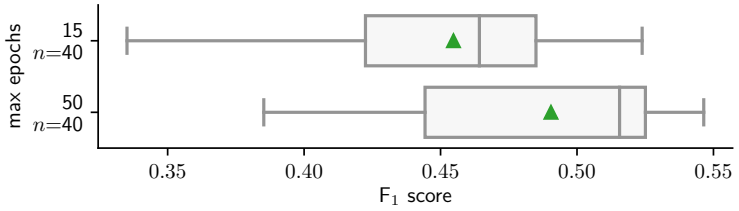


Figure 6.16: Effect of max epochs on F_1 score for averaged embeddings architecture

The hyperparameter search was rerun on the input representation that had given the best results, namely that using counts and ToN-trained embeddings of 600 dimensions. Figure 6.16 shows the benefit of increasing the maximum number of epochs to 50. This appears to be sufficient, since for all of the eight best-performing networks the model had peaked by the 40th epoch. This, combined with the use of Adam as optimizer and ReLU as the activation function, gives us the best-performing configuration for this architecture, with an F_1 score of 54.64. While this is quite a ways off the performance delivered by our other network architectures, the network itself trained in merely 6 minutes.

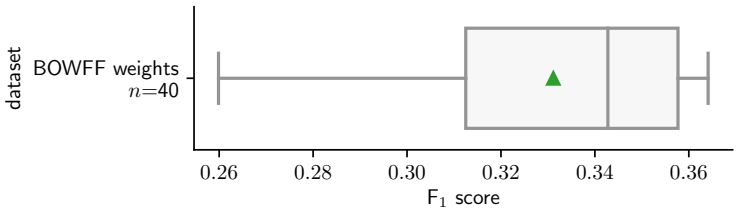


Figure 6.17: F_1 scores of models trained using the weights from the BOW feed-forward network as embeddings

Lastly an experiment was run using the weights extracted from the best BOW feed-forward network from section 6.1. In theory this could give similar performance to that network, since the averaged representations are generated in a similar manner to how the documents are transformed through the weights of the network and represented in the hidden layer of the BOW feed-forward network. In practice this worked poorly, with the

best configuration topping off at an F_1 score of 36.34, as we see in figure 6.17.

The models using averaged embeddings as input representations fail to deliver particularly impressive results when measured by F_1 score. The model is, however, very simple—while the best BOW representation used a vocabulary of 100 thousand words and therefore had that dimensionality, the averaged embeddings used only 600 dimensions. This extremely compact input representation means that even a very large dataset can be easily stored in memory, and training and prediction is very fast.

While this method does utilize semantic information through the use of word embeddings, the generated input representations still have no information on the order of or relation between the words of the text; they do no not even divulge what actual words were used, only the relations between different words as calculated through an application of the distributional hypothesis.

6.3 Hybrid network

In this chapter we have seen that a BOW feed-forward network can give results which are on par with the linear-SVM results detailed in Lapponi (2019). We have further seen that training a network on averaged embeddings is very computationally cheap. It is possible to combine the two, with parallel inputs and calculation paths, concatenating to a common vector before the last softmax layer. This then becomes a simple and cheap way to in essence augment the BOW feed-forward network with distributional semantics. Taking in this manner the best configurations for the BOW feed-forward architecture and the averaged embeddings architecture[1] yields the architecture shown in figure 6.18

The resulting model gave an F_1 score of 68.06 on the development set. This was 0.68 higher than the simple BOW feed-forward network, suggesting that enhancing such a network with averaged embeddings may potentially supply new information that is useful for the classification task.

6.4 Conclusion

The models we have seen in this chapter are all relatively simple, in that they only consist of dense layers that feed forward. They are relatively simple to analyse, and the BOW representations seem to give a good amount of signal for the models to perform the classification task. Still, the BOW representation only gives information about what words are used and with what frequency. Ideology is not only predicated on what words are used and which issues are raised, but on the sentiment that is expressed with regards to said issues. With this in mind, we would like to see if we are able to put to use more advanced architectures that can consider the content of the speeches on a deeper level.

1. with the exception of the optimizer, for which Adagrad was used for the entire model

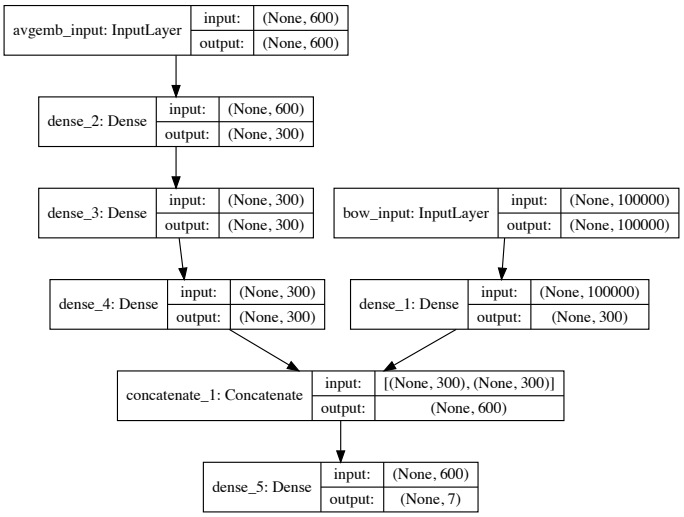


Figure 6.18: Architectural graph of the hybrid BOW and averaged embeddings feed-forward network, from Keras

Chapter 7

Advanced architectures

This chapter builds on the experiments in the previous chapter and presents experiments using somewhat more advanced neural architectures. Section 7.1 deals with convolutional neural networks (CNNs). Here we again find large differences between optimizers, and attempt to reason about the causes for these; we then go on to discuss how to measure the effect of including different window sizes. In section 7.2 we move on to recurrent neural networks (RNNs), specifically long short-term memory networks (LSTMs). We will see a tendency for our LSTM networks to train poorly when the number of parameters in the recurrent layer is high. We discuss various methods of processing the output from a recurrent layer, finding good results with max pooling and self-attention. Table 7.1 shows results for the different architectures.

Network	Accuracy	Precision	Recall	F_1 score	Train time (min)
CNN-EL	67.23	68.18	65.77	66.82	3177
CNN-PT	60.63	62.38	58.35	59.65	3074
BiLSTM-Max	67.15	68.38	65.64	66.69	1382
BiLSTM-Att	67.00	68.57	65.29	66.67	3575

Table 7.1: **CNN-EL**: CNN with an embedding layer, vocabulary size: 100,000, dropout: 0.2, convolutional windows: [1, 2, 3, 4, 5, 10, 20, 50], no. filters: 200; **CNN-PT**: CNN on fastText embeddings pre-trained on ToN corpus; otherwise same hyperparameters as CNN-EL; **BiLSTM-Max**: bidirectional LSTM with an embedding layer, 256 output dimensions per pass (total 512), no regularization, max pooling; **BiLSTM-Att**: bidirectional LSTM on fastText embeddings pre-trained on ToN corpus, 256 output dimensions per pass (total 512), 0.2 recurrent dropout, attention layer

7.1 Convolutional neural network

CNNs are, as mentioned in section 4.2, apt at extracting local feature combinations, since they consider all the words within an n-gram. This should be helpful in our study of ideology, since in addition to ascertaining

the topic at hand, we would like for our classifier to consider what is said *about* the given topic—most parties discuss the same topics, but what they have to say about them can differ greatly. Experiments were run with an embedding layer, followed by a one-layer CNN architecture with different combinations of window sizes, number of filters and regularization values. The hyperparameters tested were:

- window sizes: 4 CNN nodes, where each picked a unique window size from 1, 2, 3, 4, 5, 10 or 20
- number of filters: 100 or 200
- regularization: dropout with a value of 0 or 0.2
- optimizer: Adagrad or RMSprop
- vocabulary size for embedding layer: 50 or 100 thousand
- 300 dimensions in embedding layer
- sequences padded to a length of 1000

The longest speech in the corpus consisted of 10,005 tokens, but the CNN architecture requires a fixed input length. Even with the high-performance computing cluster used for these experiments, computational resources are not unlimited, so it would be infeasible to pad all the sequences to this length; therefore, a maximum sequence length of 1000 tokens was chosen. This caused 5941 of the texts in the training set to be abridged; for these only the first 1000 tokens were considered. This also applies to 633 texts in the development set. Out of the 560 possible combinations of the hyperparameters detailed above, 100 were selected randomly.

The smallest model had around 15 million parameters, while the largest model had around 32 million parameters. This was mainly a function of the vocabulary size, since the embedding layer output 300 dimensions, and the number of parameters in this layer is the product of the vocabulary size and output dimensionality. Training time varied between five and 22 hours.

All the models used a single convolutional layer with max pooling and rectified linear unit (ReLU) activation.

Evaluation

The model which performed best on the development set had the window sizes of 1, 2, 4 and 20, with 200 filters each, yielding a total of 31,626,707 parameters, and no dropout. The model trained in 17 hours, and gave an F_1 score of 66.82 on the development set.

Of all the the hyperparameters which were tested, the one that stood out the most when looking at the F_1 score turned out again to be which optimizer was used, as was the case for the feed-forward network discussed in section 6.1. An initial CNN run showed models using

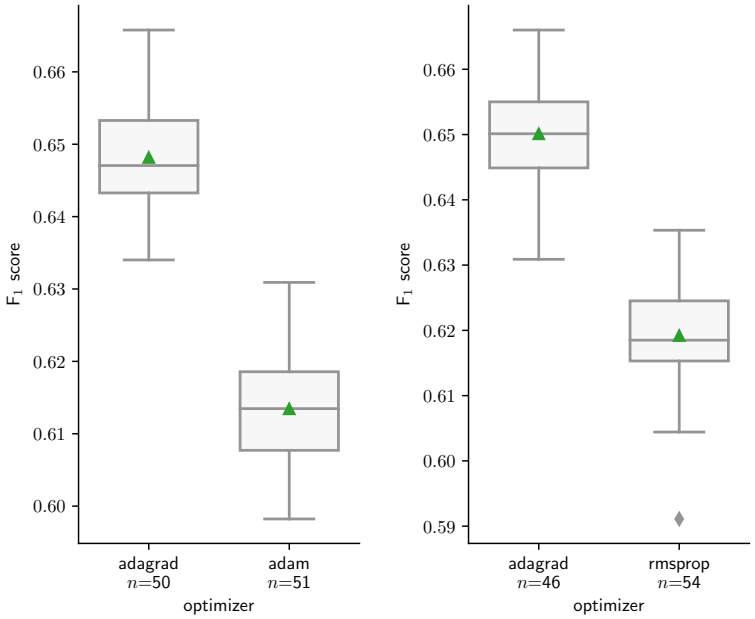


Figure 7.1: Effect of optimizer on F_1 score for CNN

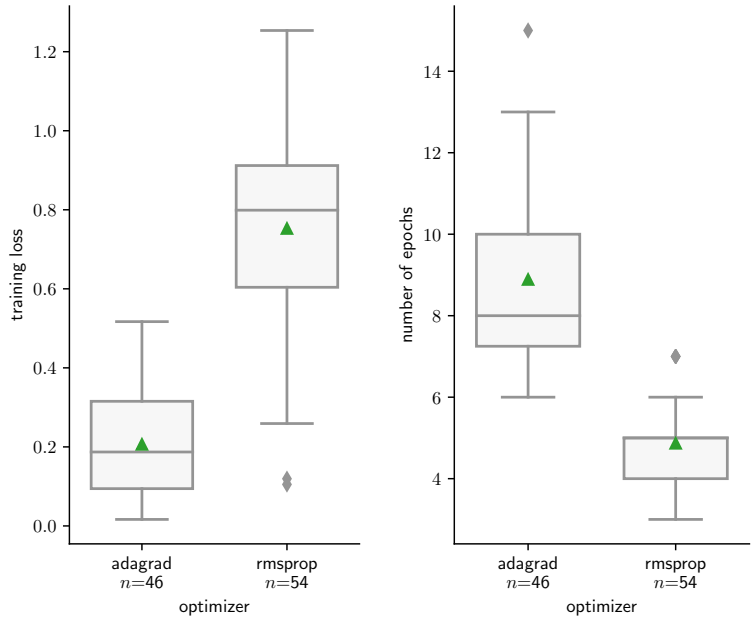


Figure 7.2: Training loss at training stop and number of epochs depending on optimizer for CNN

Adagrad outperforming every model using Adam, so RMSprop was chosen for comparison with Adagrad instead. As discussed in section 4.4.3, Adagrad will always reduce the learning rate η, while both RMSprop and Adam use an exponentially decaying average of past squared gradients to adjust it. Adam enhances this with momentum (Ruder 2016). The default

initial η for Adagrad in Keras is much higher, at 0.01, than that for Adam and RMSprop, at 0.001.

While the results from the initial experiment shown to the left in figure 7.1 cannot be directly compared to the rest, neither Adam nor RMSprop seem to be able to get anywhere near Adagrad. Since Adam and RMSprop are generally found to be better and faster optimizers, a hypothesis to explain these poor results could be that the models were in fact well-optimized on the training data but overfitted on the training data. The graphs showing loss on the training data and the number of epochs before early stopping in figure 7.2 suggests that this was, in fact, not the case; rather, training was terminated early due to to lack of improvement in F_1 score on the development set. Most likely tweaking the hyperparameters given to Adam or RMSprop would be beneficial.

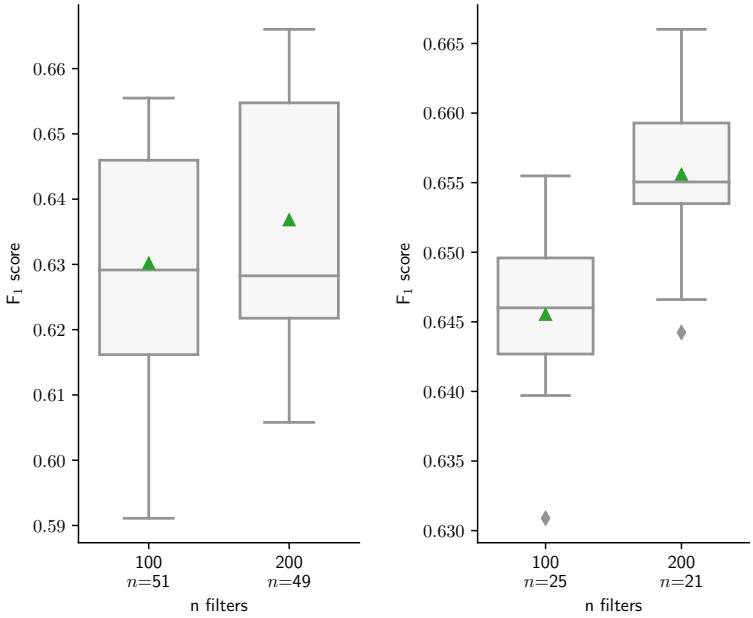


Figure 7.3: Effect of number of filters on F_1 score for CNN, all models (left) and Adagrad (right)

Since the hyperparameter search for this experiment was performed using random search rather than a full grid-search, certain hyperparameter settings occur more often with adam as optimizer than with Adagrad, and vice versa. Given that this single hyperparameter accounted for most of the divergence in results, it is likely to skew the results when looking at the other hyperparameters individually. Indeed, figure 7.3 illustrates this quite clearly: The graph to the left appears to show that that doubling the number of filters has little to no effect on the F_1 score, with the median F_1 score even falling below that of the models using 100 filters. When we look only at the models trained using Adagrad, however, we see a much clearer picture: The best network using 100 filters was beaten by nearly half of the models using 200 filters, as we can see that it is barely above the median line.

The analyses of individual hyperparameters that follow take only into account those models which were trained using Adagrad. To what extent the higher performance of the models with more filters is influenced by other variables again is, of course, another question, that will not be examined here. Overall, all the models trained using Adagrad performed similarly, with only a gap of 3.6 between the best and worst models.

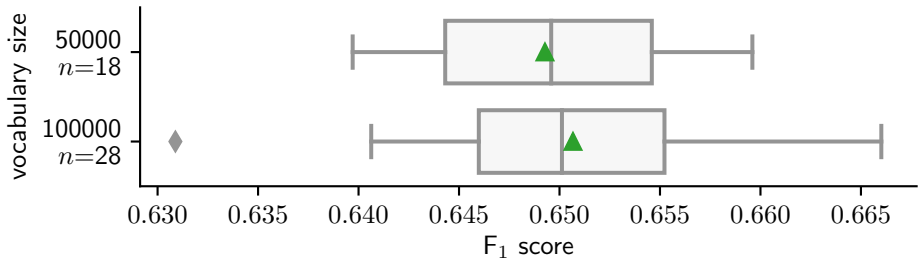


Figure 7.4: Effect of vocabulary size on F_1 score for CNN

Vocabulary size appears to have had less of an effect on performance compared to the feed-forward architecture. Since the word forms that fall below the threshold by being too rare in the training corpus are replaced by an out-of-vocabulary token, the models utilizing a smaller vocabulary receive more training in the recognizing rare words. One may imagine that this could be advantageous when classifying the development set, which is bound to have many unknown words.

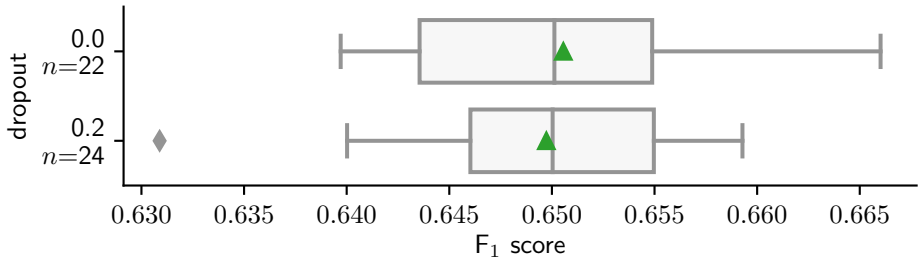


Figure 7.5: Effect of dropout on F_1 score for CNN

Dropout as regularization was also tested. Preliminary experiments had shown that dropout values of 0.5 and 0.7 tended to give poor results on the same data using pre-trained embeddings, so only the value of 0.2 was tested, along with using no regularization. There seems to be a tendency for dropout to reduce the performance of the trained models, with the six best models having no regularization.

7.1.1 Attempting to isolate effects of different window sizes

We noted that the four highest-performing models included 1 as a window size. Having 1 as a convolution window size may seem counter-intuitive,

given that it seems to go against what we think of as the purpose of a convolutional window, that is, considering local dependencies. There is an interesting finding in Jacovi, Shalom, and Goldberg (2018) which suggests that certain "slots" in a window will, for certain filters, specialize on particular words. When these words occur, they then maximize the pooling dimension, and the rest of the slots in the window are effectively ignored. Given that this is the case for individual filter slots, and the fact that the bag-of-words (BOW) experiments discussed in section 6.1 demonstrated that individual words carry much useful signal, we may be able to see an effect from filters considering only one word. The current experiment setup however made it difficult to isolate the effect of including a given window size, since most of the tested window sizes co-occurred with each other frequently and with various other hyperparameters. Further experiments were therefore performed, using the otherwise best configuration—Adagrad, vocabulary size of 100 thousand, and no dropout—in a grid search including the following hyperparameters:

- number of filters: 50, 100 or 200
- adding increasingly large window sizes, starting with 1, 2, 3, 4 and 5, and successively adding to this with window sizes of 10, 20 and 50.

A comparison was made between models that included 1 as a window size and otherwise equal models not employing this size. The results for this part can be seen in figure 7.6.

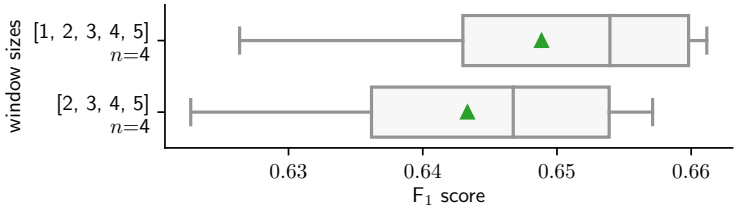
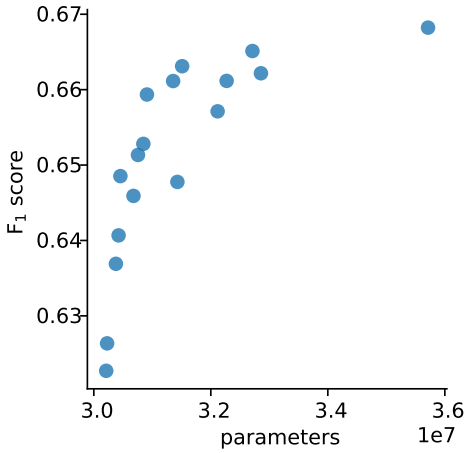


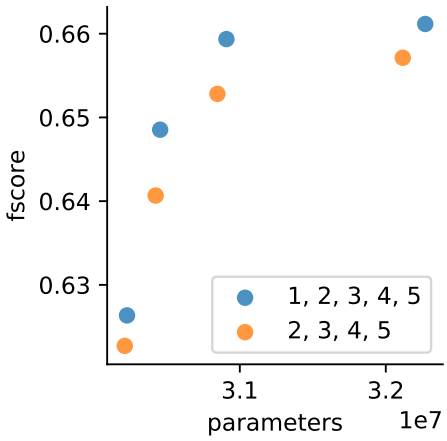
Figure 7.6: Effect of including a window size of 1 on F_1 score for CNN

Figure 7.6 appears to suggest that including convolutional windows of size 1 does indeed help the network. The question is whether this is is merely due to the fact that the number of parameters in, and thereby the learning capability of, the network is increased. Figure 7.7a shows the correlation between number of parameters and F_1 score for the models trained in this part of the experiment and supports this assumption—the models with a high number of parameters do pretty consistently outperform those with less, although adding a window size of 50 demonstrates a clear case of diminishing returns. When we compare just the models including 1 as a window size with those otherwise equal, the effect appears to outstrip the simple correlation, with an F_1 score increase

of up to 1 by including this window size at most filter sizes as can be seen in figure 7.7b. This suggests that this unconventional window size does indeed have some merit.



(a) All models



(b) only the model configurations with [1, 2, 3, 4, 5] or [2, 3, 4, 5]

Figure 7.7: Correlation between number of parameters and F_1 score for CNN (30 million parameters in embedding layer)

7.1.2 Pre-trained embeddings

The best-performing model configuration, which by a small margin was that which included a window size of 50, was subsequently tested using pre-trained fastText embeddings. Embeddings model 129 from the NLPL repository (Fares et al. 2017b), trained on Norwegian Web as Corpus (NoWaC) and the Norwegian News Corpus, was used along with our 300-dimensional embeddings trained on the ToN corpus. Both models yielded significantly poorer results than the models which trained embeddings: The model trained using ToN-trained embeddings managed an F_1 score

of 59.65, while the external embeddings gave a result of 58.34. While the fastText embeddings are arguably superior to those generated by our embedding layer in the sense that they can supply meaningful representations for unseen words through looking at character n-grams, these embeddings are trained without knowledge of the downstream task. This may indicate the usefulness of task-specific training, although better results may have been seen with a different hyperparameter configuration.

7.1.3 Effect of non-determinism

The experiments we have seen were all run without accounting for all the effects of non-determinism; that is to say, no attempt was made to fix random seeds. In order to measure the effect that this may have had on the experiments, one of the hyperparameter configurations described in this section was trained and evaluated twenty times.

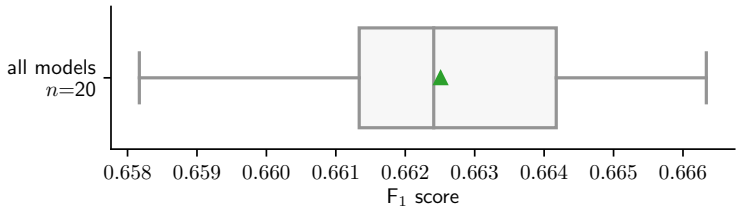


Figure 7.8: Effect of non-determinism on CNN classifier

The configuration chosen was that which provided the best results in the original random search, prior to the window size investigation in subsection 7.1.1, specifically that which had window sizes of 1, 3, 5 and 20 with 200 filters, no dropout or L_2 regularization and Adagrad as optimizer. The resulting F_1 score varied between a minimum of 65.82 and maximum of 66.63, with a mean of 66.25; this is illustrated in figure 7.8. With the margins between the highest-performing models being so narrow in the original experiment, it is prudent to keep in mind that other configurations may have easily shown an advantage given a different random initialization.

Looking at the confusion matrix of the best CNN model, seen in figure 7.9, it is quite similar to that we saw for the best feed-forward BOW network. The CNN model appears to be somewhat less eager to use the largest classes, Conservative Party (H) and Labor Party (Ap), with the scores more evenly spread. Notably, this model also found the Progress Party (Frp) to be easiest to classify, giving a recall score of 0.74 for that class.

One of our CNN models succeeded in roughly matching the performance of the feed-forward network trained on BOW representations in terms of F_1 score on the development set, but it failed to surpass it. While the CNN architecture is able to consider local dependencies within defined window sizes, it is unable to make connections beyond these boundaries. It will also generate the same representation for a given n-gram at any position in the sequence. One may suppose, though, that the position of the

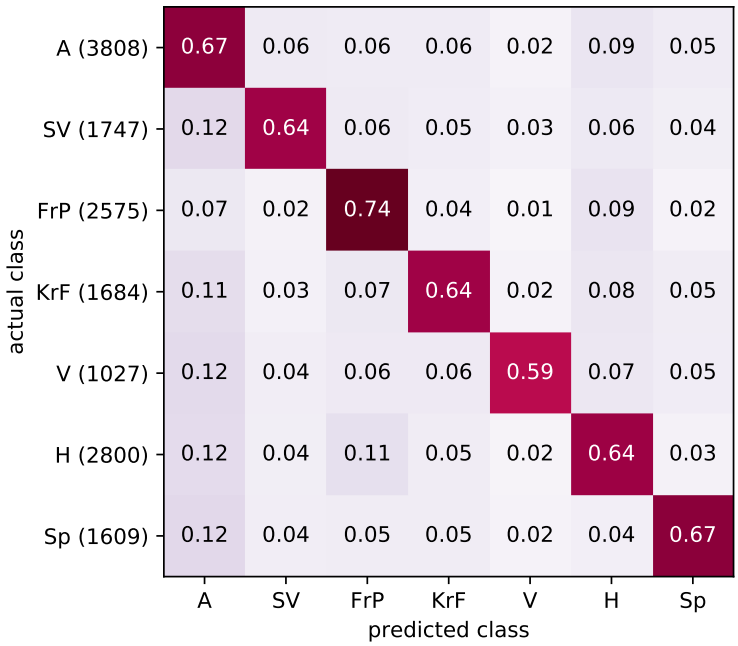


Figure 7.9: Confusion matrix of best CNN, normalized by class support (shown in parentheses)

n-gram in the speech could also be indicative of the intent of our parliamentarians; whether something is said in the beginning, middle or end of a speech could be significant. We therefore move on to an architecture which can both capture longer dependencies and consider positions in a sequence.

7.2 Recurrent neural networks

RNNs, as discussed in 4.3 have many variants in addition to the vanilla variant, such as LSTMs and gated recurrent units (GRUs). Their strength lies in capturing long-range dependencies, since information can be carried across a theoretically infinite amount of time steps, although we in practice are limited by gradient flow. For the experiments in this section, the LSTM variant of recurrent networks was used.

7.2.1 Parameters

For this set of experiments the following hyperparameter space was defined:

- recurrent unit type: LSTM
- output dimensionality: 64, 128, 256, 512, 1024 or 2048
- bidirectional: whether or not to train on backward pass of the input data

- regularization: dropout with a value of 0, 0.2 or 0.5
- optimizer: Adam or Adagrad
- vocabulary size for embedding layer: 100,000

All the models used only the output from the final hidden state for classification, and the intermediate steps were not used.

7.2.2 Initial difficulties

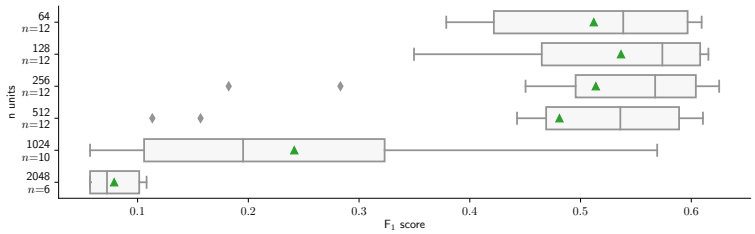


Figure 7.10: Effect of number of output dimensions on F_1 score for initial RNN experiments

An initial experiment with the RNN architecture ran into problems, which we discuss below. The best-performing input representation, i.e. an embedding layer with a vocabulary size of 100,000 was used. On this input representation we applied various numbers of recurrent units combined with other parameters in a grid search. Surprisingly, the models with a high number of recurrent units were unable to train properly. The most parameter-rich models—even though they took multiple days to train—were frequently unable to make very decent classification decisions at all. The worst-performing models, showing an F_1 score of 5.7, appeared to have devolved into majority-class classifiers after one epoch.[1] Figure 7.10 shows how the models with a high number of recurrent units frequently devolved into such networks, demonstrated by the extremely low classification scores. As discussed in section 4.3, the major weakness of RNNs is the difficulty of propagating gradients back through all the time steps in the sequence. But this does not serve as an explanation for the results seen here, since while the somewhat better-performing models had fewer output parameters, they still used the same sequence lengths.

It is possible that the dataset is simply not large enough, or the signal therein not disambiguous enough, for these additional parameters to serve any purpose. One may speculate that introducing a high number of parameters to the recurrent layer simply added noise to the signal sent to the downstream classification layer. At the same time, the models with a moderate output dimensionality units failed to significantly improve upon the simplest models, those with an output dimensionality of only 64,

1. Training was stopped for models once they showed performance indicating a uniform random classifier or a majority classifier, as described in table 5.1.

even though we in section 7.1 saw models with a much higher number of parameters performing well. Considering these initial setbacks, the hyperparameter space for the RNN architecture needed to be rethought.

7.2.3 New set of experiments

A new set of experiments was formulated, informed by the experiment above. In addition to the models using a high number of output units, results were also poor in those models which used Adagrad as an optimizer; dropout regularization with a value of 0.5 also performed poorly. An early test also showed that using static pre-trained fastText embeddings could potentially outperform the models using an embedding layer, so this was also tested, even though the same embeddings had performed poorly in the CNN experiments as reported in section 7.1.2. Thus we had our new set of hyperparameters:

- recurrent unit type: LSTM
- output dimensionality: 128, 256 or 512
- bidirectional: whether or not to train on backward pass of the input data
- regularization: no regularization, or recurrent dropout with a value of 0.2
- optimizer: Adam
- pre-trained fastText embeddings or an embedding layer with a vocabulary size of 100,000
- RNN layer output generation: last state, mean pool of all states, max pool of all states, or self-attention
- sequence length of 1000 for self-attention networks, full sequences for rest
- RNN layer output generation: last state

This set of hyperparameters gave 96 different combinations.

RNNs are able to reduce arbitrarily long sequences to a fixed output size, and Keras supports input sequences of variable length to its RNN layers, as long as all sequences within a batch are of the same length. Keras allows for batch-wise generation of training samples through its Sequence class, and therefore it was possible to pad the samples in each batch to the length of the longest sample, thereby ensuring that all the data for each sample were preserved. However, the static computational graph model imposed a limitation on the architecture of the attention layer so that these models required a fixed sequence length. For these models, the sequence length was therefore fixed at 1000 time steps; this is equivalent to what was done in the CNN experiments. Training time was—as with the other

architectures—limited to 15 epochs, but here we also need to mention that there was a clock time limit of 132 hours. There were five models which had neither reached 15 epochs nor had early stopping triggered at this point. For these models the F_1 score at the last completed epoch is used for evaluation.

7.2.4 Evaluation

The updated hyperparameter space excluded models with a very high number of output dimensions, i.e. 1024 or 2048. This was done both because training was too slow to realistically complete and because the models appeared to learn relatively poorly.

In an RNN the hidden states from preceding time steps affect the hidden state and output of subsequent steps, so that a given state is conditioned on this preceding information. This by default only works one way. A bidirectional RNN will in addition run over the sequence backwards, so that for each time step a representation is generated which is conditioned on the time steps which come after. This creates twice as many parameters and is quite computationally demanding.

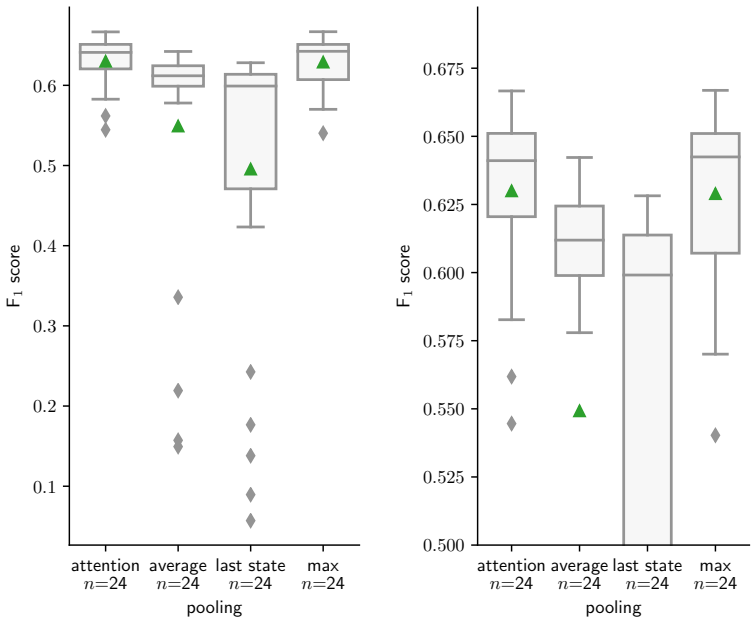


Figure 7.11: Effect of pooling method on F_1 score for RNN, all models (left) and those over an F_1 score of 50.0 (right)

An RNN creates different types of output, which may be used or combined in different manners. One may commonly use the output from the last state, as we did in the first RNN experiment, and consider this an encoding of the sequence. Apart from this method, the updated experiment also tested others, the details of which are described in section 4.3.1. All of these methods resulted in better results than only using the output from the last state, although average pooling was not particularly effective.

Compared to the other architectures examined, the RNN networks exhibited a high variance in terms of final F_1 score. The models using the output from the last state included a high number of models which were simply very poorly fit, and joining them were several of the models using averaged pooling. Many of these had F_1 scores of under 50.0, as we see to the left in figure 7.11. To more clearly see the performance of the more interesting higher-performing models, the graph to the right in figure 7.11 is cropped at the 50.0 F_1 score mark. The graphs which follow in this section will mostly continue this cropping, essentially hiding the poorer models using averaged pooling and the last state.

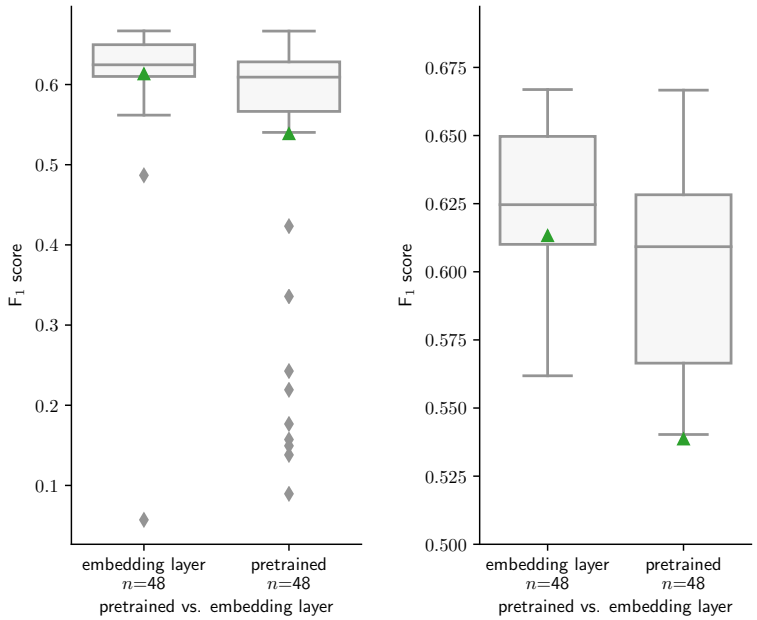


Figure 7.12: Effect of pre-trained embeddings vs. embedding layer on F_1 score for RNN, all models (left) and those over an F_1 score of 50.0 (right)

Since the sequences in the ToN dataset are very long, using only the output from the last state means that the supervision signal has to travel very far from the loss calculation in order to reach the earlier time steps, or the middle steps when running over the sequences both ways as in the bidirectional RNN models. By running an operation which takes into account all of the intermediate time steps, we establish a series of much shorter paths for the gradients, which may have helped to better preserve them. The attention models and the max pooling models performed remarkably similarly as measured by the mean, median and max F_1 scores. The best attention model gave a score of 66.67, vs 66.69 for the best model using max pooling.

For the document representations, we used two different configurations: 1) an internal embedding layer, equal to that of the CNN models, as well as 2) our 300-dimensional fastText embeddings trained on the ToN training corpus. All but two of the models performing worse than 50.0

were trained on pre-trained embeddings rather than using an internal embedding layer, as we see in figure 7.12. The best model using pre-trained embeddings performed just as well as the best model with an embedding layer, as is easier to see in the figure once we crop out the outliers.

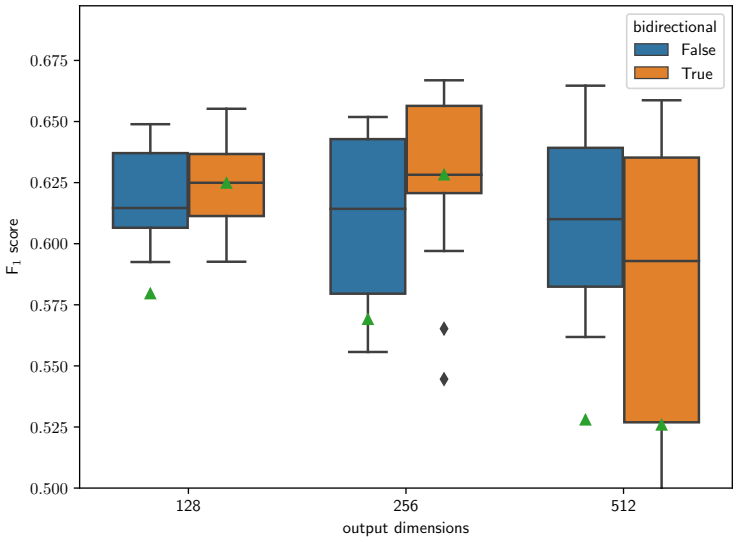


Figure 7.13: Effect of output dimensionality on F_1 score for RNN, cropped at 50.0; the effective dimensionality is doubled for the bidirectional models

Increasing the output dimensionality, and thereby number of parameters, served to increase classification performance to a certain point. Bidirectional models tended to outperform those models which were only trained on a forward pass of the sequence, but this falls apart once the number of output dimensions per pass reaches 512. The bidirectional models with 512 dimensions per pass had a total of 1024 dimensions. As we saw in the first RNN experiment a high output dimensionality from the RNN layer (as visualized in figure 7.10) was associated with a breakdown in performance. While we might expect the dense layer connecting the RNN output to the classification layer could learn to focus on the important dimensions, this appears not to be the case; past a certain threshold, adding more dimensions is detrimental to learning. Interestingly, the best unidirectional model using an output dimension of 512 nearly matched the two best models using bidirectional 256-dimensional outputs, with an F_1 score merely 0.22 points lower than the best. This may suggest that apart from the question of whether to run through the sequence bidirectionally, there is rather an optimal output dimensionality for the downstream classification layer. At the same time, the number of parameters in one LSTM layer grows exponentially with the output dimensionality, so combining two layers bidirectionally gives a simpler model which is quicker to train.

Since RNNs have many different types of parameters, there are consequently many different ways of applying regularization, depending on what type of parameter we wish to regularize. In this experiment we

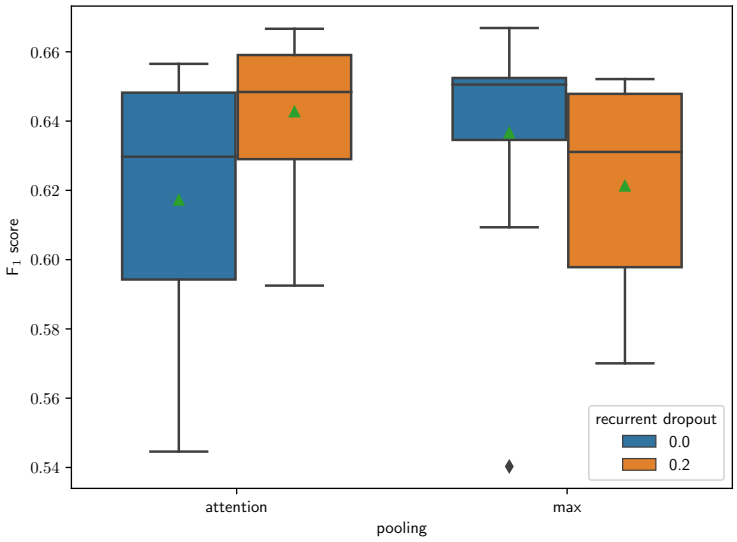


Figure 7.14: Effect of recurrent dropout on F_1 score for RNN

tested dropout with a value of 0.2 on the recurrent parameters, i.e. those connecting the different time steps. Focusing on the models using self-attention and max pooling, the usefulness of recurrent dropout appears to hinge on the way the time step outputs are combined. While the self-attention networks benefitted from dropout, with a mean score 2.5 points higher, for the max pooling networks the score decreased by 1.5 points, as we see in figure 7.14. With an RNN it is also possible to apply dropout to the input sequences or to the outputs, which can be expected to have a different impact.

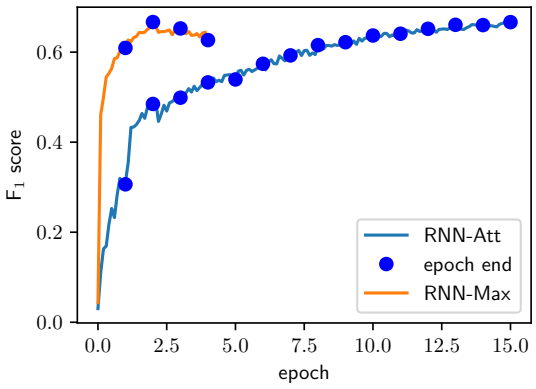


Figure 7.15: Evolution of F_1 score on development set within epochs, with the blue dots representing scores calculated at epoch boundaries

This set of experiments gave two models at the top of the ranking with nearly equal F_1 scores on the development set. What is remarkable

is that the hyperparameters used by these two networks differ in nearly every way, apart from the fact that both were bidirectional with 256 dimensions per pass over the sequence. The model which we call BiLSTM-Att, on account of employing an attention layer, used pre-trained fastText embeddings and recurrent dropout with a value of 0.2. The model using max pooling—BiLSTM-Max—on the other hand, used an embedding layer and no regularization. The former model took many more epochs to train, with the best result reported at the last, 15th epoch. The latter, on the other hand, peaked in performance after two epochs, whereafter early stopping was triggered, as we see in figure 7.15. Since the former had less parameters to update due to there being no embedding layer, time per epoch was a bit shorter at 238 minutes vs. 345 minutes per epoch, but total training time was still much longer at 60 hours vs. 23 hours.

The models using external, pre-trained embeddings need to learn how the dimensions in these embeddings relate to each other and to the downstream classification task. In the models with an internal embedding layer, on the other hand, the embeddings are learned internally from scratch and are therefore directly adapted to the particular classification task. It seems that learning to associate the external embeddings with the classification task requires more passes through the training set. Looking away from the longer training time required, the self-attention network is possibly the superior configuration: The model was still learning at epoch 15, indicating a potential for further improvement, and it was being trained on less data, the sequences having been trimmed to 1000 tokens.

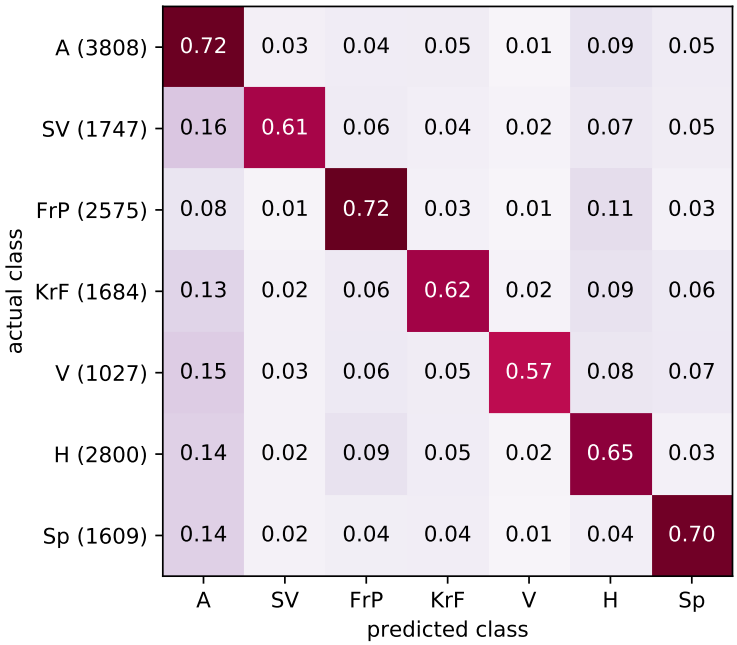


Figure 7.16: Confusion matrix of best RNN, normalized by class support (shown in parentheses)

The confusion matrix shown in figure 7.16 shows a somewhat larger

divergence between the classes, with the recall score for the Liberal Party (V) as low as 57%. Otherwise it is very similar to those of the feed-forward BOW network and CNN that we have seen before.

Figure 7.17: Visualization of attention over short text sequence. Blue indicates low weight, while red indicates high weight.

Self-attention produces a weight for each time step in a sequence, and these weights can be combined with the original text to visualize where the network's attention lay. Figure 7.17 shows one of the shortest speeches from the corpus, speech no. 229723, with each word weighted from blue (not considered) to red (heavily considered). This speech demonstrates a pattern that was frequently observed, in which later time steps were more or less ignored. In the first thousand speeches of the development set, the mean weight given to the first ten tokens was 4.8 times higher than that given to the last ten tokens. One could make a humorous analogy with humans, where we start to lose attention the longer we listen to a speech, but this is not likely to be what causes this behavior. On the contrary, we could reasonably expect the last part of a speech to be relatively dense with signal, since a speaker might want to wrap up or conclude. The explanation is likely to do with padding. Due to the fixed sequence length of the self-attention network, speeches of less than 1000 tokens are post-padded—essentially adding empty embeddings—and the attention layer learns to ignore the padded time steps, by simply assigning an insignificant weight, close to zero. What is surprising is that this appears to leak over to the last sequence elements, leading to these in many cases not being considered. The example shows how attention also works to disregard punctuation and function words. Otherwise, the similar weighting given to nearly all of the remaining tokens means that it in cases such as this works similarly to mean pooling.

For longer sequences the effect was very different. Figure 7.18 shows one of the longest sequences, which has been trimmed at 1000 tokens. In this example, which is representative for documents of this length, the attention mechanism chooses to ignore large swathes of the document, while focusing on particular areas of the text.

The RNN networks we have seen here had very divergent performance. It proved difficult to find appropriate hyperparameters, as some configurations gave rise to networks which were completely incapable of learning. Surprisingly, we saw both models that were able to make good use of pre-trained fastText embeddings, as well as models that performed well with an internal embedding layers. Although the embedding layer adds 30 million parameters to the network, these did not prove too difficult to train. Adding too many parameters in the recurrent layer, on the other hand, turned out to be disastrous.

Figure 7.18: Visualization of attention over long text sequence. Blue indicates low weight, while red indicates high weight.

7.3 Conclusion

In this chapter we have seen two types of advanced neural network architectures. None of the trained models were able to exceed the feed-forward network trained on BOW representations introduced in section 6.1, despite taking much longer to train. As the complexity of the architecture grows, the number of hyperparameters to tune grows with it. These results show the importance of considering individual hyperparameters for such advanced networks.

Chapter 8

Testing and discussion

Section 8.1 of chapter will discuss the results from the best-performing architectures on the held-out test set discussed in section 5.2. Section 8.2 presents an ensemble classifier, making use of all the other models discussed to surpass all other results.

8.1 Evaluation on held-out test set

For each type of model architecture, that configuration which performed best on the development set was used to train a new network. The trained networks were then evaluated on the held-out test set described in section 5.2.

The table in 8.1 shows the results on the development and test sets for the best hyperparameter setting for each of the various architectures we have examined. The hyperparameters were used to train new models for this evaluation, so the development scores in the table differ from those reported in the original chapters.

FF-BOW refers to the the feed-forward network using bag-of-words (BOW) representations discussed in section 6.1. It used a vocabulary size of 100,000 words, adaptive gradient algorithm (Adagrad) as optimizer, L_2 regularization with a value of 0.0001 and had one hidden layer. Despite the simple architecture, this is the network which gave the highest performance on the test set, with an F_1 score of 67.87.

FF-AE is the feed-forward network discussed in section 6.2. The best hyperparameter configuration here used representations build using 600-dimensional fastText embeddings trained on the Talk of Norway (ToN) corpus. This network used adaptive moment estimation (Adam) as optimizer, rectified linear unit (ReLU) as activation function and Kullback-Leibler as the loss function. It had three hidden layers. This gave an F_1 score of 53.63.

FF-BOWAE is a hybrid of the the two above networks and is treated in section 6.3. This took the two above networks in parallel and concatenated their output before the classification layer. Hyperparameters correspond to those used for the individual networks, with the exception that Adagrad was used as an optimizer, since using a different optimizing method on

Network	F_1 score	
	development	test
FF-BOW	**67.79**	**67.87**
FF-AE	51.97	53.63
FF-BOWAE	67.25	66.96
CNN-EL	66.75	67.00
CNN-PT	59.72	60.16
BiLSTM-Max	65.49	65.74
BiLSTM-Att	64.21	64.76
ensemble	71.25	71.55

Table 8.1: Change in performance between development set and test set for all architectures

different paths, while possible, would be non-trivial. This architecture and configuration was that which originally gave the highest score on the development set, with a score of 68.06. The model trained for this evaluation was however beaten by the pure BOW network on both the development and the test sets, its F_1 score falling to 66.96 on the latter.

CNN-EL is a convolutional neural network (CNN) with an embedding layer and a vocabulary size of 100,000. The configuration here uses dropout with a value of 0.2 and used the convolutional window sizes 1, 2, 3, 4, 5, 10, 20, 50, with 200 filters each. **CNN-PT** is a CNN with the otherwise same parameters, but instead of an embedding layer, the input was generated using 300-dimensional fastText embeddings pre-trained on the training set. Both actually performed slightly better on the test set than on the development set, with the CNN with an embedding layer only .87 points behind the feed-forward BOW network. It should be noted that the **CNN-PT** was only trained using otherwise the same hyperparameters as the **CNN-EL** network, and we can assume that a different set of hyperparameters would perform better on this input representation.

BiLSTM-Max is a bidirectional long short-term memory (LSTM) with an embedding layer. The recurrent layer had 256 output dimensions per pass, giving a total output dimensionality of 512 per time step in the combined bidirectional representation. This hyperparameter configuration included no regularization, and max pooling was employed to combine the time steps into a fixed-size representation for the classification layer. **BiLSTM-Att** is also bidirectional LSTM with the same dimensionality, but it ran on inputs from our 300-dimensional fastText embeddings pre-trained on the training set. This configuration used recurrent dropout with a value of 0.2, and self-attention was used to generate the fixed-size output representation from the time steps.

One can expect results on the test set to be somewhat below those on the development set. Because performance on the development set is used both as a stopping criteria for training and to select the "best" model

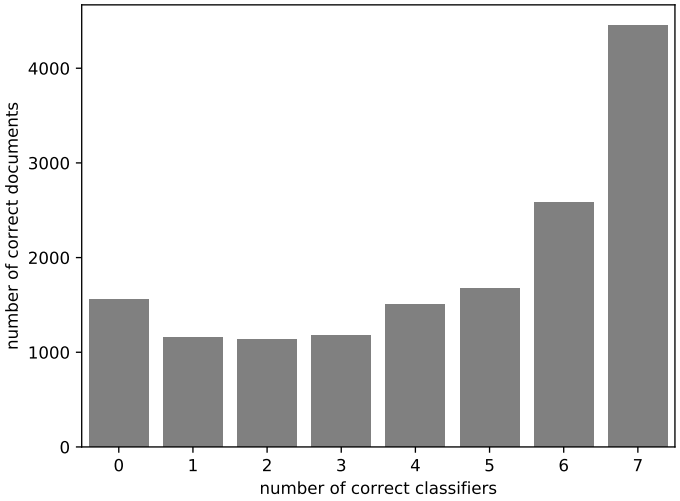


Figure 8.1: Graph showing how many documents in the test set were correctly classified by how many classifiers

configuration for the architecture, we are in essence to an extent using the development set for training, which makes our chosen models likely to be better fit on this set. The results do not entirely bear this out, since all but the **FF-AE** and **FF-BOWAE** models gave higher results on the test set. The previous champion, the feed-forward network on a combined input of averaged embeddings and BOW, sees a great drop in performance. Some of this can likely be attributed to non-determinism and the resulting initialization of the weights. In theory the **FF-BOWAE** network should have the same discriminative ability as the the **FF-BOW** network, since the former keeps all of the parameters of the latter while only adding an additional track. If the additional track does not contain information useful for classification, the model should in theory be able to learn to ignore it. In any case, the added information does not seem to be all that useful.

The LSTM models trained for the held-out testing performed significantly worse than the models trained during the original hyperparameter search. This does not mean that they simply performed better at the development set—in fact, the **BiLSTM-Max** network trained for this chapter only had an F_1 score of 65.47 on the development set, while the **BiLSTM-Att** network only achieved 64.21. Despite the fact that the networks were trained using the exactly same hyperparameters, random elements lead to these models learning much more poorly. We do not see nearly as large of an effect from this with the other architectures. In the context of our widely varying F_1 scores from section 7.2.2, we can conclude that LSTMs are challenging to train.

One striking aspect of these results is that the final F_1 scores are relatively similar across different architectures, with scores between 66 and 68 observed for the best models trained using each type of network.

This may suggest that the signal inherent in regularized word-form representations makes it difficult to exceed this threshold. While Lapponi et al. (2018) reported scores of up to 72, this was not achieved using only linguistic features on the original text; the best system also employed a set of meta-data features including, among others, the institutional role of the speaker and the county they represented.

It is apparent that there are certain speeches in the dataset that are easier to classify than others. Out of the 15250 samples in the test set, there were 1556 which none of the seven classifiers managed to correctly classify, whereas there were 4447 which all of them classified correctly; figure 8.1 shows the distribution.

Even though the networks all performed similarly when looking at the grand numbers, it is conceivable that the the various architectures have learned different ways of connecting their inputs to the various classes, seeing as they are all trained on different input representations.

8.2 Ensemble classifier

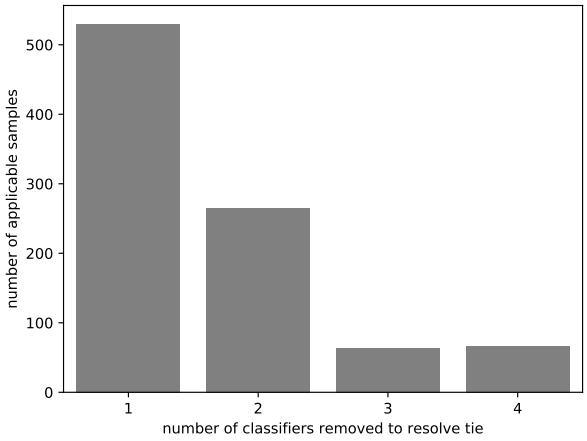


Figure 8.2: Graph showing how many documents in the test set (out of a total of 16945) lead to ties in the ensemble, and how many classifiers were removed to resolve the ties

To see how the various classifiers might complement each other, an experiment was run by combining the classifiers into a type of ensemble classifier. An ensemble classifier is a type of meta-classifier which combines the predictions from multiple classifiers to make a prediction. In our case, we created a majority[1] classifier. This entailed evaluating the development and test sets using all the classifiers and then predicting the target classes using the most commonly predicted class for each speech. Ties were resolved by removing models from consideration in ascending order of

1. technically a plurality classifier, since "majority" implies more than 50% agreement

F_1 score on the development set; i.e. if all the models disagreed, the class predicted by the FF-BOWAE model would be selected. In practice total disagreement never occurred, though there were in total 926 ties, 5.5% of the training set. Over half of the ties were resolved by removing the averaged embeddings model, but some documents caused more disagreement.

The ensemble classifier gave a significantly higher result than any of the individual models, with a score as high as 71.55 on the test set. This may suggest that the original text representations—simple lower-cased word forms—contain more signal than each of the the various representations which are derived from them. It is also likely the case that models belonging to the different architectures had learned to make different inferences.

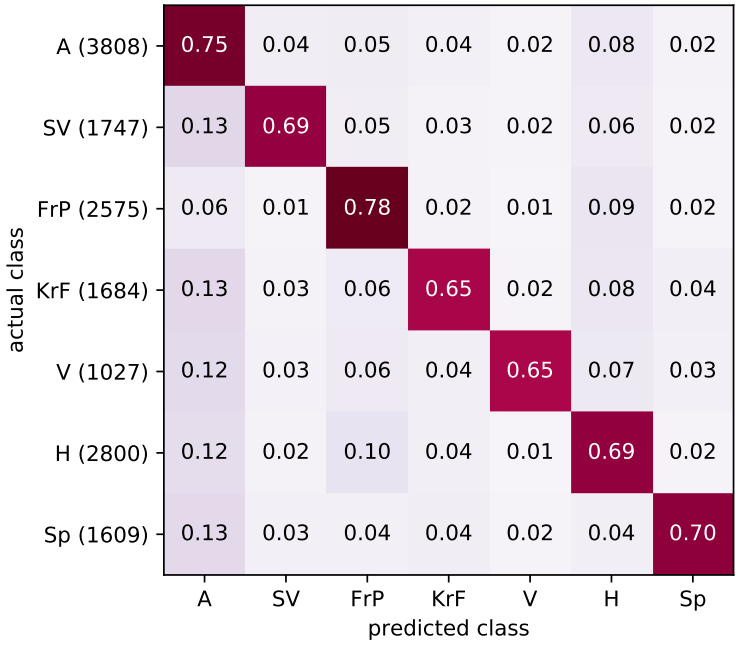


Figure 8.3: Confusion matrix of majority voting between the seven classifiers on the test set

Figure 8.3 shows the confusion matrix of our combined classifier. Overall this displays a very similar pattern to that we have seen for the individual classifiers, which is not surprising given that this represents a combination of those classifiers. That said, performance is increased for every single class, with recall for Progress Party (Frp) as high as 78.

Table 8.2 shows per-class F_1 scores for the individual classifiers and the ensemble classifier. We see that the FF-BOW model was best for most classes, but note that the class which the ensemble classifier performed best on, the Center Party (Sp), with a, F_1 score of 73.7, was also the only class for which the FF-BOW classifier was bested.

The ensemble method is a large improvement on the individual classifiers, but this method obviously still depends on the ability of the

system	SV	Ap	Sp	KrF	V	H	Frp	Macro
FF-BOW	**67.5**	**69.2**	68.8	**66.3**	**66.2**	**66.6**	**70.6**	**67.9**
FF-AE	46.2	52.8	57.0	53.0	48.7	51.3	56.8	52.3
FF-BOWAE	66.6	68.3	69.2	66.1	65.3	65.3	68.1	67.0
CNN-EL	66.1	68.1	**69.6**	63.7	65.1	**66.6**	69.9	67.0
CNN-PT	57.6	61.7	63.8	58.0	57.0	59.5	63.6	60.2
BiLSTM-Max	64.9	66.1	68.6	63.3	63.7	64.8	68.7	65.7
BiLSTM-Att	63.7	65.3	68.3	62.3	63.4	63.4	67.0	64.8
ensemble	71.6	72.3	73.7	70.3	69.6	70.5	72.9	71.6

Table 8.2: Per-class F_1 scores on the test set for the various classifiers

individual classifiers to classify. Digging into the classifying decisions, there was one speech—out of all the speeches in the test set—which none of the models correctly classified, despite the seven models producing six different answers. This speech, arguably the most difficult speech in the test set by this metric, is speech no. 106386:

> Jeg takker også for dette svaret. Slik jeg kan se det, er det mulig å reise tvil om lovligheten av å legge ned det eneste lille rehabiliteringstilbudet vi har for befolkningen i Finnmark, en liten avdeling med kun seks senger. Jeg viser bl.a. til forskrift om habilitering og rehabilitering, hvor det står presisert: «Det regionale helseforetaket skal sørge for at personer med fast bopel eller oppholdssted i helseregionen tilbys og ytes nødvendig habilitering og rehabilitering i spesialisthelsetjenesten». Videre, i merknadene til § 12: «Habilitering og rehabilitering i spesialisthelsetjenesten utgjør de tjenester som det regionale helseforetaket plikter å sørge for å tilby befolkningen», og som krever spesialistkompetanse. Og videre, i merknadene til § 1: «Tilbudet skal ytes i eller nærmest mulig brukerens vante miljø så langt det er praktisk mulig og forsvarlig.» Jeg kunne vist til flere dokumenter og planer i denne forbindelse. Mitt siste spørsmål blir da: Vil statsråden i sitt oppdragsdokument for 2007 kreve tiltak i helseforetakene for å hindre nedlegging av rehabiliteringsplasser?

A human reader would also be likely to find it difficult to find any indicators as to the party of the speaker. Knowledge of the political situation in 2007 would help, since this speech takes the shape of a question to a government minister, indicating an opposition party. The speaker also appears to be concerned with the provision of medical services in the north of Norway, an area of sparse population. This speech raises the question of to what extent the parties are distinguishable when looking at the textual content of individual speeches. The F_1 score of 71.55 given by our ensemble is unlikely to be close to the limit, but the varying nature of

the speeches' content implies that at some point the texts cease to provide enough indicative signal. Incidentally, the speech was given by a member of the Liberal Party (V).

Chapter 9

Conclusion

This thesis has examined neural networks applied to the field of classification of political speeches in parliamentary settings. It has delivered an overview over previous work in the field using traditional methods. We have reviewed a wide variety of neural methods used in the field of natural language processing (NLP), and gone into detail in describing algorithms and formulae. All of this knowledge has been applied to a specific task, namely developing neural network architectures for classifying parliamentary speeches in the Talk of Norway (ToN) corpus with regards to the political party affiliation of the speaker. This corpus a is fairly large dataset, with 250,373 speeches spanning a time period between 1998 and 2016. It exhibits much variance in terms of document length and content, and there are large differences between the sizes of the classes, as defined by the party of the speaker.

For each of the many different neural network architectures which are discussed and applied to the task, we performed experiments using a wide range of hyperparameter configurations. This extensive experimentation has been enabled by the Abel high-performance computing cluster. The data resulting from these experiments have been evaluated and analysed in detail.

We have seen that a simple feed-forward neural network based on bags-of-words (BOWs) is relatively easy to train and determine hyperparameters for. This type of architecture has also shown itself to give strong results for the task, demonstrating that newer and more complicated architectures are not always better. Correspondingly, the advanced architectures, convolutional neural networks (CNNs) and recurrent neural networks (RNNs), were more difficult to determine good hyperparameters for and to train. The final results delivered by the RNN models in particular proved difficult to reproduce, with scores dropping significantly upon re-training despite using identical hyperparameter settings.

Most worthy of attention, we have shown that an ensemble classifier based on multiple diverging architectures is able to exceed all of the individual architectures.

The ToN corpus was introduced in chapter 2, which gave background information and discussed certain statistics while putting the corpus in

the context of the classification task. Section 2.1 included a summary of previous classification efforts on the corpus, using traditional machine learning methods such as support vector machines (SVMs). Chapter 3 discussed previous work that has been done in the classification of political speech field in the contexts of other legislatures, bringing attention to certain pitfalls to avoid and considerations to take when discussing the results. Chapter 4 reviewed the basics of neural networks, with some details on how they can be applied to NLP tasks. This chapter also contained overviews of the different architectures and methods that we went on to apply in practice, while section 4.4 included detailed mathematical descriptions of the various functions we employ.

Chapter 5 presented details on the experimental setup and work environment. Section 5.1 discussed attempts to replicate the splits from Lapponi (2019) for greater replicability and laid forth the reasons why this approach was abandoned in favor of the method detailed in section 5.2. The Abel computational environment was described in section 5.4, and the Keras neural network library in 5.3. Section 5.5 provided details on certain common hyperparameters which were fixed for the experiments.

Chapter 6 documented the most basic neural network architectures that we used for our task of classifying the speeches in our corpus by political party affiliation. These consisted of two variations of feed-forward networks, but with different input representations, namely BOWs and averaged word embeddings. Analyses were given for individual hyperparameter settings, such as vocabulary size for BOW representations and variations of the input representations for the models trained on averaged word embeddings.

Chapter 7 described experiments using more advanced architectures for our task, namely CNNs and RNNs. The hyperparameters which were analysed for the former in section 7.1 included window sizes and numbers of filters. An attempt to quantify the effects of non-determinism was reported in section 7.1.3. For the long short-term memory (LSTM) networks described in section 7.2, we discussed the effects of recurrent layer output dimensionality and various ways of recombining sequences into a fixed-size output vector, including max pooling and self-attention. The self-attention mechanism was analysed in particular detail, with examples from the corpus.

In chapter 8 we discussed the results of the trained classifier models on the held-out test set. We also saw how combining the classifiers in an ensemble configuration lead to a classifier that was much stronger than its individual constituents. Using only text, our ensemble classifier delivered an F_1 score of 71.55, matching the best-performing classifier from Lapponi (2019), which used both automatically inferred linguistic features and contextual meta-data such as the county of the speaker. The strengths of our individual classification models aside, the strength of ensemble methods was thus demonstrated.

9.1 Future work

As discussed in section 5.6, the method used in the present work to separate into training, test and development sets does not account for the effects of the dependent variables person, time and issue, an issue which was raised by Yu, Kaufmann, and Diermeier (2008). Briefly summarized, if our classifier is to recognize ideology, it should be able to classify speeches made by other speakers, speaking at different time periods, and about different issues, than that which is present in the training data. Were one to update the ToN corpus to incorporate newer data, the best-performing classifier in the present text could be tested on newer data, thereby controlling for the time aspect, and to an extent the person and issue aspects.

This thesis has explored many different neural network architectures, but we have mostly used these in basic forms. We will now suggest some "deeper" network architectures that one could employ.

The literature on RNNs which has been explored for this thesis has shown them to work well on the sentence level. However, the sequences we are dealing with in this dataset are much longer, consisting of several sentences. Modeling sequences using fixed-size vectors inherently becomes more difficulty the longer the sequence is. Rather than running over the entire speech, one could instead design a network which split each document by sentence and created a fixed-length representation of each of the sentences. These representations could then be used in one of two ways. One method would be to classify each of the sentences separately, and then use the majority vote as the classification decision. Another approach would be to use the sentences as inputs for a new recurrent layer, treating each encoded sentence as a time step.

Another way we could have dealt with the long sequences in the corpus would have been to stack multiple convolutional layers on top of each other. This can be done by simply taking the output from one layer—where each dimension already covers a token n-gram—and using it as input to the next layer. In effect, this will expand the coverage of the convolution. If we for example have a window size of three in the first layer, and feed this into a second layer with a window size of four, the resulting convolution will cover 12 tokens. This may work well to cover long-range dependencies.

Other than simply tweaking the basic CNN and RNN models, we could also combine the two. For example, we could take the encoded n-grams which are output from a convolution layer, and feed these into a recurrent layer. The input at each time step that the CNN layer outputs would contain information about local dependencies, and the recurrent layer could enhance these locally informed time step representations with information from long-range dependencies.

The classification models in this thesis used word-level embeddings when training embeddings from scratch. The only classification models that were able to employ embedding models that leveraged character n-grams were those which used pre-trained, static, fastText embeddings. If a fastText embedding layer, including the character n-gram embeddings,

could instead be included as a trainable layer, we would be able to see both the benefits of generating embeddings from character *n*-grams (especially useful for Norwegian compound nouns) as well as training task-specific word embeddings in which the embeddings gain semantic values directly related to the classification task.

The above is far from an exhaustive list of all the methods we could imagine testing in order to improve classifier performance given our dataset. It is conspicuous, however, that the top models across different model architectures all had F_1 scores that were very close to each other. Trying to examine why this is the case would be interesting.

One may suspect that the texts only contain so much signal, but the higher performance of the ensemble classifier undermines this suspicion, since all of the constituent classifiers were based on the same text-based input. Apart from the simple plurality ensemble method employed here, there are also other methods that could be used, such as weighing the value of the predictions of each model based on the individual model's score on the development set, or using the means of their individual softmax outputs for classification. We could also simple use other ensemble methods. *Stacking*, for instance, lies somewhere between the majority ensemble and the combined architecture above. Rather than concatenating the individual dense outputs, stacking involves training a meta-model which uses the individual classifications as input features. It would also be interesting to see the results from using *bagging* on the best-performing FF-BOW model. This involves training a group of similar models but with random sampling, so that each of the models is trained on a different subset of the training set. Were this to surpass the performance of the ensemble of disparate models that we used here, this would serve to disprove the hypothesis that the increased performance of the ensemble method was due to the different architectures having different specialties.

It could also be a worthwhile endeavour to try to expand upon the ensemble method by integrating various architectures in a completely different manner. The FF-BOWAE model, which used two input representations with parallel feed-forward paths, concatenated before the output layer, was one attempt at integrating multiple models. Were one to do this, but using the outputs from all the different architectures, the resulting model could in theory be even stronger than the ensemble method. This would be highly dependent on the parameters between the concatenated output and the softmax layer and the network learning when to consider which parts.

The focus of this thesis has been on classification based on pure text, and all of the neural network architectures described in this thesis have used the same basic input data, i.e. lower-cased word forms. But as we discussed in chapter 2, the ToN corpus contains more linguistic information than this. It contains linguistic features such as part-of-speech (POS) tags as well as a morphological analysis for each token, all of which is automatically generated by the Oslo-Bergen Tagger (OBT). By augmenting the pure token-based representation with these linguistic data and creating a more detailed representation, a neural network may be better equipped to infer the ideological positions contained in the texts.

In addition to the linguistic meta-data, the best results reported in Lapponi (2019) were obtained when linguistic content such as that described above was joined by contextual meta-data such as the county of the speaker. It would be interesting to examine the extent to which adding this data would aid neural methods.

One could also consider adding to the linguistic data which is already contained within the corpus with other high-level inferred features. By applying topic modelling, we might be able to determine which topics a speech deals with. Sentiment analysis applied on top of this could then indicate what the speaker wishes to express regarding said topics, helping represent the ideological aspect of these speeches in a more narrow way. The things politicians say about the issues are, after all, all we (our classifiers) have to judge them by.

Acronyms

Adagrad adaptive gradient algorithm

Adam adaptive moment estimation

Ap Labor Party

BOW bag-of-words

CBOW continuous bag-of-words

CNN convolutional neural network

ELDR European Liberal Democrat and Reform Party

Frp Progress Party

GRU gated recurrent unit

H Conservative Party

KrF Christian Democrat Party

LSTM long short-term memory

MLP multi-layer perceptron

NLP natural language processing

NoWaC Norwegian Web as Corpus

OBT Oslo-Bergen Tagger

PCA principal component analysis

POS part-of-speech

ReLU rectified linear unit

RMSProp root mean square propagation

RNN recurrent neural network

SGD stochastic gradient descent

Sp Center Party

SV Socialist Left Party

SVM support vector machine

TanH hyperbolic tangent

TFIDF term frequency–inverse document frequency

ToN Talk of Norway

UEN Union for Europe of the Nations

V Liberal Party

Bibliography

Bojanowski, Piotr, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. "Enriching Word Vectors with Subword Information." *Transactions of the Association for Computational Linguistics* 5:135–146.

Cho, Kyunghyun, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation." eprint: arXiv:1406.1078.

Chollet, François, et al. 2015. *Keras.* https://keras.io.

Collobert, Ronan, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel P. Kuksa. 2011. "Natural Language Processing (almost) from Scratch." eprint: arXiv:1103.0398.

Conneau, Alexis, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. 2017. "Supervised Learning of Universal Sentence Representations from Natural Language Inference Data." In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen: Association for Computational Linguistics. eprint: arXiv:1705.02364.

Diermeier, Daniel, Jean-François Godbout, Bei Yu, and Stefan Kaufmann. 2012. "Language and ideology in Congress." *British Journal of Political Science* 42 (1): 31–55.

Fares, Murhaf, Andrey Kutuzov, Stephan Oepen, and Erik Velldal. 2017a. "Word vectors, reuse, and replicability: towards a community repository of large-text resources." In *Proceedings of the 21st Nordic Conference on Computational Linguistics*, 271–276. 131. Gothenburg.

———. 2017b. "Word vectors, reuse, and replicability: Towards a community repository of large-text resources." In *Proceedings of the 21st Nordic Conference on Computational Linguistics, NoDaLiDa, 22-24 May 2017, Gothenburg, Sweden*, 271–276. 131. Linköping University Electronic Press, Linköpings universitet.

Firth, J. R. 1957. "A synopsis of linguistic theory 1930-55.," (Oxford) 1952-59:1–32.

Gardner, Matt, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke S. Zettlemoyer. 2017. "AllenNLP: A Deep Semantic Natural Language Processing Platform." eprint: arXiv:1803.07640.

Goldberg, Yoav. 2016. "A Primer on Neural Network Models for Natural Language Processing." *Journal of Articial Intelligence Research* 57:345–420.

———. 2017. *Neural Network Methods for Natural Language Processing.* Vol. 10. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers. ISBN: 9781627052986.

Grave, Edouard, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. 2018. "Learning Word Vectors for 157 Languages." In *Proceedings of the International Conference on Language Resources and Evaluation 2018.* Miyazaki.

Hirst, Graeme, Yaroslav Riabinin, and Jory Graham. 2010. "Party status as a confound in the automatic classification of political speech by ideology." In *Proceeding of the 10th International Conference on Statistical Analysis of Textual Data,* 731–742. Rome.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. "Long Short-Term Memory." *Neural Computation* (Cambridge, MA) 9 (8): 1735–1780. ISSN: 0899-7667.

Høyland, Bjørn, Jean-François Godbout, Emanuele Lapponi, and Erik Velldal. 2014. "Predicting Party Affiliations from European Parliament Debates." In *Proceedings of the ACL 2014 Workshop on Language Technologies and Computational Social Science,* 56–60. Baltimore.

Jacovi, Alon, Oren Sar Shalom, and Yoav Goldberg. 2018. *Understanding Convolutional Neural Networks for Text Classification.* eprint: arXiv:1809.08037.

Johannessen, Janne Bondi, Kristin Hagen, André Lynum, and Anders Nøklestad. 2012. "OBT+stat. A combined rule-based and statistical tagger." In *Exploring Newspaper Language: Using the web to create and investigate a large corpus of modern Norwegian,* edited by Gisle Andersen, 51–65. John Benjamins Publishing Company.

Joulin, Armand, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2017. "Bag of Tricks for Efficient Text Classification." In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers.* Valencia: Association for Computational Linguistics. doi:10.18653/v1/e17-2068. http://dx.doi.org/10.18653/v1/e17-2068.

Kingma, Diederik P., and Jimmy Ba. 2014. "Adam: A Method for Stochastic Optimization." eprint: arXiv:1412.6980.

Kutuzov, Andrey, Mikhail Kopotev, Tatyana Sviridenko, and Lyubov Ivanova. 2016. "Clustering Comparable Corpora of Russian and Ukrainian Academic Texts: Word Embeddings and Semantic Fingerprints." eprint: arXiv:1604.05372.

Lapponi, Emanuele. 2019. "A Bridge Too Far: From Portal Design to NLP-powered Political Science." Unpublished thesis. PhD diss., University of Oslo.

Lapponi, Emanuele, and Martin G. Søyland. 2016. "Talk of Norway." Accessed October 29, 2016. https://github.com/ltgoslo/talk-of-norway.

Lapponi, Emanuele, Martin G. Søyland, Erik Velldal, and Stephan Oepen. 2018. "The Talk of Norway: a richly annotated corpus of the Norwegian parliament, 1998-2016." *Language Resources and Evaluation:* 1–21.

Lin, Zhouhan, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. 2017. "A Structured Self-attentive Sentence Embedding." eprint: arXiv:1703.03130.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2015. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. https://www.tensorflow.org/.

McDowall, Fergus. 2016. *stopword.* https://github.com/fergiemcdowall/stopword.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. "Efficient Estimation of Word Representations in Vector Space." eprint: arXiv:1301.3781.

———. 2013b. "Efficient Estimation of Word Representations in Vector Space." eprint: arXiv:1301.3781.

Mikolov, Tomas, Wen-tau Yih, and Geoffrey Zweig. 2013. "Linguistic regularities in continuous space word representations." In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 746–751. Atlanta.

Paszke, Adam, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. "Automatic differentiation in PyTorch." In *NIPS-W*.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al. 2011. "Scikit-learn: Machine Learning in Python." *Journal of Machine Learning Research* 12:2825–2830.

Peterson, Andrew, and Arthur Spirling. 2018. "Classification Accuracy as a Substantive Quantity of Interest: Measuring Polarization in Westminster Systems." *Political Analysis* 26 (1): 120–128.

Al-Rfou, Rami, Bryan Perozzi, and Steven Skiena. 2013. "Polyglot: Distributed Word Representations for Multilingual NLP." In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, 183–192. Sofia.

Ruder, Sebastian. 2016. "An overview of gradient descent optimization algorithms." *arXiv preprint arXiv:1609.04747.*

Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1986. "Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1." Chap. Learning Internal Representations by Error Propagation, edited by David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, 318–362. Cambridge, MA, USA: MIT Press. ISBN: 0-262-68053-X. http://dl.acm.org/citation.cfm?id=104279.104293.

Wittgenstein, Ludwig. 1997. *Philosophical Investigations / Philosophische Untersuchungen.* 2nd ed. Translated by Gertrude Elizabeth Margaret Anscombe. Blackwell. ISBN: 0-631-20569-1.

Yu, B., S. Kaufmann, and D. Diermeier. 2008. "Classifying Party Affiliation from Political Speech." *Journal of Information Technology and Politics* 5 (1): 33–48.