University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Data Science	Spring/ ~~Autumn~~ semester, 20 20 ...... Open / ~~Confidential~~
Author: Nils Magne Fossåen	
Programme coordinator: Patricia Retamal Supervisor(s): Vinay Setty	
Title of master's thesis: Semi-supervised learning for classification of Nordic news articles	
Credits: 30	
Keywords: semi-supervised learning, classification	Number of pages: 73 ………………… + supplemental material/other: 0 ………… Stavanger, 15. july / 2020 ……………….. date/year

University of Stavanger

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Semi-supervised learning for classification of Nordic news articles

Master's Thesis in Computer Science
by

Nils Magne Fossåen

Internal Supervisor

Vinay Setty

July 15, 2020

Abstract

Semi-supervised learning defines the techniques that fall in between supervised and unsupervised learning. It is commonly used in classification settings where one has a lesser amount of labeled data compared to unlabeled. The goal is to extract extra learning from the unlabeled data to improve on the supervised classification.

We will explore some of the approaches to semi-supervised learning to improve on the classification of Nordic news articles in the corpus provided. We will be exploring the methods of self-training in several different configurations and methods of feature extraction and engineering.

We will also provide some background and baseline using common supervised methods for improving results as well as different document representations like word-embedding so that we will be able to compare and put our semi-supervised results in relation to these methods.

We will see that while some of the methods explored did not succeed, others did and in relation to some of the supervised methods their performance is comparable. We will also see some promising approaches for countering the imbalance problem when considering confident pseudo-labels.

Table of contents

For my wife and kids, thank you!

1. Introduction

1.1. Motivation

In our dataset of 1 561 443 articles, only 37 323 are labeled. That is a miniscule 2.390%. That leaves 97.610% of the data and possible patterns contained there unavailable for our supervised learning algorithms to learn from. If it was possible to leverage only some percentage of that data it could give us additional learning and increased performance in our supervised models. This applies to most supervised learning scenarios where the labeled data compared to unlabeled is very sparse.

This is the task of semi-supervised learning techniques, some of which we will explore in this work. To be able to compare results we will also look at supervised methods of improving performance and create a baseline for comparison.

1.2. Problem definition

Text classification of Nordic newspapers following the CAP labeling scheme has proved difficult because of imbalance in the labeled data in the corpus available and few samples for the minority labels. Supervised methods of countering this imbalance has had mixed results. We will see if we can improve upon these supervised methods with additional learning from the unlabeled part of the corpus.

Q1: Are we able to create an algorithm that is able to extract additional learning using self-training methods, an inductive wrapper class of semi-supervised methodology.

Q2: Can we use unsupervised preprocessing techniques for feature extraction and engineering.

Q3: Will applying these techniques improve accuracy in classification

2. Background

Here we will write about the news corpus that has been the focus of our work. It was delivered by the Faculty of Social Sciences at UiS and has been a part of research done by one of their Ph.D. students. To make our contributions as comparable to their research we have also received their formatted and lemmatized version of the corpus. We have used this lemmatized version in all our experiments.

The corpus itself is a selection of news articles from two major newspaper outlets in Norway; Aftenposten and VG. Each contributing 60.5% and 39.5% to the total 1 561 443 articles in the corpus. Of these, 37 323 articles are labeled according to the CAP scheme in the respective 45 labels (see table).

2.1. Labeling scheme

Below is a table of the original CAP labeling scheme.

Code	Label
1|1	Macroeconomics
101|101	Employment and unemployment
102|102	Taxes, tax policy and reforms
103|103	Public spending, National Budget and debt
104|104	Industrial policy
2|2	Civil rights and liberties
3|3	Health

4|4	Agriculture, fisheries and food
401|401	Foods and food industry
5|5	Labour
501|501	Unemployment benefits and sickness benefits/pay
6|6	Education
7|7	Environment
8|8	Energy
9|9	Refugees and immigration
901|901	Refugees and asylumseekers
10|10	Transport
1001|1001	Public transport
12|12	Crime and justice
13|13	Social welfare and social affairs
1301|1301	Elderly care, retirement and other benefits for elderly
1302|1302	Children and families, rights and conditions of children
14|14	Housing and urban/rural development
1401|1401	Public funding for housing
15|15	Commerce, banking and consumer issues
16|16	Defense and security
1601|1601	Terrorism
17|17	Research, technology, IT and mass media
18|18	Foreign trade
19|19	Foreign affairs, development aid and international economy
1901|1901	Aid assistance policy to developing countries and assistance to other countries
1902|1902	International economy and finance
1903|1903	EU and EEA
20|20	Public sector and politics in general
2002|2002	Politics in general
2001|2001	Relationship between the central and local level, regional policy and local politics
21|21	Public land, spatial planning and resource management
23|23	Culture, art
91|91	Culture / art events and entertainment
24|24	Sports
92|92	Sporting events, athletes
25|25	Natural disasters, fires, preparedness
2501|2501	Other accidents
26/26	Religion and churches
93|93	Miscellaneous

Table 1 CAP labeling scheme

As we can see the list of labels is very exhaustive and it is of a hierarchical structure. There is a lot of overlapping subjects especially for subgroups of labels (3 and 4 digit labels). To simplify classification and make the labeling scheme less "noisy" we aggregate sublabels into parent labels. Then we get the updated labelling scheme we will be using further on in this thesis. This aggregation is done in accordance with the Faculty of Social sciences directions.

Code	Label
1	Macroeconomics
2	Civil rights and liberties
3	Health
4	Agriculture, fisheries and food
5	Labour
6	Education
7	Environment
8	Energy
9	Refugees and immigration
10	Transport
12	Crime and justice
13	Social welfare and social affairs
14	Housing and urban/rural development
15	Commerce, banking and consumer issues
16	Defense and security
17	Research, technology, IT and mass media
18	Foreign trade
19	Foreign affairs, development aid and international economy
20	Public sector and politics in general
21	Public land, spatial planning and resource management
23	Culture, art
91	Culture / art events and entertainment
24	Sports
92	Sporting events, athletes
25	Natural disasters, fires, preparedness
26	Religion and churches
93	Miscellaneous

Table 2 Aggregated labels

We should also note that the labels 91, 92 and 93 are to be considered as trash/noise and we are not ultimately interested in these labels, but since we are going to be working with the unlabeled part of the corpus we keep them as they have a considerable large portion of the labeled corpus and we can expect that the distribution is similar in the unlabeled corpus. Being able to discriminate them would be important.

2.2. Class distribution

The corpus is heavily imbalanced as we can see from figure below.

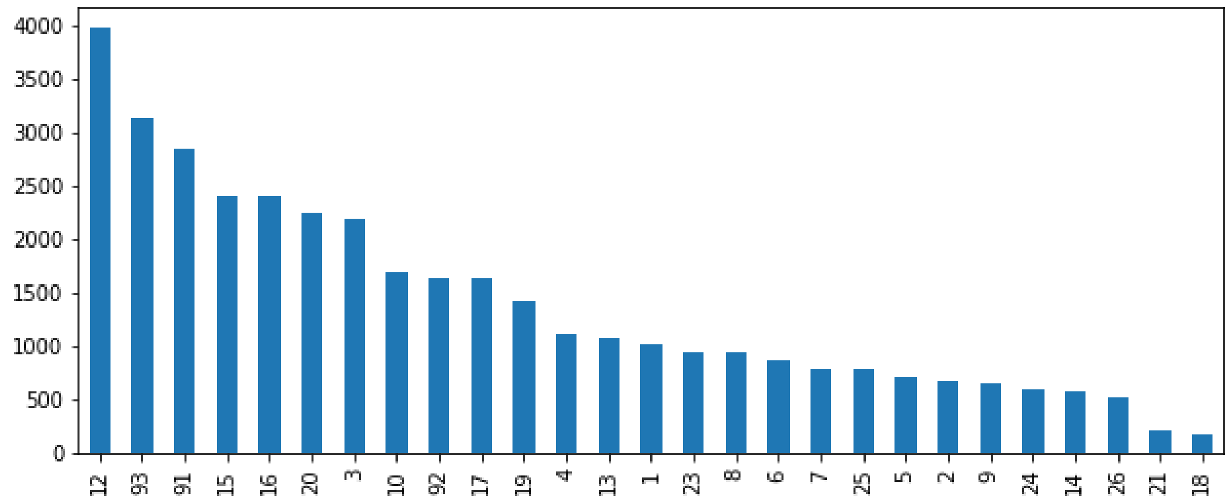

Figure 1 Label distribution

We can also observe that a large portion of the labeled articles fall in the "trash" category (91, 92, 93) and when we merge them we can see how they compare to the rest of the label distribution in the figure below.

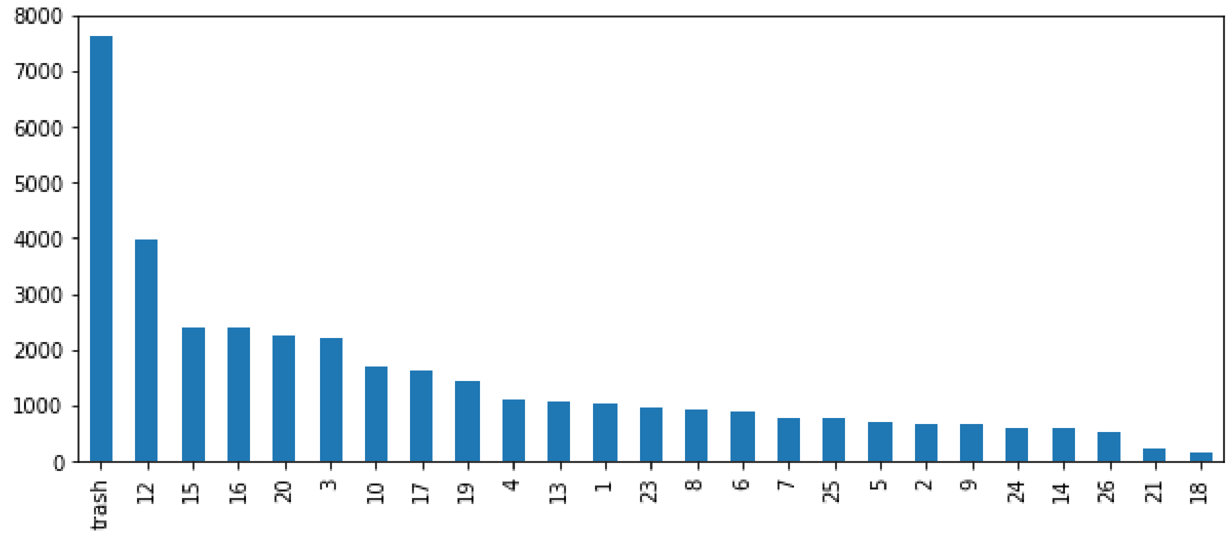

Figure 2 Trash distribution

Our main concern here are the labels 18 and 21 with very low numbers of samples, 167 samples from label 18 and 214 labeled 21. These numbers are low with respect to ability to learn enough patterns to discriminate, but they are also low compared to the other labels and would be prone to be ignored by a classifier due to such a low prior probability compared to the other labels.

2.3. Length metrics

We can count number of words per article or just take the character length of the entire article as measurements of length. We use the mean values and also standard deviation to see if there is any discriminatory effects. We use label -1 for the unlabeled data and we choose character length as metric.

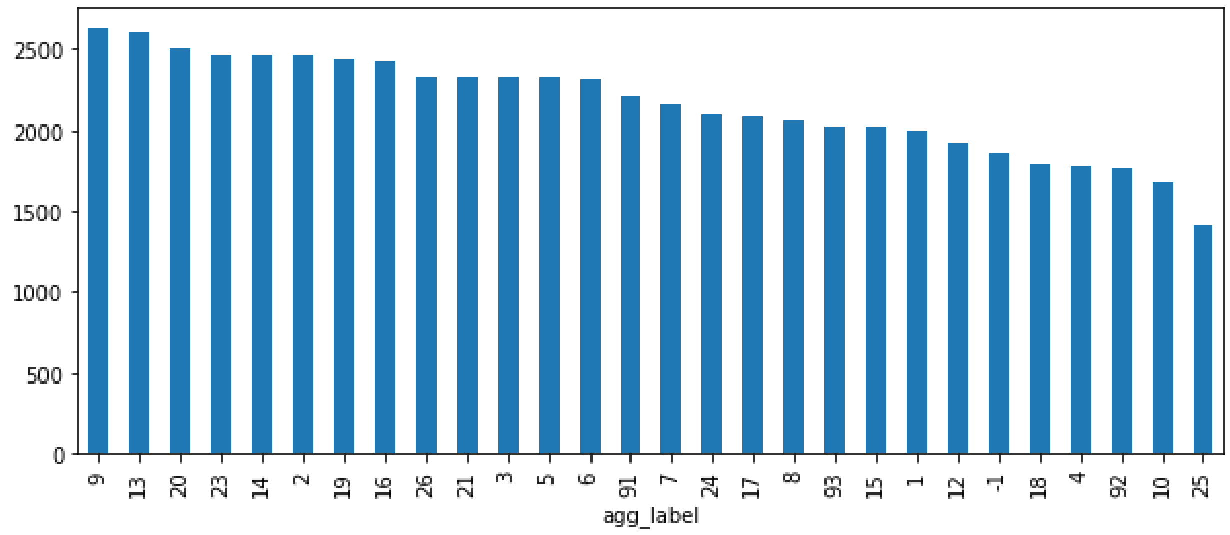

Figure 3 Mean character length

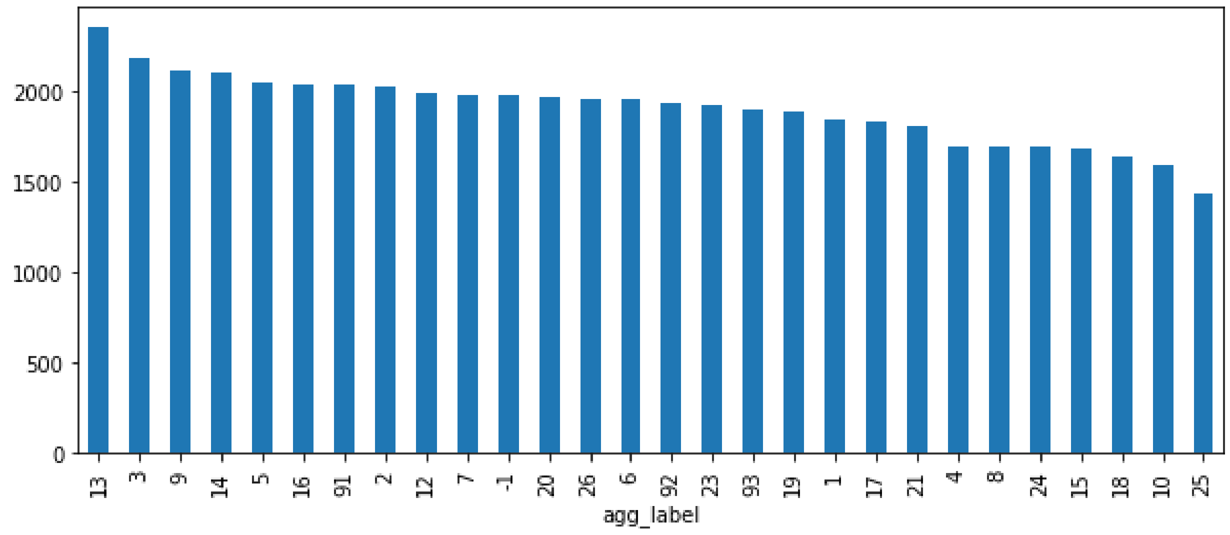

Figure 4 Standard deviation character length

As we can see there is minimal discriminatory effect from these metrics. We could perhaps see some difference in mean length, but when we have such a high standard deviation it becomes impossible to discriminate on this metric.

We can also compare between labeled and unlabeled to check the distribution.

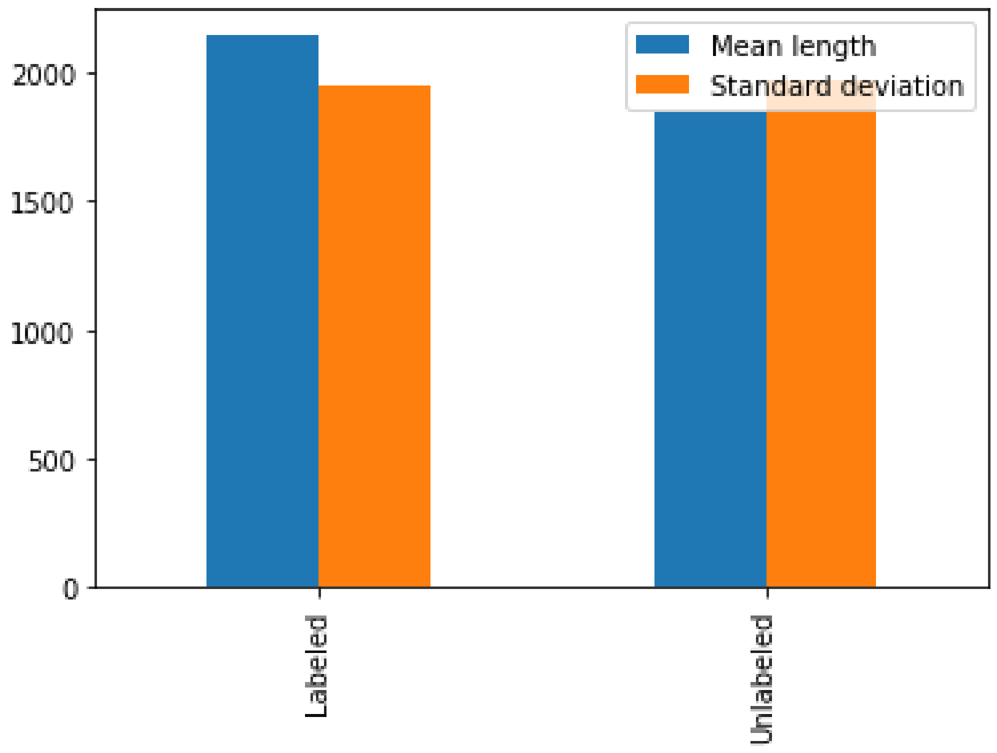

Figure 5 Mean length and standard deviation for labeled and unlabeled data

Here we can see a slight difference indicating that the distributions are not entirely equal. We can perhaps expect a higher presence of the lower length labels with a higher standard deviation.

2.4. Nynorsk/Bokmål distribution

Since Norwegian language have two distinct written languages it is important to check for the distribution of these in the corpus. There is no indicator for NN (Nynorsk) or NB (Bokmål)) in the dataset so we need to make a discriminator for this. An easy way to discriminate would be to take the 500 most common words for NB and translate this to NN. Then we can reduce these lists to words that are different. When we iterate over articles we can take the intersection between these two lists, and the list with the greatest intersection would imply the written language used.

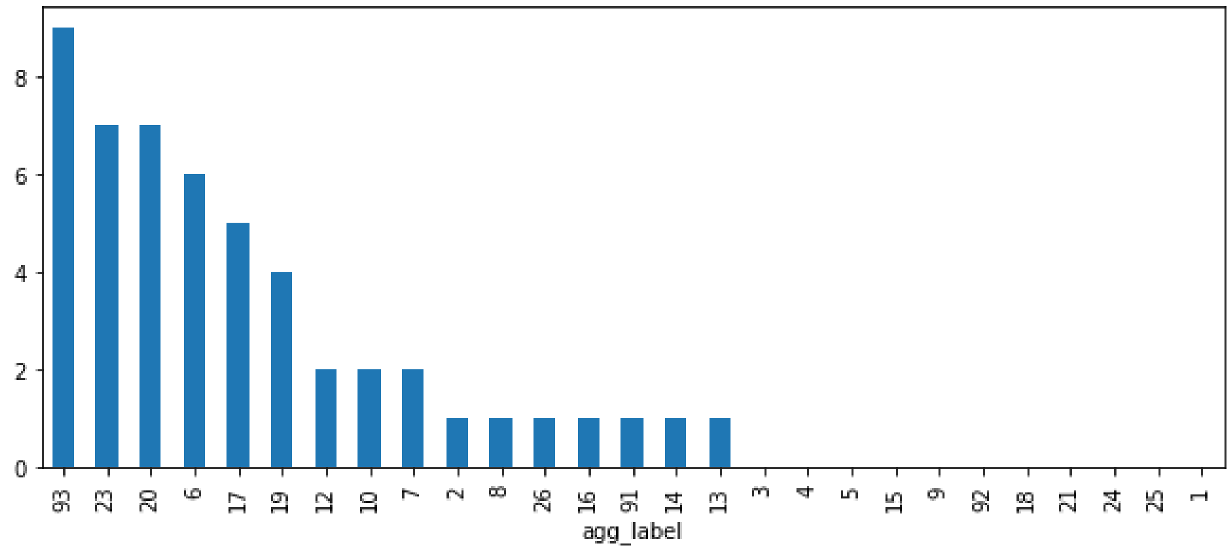

Figure 6 Nynorsk distribution

As we can see the NN presence is very low in the labeled data, and at a total of 1957 articles in the whole corpus. We are safe to remove them as they would only introduce noise to our feature space. All further experiments in this thesis is done with these NN articles removed.

2.5. Date distribution

Our corpus ranges in date from 2000-01-02 until 2017-09-16. During these 17 years there might be some development in language usage and perhaps also newspaper focus. New subjects with internet related terminology might grow more dominant in later times and so on.

Let us also check the corpus distribution per year to see if we have representative number of articles for each time-period.

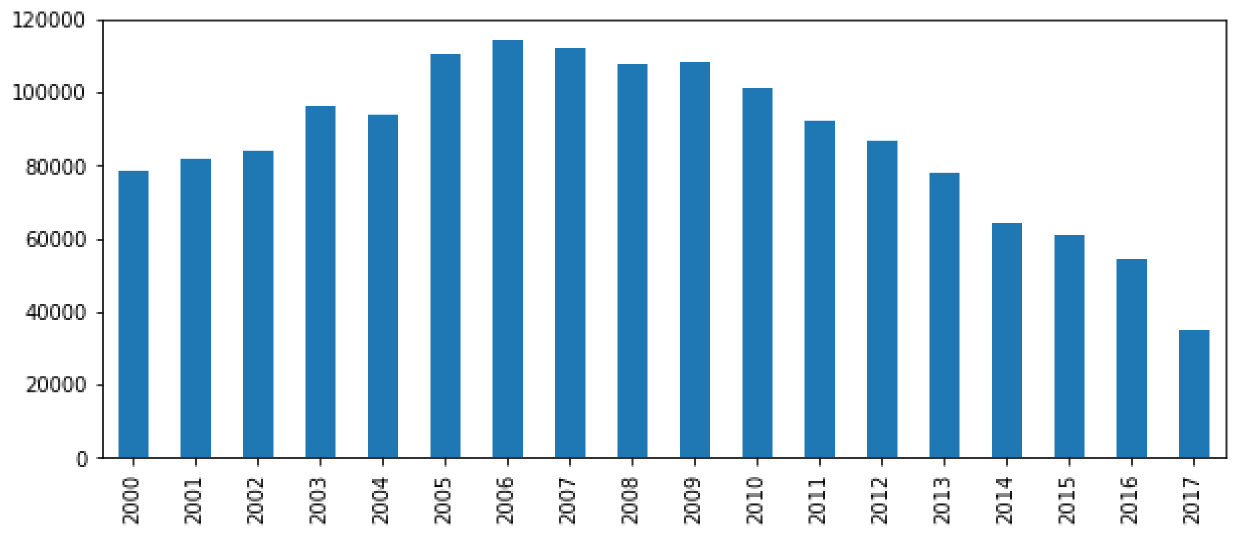

Figure 7 Entire corpus articles per year distribution

As we can see there is no time period that is not well represented, perhaps the later years have a little less then the average. Let us compare the corpus distribution with the labeled data to see if we have any differences there.

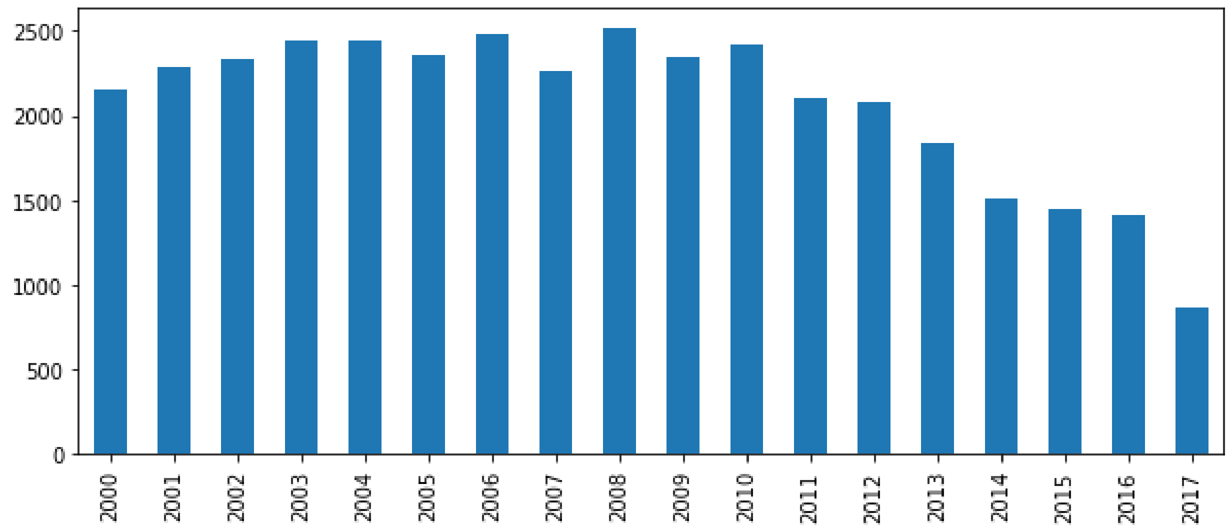

Figure 8 Labeled corpus per year distribution

Our labeled corpus follow the overall distribution well. We will keep this time distribution in mind later and check if there is any differences in perhaps binning the corpus by time and train separate models.

2.6. Vectorization

We have a corpus of news articles containing Norwegian language text. To be able to run any machine learning algorithm on these we need to reduce these articles to a numerical feature space that our machine learning algorithms can understand. A simple and intuitive way to solve this is to break down our articles into the words that it contains. Then we can represent our articles as a set of words with a corresponding count of the number of times the word appeared in the article. When we have done so with all the articles we can merge the sets of words into one great set of words spanning all the words in the entire corpus. This is now our vocabulary or feature space and each article is then represented in this feature space with the counts of words appearing in the articles as a vector we can feed to our machine learning algorithm. This method is called a bag-of-words method, BoW.

This method has some weaknesses though. It will favor the most frequent words in the corpus and longer documents as well. Luckily there is a method that counters these biases.

2.6.1. TF-IDF

TF-IDF or Term Frequency – Inverse Document Frequency, counters these problems by converting the counts to internal document frequencies, countering the bias towards long documents. The IDF counters the bias towards most frequently used words by punishing words that occur frequently in the corpus. See formula below.

$$\text{TF-IDF} = \text{TF}(t, d) \times \text{IDF}(t)$$

$$\text{TF} = \frac{\text{num. of times term } t \text{ occur in doc. } d}{\text{num. of terms in doc. } d}$$

$$\text{IDF} = \log \frac{1 + \text{total num. of doc.}}{1 + \text{num. of doc. term } t \text{ appears in}} + 1$$

Figure 9 TF-IDF formula

A popular library to use for TF-IDF vectorization is scikit-learns[1] TfidfVectorizer [2]. It comes with a lot of customizable hyperparameters, but the most important ones that we will focus on is ‘lowercase’ and ‘max_df’.

When we apply our vectorizer to the corpus we will tokenize the articles by splitting them on the space character. The way this tokenization is done can also be customized and one very common thing to do at this stage is to lowercase all words. This is to reduce noise in the feature space, but since we have a pre-formated lemmatized corpus where such noise is already taken care of we do not want to lowercase words because capitalized words are here recognized as entities and should be vectorized as independent terms. So we set ‘lowercase’ to false.

The next hyperparameter ‘max_df’ is a feature selection parameter with similar function as removing stop words. It will remove terms that occur at a maximum frequency of documents. This is also to reduce noise as very frequent words bring little discriminatory effect. We set our ‘max_df’ to 0.25, meaning we remove terms that occur in ¼ of the documents.

More about how we come to these hyperparameter-settings in next chapter, Baseline, where we tune our model.

With help from Singular Value Decomposition (SVD) we can plot our TF-IDF vectors and see if there is any separation between classes.

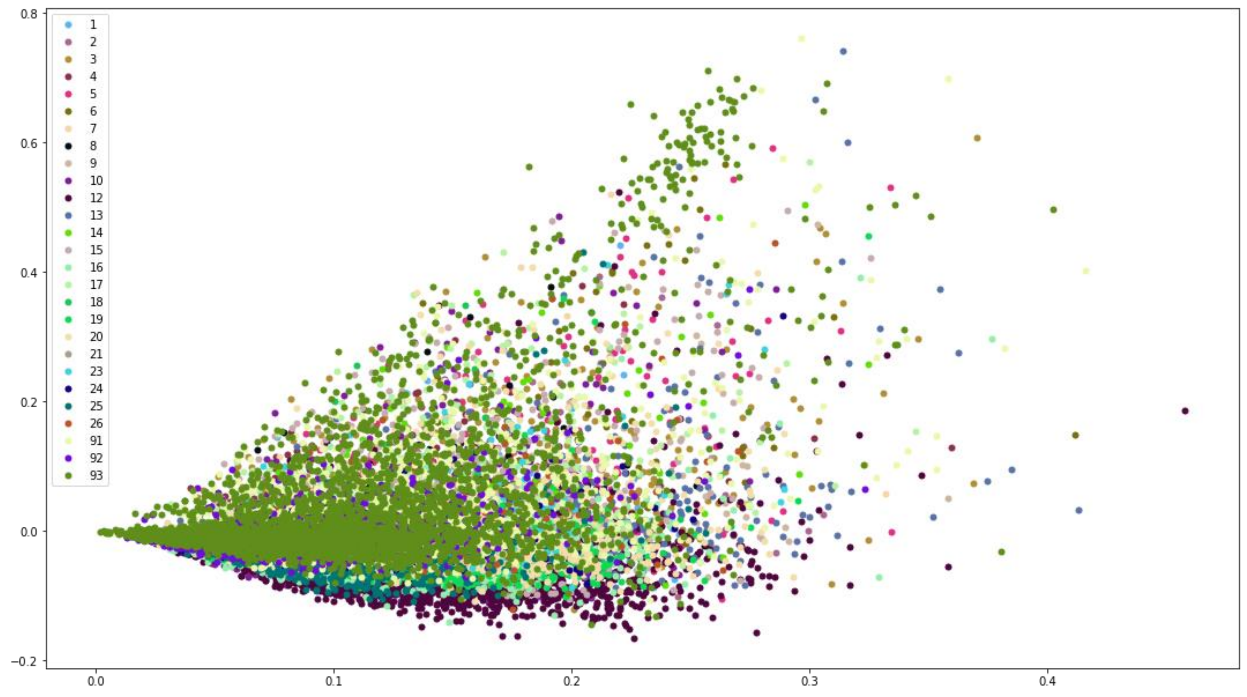

Figure 10 SVD plot of TF-IDF vectors of labeled data

There is little separation between classes in this plot. Let us see if there is a difference in distribution for the unlabeled data.

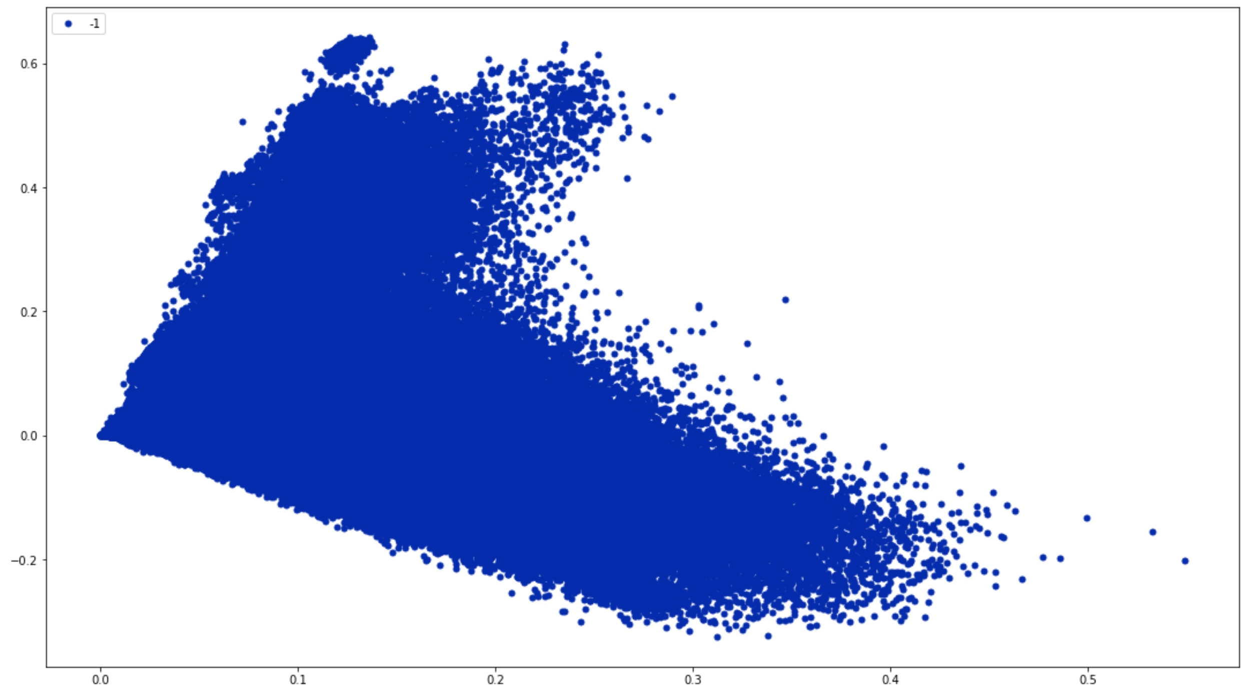

Figure 11 SVD plot of TF-IDF vectors of unlabeled data

Difficult to say much about the distribution of the unlabeled data, but it seems to be similar to the labeled distribution.

2.6.2. Word embedding

As TF-IDF gives discrete representations in vector space it does not capture any relationships between words. Word embeddings capture these relationships with help of neural nets looking at the context the words appear in and project these words into a defined dimensional space.

As word embedding models are resource heavy to train it is very useful to make use of transfer learning and use already trained models from different datasets. We can then optionally do fine tuning to better adapt to our specific dataset.

The Language Technology Group at University of Oslo [3] has a repository [4] of several embedding models trained on multiple datasets available for download. We will look at model 80 and 81 which are both fastText [5] skipgram models trained on Norsk Aviskorpus, NoWaC and NBDigital corpuses. These corpuses are highly related to our context so we will not do any fine tuning of these models.

Model 80 is a lemmatized model with vector size 100 and a vocabulary size of 3 998 140 words.

Model 81 is not lemmatized with vector size 100 and a vocabulary size of 4 428 648 words.

To convert our corpus with the embedding models we need to follow a few steps. First we need to identify the vocabulary to embed. This can be done using scikit-learns CountVectorizer [6]. We then iterate over the vocabulary and query the model for the embedding vector for that word. When done we will have a matrix representing our vocabulary.

Then we need to convert our document representations to fit into the embedded space. This can most easily be done by converting our document vector holding counts of words for each document into a subset of the embedded matrix vocabulary. Then we can take the mean values over each dimension and we have a single vector representation of a document. This approach has the same weaknesses as the simple approach to vectorization explained earlier. Word frequencies both internal to document and over corpus are not taken into consideration and therefore our mean vectors will be very noisy as every word "pulls" equally in their direction.

To further sophisticate the embedding document representations we can make use of the TF-IDF term frequencies we already have made to represent our documents. Now instead of just taking the mean we take the TF-IDF weighted sum of the embedded subset matrix. To do this we simply take the dot product of the TF-IDF vector with the embedded matrix and if the TF-IDF weights are not regularized (sum to 1) we divide that with the sum of the TF-IDF vector.

Then we get a better representation of the documents in embedded space. It will give us approx. 2% increase in accuracy over the naive mean representation.

Below are some plots of both lemmatized and not lemmatized embedding vectors.

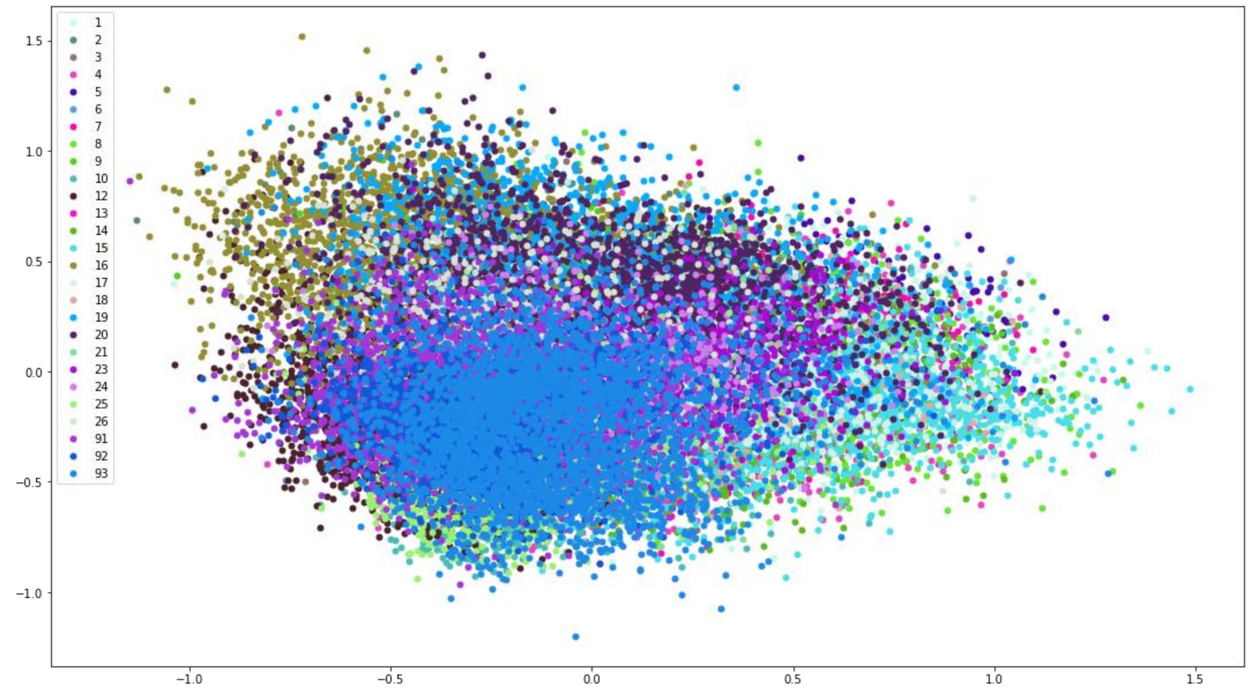

Figure 12 PCA plot of lemmatized fastText vectors of labeled data

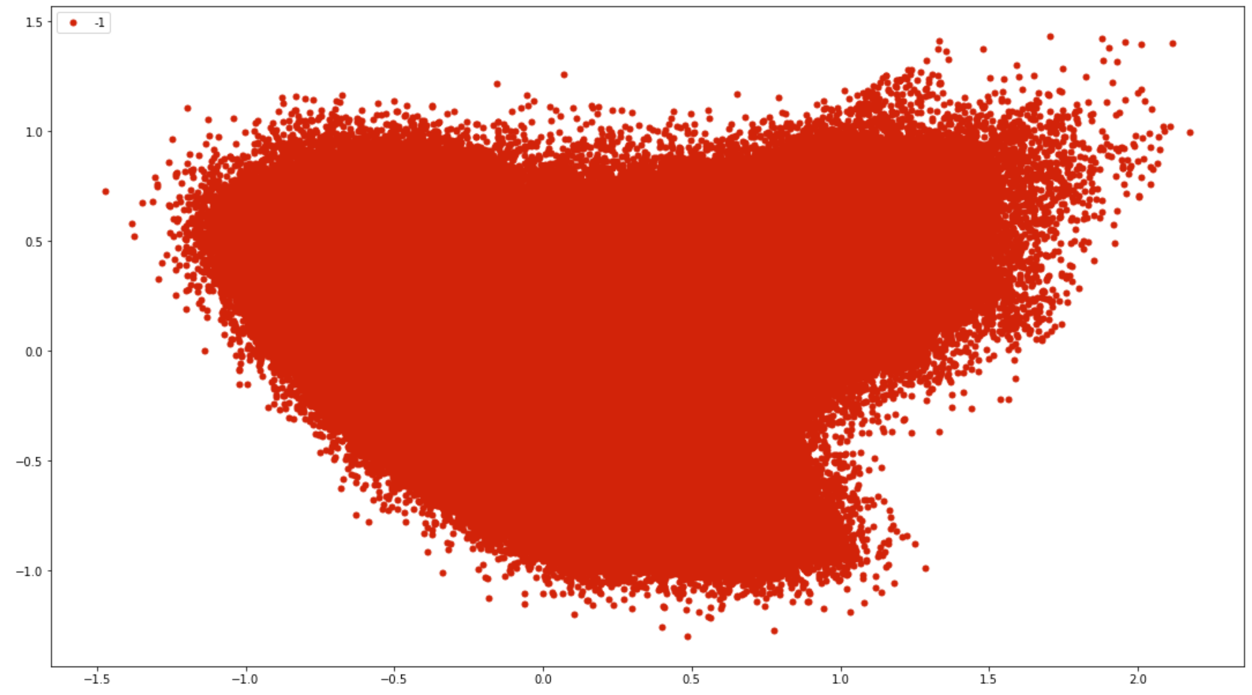

Figure 13 PCA plot of lemmatized fastText vectors of unlabeled data

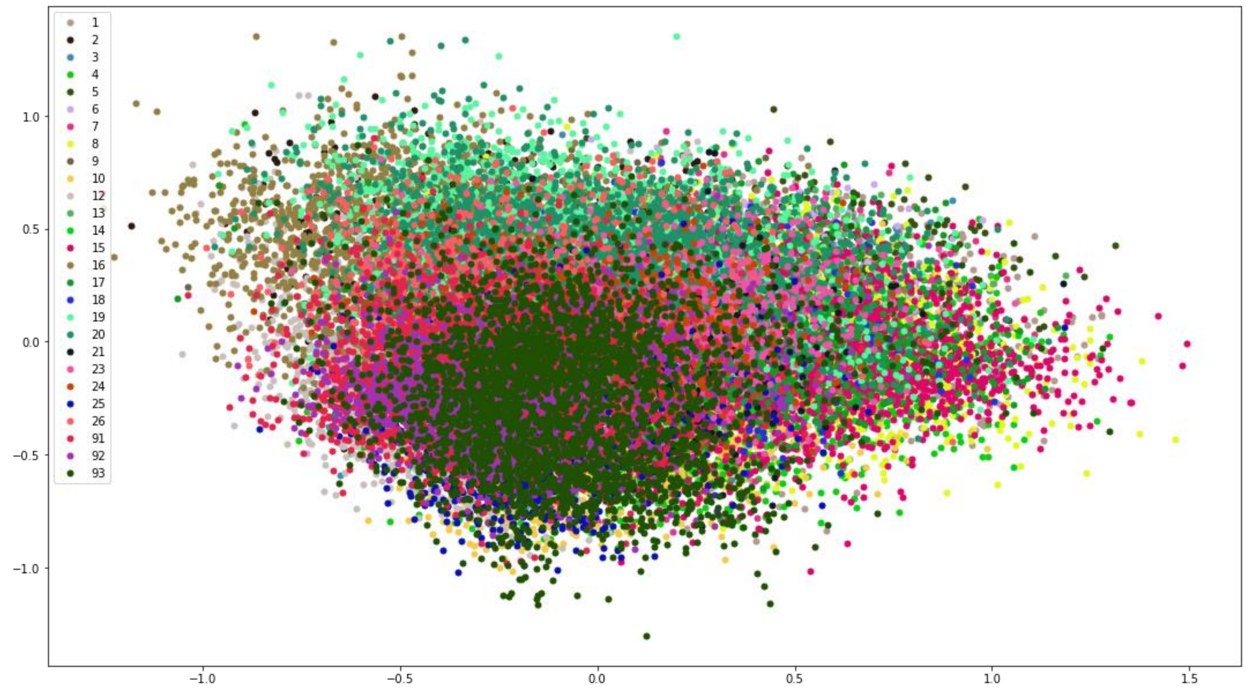

Figure 14 PCA plot of non-lemmatized fastText vectors of labeled data

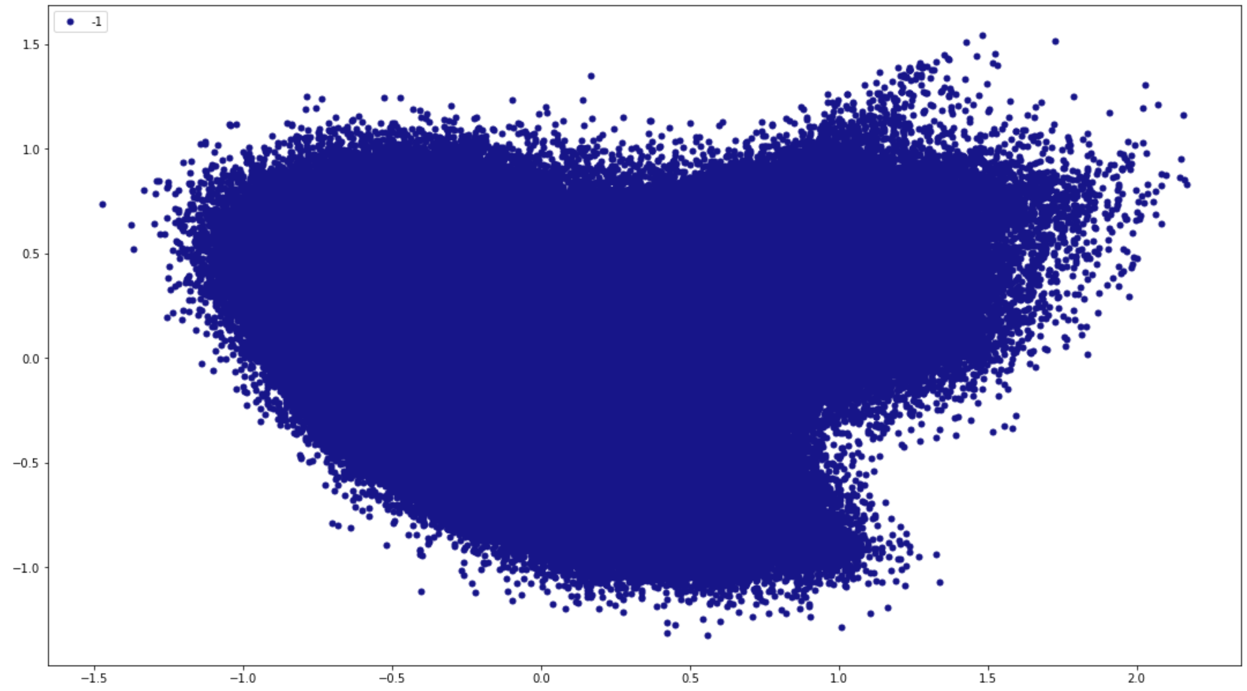

Figure 15 PCA plot of non-lemmatized fastText vectors of unlabeled data

3. Baseline

Before we apply our supervised and semi-supervised algorithms it is good to establish a baseline for us to refer to. Our main task is text classification so we will need a model for classification. Here we have a few choices to make. Classical text classification models are Naïve Bayes (NB) and Support Vector Machines (SVM). These are fast and reliable and have proved effective for such tasks. In later years it has also become popular with huge deep neural networks like BERT that are very heavy and expensive to train. We will not look into such heavy models here because as they do deliver better accuracy, so will any newer and better classifiers in the near future (and the bar moves fast). Our goal is not to find the best classifier, but to device a model for exploiting unlabeled data. A classifier is part of such a model, so replacing it will certainly improve results. We are more interested in the increase in accuracy we can achieve regardless of what type of classifier is used by leveraging the unlabeled data.

3.1. Train, validation and test splits

To be able to compare and measure results we will need to split our labeled data into 3 disjoint sets. One for training, this will be the data available for our machine learning algorithm to learn patterns from. One for validation, this set is used during training to evaluate the performance of the model and to perform hyper-parameter tuning based on results on this validation set. Lastly we have a test set that we do the final performance evaluation of the model on. This way we will not be overly optimistic by tuning our model to fit our validation set, but get a better understanding of our models general performance on unseen data.

For our experiments we will use a 60% train, 20% validation and 20% test split of our labeled data. We use scikit-learns train_test_split [35] library to do a stratified split, meaning we try to balance the label distribution in the splits. This is important since we have a lot of labels and also since our dataset is imbalanced we want to keep the prior probabilities similar in our splits. We do an initial split with random_state=1 and a test_size=0.4, this gives us the 60% training data. Then we further split the 0.4 test_size data in half to get our 20% validation and test sets, also with random_state=1.

We will use these splits throughout the experiments so that results are comparable.

3.2. TF-IDF vectorization

For our experiments we will use scikit-learns TfidfVectorizer referenced earlier. It comes with a set of hyper-parameters we can set, for further explanation on what the parameters do we reference to scikit-learns documentation. We will focus on the following parameters:

Hyper-parameter	Options
‘strip_accents’	[‘ascii’, ‘unicode’, ‘None’]
‘max_df’	Float, in range [0.0, 1.0]
‘lowercase’	[‘True’, ‘False’]
‘max_features’	Int, number of features
‘stop_words’	List, list of stopwords

Table 3 scikit-learn TfidfVectorizer hyper-parameters

In order to evaluate the vectorization we need a classifier and for our initial evaluation we just run the TfidfVectorizer with default hyper-parameters together with 5 different classifiers also running with default hyper-parameteres on validation data split. The classifiers are as follow, and subsequent accuracy scores:

Model	Accuracy score
LinearSVC	**0.6754**
SGDClassifier	0.6671
LogisticRegression	0.6357
ComplementNB	0.6224
MultinomialNB	0.3182

Table 4 Initial model default values accuracy scores

As we can see both the SVM based classifiers come out ahead of both logistic regression and NB based classifiers. These results are only preliminary results though and we will go more thorough into exploring hyper-parameters for each of these classifiers later. But for our initial tuning of TfidfVectorizer we choose LinearSVC as our classifier to tune with.

The hyper-parameter values shown in the table below will be run in a grid-search with LinearSVC as classifier using only default parameters on the validation data split.

Hyper-parameter	Values
'strip_accents'	['ascii', 'unicode', None]
'max_df'	[0.05, 0.15, 0.25, 0.35]
'lowercase'	[True, False]
'max_features'	[10000, 20000, 30000, 40000, None]

Table 5 TfidfVectorizer initial grid-search hyper-parameter values

We skip running stopwords in the grid-search as this parameter is dependent on the 'strip_accents' hyper-parameter to properly function. Instead we get the best performing hyper-parameters of this initial grid-search and see if they perform better with or without stopwords afterwards.

The best performing hyper-parameters of this initial grid-search:

Hyper-parameter	Value
'strip_accents'	None
'max_df'	0.25
'lowercase'	False
'max_features'	None

Table 6 TfidfVectorizer initial grid-search best values

Returned with an accuracy of: **0.6775**

3.2.1. Stopwords

To reduce noise in a corpus it can sometimes be beneficial to remove what is known as stopwords. These are words that are so common in language that they carry little discriminatory effect to any text. They are mostly function words as "the", "is", and "at". Removing them will shrink the size of the corpus and of course the vocabulary and can therefore make modelling more efficient. But removing such words can also be detrimental to catching phrases or entities where these words are used.

From our hyper-parameters we already have used 'max_df' which could be thought of as a corpus-specific stopwords list and we saw that a value of 0.25 returned the best score. This means that the words appearing in >=25% of the documents will be considered stopwords and removed.

To further investigate if we can get benefits of removing stopwords other then the corpus-specific ones we need to get a list of stopwords not based entirely on our corpus. Such a list is not easily obtained for Norwegian language as most NLP frameworks operate with English vocabulary. But there is a resource online that provide a list of the 500 most frequent words in Norwegian language based on both newspaper articles and literature. This list is compiled on behalf of University of Bergen and can be found at this location [7].

We also need to process the list of stopwords with the same lemmatization as is applied to the corpus for it to match correctly. We will also see what results we get by using different lengths of the list, i.e. all 500, top 250, top 100, top 50 etc.

The results applied with the previously mentioned best hyper-parameters for TfidfVectorizer and LinearSVC default settings on the validation set below:

Number of stopwords	Accuracy score
20	**0.6775**
50	0.6763
100	0.6767
150	0.6763
250	0.6724
500	0.6687

Table 7 Stopwords accuracy scores

As we can see there is no extra benefit from using an external stopwords list and we will not apply it further in our experiments.

3.3. Classifier

As mentioned earlier we have focused on the common classical classifiers for Natural Language Processing (NLP) for this thesis. We will also use implementations provided by scikit-learns package of machine learning libraries. Scikit-learn is a reputable and reliable source of such implementations and also provide an extensive library of other functionality we can benefit from in our experiments. Sticking with one provider also simplifies compatibility issues between different modules.

The classifiers we will look at are the following:

3.3.1. LinearSVC

This classifier [8] is a SVM classifier based on the C-library liblinear [9]. It uses a linear kernel and is implemented to support a large number of samples. This is important to us as our entire corpus is more then 1.5 million samples. It also support sparse representations which is also important as our TfidfVectorization will produce a sparse matrix. Multiclass classification , which is our case, is handled by one-vs-rest.

We will look at the following hyper-parameters and perform a grid-search over the value ranges in the table below:

Hyper-parameter	Values
‘loss’	[‘hinge’, ‘squared_hinge’]
‘tol’	[0.01, 0.005, 0.001, 0.0005]
‘C’	[0.5, 1, 1.5, 2]

Table 8 LinearSVC initial hyper-parameter grid-search values

The best performing hyper-parameter values were:

Hyper-parameter	Value
‘loss’	‘squared_hinge’
‘tol’	0.01
‘C’	0.5

Table 9 LinearSVC initial best hyper-parameter values

With an accuracy score of: **0.6798**

As we can see two of our best performing hyper-parameters are edge values so we do further investigation and run a second grid-search with following parameters:

Hyper-parameter	Values
‘loss’	[‘squared_hinge’]
‘tol’	[0.008, 0.01, 0.015, 0.02]
‘C’	[0.1, 0.3, 0.5, 0.8]

Table 10 LinearSVC second hyper-parameter grid-search values

The best performing hyper-parameter values were:

Hyper-parameter	Value
‘loss’	‘squared_hinge’
‘tol’	0.01
‘C’	0.5

Table 11 LinearSVC best hyper-parameter values

With an accuracy score of: **0.6796**

Here we get a confusing result as we have the same hyper-parameter values, but different accuracy scores. This is a result of the underlying liblinear library random state that we are unable to do anything with, so scores will have minor fluctuations (+- 0.0005). But we did get non-edge values so we settle for these hyper-parameters as our best.

3.3.2. SGDClassifier

This is a Stochastic Gradient Decent (SGD) wrapper [10] for linear classifiers that enables SGD training. Depending on what loss function defined by hyper-parameter it can work as a SVM, Logistic Regression (LR) or Perceptron classifier. It works with large data and can handle sparse representations.

We will focus on the following hyper-parameters and perform grid-search over the value ranges in the table below:

Hyper-parameter	Values
'loss'	['hinge', 'log', 'squared_hinge', 'perceptron']
'alpha'	[0.001, 0.0001, 0.00001, 0.000001]
'random_state'	[1]

Table 12 SGDClassifier hyper-parameter grid-search values

The best performing hyper-parameters were:

Hyper-parameter	Value
'loss'	'log'
'alpha'	0.00001

Table 13 SGDClassifier best hyper-parameter values

With an accuracy score of: **0.6711**

The LR loss function performed best and also worth noting was it got a precision of 0.7125 which is 2-4% above what previous models achieve.

3.3.3. LogisticRegression

A probabilistic model for classification [36] also based on the liblinear library. It uses a one-vs-rest implementation of multiclass problems. It handles sparse data and scales well and it has regularization.

We will focus on the following hyper-parameters:

Hyper-parameter	Values
'tol'	[0.001, 0.0001, 0.00001, 0.000001]
'C'	[6, 8, 10, 12]
'solver'	['newton-cg', 'lbfgs', 'liblinear', 'saga']
'random_state'	[1]

Table 14 LogisticRegression hyper-parameter grid-search values

The best performing hyper-parameters were:

Hyper-parameter	Value
'tol'	0.0001
'C'	8
'solver'	'saga'

Table 15 LogisticRegression best hyper-parameter values

With an accuracy score of: **0.6728**

The solver 'saga' is a stochastic average gradient decent solver and is optimized for large datasets with high dimensionality so it should fit our purpose well.

3.3.4. ComplementNB

This Naïve Bayes implementation [11] is optimized for imbalanced datasets and could therefore be interesting for us. It is an adaptation of Multinomial Naïve Bayes (MNB) where weights are based on statistics of the complement of each class. It should be more stable and is also considered to be better at text classification then MNB.

Naïve Bayes classifiers also have an advantage that they have only one hyper-parameter to tune, the alpha smoothing parameter:

Hyper-parameter	Values
'alpha'	[0.1, 0.2, 0.5, 1]

Table 16 ComplementNB hyper-parameter grid-search values

The best performing alpha smoothing parameter was:

Hyper-parameter	Value
'alpha'	0.2

Table 17 ComplementNB best hyper-parameter value

With an accuracy score of: **0.6573**

We see that we get a lesser score then the prior classifiers, but an advantage to the NB classifiers is that they are very fast and probabilistic, something that is advantageous for our application where we have a very large dataset and we will also need to deal with probability for judging confidence in predictions.

3.3.5. MultinomialNB

One of the most classical NB variants [12] for text-classification. We will see how it performs compared to its complement counterpart that is more optimized for imbalanced data.

Again, we focus only on the alpha smoothing parameter for our grid-search:

Hyper-parameter	Values
'alpha'	[0.01, 0.0015, 0.001, 0.0005, 0.0001]

Table 18 MultinomialNB hyper-parameter grid-search values

The best performing alpha smoothing parameter was:

Hyper-parameter	Value
'alpha'	0.0015

Table 19 MultinomialNB best hyper-parameter value

With an accuracy score of: **0.6463**

As we can see there was a slight underperformance compared to the complement version of NB, though MNB is the fastest of all the classifiers we have tested it is also the bottom performer.

3.4. Conclusion

Now we have tuned and compared some of the most classical classifiers for text-classification and we have the current ranking from tuning on the validation set:

Classifier	Accuracy score
LinearSVC	**0.6797**
LogisticRegression	0.6728
SGDClassifier	0.6711
ComplementNB	0.6573
MultinomialNB	0.6463

Table 20 Classifiers best performance on validation set

As a final test we will run these tuned classifiers on the holdout test set to get a more realistic view of performance:

Classifier	Accuracy score
LinearSVC	**0.6759**
LogisticRegression	0.6672
SGDClassifier	0.6660
ComplementNB	0.6605
MultinomialNB	0.6456

Table 21 Classifiers performance on holdout test set

Not any surprises here on the test set, most classifiers have slightly poorer performance compared to validation set in accuracy and ComplementNB a slight increase, but nothing more then what to be expected. The ranking is still the same.

So in conclusion we can say that for our labeled data the best performing classifier is the SVM family of classifiers and LinearSVC being the top performer. We can then say that **0.6759** is our initial baseline score. Our supervised experiments will be judged on performance in comparison to this score.

We may use some of the lesser performing classifiers in the experiments due to for instance performance and speed issues or for probabilistic issues. As mentioned earlier, probabilistic classifiers as NB and LR give good probabilities that we can use for confidence scores. Though there are ways of computing approximations to probabilities for the SVM based classifiers. More on this later in the thesis.

4. Supervised methods for improvement

In order for us to be able to compare the effectiveness of our semi-supervised techniques we will also apply some supervised (or rather non-semi-supervised) measures for increasing accuracy. This way we can compare the results in relation to these measures.

We will adapt some techniques used in a precursor project to this thesis [37] with some adjustments, but due to a problem in file naming and ID's we were not able to re-use any of the results and conclusions from that project here.

4.1. Feature selection

Perhaps the simplest and most common practice for better performance. Feature selection decrease the complexity of our model and could lead to less variance. We already do a bit of feature selection in the vectorization stage of the process, mentioned in the chapter about vectorization. There we do feature selection based on frequency in features alone, but at this stage we will perform more sophisticated statistical measures by also looking at the labels when we perform selection. This way we will hopefully choose features that are most important for classification.

A caveat here is that sometimes, especially if we have few samples, feature selection like this can lead to picking features that are very rare, and therefore very discriminatory in our dataset, but not really generally discriminatory "in real life". For instance if an adjective not often used is used by coincidence only sometimes within a category and not anywhere else, it would become a very significant discriminatory feature for this category. But "in real life" this adjective might be used in any category and therefore it would lead to misclassifications.

Possible countermeasures to this could be done at the vectorization stage where we can filter out very rare words by setting a minimum document frequency threshold. We have not done this and we will not be doing any particular measures to prevent these caveats in this feature selection process as well, but it is mentioned so that it is something to be aware of.

For all our experiments we will be using the previous experiments best performing setup of TfidfVectorization and LinearSVC classification. We will be looking at percentage increase (or decrease) in accuracy compared to our current initial baseline.

4.1.1. SelectKBest

This [13] module from scikit-learns library enable us to perform feature selection based on a scoring function. We will look at chi-square, ANOVA F-value and mutual info scoring functions and perform a grid-search over them looking at k different thresholds for number of features.

Hyper-parameter	Values
'score_func'	[chi2, f_classif, mutual_info_classif]
'k'	[10000, 25000, 40000, 55000]

Table 22 SelectKBest grid-search hyper-parameter values

The feature selection is done on the training set and then the validation and test set is transformed into the selected features of the training set.

The performance on the validation set was as follow, grouped by 'k':

'k'	'score_func'	Accuracy score
10000	chi2	0.6669
25000	f_classif	0.6751
40000	chi2	**0.6781**
55000	chi2	0.6777

And the result running with best performing feature selection on test set: **0.6704**

4.1.2. Conclusion

As we see there is a **0.814% decrease** in performance using feature selection.

4.2. Sampling

Since our dataset is heavily imbalanced we can introduce techniques for dealing with that as a means for improving accuracy. One such technique is sampling and we will take a look at some popular sampling strategies for both over-sampling minority labels and under-sampling the majority labels.

To do sampling we will look outside the scikit-learn libraries and use a package called imbalanced-learn [14] that has a lot of modules for doing different sampling techniques. A great thing about this package is that it is developed with compatibility with scikit-learn in mind so we can use the modules seamlessly with our models.

4.2.1. ClusterCentroids

This [15] is sort of a hybrid of under-sampling as it introduce new synthetic samples. By creating k centroids using a KMeans clustering algorithm applied to samples of same label and using them as representatives for the label. It is sensitive to how well the data is able to be clustered and there are a couple strategies as to how much of this is applied. This is represented in the hyper-parameter sampling_strategy. We will be doing a grid-search over this hyper-parameter.

Hyper-parameter	Values
'sampling_strategy'	['majority', 'not_minority', 'all']
'random_state'	1

Table 23 ClusterCentroids grid-search hyper-parameter values

The best performing hyper-parameter value being:

Hyper-parameter	Value
'sampling_strategy'	'majority'

Table 24 ClusterCentroids best hyper-parameter values

With an accuracy score on validation set: **0.6326**

Accuracy score on test set: **0.6332**

This gives a **6.318% decrease** in performance.

Since this technique use clustering techniques for creating synthetic samples it could benefit from reducing feature space using feature selection. We will also try with best performing hyper-parameters for both ClusterCentroids and SelectKBest to see if this is the case. The sampling will then be applied after feature selection in the pipeline.

Accuracy score on test set with feature selection: **0.6147**

And we see that feature selection gained no extra benefits.

4.2.2. RandomUnderSampler

The most basic under-sampling technique. As the name implies it [16] performs under-sampling by selecting random subsets from the different labels. Depending on sampling strategy it will do this from either only the majority label or from all but the smallest label. It can also adjust for replacement or not.

We will look at the two hyper-parameters available and perform a grid-search:

Hyper-parameter	Values
'sampling_strategy'	['majority', 'auto']
'replacement'	[True, False]
'random_state'	1

Table 25 RandomUnderSampler grid-search hyper-parameter values

With best performing hyper-parameters:

Hyper-parameter	Value
‘sampling_strategy’	‘majority’
‘replacement’	False

Table 26 RandomUnderSampler best hyper-parameter values

Accuracy score on validation set: **0.6258**

Accuracy score on test set: **0.6256**

This gives us a **7.442% decrease** in performance.

4.2.3. TomekLinks

Perhaps the most interesting under-sampling technique [17]. A Tomek’s link is when two samples of different labels are each other’s closest neighbors according to a chosen distance metric. The under-sampling is done by removing either one or both of these samples depending on what sampling strategy is chosen. With ‘majority’ strategy only the majority label samples are removed, with ‘auto’ strategy both are removed, except for the minority label samples.

We can grid-search only one hyper-parameter for TomekLinks:

Hyper-parameter	Values
‘sampling_strategy’	[‘majority’, ‘auto’]
‘random_state’	1

Table 27 TomekLinks grid-search hyper-parameter values

The best performing hyper-parameter value being:

Hyper-parameter	Value
‘sampling_strategy’	‘majority’

Table 28 TomekLinks best hyper-parameter value

With an accuracy score on validation set: **0.6803**

And accuracy on test set: **0.6770**

Giving us a **0.163% increase** in performance. So far the only technique that has actually given us an increase in performance.

4.2.4. ADASYN

Our first over-sampling method. Short for Adaptive Synthetic sampling method [18]. It use k-nearest neighbors to calculate the density for a region and populates it with synthetic samples until a density threshold is met.

It has two hyper-parameters to tune, sampling_strategy same as before and n_neighbors. The latter is used to define how many neighbors in KNN is used to determine the density of a region.

Due to a software error in this implementation other sampling strategies then minority return an error. Because of this we tried searching using a dictionary defining manually sample thresholds for each label. Every effort to tune this with either all labels with same threshold as majority or different grading of threshold for each label returned worse results then using minority as sample strategy. We therefore focus only on this strategy.

We will perform a grid-search over the hyper-parameters:

Hyper-parameter	Values
‘sampling_strategy’	[‘minority’]
‘n_neighbors’	[23, 27, 31, 33]
‘random_state’	1

Table 29 ADASYN grid-search hyper-parameter values

With best performing hyper-parameters:

Hyper-parameter	Value
‘sampling_strategy’	‘minority’
‘n_neighbors’	27

Table 30 ADASYN best hyper-parameter values

Accuracy score on validation set: **0.6797**

And accuracy on test set: **0.6761**

Giving us a **0.030% increase** in performance.

4.2.5. SMOTE

Short for Synthetic Minority Oversampling Technique [19]. It is same as ADASYN, but differs by not using a density function for regions but rather a uniform grid like structure. It creates new synthetic samples on this grid until a label imbalance threshold is met. KNN is also used for deciding what labels are being synthesized on the grid.

Hyper-parameters are same as for ADASYN:

Hyper-parameter	Values
‘sampling_strategy’	[‘minority’, ‘auto’]
‘k_neighbors’	[3,5,7,13]
‘random_state’	1

Table 31 SMOTE grid-search hyper-parameter values

With best performing hyper-parameters:

Hyper-parameter	Value
‘sampling_strategy’	‘minority’
‘k_neighbors’	3

Table 32 SMOTE best hyper-parameter values

Here we get a lower end edge value of 3 as ‘k_neighbors’, but we do not search at lower values of k.

Accuracy score on validation set: **0.6795**

And accuracy score on test set: **0.6761**

Giving us a **0.030% increase** in performance, exactly same as ADASYN.

4.2.6. RandomOverSampler

The most basic over-sampling technique [20]. Creates new synthetic samples by random sampling with replacement from the minority classes depending on what sampling strategy is chosen.

Grid-search over hyper-parameters:

Hyper-parameter	Values
‘sampling_strategy’	[‘minority’, ‘auto’]
‘random_state’	1

Table 33 RandomOverSampler grid-search hyper-parameter values

With best performing hyper-parameter:

Hyper-parameter	Value
‘sampling_strategy’	‘minority’

Table 34 RandomOverSampler best hyper-parameter values

Accuracy score on validation set: **0.6794**

Accuracy score on test set: **0.6761**

Giving an **0.030% increase** in performance.

4.2.7. Conclusion

If we look at the results in the table below we can see that none of the sampling techniques gave any substantial increase in performance.

Sampling technique	Change in performance
ClusterCentroids	-6.318%
RandomUnderSampler	-7.442%
TomekLinks	**0.163%**
ADASYN	0.030%
SMOTE	0.030%
RandomOverSampler	0.030%

Table 35 Resulting performance change of sampling techniques

As we can see our best choice here would be TomekLinks that gave a small increase in performance. We can conclude that the sampling techniques used here have little to give in regards of performance.

4.3. Word-embedding

As mentioned earlier word-embedding try to capture more of the contextual information of the words and could therefore provide additional performance by use of this additional information. We will look at two different models, with and without lemmatization. The reason for this is that we can not be quite sure that the trained embedding model follows the exact same lemmatization process as we have for our corpus. Also the nature of how fasttext embedding is done by splitting up words into skipgrams it could prove that an unlemmatized model with more words in the dictionary handles our corpus better.

When we create our baseline word-embedding we will use the TF-IDF values and vocabulary of the train set.

First we will see how well the two different models perform alone. We will use the LinearSVC classifier with same settings as used in earlier experiments.

Word-embedding model	Validation set score	Test set score
80 (lemmatized)	**0.6667**	**0.6728**
81 (non-lemmatized)	0.6606	0.6701

Table 36 Word-embedding models performance alone

As we can see the lemmatized model slightly outperforms the non-lemmatized model. Though the difference is minimal we will go further with model 80 in further experiments.

We also ran our grid-search on hyper-parameters for classifiers and can conclude that the best performing hyper-parameters and classifier, LinearSVC, also perform close to best on the embedded data alone.

Now we will use the word-embeddings in addition to our TF-IDF vectorizations of our data and see if they combined will perform better. TfidfVectorizer and LinearSVC is run with best performing hyper-parameters from previous experiments.

Set	Accuracy score
Validation	0.6983
Test	**0.7030**

Table 37 TF-IDF and word-embedding combined accuracy scores

Here we see we get a substantial **4.009% increase** in performance on the test set from initial baseline.

Since we are running with a SVM model we need to be careful with the scale of our features, if we do a quick test we can see that the word-embeddings have a maximum value of 1.844 and a minimum value of -1.535. This will make our SVM model unstable as our values are TF-IDF frequencies between 0 and 1. So we will apply a scaler to our word-embeddings so that the values range from 0 to 1. To do this we use another package from scikit-learns library, MinMaxScaler [21] and we set it to scale to the range [0,1]. Then we run again.

Set	Accuracy score
Validation	0.6995
Test	**0.7039**

Table 38 TF-IDF and word-embedding scaled combined accuracy scores

And we see that the get a slight increase from without scaling, **4.143% increase** in performance from initial baseline.

4.3.1. Conclusion

Adding word-embedded representation gave us a substantial increase in performance combined with the traditional TF-IDF values.

4.4. Calibration

Since we are using a SVM model for our classifications, which is a margin-maximizer model it does not inherently have probabilities. This is something we will need to have when we later want to perform semi-supervised methods. A calibrator can also improve classifications so we will apply it here and see how it affects the results.

4.4.1. CalibratedClassifierCV

There is a module for calibration in scikit-learns library called CalibratedClassifierCV [22]. It will wrap around our LinearSVC classifier and use cross-validation to split the input train data in k-fold internal train and test sets. Using the internal train set it will perform classification using the LinearSVC classifier, and in our multiclass context it will do so in 1-vs-rest fashion. It will use a binary class of 0 and 1 for the predictions and using the test set it will fit a regressor on these predictions. Read more about this module in the referenced manual.

There are two hyper-parameters to tune for CalibratedClassifierCV and those are the number of folds for cross-validation and method for calibration. We will use 3 fold cross-validation for performance reasons. But we will check what method performs best on validation set.

Hyper-parameter	Values
'method'	['sigmoid', 'isotonic']

Table 39 CalibratedClassifierCV hyper-parameter values

Best performing method:

Hyper-parameter	Value
'method'	'isotonic'

Table 40 CalibratedClassifierCV best performing hyper-parameter

Accuracy on validation set: **0.6791**

Accuracy on test set: **0.6800**

Calibration benefited our classification as well as providing us with the needed probabilities we need later. It got a **0.607% increase** in performance on test set.

4.5. Conclusion

If we now combine all that we have tried into one model, we can get what we will call our supervised model baseline.

Our supervised model consists of the following modules:

Module	Hyper-parameters
TfidfVectorizer	['lowercase': False, 'max_df': 0.25]
LinearSVC	['C': 0.5, 'loss': 'squared_hinge', 'tol': 0.01]
TomekLinks	['sampling_strategy': 'majority', 'random_state': 1]
fastText word-embedding	Model 80 (lemmatized)
MinMaxScaler	['feature_range': (0,1)]
CalibratedClassifierCV	['method': 'isotonic', 'cv': 3]

Table 41 Final model with conventional means of improvements.

With accuracy score on validation set: **0.6987**

And accuracy score on test set: **0.7066**

Giving us a total improvement from initial baseline by **4.542% increase** in performance.

Now we have some good results to compare our further experiments with. We have an initial baseline with just vectorization and classifier and we also have numbers to compare individually how well supervised methods and representations improved our performance.

We will now use this model in our further experiments with semi-supervised methods.

5. Semi-supervised methods

Semi-supervised techniques are techniques that involve both labeled and unlabeled data. It is therefore considered to be in between supervised and unsupervised learning. The motivation for semi-supervised learning (SSL) is that supervised learning requires labeled data which can be expensive and time consuming to attain, while unlabeled data is readily available in plentiful.

For semi-supervised learning to be effective it relies on at least one of these basic assumptions about the relationships between the labeled and unlabeled data;

Smoothness/continuity assumption: this means that samples close to each other in feature space should be close to each other in label space. It also imply that samples of same label exists in a clustered region of feature space.

Low density assumption: meaning that decision borders should exist in areas of feature space with low density. This assumption implies that there exists some separation between labels in feature space and that these areas are sparsely populated.

Manifold assumption: Here we assume that the samples we observe in feature space exists on lower-dimensional manifolds and that samples of same label share the same manifold.

SSL methods can be divided into two main categories [23] each with their own sub-categories. These main categories are; Inductive methods and transductive methods. Inductive methods provide models for predictions of samples that cover the full feature space, while transductive methods can only predict what it has seen before.

We will focus on the inductive methods in this work.

First we will look at the oldest and most known SSL methods under the sub-category called wrapper methods. These are methods that first train a model on the available labeled data for then to iteratively classify unlabeled data and re-train the model on a combination of labeled and pseudo-labeled data. They are very versatile as they can be used with any classification algorithm. These methods can again be sub-categorized as either self-training methods or co-training methods. We will be working with self-training methods, but also inspired by co-training methodology.

We will also look into the sub-category of unsupervised preprocessing. These are methods that can be used for feature extraction from the unlabeled data, unsupervised feature engineering from the labeled data or as a method for setting initial parameters for the supervised model. We will look into methods for feature extraction and engineering.

5.1. Self-training

Self-training was introduced by Yarowski [24] for word sense disambiguation. It has since been applied to many problems with success. [25]. Since it is such a versatile method it gives rise to many configurations of how to attain pseudo-labeled data, how to use it and how to increment and stop the process.

Self-training can be considered the most basic wrapper method. It attains pseudo-labels by a supervised classification algorithm. Since it is a wrapper function, the classifier is not aware of any difference in labeled and pseudo-labeled data and we can also therefore apply this method with any classifier available. The process is iterative and for each iteration the classifier will create new pseudo-labels. Depending on the algorithm it will append all or portions of the pseudo-labeled data to the next iterations train data. It is natural to consider some type of confidence in the predictions as a parameter to discriminate what is included in the training data.

If the process of attaining pseudo-labeled samples is based on probabilities it can resemble Expectation-Maximization [26]. Here there has been done work with NB-classifiers on detecting fake product reviews [27].

The process is also sensitive to reinforcing any mistakes made in the pseudo-labeling and it is therefore important to either get as confident pseudo-labels as possible or have some way of dealing with the mis-labeled samples.

5.1.1. Relation to co-training

Co-training [28] is an extension of self-training with some set conditions on how to use the pseudo-labeled data. It involves two or more classifiers, each trained on their separate views of the data. It was originally developed for dataset with native multiple views of the data, for instance in web-related tasks where one could consider link text and target text as two separate views sharing the same label space. The co-trainers then creates pseudo-labeled samples that they append to their counterparts training data. A distinction to be made here is that in co-training we are interested in the samples where the co-trainers disagree on the label.

For co-training to be effective it relies on the assumptions that the different views of the data are independent and not highly correlated. It also assumes that each view is sufficient to make credible predictions.

It has been proven that it is possible to achieve successful co-training with only single view data by way of splitting the feature space into separate views [29].

We will in our experiments make self-training models that take inspiration from these conditions, by way of splitting feature space for multiple trainers and also by splitting the training data. We will not follow the disagreement condition for exchanging information through unlabeled data.

5.2. Unsupervised pre-processing

Feature extraction is an old and extensive subject and we will not describe it other then in how we will use it in regards to our unlabeled data. We do not care about the labels or pseudo-labels of unlabeled data, but we rather want to extract features existing in the unlabeled portion of the data at the vectorization stage.

For feature engineering we will look into cluster-then-label techniques of improving classification. Here we are interested in if mixing the labeled and unlabeled data will improve results in the clustering stage.

6. Experiments

We will now perform semi-supervised methods to further improve on our classification accuracy score.

6.1. Feature extraction, new semi-supervised baseline

Our first experiment with the unlabeled data is also one of convenience. Since we will be doing a lot of iterations and re-training of our model with different combinations of our labeled and pseudo-labeled data it would be very time and resource consuming to perform the word-embedding conversion of the vocabulary for each iteration. Instead we can pre-calculate these for each document by applying TF-IDF vectorization to all the training data and unlabeled data combined and create the fasttext word-embedding representations for each sample and store these vectors as a feature we can extract directly.

This would be considered feature extraction as we get the extended vocabulary from the unlabeled data as well as more regularized IDF values from the much larger dataset.

Note that we do not include the validation and test set in this pre-calculation, these sets are transformed into the train plus unlabeled vocabulary after fitting the vectorizer.

Our baseline model is then as follows:

Module	Hyper-parameters
TfidfVectorizer	[‘lowercase’: False, ‘max_df’: 0.25]
LinearSVC	[‘C’: 0.5, ‘loss’: ‘squared_hinge’, ‘tol’: 0.01]
TomekLinks	[‘sampling_strategy’: ‘majority’, ‘random_state’: 1]
fastText word-embedding	Model 80 (lemmatized) pre-run on all unlabeled+train set
MinMaxScaler	[‘feature_range’: (0,1)]
CalibratedClassifierCV	[‘method’: ‘isotonic’, ‘cv’: 3]

Table 42 Semi-supervised baseline model with feature-extracted word embeddings

This model accuracy on validation set: **0.7038**

And accuracy on test set: **0.7122**

And we see that word-embeddings with TF-IDF weights and vocabulary from the unlabeled data already give us a considerable increase from supervised baseline, a **1.007% increase**.

We now consider this as our new baseline as we will be using this model for our further experiments.

Note; we did try this same feature extraction method to our regular bag-of-words representation with no change in performance.

6.2. Self-training, single model

The most intuitive way of leveraging the unlabeled data is by applying a classifier to it, identify the most confident samples, add them to the train set and reapply. Then we only have to find the threshold of confidence and let it run for as many iterations as we want or until there is no more unlabeled data to append.

We can decide if we want to classify all the unlabeled data at once and rerun the algorithm on all the labeled data until there are no samples above confidence threshold.

Another approach is to apply the algorithm to a sample of n size of the unlabeled data, append the samples above confidence threshold to train set and remove them from the sampling pool of unlabeled data and repeat until either no more samples reach above threshold or the sampling pool is empty.

We will try both and compare results, but we can speculate on some intuitions about these methods.

If we classify all the unlabeled data we will either have to set the confidence threshold very high so that we get very few samples and the algorithm will converge quickly and we will not gain much from the unlabeled data. If we lower the threshold we will get more samples, but of less quality and the model will become very unstable and deteriorate quickly. It is perhaps applicable to think about it as a gradient decent, if we find a optimal threshold we will stabilize at some local minimum.

If we perform the algorithm on subsets of the unlabeled data instead we might mitigate bad predictions that would be inevitable when we predict all at once. And also by moving more slow we can learn more on the way which may make more of the unlabeled data available at a given threshold.

6.2.1. All data

First we will run our model in 5 iterations and with confidence (probability) threshold of 0.99 and 0.98 on all the unlabeled data.

Unlabeled data	Iterations	Confidence	Total samples added	Final acc. on test set
All	5	0.99	6116	**0.6951**
All	5	0.98	61321	0.6744

Table 43 Self-training on all unlabeled data

As we can see self-training on all the unlabeled data at once get deteriorating performance. Let us inspect the distribution of the new pseudo-labeled samples added in the last run with confidence of 0.98 and with a **2.401% decrease** in performance.

Pseudo-label	Samples added
1	1
2	0
3	1510
4	38
5	0
6	312
7	31
8	318
9	63
10	628
12	9085
13	1
14	6
15	99
16	1131
17	51
18	0
19	2
20	91
21	0
23	16
24	35
25	11
26	1
91	634
92	47083
93	174

Table 44 Self-training with all unlabeled data and confidence at 0.98 pseudo-label distribution

We can clearly see that our pseudo-labeling distribution is not even. We can see some correlation with the label distribution in the labeled data, but not completely. For instance label 93 is the second largest label after 12, but it gets only 174 pseudo-labeled samples. Label 92 is the 9th largest label in the labeled data and it gets 76.8% of all the pseudo-labeled samples.

We could reason that this is due to a difference in distribution from the labeled data to the unlabeled data. We did mention earlier that we saw a slight skew in mean length and standard deviations indicating that the unlabeled data consists of slightly shorter articles with greater standard deviations. This would correlate with the distribution of length metrics for label 92.

If we look at the content of label 92, which is "Sporting events, athletes", we could perhaps imagine that this is a category filled with names of sports clubs and athletes. Such names would be very discriminatory as they rarely appear in any other circumstances. They also often appear in articles many at once creating good patterns to enforce their discriminatory effect.

If we look at the confusion matrix for our baseline results and compare it with this self-training result the label 92 accuracy has declined from 0.87 to 0.52 and the labels most confused with label 92 are the labels 24, 91, 93 and 23. These are all sport, events and cultural related labels, with exception of 93 which is a miscellaneous label containing anything. We could infer from this that we have wrongfully pseudo-labeled samples from these related classes, not only because of relation and similarities, but also because of the magnitude of number of samples that have crossed the threshold. We would be moving the decision border wrongfully in wrong direction just by the density of samples in this label.

6.2.2. Sample from pool

When we use the unlabeled data as a pool to sample from we can regulate how many new pseudo-labeled samples we add to the train set per iteration. This way we can leverage any additional learning incrementally as opposed to when we sample all at once.

We will perform a grid-search on the following parameters:

Parameter	Values
Iterations	[5, 10, 15]
Confidence	[0.99, 0.98]
Sample size	[50000, 100000, 150000]

Table 45 Self-training sample from pool parameter grid-search values

Below is a table with top 5 and bottom 5 results from this grid-search, ranked by accuracy.

Unlabeled sample size	Iterations	Confidence	Total samples added	Final acc. on test set
50 000	5	0.99	786	**0.7125**
50 000	10	0.99	1429	0.7102
100 000	5	0.99	1428	0.7101
50 000	15	0.99	1886	0.7098
150 000	5	0.99	1995	0.7089
…	…	…	…	…
50 000	15	0.98	7220	0.6948
100 000	15	0.98	13776	0.6943
150 000	10	0.98	14878	0.6941
100 000	10	0.98	9684	0.6931
150 000	15	0.98	22051	0.6920

Table 46 Self-training sample from pool grid-search results

Pseudo-label	Top rank	Bot. rank
1	0	0
2	0	0
3	35	835
4	0	15
5	0	0
6	4	148
7	0	2
8	9	52
9	0	26
10	15	237
12	182	3752
13	0	1
14	0	4
15	0	12
16	35	382
17	0	21
18	0	0
19	0	0
20	0	10
21	0	0
23	0	3
24	0	11
25	0	2
26	0	1
91	5	98
92	501	16431
93	0	8

Table 47 Self-training sample from pool pseudo-label distribution

We see that the top result gives a slight, **0.042% increase** in performance, but as we add samples performance deteriorates. The trend is clearly that with added samples and lowering confidence performance worsen.

And from the pseudo-label distribution table to the right we clearly see the same pattern as when we sampled all the data at once. And label 92 is still overly represented.

6.2.3. Sample from pool with validation

To counter the deterioration with added samples we can use the validation set to train a model and check whether it improves or not. If samples added to train set improves accuracy on validation set we include them in the train set, if not we put them back in the pool.

In the most extreme case here we could check every single sample and train a model for each, but that would be very impractical and resource demanding. But we would benefit from having a smaller set of samples to evaluate for each iteration. We will use the parameters of the top ranked model as it will give approximately 150-200 samples per iteration.

If we follow the statistical chance of getting 1% error we should have about 1-2 badly classified samples per iteration. Now depending on how bad they are on overall performance we have a way of discarding them.

Now we have the same parameters as previous, number of iterations, sample size and confidence threshold, but we also set an initial validation accuracy threshold.

Unlabeled sample size	Iterations	Confidence	Total samples added	Initial validation acc.	Final acc. on test set
50 000	5	0.99	178	0.7038	**0.7122**
50 000	15	0.99	178	0.7038	0.7121

Table 48 Sample from pool with validation initial parameters results

As we can see the model did not achieve any more samples that increased the validation accuracy from 5 iterations to 15, in fact it achieved the top score in the first iteration and discarded all subsequent iterations samples. We will have to increase the amount of iterations considerably to be able to gain more from this procedure. In this regard it becomes impractical to use our best performing model in this iterative selection process.

Rather then using the full model for the selection of samples to append to the training set we will use a model of just TfidfVectorizer for vectorization and the ComplementNB classifier as it provides quick classification and has probability inference inherently. To make the probabilities most accurate we will also use the CalibratedClassifierCV module to perform scaling.

Module	Hyper-parameters
TfidfVectorizer	['lowercase': False, 'max_df': 0.25]
ComplementNB	['alpha': 0.2]
CalibratedClassifierCV	['cv': 3]

Table 49 Quick model for validation

We will of course suffer by less accurate classifications, but in this process we are filtering out the majority of less probable classifications and we are also filtering out by judging improvement in validation accuracy. We assume that samples that improve this simpler quick model will also be beneficial for our baseline model.

Unlabeled sample size	Iterations	Confidence	Total samples added	Init./final valid. acc.	Final acc. on test set
250 000	150	0.98	37	0.6598/0.6604	**0.7124**
50 000	150	0.97	72	0.6598/0.6602	0.7110

Table 50 Sample from pool with validation by quick model

As we can see the first run using a high (0.98) confidence we got a **0.028% increase** in performance, but achieves very few added samples, 37 in total. It got approximately 10-30 candidates per iterations, but only added them to train set 4 times and lastly as early as the 18th iteration. One could argue that the sample size vs total amount of samples in the pool, roughly 1/6th of the unlabeled data would indicate that most samples even though randomly selected at each iteration (but deterministic in between experiments to be able to compare) have been evaluated early on. But an argument for performing as many iterations would be that at each iteration a new set of candidates are chosen, and since we only check whether the set of candidates as a whole increase or decrease performance it would be beneficial to have many random sets of these candidates so that we increase the chance of getting a good set.

Another thing to notice about the high confidence run is that all the pseudo-labels added were label 12. This could indicate that the effect we are seeing with an overrepresentation of pseudo-labels in label 92 is a reinforcement phenomenon.

When we look at the lower (0.97) confidence run we decreased the iteration sample size to 50 000 so that we get roughly the same amount of confident candidates per iterations as previous run (20-40). We did get more samples added to the train set, but performance was worse on the test set even though performance on validation set with the quick model was approximately same as the higher confidence run. For the pseudo-label distribution we got 65 samples with label 12 and 7 samples with label 92.

Now we will forget about confidence and just classify 30 samples per iteration and add them to train set if the validation accuracy is improved.

Unlabeled sample size	Iterations	Total samples added	Init./final valid. Acc	Final acc. on test set
30	150	420	0.6598/0.6649	0.7110
30	300	450	0.6598/0.6651	**0.7118**

Table 51 Sample from pool with validation and no confidence threshold

Pseudo-label	150-iter.	300-iter.
1	8	8
2	8	9
3	16	16
4	9	10
5	2	2
6	4	5
7	4	4
8	3	3
9	5	5
10	7	8
12	39	42
13	7	7
14	3	3
15	12	14
16	10	12
17	9	9
18	2	2
19	10	10
20	16	18
21	3	3
23	5	6
24	8	8
25	3	3
26	4	4
91	79	84
92	76	82
93	68	73

Table 52 Sample from pool with validation and no confidence threshold pseudo-label distribution

Here we can see that we get further in validation accuracy and we also get more samples added. But we get a **0.056% decrease** in accuracy from the baseline on test set.

We can see from the pseudo-label distribution that we get a better representation of labels, but we see a overrepresentation of the so-called 'trash' labels, 91, 92 and 93. This could indicate a difference in distribution in the unlabeled vs labeled corpus.

Let us do a classification over the entire corpus using the quick model and see how our pseudo-labels distribute.

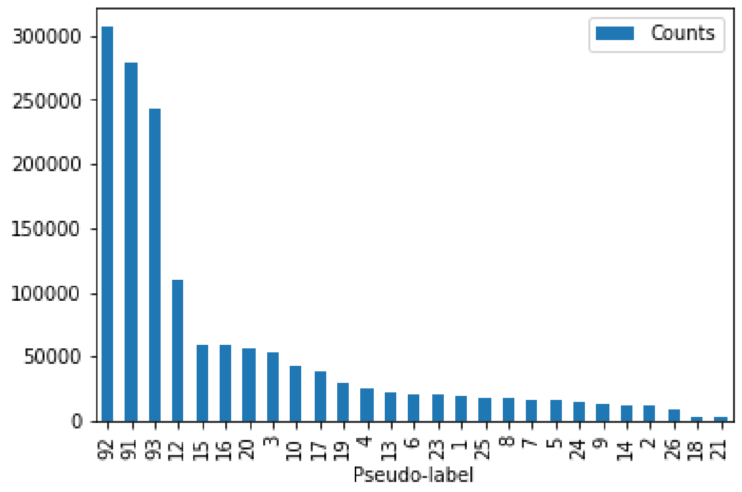

Figure 16 Quick model predictions over unlabeled corpus

From this we clearly see that our unlabeled corpus most likely have a different distribution of the so called 'trash' labels. Of course we can not trust our predictions to be accurate, but they give us a strong indication and we also see that the distribution of the other labels follow the labeled corpus distribution closely.

6.2.4. Trash filter

Let us try to filter out the trash labels from the unlabeled corpus before we apply the pseudo-labeling. To do this we create a binary label for the train set to be 1 if label is in [91,92,93] and 0 if else. We then use these labels to train our baseline model and apply it to the unlabeled data.

When we apply it to the validation and test set we get results at 90% and 91% so we should have a high accuracy for this filter.

When applied to entire corpus, both unlabeled and labeled we can create a confusion matrix from the labeled data.

	Is trash	Is not trash
Predicted trash	6145	793
Predicted not trash	1478	28856

Table 53 Trash filter confusion matrix over labeled data

And we see that we get a good precision of 0.886. Our filter should be conservative with low Type 1 errors and also a reasonable Type 2 error rate.

When filter is applied we get 764 869 positives, meaning we filter out 50.2% of the unlabeled data.

Lets apply the trash filter and run the grid-search performed earlier and see if get better results.

Sample size	Iterations	Confidence	Total samples added	Final acc. on test set
50 000	10	0.99	948	**0.7133**
150 000	5	0.99	1267	0.7125
100 000	5	0.99	956	0.7124
50 000	15	0.99	1232	0.7118
50 000	5	0.99	562	0.7118
…	…	…	…	…
150 000	5	0.98	5349	0.7031
100 000	10	0.98	6118	0.7008
100 000	15	0.98	7110	0.6920
150 000	10	0.98	7236	0.6886
150 000	15	0.98	7737	0.6835

Table 54 Self-training sample from pool with trash filter grid-search results

Here we see that we now get a top result with **0.154% increase** in performance and top 3 are all increasing in performance from baseline. Overall the performance is improved with the trash filter and we also observe that we get a considerable lower amount of samples added in the lower ranked results indicating that we have curbed some of the reinforcement phenomenon we saw with label 92.

Let us look at the pseudo-label distributions.

Pseudo-label	Top rank	Bot. rank
1	0	0
2	0	0
3	119	985
4	0	15
5	0	0
6	7	247
7	0	5
8	21	293
9	4	34
10	51	443
12	670	4811
13	0	3
14	0	6
15	0	53
16	76	748
17	0	38
18	0	0
19	0	0
20	0	43
21	0	0
23	0	3
24	0	1
25	0	9
26	0	0
91	0	0
92	0	0
93	0	0

Table 55 Self-training sample from pool with trash filter pseudo-label distribution

Here we see that our trash filter have effectively eliminated all samples from the trash labels something we should be wary of. We see a stronger distribution over the other major labels where now label 12 is the largest label which correlates with the labeled data distribution. We do not see the same degree of reinforcement phenomenon in label 12 as we did with label 92.

6.2.5. Conclusion

We have seen that our self-training method has problems gaining any advantage from the unlabeled data. It is difficult to get samples that we can be confident about and when lowering confidence we see that performance deteriorates. We have also seen that we get a disproportionate amount of samples with pseudo-label 92. This trend was also present in the run where we just classified 30 samples and added if performance on validation set improved, but here accompanied by the other two 'trash' labels, 91 and 93. By doing a classification over the entire unlabeled corpus we saw that we have a strong indication that there is a different distribution of the 'trash' labels in the unlabeled corpus from the labeled corpus.

We created a 'trash' filter and we saw that applying this increased performance and eliminated the disproportion and reinforcement phenomenon in trash labels, but we also eliminated any potential learning from 50% of the unlabeled data.

We will apply the 'trash' filter to our further experiments.

6.3. Self-training, multiple models

Inspired by co-training and Random Forrest algorithm [30] we will now try and train multiple models and see if we can gain some benefits by creating diversity in the models predictions by either splitting the feature space or sample space.

6.3.1. Multiple models, feature-split

First we will create a method of splitting the feature space in two or more separate views in a semi-randomized fashion. We will use chi-squared scoring of the features and rank them according to score. Then we take the top n ranked features and evenly distribute them among the views so that each view gets similar distributions of top ranked features. We do this by simply taking the ranked position modulus the number of views to decide which view to assign the feature. The rest of the features not in the top n ranked we assign randomly, but even among views.

When features are split we train a model for each view using the split feature-sets as vocabulary in the vectorization stage. We will also try using different models for our views to see if different styles of classification algorithms could be beneficial. We will also see if using word-embeddings as a separate view is beneficial in combination with regular bag-of-words representations.

We now have a set of parameters for our feature-split algorithm that we should perform a grid-search on to see how it performs. But before we do this we want to create a metric measuring the aspect of the algorithm that we consider important. This way we have a way of ranking our results from the grid-search.

We have seen in previous experiments that when using probabilities for measuring confidence in sample predictions there is a natural tendency to end up with an overrepresentation of samples from the majority labels when we set a high threshold of confidence. This threshold is necessary because we also see that lowering it will result in too many misclassifications and the model deteriorates quickly. There is also possibilities of negative reinforcement phenomena where one label reinforce itself and quickly become so dense or get such a large prior that it dwarfs all other classes and performance deteriorates.

So what we want to see in our algorithm is as evenly distributed pseudo-labels as possible with as little bias towards any one label and with as many labels represented as possible. We propose that such a metric could be created by taking the entropy of the pseudo-label-counts (number of predictions per pseudo-label) that meet the confidence threshold. The entropy will be small when any pseudo-label is overrepresented and larger when the pseudo-label-counts are as even as possible. This again we multiply with the number of unique pseudo-labels that met the confidence threshold, because it is most important to get as broad a distribution as possible.

Our metric is then set so that we can rank our grid-search results and get the algorithm that creates the most broadly distributed pseudo-labels while dampening the bias towards the majority labels as much as possible.

When we have the predictions and probabilities from our feature-split algorithm we also need to tune the threshold we set for what we consider confident predictions. This can be done by a stepwise function where we iteratively lower the threshold from 1 by some value and check the accuracy of the resulting confident pseudo-labeled samples using the validation set labels. When we reach a desired accuracy we break the step-function and return the threshold.

In considering what is a confident sample we can also choose whether to let all the models meet the probability threshold or let at least one model meet the threshold. We will perform the grid-search separately for each of these alternatives (we call them ‘all’ and ‘any’).

So now we have described our feature-split algorithm and the parameters we want to perform grid-search on.

6.3.1.1. Parameter tuning

Parameter	Values
‘top_n’	[200, 500, 2000]
‘splits’	[1, 2, 3]
‘wordembedding’	[True, False]
‘models’	[‘quick_model’, ‘baseline_model’, ‘quick_model+baseline_model’]
‘accuracy’	[0.98, 0.99]
‘confidence’	[‘all’, ‘any’]

Table 56 Multiple models, feature-split grid-search values

Here we see the parameters and values for our grid-search. The ‘splits’ parameter decides how many feature splits and models are created. We have 1 as a value here because in combination with ‘wordembedding’=True will give us the combination of bag-of-words representation and word-embedded representation as two separate views. The feature space will not be split between the two models in this case.

When we get the combination ‘models’=’quick_model’ and ‘splits’=2 for instance then we will have 2 quick_models trained on their respective split of the feature space.

When ‘models’=’quick_model+baseline_model’ and ‘splits’=3 we will get a feature space split in 3 for 2 quick_models and 1 baseline_model.

The case where ‘wordembedding’=False and ‘splits’=1 will just create a single regular classification model as in earlier experiments. We include this for reference.

Accuracy here refers to the target accuracy in validation set when performing probability confidence threshold tuning.

We will view the results grouped by ‘accuracy’ and ‘confidence’ separately as we consider these 4 groups of results to be in their own confidence group. Where ‘accuracy’=0.99 and ‘confidence’=’all’ is considered the most confident group and so on. We will refer to the different groups as Group **A, B, C** and **D**. We include the top 5 and bottom 5 ranked by our metric in the tables for reference.

Group A

Results for ‘accuracy’=0.99 and ‘confidence’=’all’ below:

‘top_n’	‘splits’	‘word embedding’	‘models’	‘accuracy’	‘confidence’	Metric
200	2	False	‘baseline_model’	0.99	‘all’, 0.928	**29.323**
200	2	False	‘quick_model+baseline_model’	0.99	‘all’, 0.909	25.002
0	1	False	‘baseline_model’	0.99	‘all’, 0.962	23.348
2000	2	False	‘baseline_model’	0.99	‘all’, 0.939	22.529
500	2	True	‘quick_model+baseline_model’	0.99	‘all’, 0.896	21.500
…	…	…	…	…	…	…
200	3	False	‘quick_model’	0.99	‘all’, 0.929	5.510
2000	2	False	‘quick_model’	0.99	‘all’, 0.982	4.081
500	3	False	‘quick_model+baseline_model’	0.99	‘all’, 0.904	4.065
500	3	True	‘quick_model’	0.99	‘all’, 0.889	3.954
500	3	False	‘quick_model’	0.99	‘all’, 0.899	3.816

Table 57 Multiple models, feature-split grid-search results for Group A

This could be considered the most conservative group as it has the highest accuracy and requires all trainers to be above the confidence threshold. We include the resulting probability threshold under the ‘confidence’ column for this table.

Pseudo-label	#1	#2	#3	#4	#5
3	29	30	23	23	17
4	1	1		1	1
6	10	12	9	6	9
7		2	2	1	1
8	6	5	7	5	4
9	2	3			
10	23	15	17	17	18
12	128	131	95	93	126
15	6	4	1	3	6
16	22	29	16	13	13
17	2		2	1	
20	7	10	5	5	11
25	1				
26	1	1			
91	2	2	2	1	1
92	45	30	26	25	27
93	1		1		1
SUM	**287**	**275**	**206**	**194**	**235**

Table 58 Multiple models, feature-split grid-search results pseudo-label distributions for top 5 from Group A

It might be a little confusing when we talk about accuracy and confidence at this point so let us clarify this a bit more. As mentioned above we use a step function where we iteratively lower the threshold for the classification probability or confidence on predictions on the validation set. We chose the steps to be 0.001. We will then first set the threshold to 0.999, then we check if there are any samples that meet the threshold, if not we continue to next step, 0.998, and so on. When we get samples that meet the threshold we calculate the accuracy of the samples predictions using the y-labels from the validation set. We continue lowering the threshold until we reach the accuracy we set as parameter, i.e. 0.99 or 0.98.

When we do this to the cases where we have only 1 split and ‘wordembedding’=False we are technically just empirically checking the validity of the classification probabilities. We get a couple interesting insights from this.

First we can see that the quick_model that is based on ComplementNB is pretty much spot on with its probabilities, meaning the threshold is at 0.99 when accuracy is at 0.99. The baseline_model tends to get a lower threshold then accuracy, approx. 0.03 or 3%. This effect is because the baseline_model uses a SVM classifier and SVM focus on the samples close to the decision border. It is therefore hard to calibrate it properly at the extremes of probability (close to 0 or 1) [31]. So we have learned that our NB based model gives more reliable probabilities then our SVM based model.

We can also see that threshold decreases significantly with the number of splits. At 3 splits we can see that the threshold drops to 0.929 to get an accuracy of 0.99 for the NB based quick_model. We could infer from this that if we assume independence between models then according to the multiplication rule of independent probabilities using the formula 1-(1-x)^3=0.99 and solve for x we get that x=0.785. This is what we should have as threshold if our models were perfectly independent. Then when we look at what we got empirically we can see that we are off from these numbers. This tells us that we can not assume independence between our models, though we can not tell how strongly they are dependent.

Now we look at the pseudo-label distributions and we see that our metric is working well, it prioritize number of different labels in the predictions over number of samples as we can see by looking at rank #5 that has more samples then both rank #3 and #4. When number of different labels is the same as with rank #3 and #4, it prioritize the one with the most even distribution.

Following is the tables for the rest of the groups:

Group B

Results for ‘accuracy’=0.98 and ‘confidence’=’all’ below:

‘top_n’	‘splits’	‘word embedding’	‘models’	‘accuracy’	‘confidence’	Metric
200	2	False	‘baseline_model’	0.98	‘all’, 0.902	**43.193**
200	3	False	‘baseline_model’	0.98	‘all’, 0.893	35.068
0	1	False	‘baseline_model’	0.98	‘all’, 0.943	33.872
500	3	False	‘baseline_model’	0.98	‘all’, 0.893	33.215
200	2	False	‘baseline_model’	0.98	‘all’, 0.901	32.280
…	…	…	…	…	…	…
2000	3	True	‘quick_model’	0.98	‘all’, 0.894	5.670
0	1	True	‘quick_model’	0.98	‘all’, 0.977	5.668
200	3	False	‘quick_model’	0.98	‘all’, 0.929	5.510
500	3	True	‘quick_model’	0.98	‘all’, 0.883	4.036
500	3	False	‘quick_model’	0.98	‘all’, 0.899	3.817

Table 59 Multiple models, feature-split grid-search results for Group B

Group C

Results for ‘accuracy’=0.99 and ‘confidence’=’any’ below:

‘top_n’	‘splits’	‘word embedding’	‘models’	‘accuracy’	‘confidence’	Metric
200	3	False	‘quick_model+baseline_model’	0.99	‘any’, 0.984	**26.996**
200	3	True	‘quick_model+baseline_model’	0.99	‘any’, 0.984	26.580
0	1	False	‘baseline_model’	0.99	‘any’, 0.962	25.168
0	1	True	‘baseline_model’	0.99	‘any’, 0.977	21.969
200	2	False	‘baseline_model’	0.99	‘any’, 0.966	20.162
…	…	…	…	…	…	…
2000	3	True	‘quick_model+baseline_model’	0.99	‘any’, 0.997	8.862
2000	2	True	‘quick_model+baseline_model’	0.99	‘any’, 0.993	7.613
2000	2	False	‘quick_model+baseline_model’	0.99	‘any’, 0.993	7.490
500	2	False	‘quick_model+baseline_model’	0.99	‘any’, 0.998	6.758
500	2	True	‘quick_model+baseline_model’	0.99	‘any’, 0.998	6.758

Table 60 Multiple models, feature-split grid-search results for Group C

Group D

Results for ‘accuracy’=0.98 and ‘confidence’=’any’ below:

‘top_n’	‘splits’	‘word embedding’	‘models’	‘accuracy’	‘confidence’	Metric
2000	2	False	‘baseline_model’	0.98	‘any’	**43.576**
200	3	True	‘quick_model+baseline_model’	0.98	‘any’	43.258
200	3	False	‘quick_model+baseline_model’	0.98	‘any’	43.217
2000	3	False	‘baseline_model’	0.98	‘any’	41.618
500	2	False	‘baseline_model’	0.98	‘any’	39.800
…	…	…	…	…	…	…
200	3	False	‘baseline_model’	0.98	‘any’	10.767
2000	2	True	‘quick_model+baseline_model’	0.98	‘any’	7.612
2000	2	False	‘quick_model+baseline_model’	0.98	‘any’	7.490
500	2	False	‘quick_model+baseline_model’	0.98	‘any’	6.758
500	2	True	‘quick_model+baseline_model’	0.98	‘any’	6.758

Table 61 Multiple models, feature-split grid-search results for Group D

And pseudo-label distributions for groups **B**, **C** and **D**:

Pseudo-label	#1	#2	#3	#4	#5
1				1	
3	45	36	36	36	43
4	4	2	4	1	1
6	16	11	11	13	13
7	3	2	3	1	2
8	11	10	11	12	10
9	4	2	5		
10	32	27	33	34	29
12	183	155	163	139	170
13	1				
14	1		1		
15	10	8	9	3	8
16	36	29	47	30	25
17	6	2	4	2	4
20	12	9	7	8	8
23	1	2			2
24	1				
25	3	1	1	1	1
26	1	1		1	1
91	11	7	8	10	10
92	71	56	57	62	64
93	3	3	3	4	2
SUM	**455**	**363**	**403**	**358**	**393**

Table 62 Multiple models, feature-split grid-search results pseudo-label distributions for top 5 from Group B

Pseudo-label	#1	#2	#3	#4	#5
3	13	14	23	20	19
4	1	1			
5	1	1			
6	3	3	9	6	5
7	1	1	2	2	
8	4	5	7	7	7
9	1	1	1		1
10	8	8	17	11	12
12	105	109	96	105	90
15	1	1	1	2	3
16	18	18	16	18	16
17	2	1	2	2	1
20	3	3	5	4	3
24	1	1			
26				1	
91	1	1	1	1	1
92	10	10	26	29	20
93	3	3	1		
SUM	**176**	**181**	**207**	**208**	**178**

Table 63 Multiple models, feature-split grid-search results pseudo-label distributions for top 5 from Group C

Pseudo-label	#1	#2	#3	#4	#5
3	46	61	54	47	45
4	4	6	6	6	1
5		2	2		
6	16	13	13	18	17
7	3	5	5	3	3
8	12	14	14	13	11
9	6	3	3	5	3
10	39	43	36	44	34
12	183	245	232	194	181
13	1	1	1	2	1
14	1			1	2
15	10	7	7	9	10
16	42	54	49	57	43
17	6	7	7	6	3
19	1				
20	8	19	18	8	9
23	1	5	5	3	1
24		3	3		1
25	2	1	1	2	
26	3	2	2	1	1
91	10	8	5	13	7
92	66	69	62	76	60
93	3	17	17	4	4
SUM	**463**	**585**	**542**	**512**	**437**

Table 64 Multiple models, feature-split grid-search results pseudo-label distributions for top 5 from Group D

If we look at the tables for group **C** and **D** we see that even though intuitively we considered these less strict because of the 'any' condition, it turns out this only forced the models to be even more confident to compensate for this condition. It is interesting to see that this gave us different constellations of models then in the 'all' condition groups. We also see that lowering our accuracy to 0.98 approximately doubles the amount of samples that meet the criteria.

Now we will test our best model from each of the four groups on the test set to see how they perform. We can perhaps expect that our models are somewhat overfit to our validation set.

6.3.1.2. Performance on test set

In the table below we see the 4 best performing models from our 4 groups. In the column 'accuracy' we see the target accuracy that we tuned our confidence threshold to meet. The values that come most close to this target accuracy is made out in bold.

Group	'top_n'	'splits'	'word embedding'	'models'	'accuracy'	'confidence', 'threshold'	Acc. on test set
A	200	2	False	'baseline_model'	0.99	'all', 0.928	**0.990**
B	200	2	False	'baseline_model'	0.98	'all', 0.902	0.985
C	200	3	False	'quick_model+ baseline_model'	0.99	'any', 0.984	0.971
D	2000	2	False	'baseline_model'	0.98	'any', 0.938	**0.980**

Table 65 Multiple models, feature-split best models performance on test set.

As we can see models **A** and **D** was spot on with their target accuracies, while model **B** overshot it a tiny bit. Model **C** is the model that is most off, but it is also the most complex of all the models and therefore probably most overfit to the validation set.

Pseudo-labels	A	B	C	D
1				1
3	30	53	16	56
4		3		4
6	8	17	4	18
7		1	2	2
8	9	16		18
9	1	1		3
10	14	27	6	30
12	127	183	129	183
13	2	3		3
14		1		
15	5	12	2	13
16	28	48	19	58
17	2	6	1	6
20	4	10	3	7
23		1		
24		1	1	
25		1		1
26			1	
91	3	11	1	10
92	52	78	19	79
93	1	3	5	4
SUM	**286**	**476**	**209**	**496**

Table 66 Multiple models, feature-split best models performance on test set pseudo-label distribution

Looking at the pseudo-label distributions we see no surprises.

We are happy with our results and now we will see how these feature-split models perform on unlabeled data using the best performing self-training method we got in previous experiment.

6.3.1.3. Performance in self-training

We will run our feature-split model from group **A** and group **D** in a self-training method to see if we get any better results. The column 'Model' refers to the model from that group.

Sample size	Iterations	Model	Total samples added	Final acc. on test set
50 000	10	A	20 838	0.6780
50 000	10	D	44 864	**0.6800**

Table 67 Multiple models, feature-split in self-training method initial results

Pseudo-label	A	B
1	5	38
2		4
3	2799	6382
4	163	746
5	1	19
6	847	1946
7	46	231
8	764	1799
9	87	557
10	1547	3649
12	10570	17639
13	16	208
14	32	288
15	412	1709
16	2493	5950
17	183	672
19	21	247
20	690	1955
23	48	194
24	20	112
25	72	309
26	12	163
91	3	13
92	5	10
93	2	24
SUM	**20838**	**44864**

Table 68 Multiple models, feature-split in self-training method initial results pseudo-label distribution

As we can see our initial results with the calibrated thresholds did not improve results and surprisingly the model D, calibrated for 0.98 accuracy gave the best results, but with a **4.521% decrease** in performance from baseline.

We will have to tune our threshold parameter further with the self-training method. We will do so manually incrementally and see if the accuracy converges.

Sample size	Iterations	Group	Threshold	Total samples added	Final acc. on test set
50 000	10	A	0.96	6063	0.6941
50 000	10	A	0.98	1273	**0.7002**
50 000	10	A	0.99	264	0.7000
50 000	10	D	0.96	18312	0.6814
50 000	10	D	0.98	4273	0.6918
50 000	10	D	0.99	880	0.6942
50 000	10	D	0.995	193	0.6952

Table 69 Multiple models, feature-split in self-training method manual threshold tuning results

As we can see increasing the threshold improves performance peaking in model **A** with threshold at 0.98 getting a **1.685% decrease** in performance from baseline. We did not achieve convergence with model **D**, but judging by the number of samples added and the level of improvement there is not much chance of it ever gaining any traction to give it any more significant improvements. Another thing we can observe is that the total number of samples decreased dramatically in the better performing threshold, ending up with only samples coming from the major labels 3, 10, 12 and 16. With a severe overrepresentation of samples from label 12.

Note, we performed the manual tuning of the threshold using the test set as reference and therefore these numbers should be considered optimistic.

6.3.1.4. Conclusion

I think we can conclude that multiple models with feature-split in the manner we have tried here did not succeed in gaining any advantage over the single-model approach. Splitting the feature space probably decreased each models performance and ability to correctly classify more then the advantage of having separate classifiers agree on predictions. We also saw good indications that the feature splits were far from being independent, as indicated by the multiplication rule of independent probabilities, which was expected, but probably to a degree in this corpus that made it difficult to make any advantage.

6.3.2. Multiple models, sample split

Our next approach is rather then splitting the feature space we split the sample space. By partitioning the samples on different heuristics we hope to create two different learners that have separate knowledge about the sample space as a whole.

We will compare two different heuristics and compare them to each other and also do a random split and in the end see how they compare when put in a self-training method.

6.3.2.1. Random split

For reference we will first perform a random split of the sample space into two partitions that we train two separate trainers on. We use the 'all' method for agreeing on predictions, meaning both trainers must agree on prediction. Then we filter on a threshold of prediction probability.

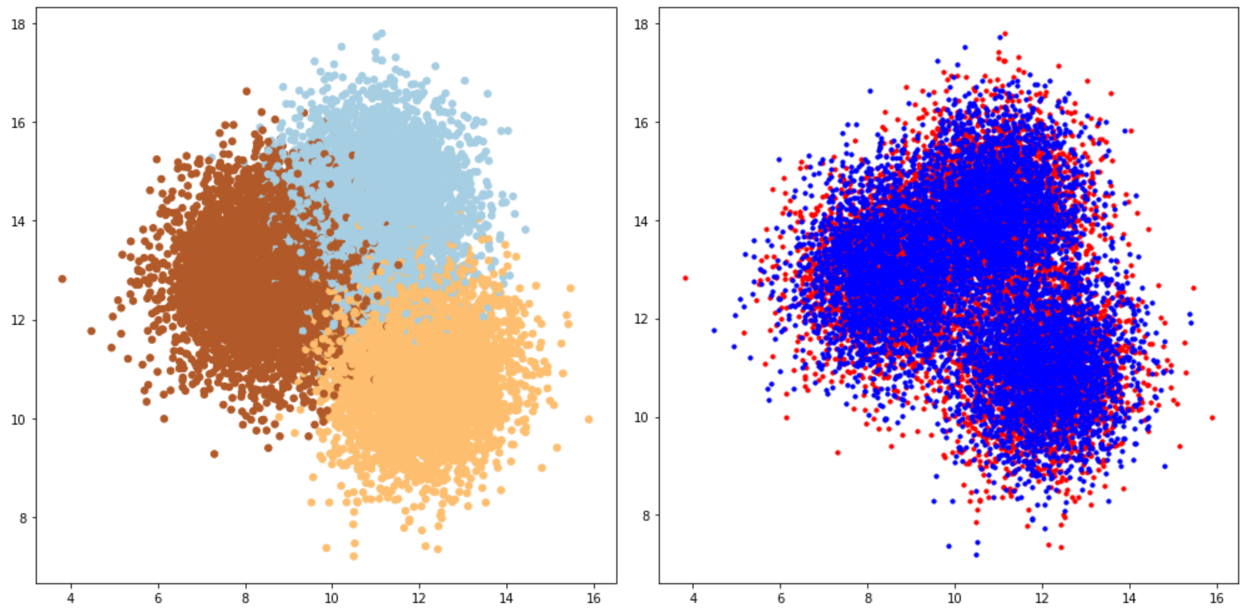

Figure 17 Sample space split, random split on synthetic data for illustration, 3 labels

With these parameters we run a self-training method for 10 iterations and check our results.

Sample size	Iterations	Threshold	Total samples added	Final acc. on test set
50 000	10	0.98	1675	**0.7077**

Table 70 Sample space random split results in self-training method

Here we get a **0.632% decrease** in performance from baseline. Not an improvement, but we do get better scores then from our attempts at doing feature split.

6.3.2.2. Entropy split

This heuristic we will find by first performing a stratified split of the training data in two sets of same size. We then train a classifier separately on each split and perform classification on the other set. We then have predictions and probabilities for each sample. We join the two splits back together into one training set again with the probabilities stored and based on the heuristic of entropy in the probabilities for each sample we split the training set again into two separate train sets.

The intuition of this approach is that samples with a high entropy in probabilities are harder to explain then samples with low entropy. This would translate to samples being closer and further away from the decision border in feature space. If we then split on a threshold of entropy, here we will use the mean entropy of all the samples, we will then have a learner that see only samples closer to the decision border and a learner that see only samples further away from decision border.

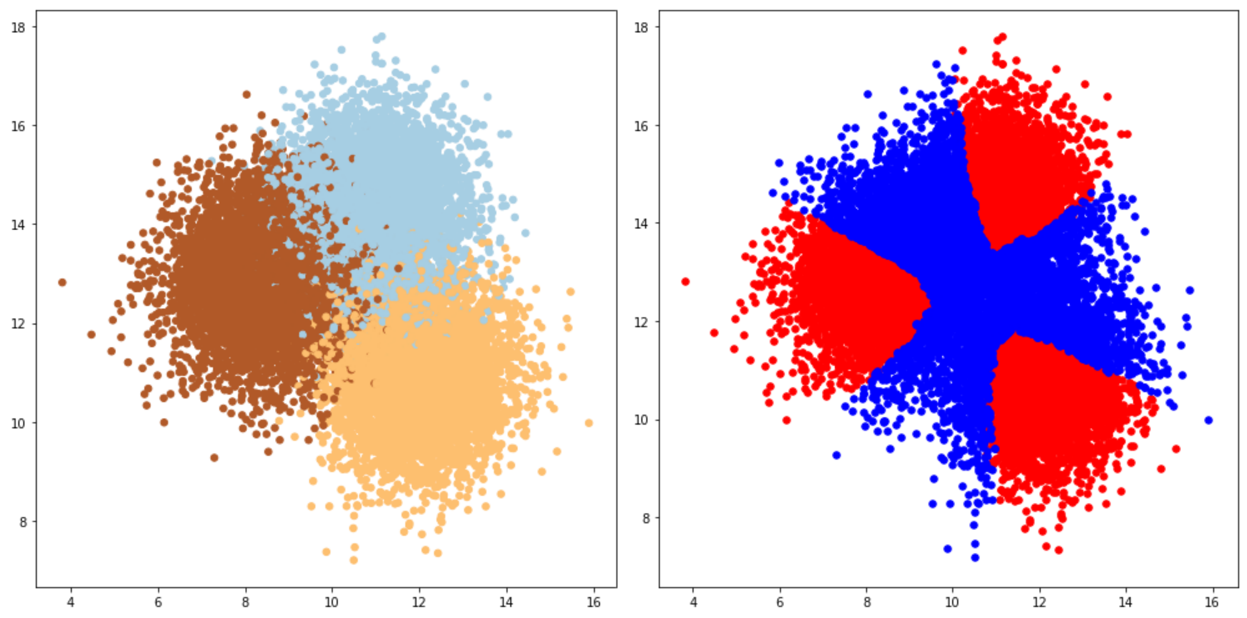

Figure 18 Sample space split, split on entropy mean on synthetic data, 3 labels

Above we can see from the figure that we get a nice partitioning of the feature space where the red and blue dots represents samples in the two different splits of the sample space.

We are using the baseline model for classification in both learners.

Sample size	Iterations	Threshold	Total samples added	Final acc. on test set
50 000	10	0.55	3541	**0.7079**

Table 71 Sample space entropy split results in self-training method

Here we got a **0.604% decrease** in performance from baseline, but we got more then double the amount of samples compared to the random split. We also see that we need a much lower threshold here because the trainer that is taught on the samples above the entropy threshold has a much lower confidence for its predictions with a mean of about 0.30, compared to the trainer with samples below entropy threshold that has a mean probability of 0.66 in its predictions.

Pseudo-l.		Pseudo-l.	
2	52	**17**	99
2	53	**19**	199
5	142	**20**	1568
6	21	**23**	93
7	3	**24**	19
10	13	**25**	20
12	611	**91**	11
13	199	**92**	4
14	155	**93**	2
15	213		
16	64		

Table 72 Sample space entropy split result pseudo-label distribution

Another thing to note about these predictions is the distribution. It is very different from the previous distributions that were very biased towards the major labels. Here we see much better spread. The most samples come with pseudo-label 20 which is not seen before.

This an interesting result, judging by the small decrease in performance compared to the amount of samples added we could consider the quality of the samples to be pretty good.

6.3.2.3. Support vector split

With the same intuition as with the entropy split we can do a split on distance to the hyper-plane in a SVM model of the train data. We first fit a SVM model on the train data and then we calculate the absolute distance to the hyper-plane and based on a threshold we split the data in two. One split contains data closer to the hyper-plane the other further away.

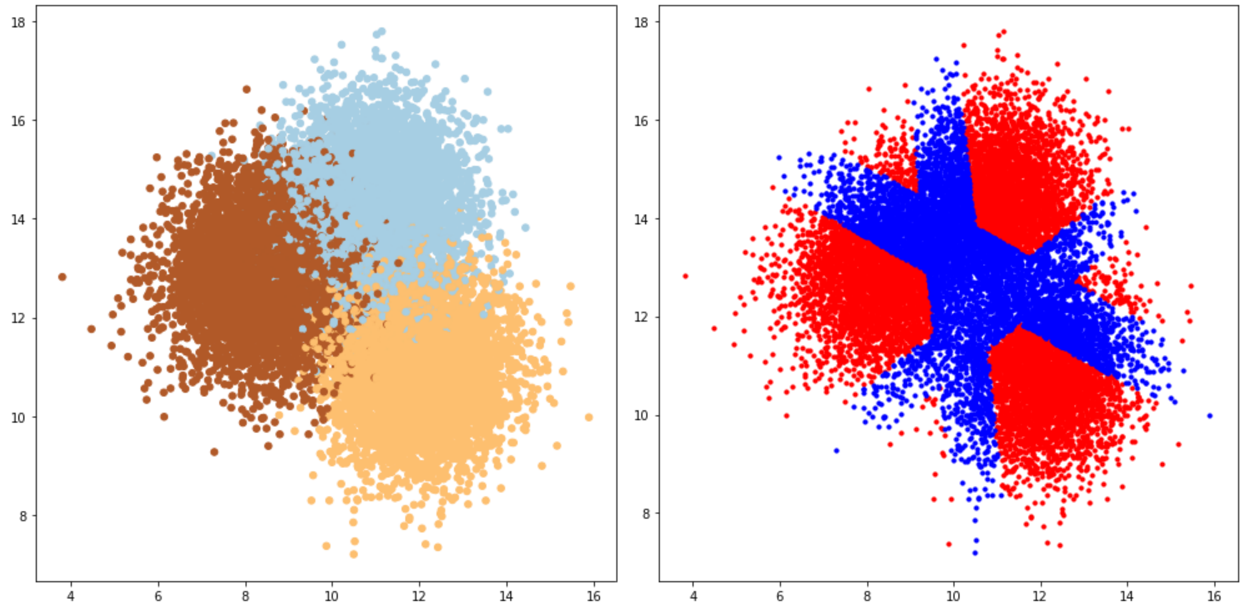

Figure 19 Sample space split, distance to hyper-plane split on synthetic data, 3 labels

As we can see from the illustration the split is much like the entropy based split, but less smooth and we get some sections with samples mixing in labels.

Let us see how it performs in the self-training method.

Sample size	Iterations	Threshold	Total samples added	Final acc. on test set
50 000	10	0.88	1548	**0.7118**

Table 73 Sample space support vector split results in self-training method

We got a **0.056% decrease** in performance while adding 1548 samples and this is so far the best result of the sample split methods. Let us look at the pseudo-label distribution.

Pseudo-l.		Pseudo-l.	
1	6	**17**	101
3	550	**19**	1
4	35	**20**	24
6	161	**23**	43
7	24	**25**	11
8	167	**26**	3
9	5	**93**	6
10	165		
12	180		
13	1		
14	6		
15	13		
16	46		

Table 74 Sample space support vector split results pseudo-label distribution

The pseudo-label distribution is also good, showing a good spread of labels. We do not get a spike in one of the pseudo-labels as in the entropy split did with pseudo-label 20.

Here we also needed to lower threshold because of the average probabilities in predictions for the model trained on samples above the threshold for distance to the hyper-plane were lower on average.

6.3.3. Conclusion

In conclusion for all our attempts at multiple models by doing both feature and sample splits we have not been able to gain any advantage in performance. But we have gained some insights to our problem by doing so. From earlier experiments we saw that lowering threshold of confidence in pseudo-labeled samples caused rapid deterioration in performance. And by keeping the confidence level high we only gained pseudo-labeled samples from the already majority labels only enforcing the negative effects of the imbalance in labels. But we found that by using the entropy and support vector hyper-plane distance splits we were able to amass pseudo-labels from a broader distribution without a major deterioration in performance.

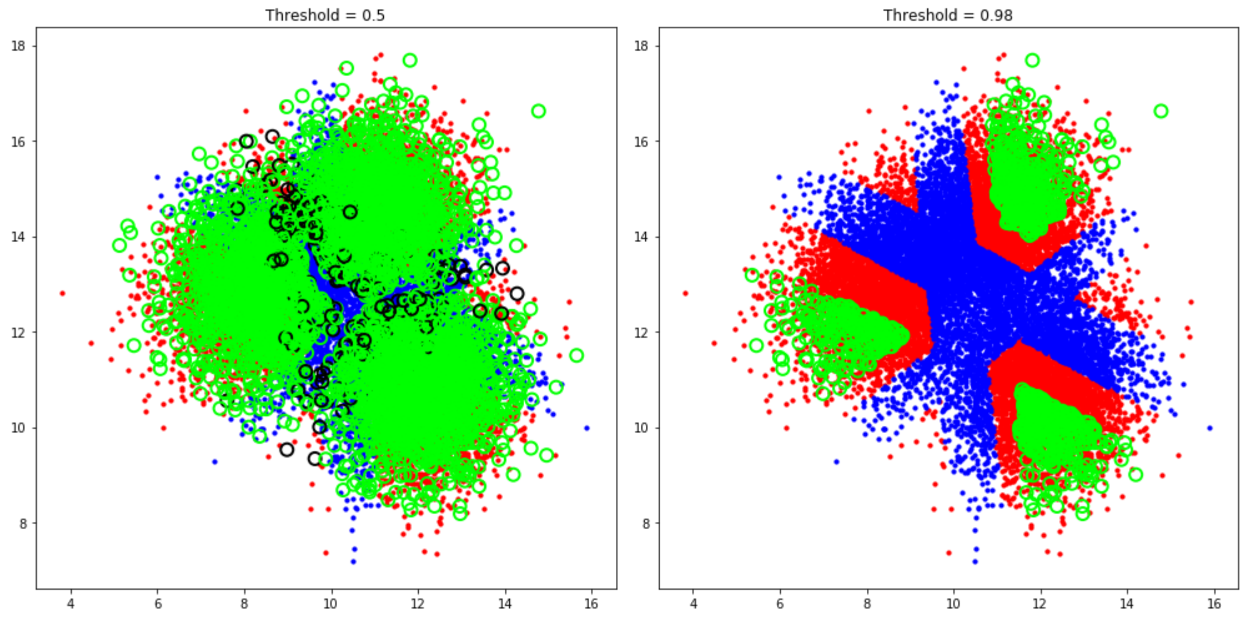

Figure 20 Sample space split, distance to hyper-plane split on synthetic data, 3 labels. Here shown with correct predictions in green and wrong predictions in black. Illustrating confidence threshold levels effect on predictions distance from hyper-plane.

Our problem is in the dilemma of confidence vs distribution and we also need to consider that in order to improve our model with new samples we will need to gain samples with new and valuable information. This would translate in our SVM models to samples closer to the hyper-plane to better define the decision border. But samples closer to the hyper-plane are also samples with low probability in prediction so by having a high threshold will remove them from our pseudo-labeled candidates.

We need to further investigate our definition of confidence threshold and also look into how to mitigate the negative effects of gaining badly predicted pseudo-labels if we attempt to move towards the hyper-plane for our pseudo-labeled samples.

6.4. Dynamic thresholding

As mentioned earlier we should look into how we define our threshold for confidence in probability of our predictions. One such way could be to look at each of the pseudo-label predictions maximum probability.

By doing predictions on the validation set we see that we have a variety of maximum values with a good correlation between number of samples in training data and maximum probability for predictions of that label. More training data give higher maximum probability. This is intuitive as there are more data to properly align the hyper-plane and also there is a higher distribution of the major labels in the corpus therefore more chances to get a higher probability sample in the validation set. This confirms the bias towards the major labels.

We can then make a dictionary of these maximum values for each label and instead of setting a flat threshold over all labels we set a ratio of the maximum value for each individual label. This way we siphon samples distributed over all labels mitigating our problem with only getting the major labels in our resulting pseudo-labeled candidates.

6.4.1. Tuning ratio

Of course this will also result in a greater risk of mislabeling so we will run a sequence of ratios from 0.5 to 1 in increments of 0.001. The threshold is then set per label with the formula; maximum probability multiplied with ratio. This is from predictions on the validation set.

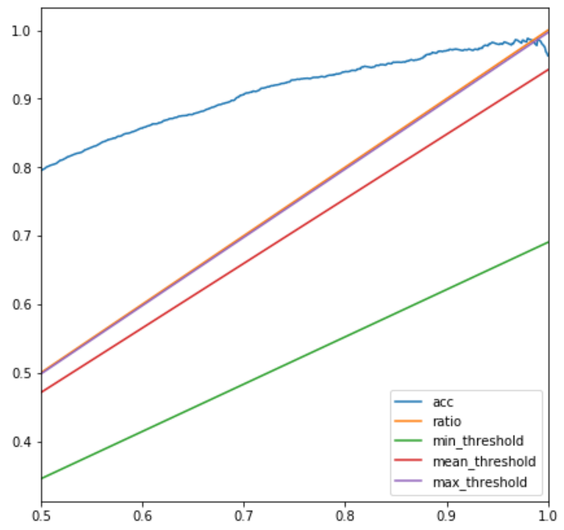

Figure 21 Dynamic thresholding results on validation set

Here in the graph on the left we see a plot of the ratio barely visible in yellow with the maximum threshold from all labels following tightly. We see the mean threshold over labels in red and the minimum threshold in green indicating the span over which the labels probability cover. In blue we see the resulting accuracy of the subset of predictions meeting their thresholds. The X-axis is the ratio.

As we can see the accuracy dips right before the end, with peak exactly at ratio=0.98.

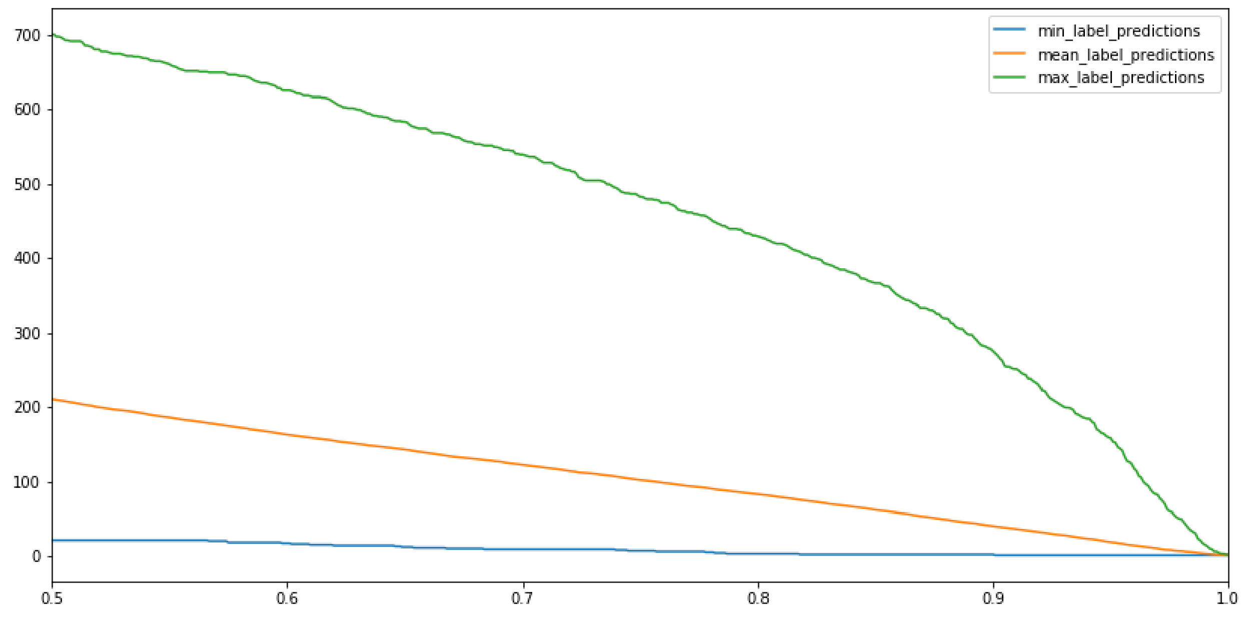

Figure 22 Dynamic thresholding results on validation set, predictions grouped by label

Above we see plots of the min, max and mean number of predictions per label with ratio as X-axis. From this graph we see that we still have a large difference in the number of samples per label, though decreasing rapidly as the ratio moves close to 1 resulting in only the samples with maximum probability in each label (mostly 1 per label).

Pseudo-l.		Pseudo-l.	
12	49	**26**	3
92	17	**17**	3
3	15	**13**	2
16	10	**2**	2
91	8	**21**	2
15	8	**24**	2
6	7	**7**	2
8	6	**93**	1
10	6	**5**	1
20	6	**18**	1
9	5	**19**	1
23	5	**1**	1
4	4		
14	5		

Table 75 Dynamic thresholding results on validation set with ratio=0.98 prediction subset pseudo-label distribution

Now we look at the results we get at ratio=0.98, which had a peak accuracy=0.9882 for the resulting subset of predictions on validation set we see that we get a total of 170 samples, and only 2 samples mislabeled from the labels 6 and 8.

We run dynamic thresholding on the test set with ratio=0.98 and we get an accuracy=0.9940 and a pseudo-label distribution almost identical to the validation set.

Before we try dynamic thresholding in a self-training method we will also implement a method for mitigating the imbalance we get in pseudo-labels. But now we have a method of getting a better distribution of pseudo-labels.

6.5. Ensemble, bagging

To mitigate the imbalance we get in pseudo-label distribution we will create an ensemble structure containing a number of models trained on separate combinations of the labeled training data and a stratified random subset of available pseudo-labeled data. This way we can both regulate the number of pseudo-labeled samples we add to the training data, but we also get good regularization by combining the predictions from several models mitigating the destabilizing effect of mislabeled samples.

6.5.1. Algorithm

We first initialize the pseudo-labeled pool by performing predictions and probabilities with our baseline model on the unlabeled data. We then run these predictions and probabilities through our dynamic thresholding method and get our pseudo-labeled candidates in return. We then add these candidates to the pseudo-labeled pool. Then we iterate the number of models we want to fit with training data combined with a random sample from the pool of pseudo-labeled data. We do this sample stratified by iterating over the labels in the training data and counting the number of samples per label, this number we multiply with a set sample rate and we get the number of samples we pick from the pool. If there are less samples then what the sample rate indicate, we choose to add all the samples from the pool.

We then have a number of models trained with different combinations of labeled and pseudo-labeled data. By using a sample rate we can regulate how much pseudo-labeled data we put in the mix and we can tune this parameter to get as stable models as possible.

When the models have been fit they perform predictions and probabilities on either more unlabeled data or the final test set. We combine all the probabilities from each model in a tensor and get the mean probabilities for each sample and this way we get the regularization of the predictions.

We have then some parameters to tune. The ratio parameter for the dynamic threshold, this controls how much pseudo-labeled data we get in the pool and to an extent how close the samples will get to the hyper-plane. The number of models we want to train in our ensemble, this controls how much regularization we get. The sample rate for controlling how much pseudo-labeled data we add proportional to the number of samples per label in the training data. We can also perform individual sample rates per label or for sets of labels if we want to counter the imbalance already in the training data by adding more pseudo-labeled samples. This could also be done for the dynamic thresholding, as we could set individual threshold levels per label or groups of labels.

6.5.2. Parameter tuning

We will run a sparse grid-search over the parameters for number of ensembles, the sample rate and the dynamic threshold ratio. For this grid search we use the validation set as we will not be using validation inside the loop at any point. We also only run this for 1 iteration, meaning we only add pseudo-labeled samples to the pool once before we run the ensemble on the validation set. We do predictions on all the unlabeled data.

Parameter	Values
Number of ensembles	[5, 10]
Sample rate	[0.5, 1, 2]
Dynamic threshold ratio	[0.65, 0.80, 0.98]

Table 76 Ensemble sparse grid-search parameter values

Grid-search results on validation set, showing top 5 and bottom 5 entries for reference.

Number of ensembles	Sample rate	Dynamic threshold ratio	Accuracy on validation set
5	0.5	0.65	**0.7078**
5	0.5	0.80	0.7077
10	0.5	0.65	0.7077
10	0.5	0.80	0.7072
10	1	0.65	0.7069
…	…	…	…
10	0.5	0.98	0.7029
10	1	0.98	0.7025
5	1	0.98	0.7014
10	2	0.98	0.6995
5	2	0.98	0.6990

Table 77 Ensemble sparse grid-search results

Here we can see that we clearly get a benefit from a lower dynamic threshold ratio. We are therefore able to learn from samples more close to the hyper-plane and mitigate the mis-labeling with the regularization. The top ranked gets a 0.568% increase in performance over baseline model on validation set, and we see that all the top 5 and in fact all the way to rank 12 have an improvement.

Pseudo-l.		Pseudo-l.	
12	69262	**13**	7412
16	36568	**14**	6871
3	35321	**5**	6833
20	28888	**1**	6806
10	26568	**7**	6760
15	25234	**24**	5471
17	17957	**26**	4955
6	15257	**93**	3455
8	12511	**2**	2644
4	12123	**91**	1755
19	9705	**92**	1169
25	9580	**18**	579
23	7905	**21**	505
9	7754		
SUM			369848

Table 78 Ensemble sparse grid-search top rank pseudo-label distribution

If we look at the pseudo-label distribution for the top ranked model with dynamic threshold ratio of 0.65 we see that we have plenty of pseudo-labeled samples to pick from and especially from the major labels. We could easily increase the dynamic threshold ratio for some of the top labels so that their pseudo-labeled samples had a slight higher confidence. This will result in samples being further from the hyper-plane. But if we assume that the major labels already have a reasonable amount of samples to properly align the hyper-plane doing this distinction in thresholding could be beneficial. We then take higher risks with the minority labels in an attempt at aligning the hyper-plane, while lowering the risk with the majority labels.

We will create such a progressive ratio by looking at the label distribution and manually create increments in the ratio by using conditional statements on the label. We create 3 conditional groups of labels with three increments.

Group of labels	Increment in ratio
12	0.15
15, 16, 20, 3	0.10
10, 17, 19	0.05

Table 79 Dynamic threshold label groups with incremented ratio

If we now run again with the settings of the top ranked model on validation set, we should get the thresholds for label 12 = 0.95, for label 15, 16, 20 and 3 = 0.85 and so on. When we run this configuration we get an accuracy of 0.7081 on the validation set, a slight improvement and a 0.611% increase from baseline on validation set. We will keep this configuration and run it on the test set.

Number of ensembles	Sample rate	Dynamic threshold ratio	Conditional ratio groups	Final accuracy on test set
5	0.5	0.65	Yes	**0.7121**

Table 80 Ensemble score on test set

Here we can see that we got pretty much the same result as our baseline, **0.014% decrease** in performance.

6.5.3. Conclusion

The results we got on both the validation and test sets are interesting, even though the performance on the test set did not improve. We got improvement on most configurations on the validation set and we pretty much maintained the level of performance on the test set. Also the better performing configurations had a low dynamic threshold ratio, giving us more data closer to the hyper-plane, but from previous experiments we have seen that lowering confidence threshold quickly deteriorates the model. This we managed to mitigate with regularization from the ensemble structure with random subsets of the pseudo-labeled samples added to each models training data, maintaining and on validation set increasing performance.

6.6. Clustering

Clustering is a unsupervised method of grouping together samples that have certain similarities. The goal of a clustering method is to create such groups so that all the elements in them are more similar to each other then elements in other groups. This is can be achieved with many different techniques, but we will focus on the K-Means algorithm. This algorithm has 1 parameter, the K, which decides how many clusters to form. To do this we create K cluster centers, called centroids, at random locations in the feature space, or we can pick K random samples locations as initial centroids. Then we start an iterative process of assigning samples to their nearest centroid based on a distance metric, when all samples are assigned we recalculate the centroid to be the mean of all its assigned samples. This process continues until there is no change in assignments to centroids (or less change then some threshold or a maximum number of iterations).

Because there is no defined way of initializing the centroids, they are random, there is also no optimal solution to the problem and it can converge at a local optimum. Because of this it is normal to re-run the algorithm several times with different initial centroids and then pick the one with the smallest sum of all squared distances from the centroids.

We will use an implementation of K-Means from the scikit-learn library called KMeans [32].

Also because K-Means is known for poor performance in high dimensionality we will use the word-embedding representations for clustering.

6.6.1. Cluster features

The way we will use clustering is to engineer additional features for us to see if we can improve classification with. We will cluster our samples several times using different values for K and save each cluster assignment as a feature. The intuition here is that there might be additional information in knowing what cluster a sample was in at different scales of clustering. We will sort of create a "path" from larger clusters down to smaller clusters.

In addition we will see if mixing in unlabeled data with the labeled data will improve the clustering and give better results.

We then have two parameters to tune, the K in K-means and number of different Ks, let us call that parameter N. We will tune these parameters first and then look at adding unlabeled data later.

6.6.1.1. Tuning K and N

We could do an exhaustive grid-search on all different K's and N's within some set range, but because that would be very time and resource consuming we will choose some values that we think might be suitable to what we want to achieve. We want to create clusters of different scale to create a "path" from big clusters down to small clusters. N controls the length of the path and K controls the scale of the clusters. To make it easy for us and also with a bit of hope that nature will be on our side we choose the Fibonacci sequence to give us our K's. We choose some N-length subsets of this sequence to be our parameter values. We represent K and N as a list of K's; [K_1, K_2, K_3, … , K_n]

Parameter	Values
K and N	[5, 8, 13]
	[5, 8, 13, 21]
	[8, 13, 21]
	[13, 21, 34]
	[21, 34, 55, 89]

Table 81 K and N parameter values

Since we are using a SVM model we will have to convert the cluster assignments to one-hot representations for us to be able to use them as features in our model.

We will run the parameters on the validation set and add the features to our baseline model to see the resulting accuracy. Below is the results ranked by accuracy.

K and N	Accuracy on validation set
[5, 8, 13]	**0.7053**
[13, 21, 34]	0.7049
[5, 8, 13, 21]	0.7033
[8, 13, 21]	0.7022
[21, 34, 55, 89]	0.6967

Table 82 Cluster features K and N tuning results

Here we see that we get a positive result with improvement in the accuracy for the two top ranked results on the validation set. Let us see if the best ranked parameter also performs well on the test set.

K and N	Accuracy on test set
[5, 8, 13]	**0.7129**

Table 83 Cluster features best K and N on test set result

As we can see we also got an improvement on the test set with **0.098% increase** in performance.

Now we will see if we introduce unlabeled samples together with the labeled if it could improve clustering and give us better features. We will try with different multiples of the size of the train set to see if there is an optimal amount of unlabeled vs labeled data. At the end we will use all the unlabeled data. We will use the K and N from the best ranked of previous results. 'all' in this setting means we add all the unlabeled data to the train set while performing clustering.

Parameter	Values
Multiples of train set size	[0.5, 1, 1.5, 2, 5, 10, all]

Table 84 Unlabeled mix with labeled data cluster features parameter values

Results on validation set ranked by accuracy.

Multiple	Accuracy on validation set
10	**0.7041**
All	0.7041
5	0.7040
1	0.7032
1.5	0.7032
2	0.7032
0.5	0.7016

Table 85 Unlabeled mix with labeled data cluster features results

We see no improvement, but we see that the more unlabeled data we add the less the damage is. We will not try this with the test set as we saw no improvement.

6.7. Other experiments

We also tried some other techniques that had less success then the ones described earlier in this writing. We will shortly describe some of them here.

6.7.1. Directed Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is a popular topic modelling algorithm [33]. It is used for discovering latent topics in the data at hand. It could be considered a way of soft-clustering both documents and words into a defined number of clusters/topics so that the words that appear in similar documents are bundled together with probabilities of appearing in each cluster/topic. It could also be considered as dimensionality reduction where each document is reduced to the probabilities of appearing in the defined number of topics. A document could be interpreted as being 5% politics, 4% finance and 91% sports for instance.

When LDA is applied on its own it is a un-supervised algorithm where you define the number of topics and then LDA will discover these topics on its own. This is useful if you want to achieve some inference about what a corpus themes are.

In practice LDA is an algorithm performed on two matrices, a word-by-topic matrix and a document-by-topic matrix. These matrices describe the probabilities for a word to belong to a topic and a document to belong to a topic. Usually these matrices are instantiated with random variables and through iterations the latent distributions are discovered.

We can exploit the word-by-topic matrix initialization by not instantiating it randomly, but with defined values.

We will be using the LDA module from the scikit-learn library [34], but we will create a super-class around it and modify its behavior when initializing the latent word-by-topic matrix. We will add the possibility to define this matrix manually.

After enabling this functionality we will use a scoring function to score the words in our vocabulary by label. To do this we will iterate over each label in a one-vs-rest fashion where we create a binary label and score our words using chi-squared. After iterating over each label in this fashion we have our word-by-topic matrix that we can use to instantiate the LDA. We will of course also define our topics (or components as the hyper-parameter is called in the module) to be the same number as the number of labels.

We can also mix in unlabeled samples and leave their unique words at a set default score.

Now we let the LDA run its course and we can transform our validation or test set to get the probabilities for a sample to belong to a certain topic. By remembering what topics correspond to what label when we instantiated we can classify either the sample by taking the topic/label with the highest probability or we can train a separate classifier on these probabilities using the LDA as a dimensionality reducer.

Neither of these techniques proved beneficial to our problem as we were not able to get any performance over 50% accuracy.

6.7.2. Date distribution

As mentioned earlier in the thesis we should look at how the articles change over time. We did this in two different ways. We first split the corpus in half on data and trained separate models for each split, and we tried splitting in three with three separate models achieving nothing but less accuracy probably due to less training data.

We also tried using the date as a feature directly with no luck. And we tried binning the dates in 10 bins and use that as a feature with no gains in performance.

I think we can conclude that date is not discriminatory in this corpus.

7. Conclusion

In this work we have presented several techniques for improving classification, both supervised and semi-supervised. We will try and compare how well they have succeeded on our dataset. Let us first review some of the results in the table below.

Method	Model	Pseudo-labeled added	Accuracy on test set	Relative to baseline
Supervised methods				
Vectorization & classification	TfidfVectorizer + LinearSVC		0.6759	*Initial baseline
Feature selection	chi2, 40 000		0.6704	0.325% decrease
Sampling	TomekLinks		0.6770	**0.163% increase**
Word-embedding	model 80, lemmatized, TF-IDF weighted, min-max scaled		0.7039	**4.143% increase**
Calibration	CalibratedClassifierCV, isotonic		0.6800	**0.607% increase**
Combined supervised model			0.7066	**4.542% increase** *Supervised baseline
Semi-supervised methods				
Feature extraction	TF-IDF weights for word embeddings		0.7122	**1.007% increase** *baseline
Self-training, single model	All unlabeled, 5 iterations, 0.99 confidence	6 116	0.6951	2.401% decrease
Self-training, single model	50 000 samples from pool, 5 iterations, 0.99 confidence	786	0.7125	**0.042% increase**
Self-training, single model	50 000 samples from pool with validation, 5 iterations, 0.99 confidence	178	0.7122	0% difference
Self-training, single model	250 000 samples from pool with validation, quick model, 150 iterations, 0.98 confidence	37	0.7124	**0.028% increase**
Self-training, single model	30 samples from pool with validation, quick model, 300 iterations, 0.00 confidence	450	0.7118	0.056% decrease

Self-training, single model	50 000 samples from pool, 5 iterations, 0.99 confidence, trash filter	948	0.7133	**0.154% increase**
Self-training, multiple models, feature split	50 000 samples from pool, 10 iterations, confidence tuning with validation set	20 838	0.6800	4.521% decrease
Self-training, multiple models, feature split	50 000 samples from pool, 10 iterations, 0.98 confidence	1273	0.7002	1.685% decrease
Self-training, multiple models, sample split, random	50 000 samples from pool, 10 iterations, 0.98 confidence	1675	0.7077	0.632% decrease
Self-training, multiple models, sample split, entropy	50 000 samples from pool, 10 iterations, 0.55 confidence	3541	0.7079	0.604% decrease
Self-training, multiple models, sample split, support vector	50 000 samples from pool, 10 iterations, 0.88 confidence	1548	0.7118	0.056% decrease
Self-training, ensemble	Dynamic threshold ratio 0.65	Sample rate=0.5	0.7121	0.014% decrease
Cluster feature engineering		0	0.7129	**0.098% increase**

Table 86 Selection of results

As we can see among the supervised methods (or rather non-semi-supervised) word-embedding representation in combination with BoW clearly gave the best performance increase.

Among the semi-supervised methods tried in this work the TF-IDF feature extraction for weighting the word-embedding vectors gave the best improvement from the combined supervised model.

It has proven difficult to gain any extra learning from the unlabeled samples in this dataset using the basic wrapper self-training approach, but we did achieve increases in performance with some of the methods.

We should also consider that increments in accuracy would get progressively more difficult to gain. As we are adding our semi-supervised methods on top of the supervised baseline.

Our best results with self-training were the single model with a flat confidence threshold, but this model would only append pseudo-labels from the majority labels and would in the long run only enforce the imbalance problem further.

Our ensemble approach did not succeed in improving accuracy on the test set, but it did for many configurations on the validation set. This could indicate that there is less accuracy to be gained on this particular split of test data. The validation accuracy is approximately 1% less then on the test set giving more room for cheaper gains in accuracy here.

We were able to counter the challenges we had with distribution in confident pseudo-labels and also found an effective way of mitigating the destabilizing effect of mislabeled samples.

If we compare the results of our semi-supervised techniques with the common supervised sampling techniques we see that our semi-supervised methods are at least comparable if not better.

A weakness in our strategy could be that even though we did not use the test set for parameter-tuning at most stages, we did use it to draw conclusions about individual components of the final models, so our results should be considered somewhat optimistic.

Ansver Q1: Yes, we are able to create an algorithm that is able to extract additional learning from the unlabeled data.

Ansver Q2: Yes, we are able to make use of unsupervised pre-processing techniques

Ansver Q3: Applying these techniques can improve accuracy.

We can conclude that it is possible to gain extra learning using the unlabeled data and that it is also possible to mitigate the imbalance problem, but further work needs to be done to find the correct balance between confidence, distribution and getting samples that are closer to the decision border.

Files used in this thesis can be found at this repository: https://github.com/nmfoss/master_public

8. Future work

We think that the most interesting direction for future work would be in exploring the multiple models approach with focus on how to create better diversity in the trainers to counter the dilemma of confidence vs. distance to decision border.

We also think that the ensemble and dynamic threshold approach should be further explored to find the optimal balance of regularization and sample rate. As well as different techniques for filling the pseudo-labeled pool.

9. References

[1] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[2] TfidfVectorizer, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

[3] Language Technology Group (LTG), web-page
https://www.mn.uio.no/ifi/english/research/groups/ltg/

[4] NLPL word embeddings repository, web-page
http://vectors.nlpl.eu/repository/

[5] A. Joulin, E. Grave, P. Bojanowski, T. Mikolov, Bag of Tricks for Efficient Text Classification

[6] CountVectorizer, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

[7] Frekvensordliste: de vanligste norske ord, web-page
https://www.korrekturavdelingen.no/ord-uttrykk-frekvensordliste-500-vanligste-norsk.htm

[8] LinearSVC, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

[9] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification Journal of Machine Learning Research 9 (2008), 1871-1874.

[10] SGDClassifier, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html

[11] ComplementNB, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.ComplementNB.html

[12] MultinomialNB, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html

[13] SelectKBest, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html

[14] Lemaitre, G., Nogueira, F., & Aridas, C. K. (2017). Imbalanced-learn: A python toolboxto tackle the curse of imbalanced datasets in machine learning. Journal of Machine Learning Research, 18, 1–5.

[15] ClusterCentroids, imbalanced-learn module manual, web-page
https://imbalanced-learn.readthedocs.io/en/stable/generated/imblearn.under_sampling.ClusterCentroids.html

[16] RandomUnderSampler, imbalanced-learn module manual, web-page
https://imbalanced-learn.readthedocs.io/en/stable/generated/imblearn.under_sampling.RandomUnderSampler.html

[17] TomekLinks, imbalanced-learn module manual, web-page
https://imbalanced-learn.readthedocs.io/en/stable/generated/imblearn.under_sampling.TomekLinks.html

[18] ADASYN, imbalanced-learn module manual, web-page
https://imbalanced-learn.readthedocs.io/en/stable/generated/imblearn.over_sampling.ADASYN.html

[19] SMOTE, imbalanced-learn module manual, web-page
https://imbalanced-learn.readthedocs.io/en/stable/generated/imblearn.over_sampling.SMOTE.html

[20] RandomOverSampler, imabalanced-learn module manual, web-page
https://imbalanced-learn.readthedocs.io/en/stable/generated/imblearn.over_sampling.RandomOverSampler.html

[21] MinMaxScaler, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

[22] CalibratedClassifierCV, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html

[23] van Engelen, J.E., Hoos, H.H. A survey on semi-supervised learning. Mach Learn 109, 373–440 (2020)

[24] Yarowsky, D.(1995).Unsupervised word sense disambiguation rivaling supervised methods.In Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics.

[25] Zhu, X. (2008). Semi-supervised learning literature survey. Technical Report. 1530, University of Wisconsin Madison

[26] Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. Journal of the Royal statistical society, Series B, 39, 1–38

[27] Wu, Z., Wu, J., Cao, J., & Tao, D. (2012b). Hysad: A semi-supervised hybrid shilling attack detector for trustworthy product recommendation. In Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining (pp. 985–993). ACM

[28] Blum, A., & Mitchell, T. (1998). Combining labeled and unlabeled data with co-training. In Proceedings of the 11th annual conference on computational learning theory (pp. 92–100). ACM

[29] Yaslan, Y., & Cataltepe, Z. (2010). Co-training with relevant random subspaces. Neurocomputing, 73(10), 1652–1661

[30] Breiman, L. Random Forests. Machine Learning 45, 5–32 (2001)

[31] Predicting Good Probabilities with Supervised Learning, A. Niculescu-Mizil & R. Caruana, ICML 2005

[32] KMeans, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

[33] Zhai, C.H. and Massung, S. (2016), Text data management and analysis: a practical introduction to information retrieval and text mining, Association for Computing Machinery, ACM books, 329-388

[34] LatentDirichletAllocation, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.LatentDirichletAllocation.html

[35] train_test_split, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

[36] LogisticRegression, scikit-learn module manual, web-page
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[37] Fossåen, (2019). Classifying Nordic language news articles into hierarchical topics, DAT620 project report, UiS