Table of Contents

DAT620 Project Report

Classifying nordic language news articles into hierarchical topics

Nils Magne Fossåen

University of Stavanger, Norway

nm.fossaen@stud.uis.no

Abstract

We will explore several techniques for overcoming class imbalance, noisy data and overlapping class domains. Our goal is to classify norwegian language news articles into a large scope of topics. We will try several classifiers and also experiment trying to make some ensemble classifiers. The task has shown to be difficult and the main goal of achieving an accuracy of 67% was not met. Our best classifier achieved 51% accuracy, using both discrete and embedded word vectors combined.

Introduction

Classifying nordic (norwegian bokmål and nynorsk) language news articles is an extra challenging text classification task as most frameworks, pretrained models and literature is focusing on english language. Another issue is that the articles are a mix of the two forms of written norwegian; bokmål and nynorsk so there is not a uniform language representation.

The corpus is a collection of 37 323 articles from two major newspapers in Norway; Aftenposten and VG. It has been manually labeled using an adjusted version of the Comparative Agendas Project (CAP) labeling scheme consisting of 45 hiarchical classes. After class aggregation and removal of unwanted "junk" classes and also removal of nynorsk articles we are left with 24 classes and 29 587 articles.

Classes range in topics such as economic, health, crime, foreign affairs, politics, culture and sports amongst others.

The corpus is also very unbalanced ranging from one dominant class of about 4000 articles to 5 rather even classes with approx. 2000-2500 articles to a long tail of classes averaging around 500 articles, but some as low as below 100.

The main focus of this project has been on the modeling part and not so much on the preprocessing. I was delivered a lemmatized corpus already preprocessed by Erik De Vries. The corpus has also been modeled before with a Naive Bayes classifier with an overall accuracy of 67-68%.

My goal has been to acheive this accuracy, and exceed it if possible. This I have not been able to do with any of the tested modeling techniques so far. My results have been very stable around 48-52% accuracy.

This report will document what techniques I have used when modeling at different stages such as different vectorising techniques, feature engineering, feature selection, scaling of features, classifiers and different up- and down-sampling techniques. I have also tried to transform the corpus to a embedded space using FastText word vectors and applying similar models there.

One novel modeling technique has also been tried out based on a lecture by Anshumali Shrivastava. [1]

Technologies I have used in this project is Python, Pandas Dataframes and mainly Sci-Kit Learns [2] machine learning libraries.

The corpus (Background)

The labeled corpus is part of a greater dataset consiting of a total 1 561 443 articles from two major newspapers in Norway; Aftenposten and VG. Their total contribution to the dataset is 53.4% and 46.6%.

The labeled corpus is of 37 323 articles where 60.5% from Aftenposten and 39.5% from VG. These articles are divided into 45 classes following the CAP 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

To simplify things and make classification easier we aggregate subclasses to parents and we also get rid of the "junk" (not in the scope of our interest) classes 91, 92 and 93. We are then left with 24 classes in total.

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
24	Sports
25	Natural disasters, fires, preparedness
26	Religion and churches

Table 2: Aggregated and filtered classes

The nynorsk/bokmål distribution over the classes is described in the following figure by indicating how many nynorsk articles there is per class.

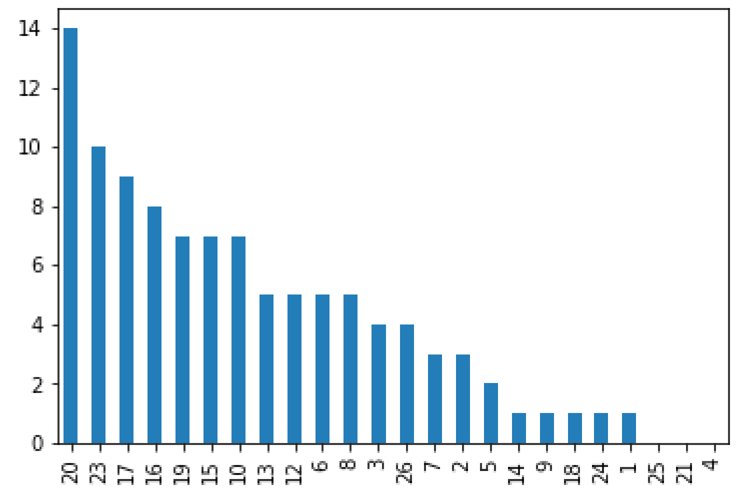

Illustration 1: Nynorsk articles per class

As we can see there is not a lot of nynorsk articles and considering how they would complicate the classification further we will consider removing them from the corpus.

As we can see from the class distribution we have a very unbalanced dataset.

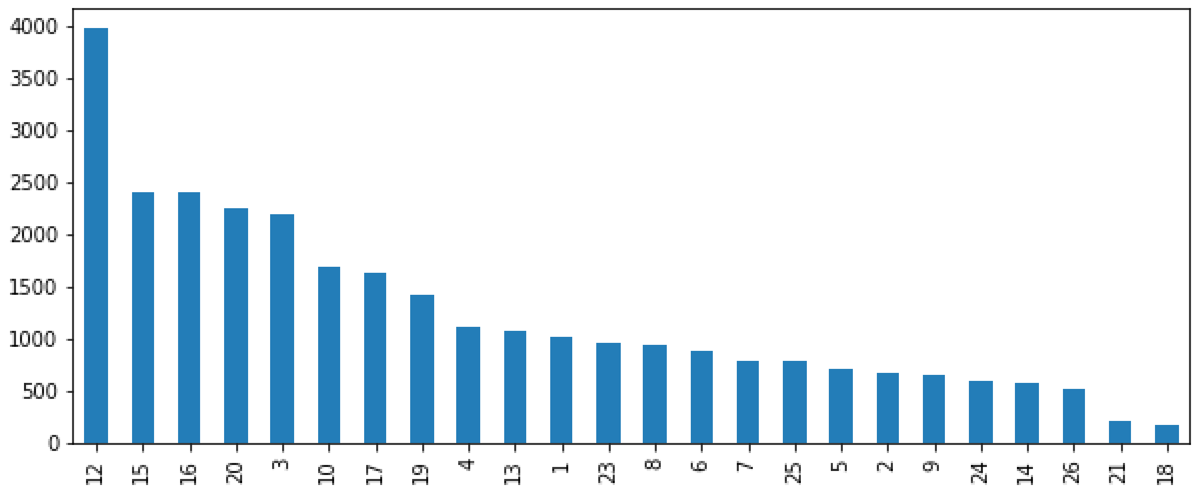

Illustration 2: Class distribution

This will give us some challenges and opportunities to try different techniques for dealing with this problem.

Further we look at the distribution of document lengths to see if there is anything to be aware of here.

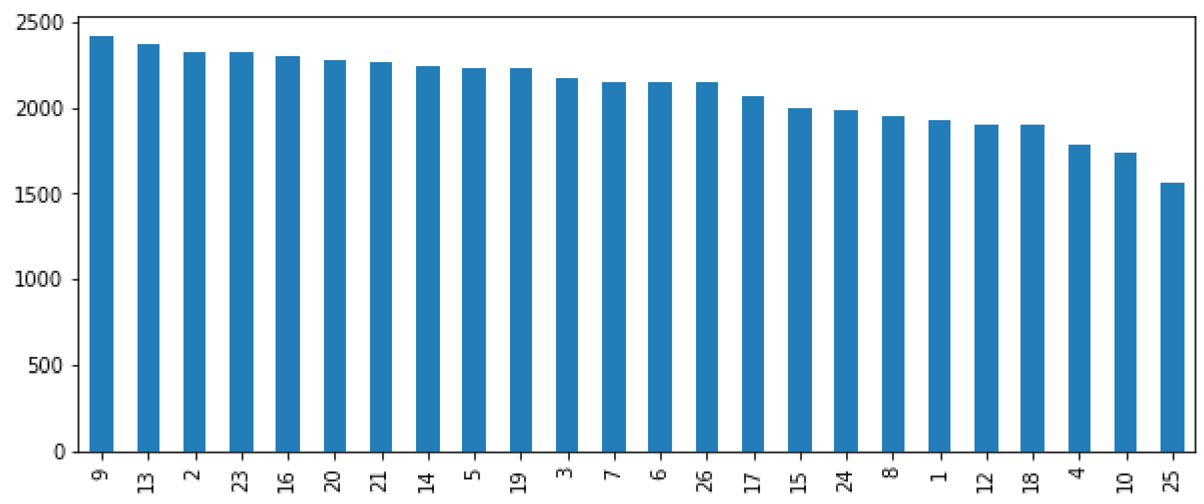

Illustration 3: Mean document length

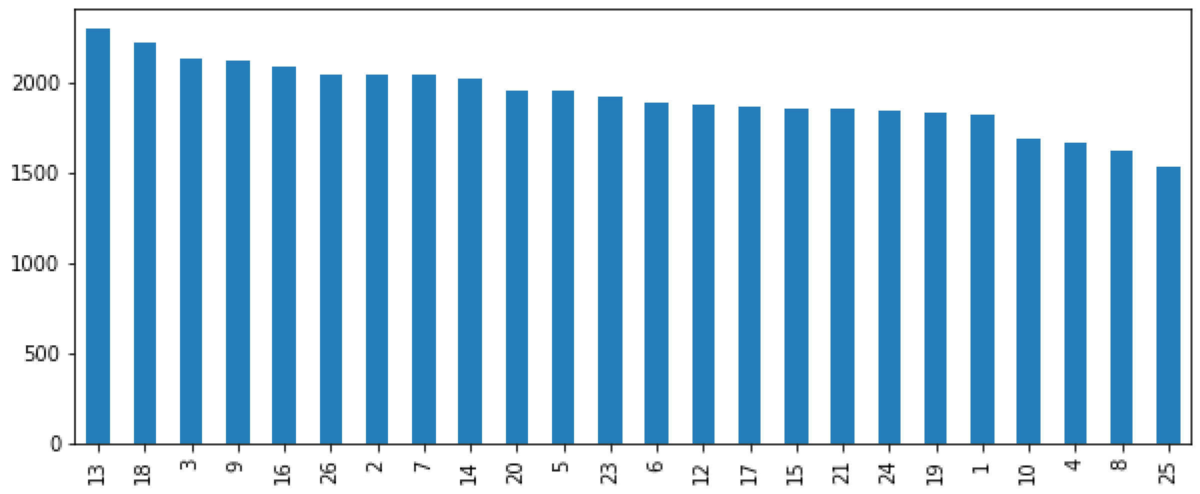

Illustration 4: Document length STD

As we can see there is a rather even distribution of the document lengths and as we can see from the standard deviations the articles vary greatly in size from very short articles to longer within each class. So there is little indication of any discriminatory effect from looking at document lengths.

By doing POS tagging we can also get a metric for our dataset distribution, but here there is also very little sign of any discriminatory effect.

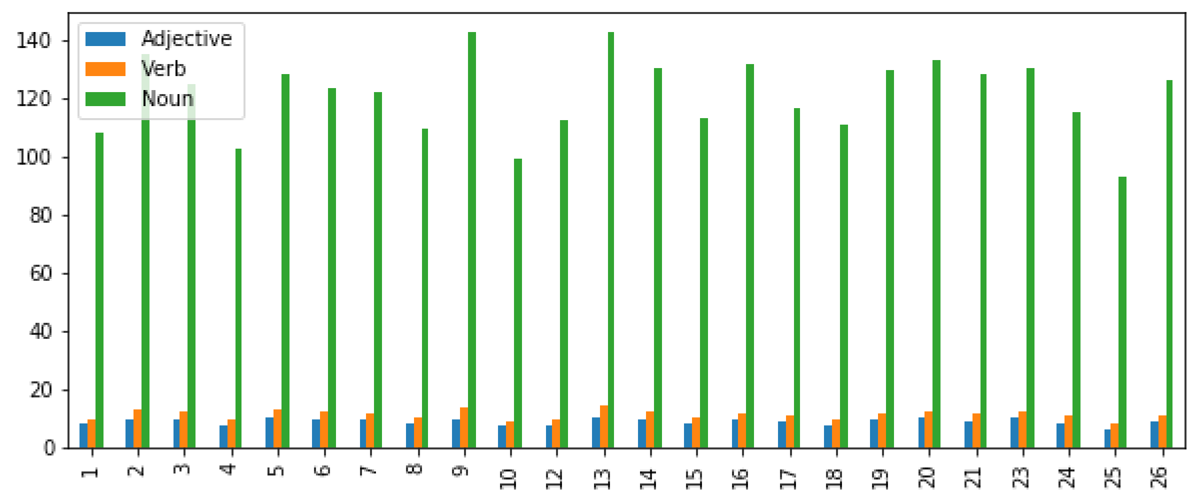

Illustration 5: POS tags means distribution

Word ontologies is also a way to get metrics about our dataset, but by looking at the distribution of some of the more frequent ontologies it gives little hope of any discriminating effect.

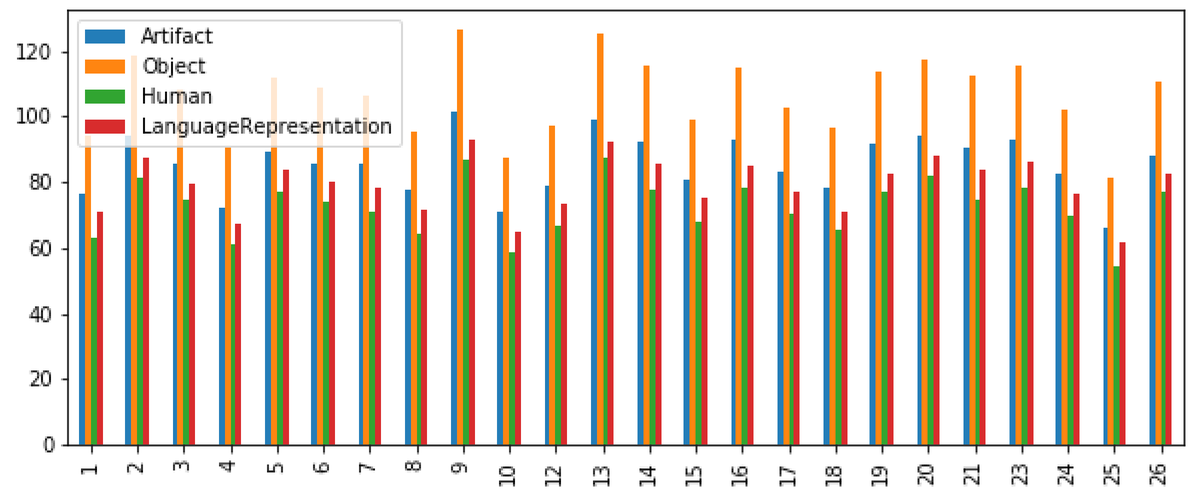

Illustration 6: Most frequent ontologies tags means

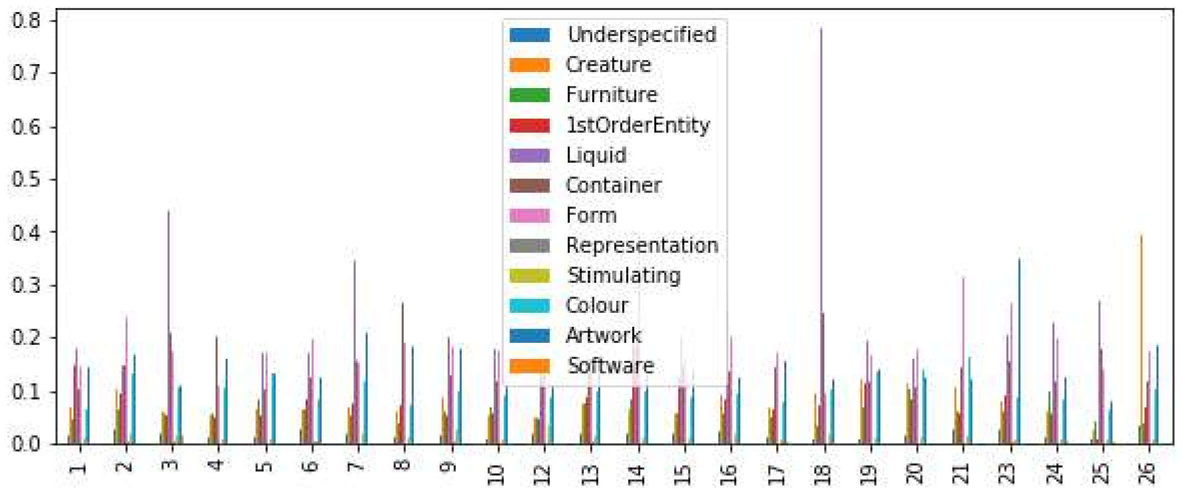

Illustration 7: Most infrequent ontologies tag means

To further illustrate our dataset we can transform our articles to word embedded space and through PCA get a 2D-scatterplot illustrating the distribution. This is perfomed on the mean vectors of each article.

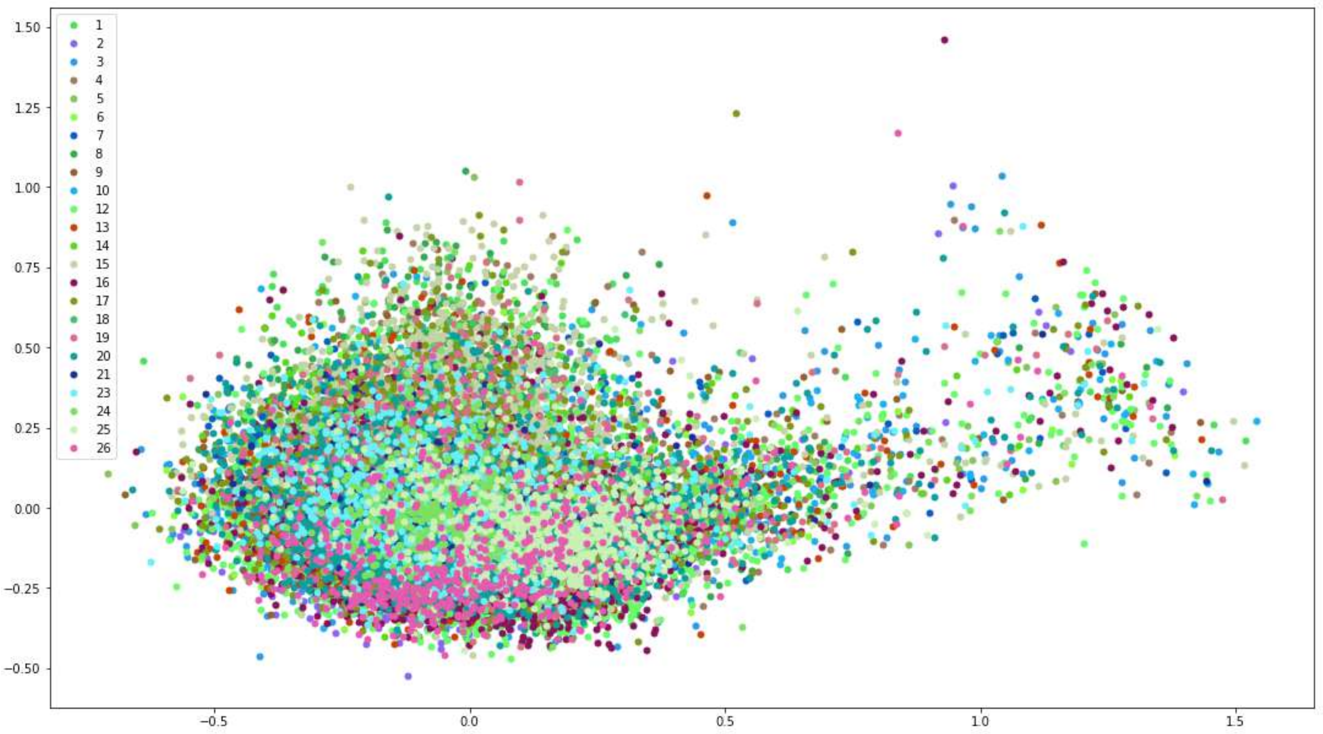

Illustration 8: FastText document distribution

As we can see there is little indication of any discriminating grouping from this plot. Though we see some good indication of intraclass grouping the interclass grouping is not so promising as classes are grouped mostly ontop of eachother.

Let us take a look at the mean and median over each class.

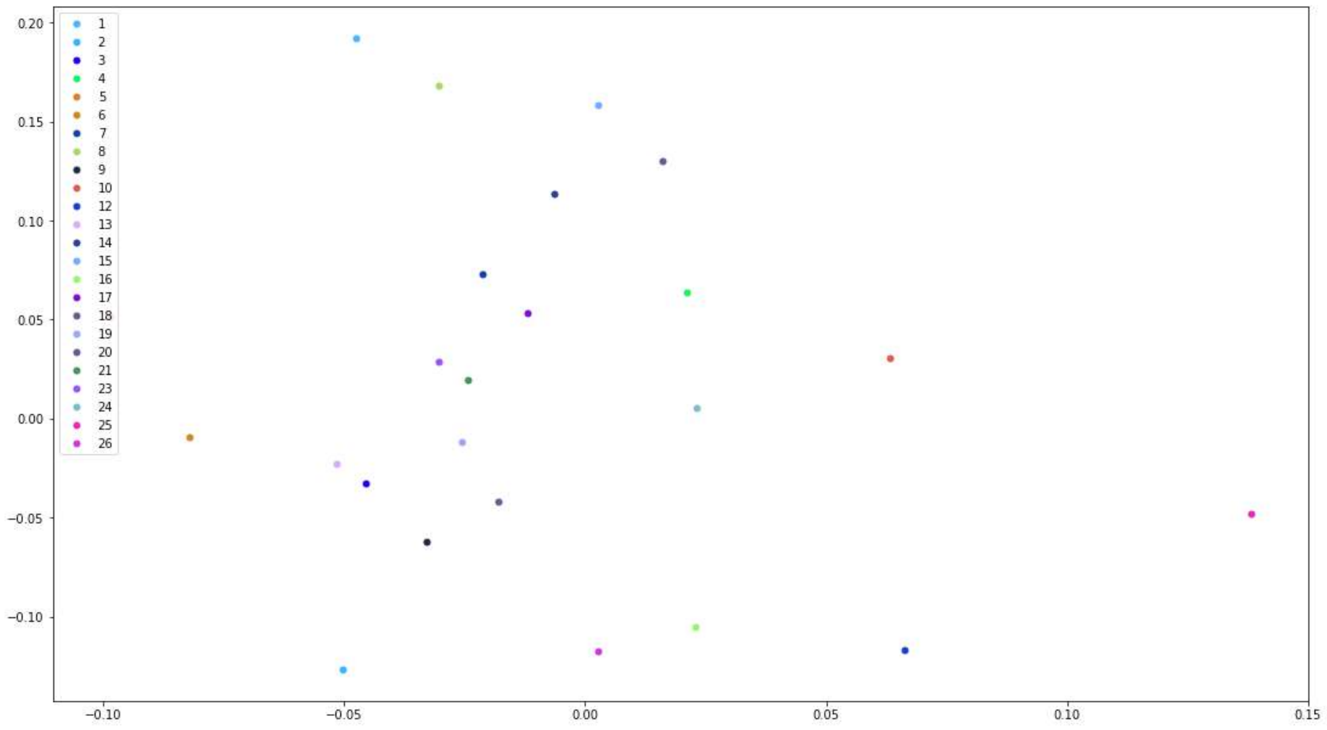

Illustration 9: FastText class mean distribution

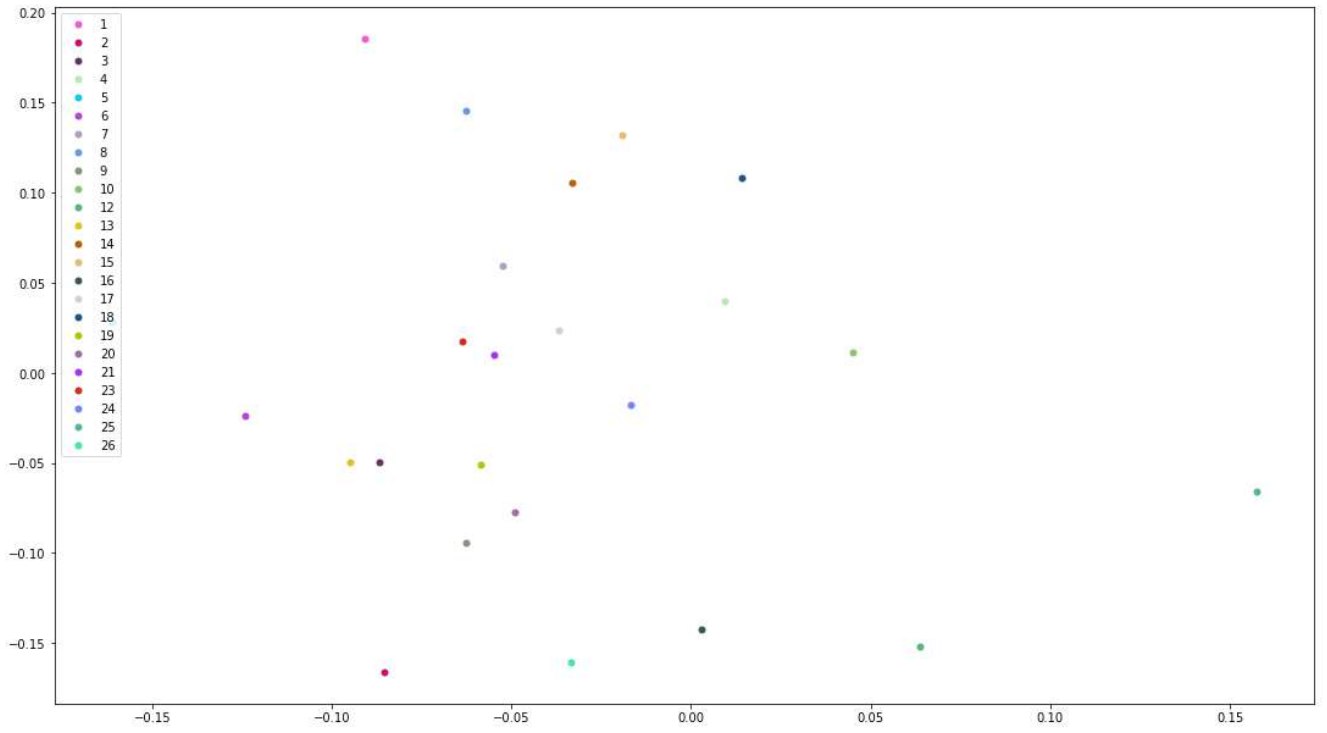

Illustration 10: FastText class median distribution

As we can see from the axis there is very little spread, but we do get some "outliers" with some good indication of separation between classes. The problem is the clustered majority in the middle. This is what one could expect though from any text classification problem as majority of language would be centered around a shared vocabulary.

A couple more plots describing the corpus distribution.

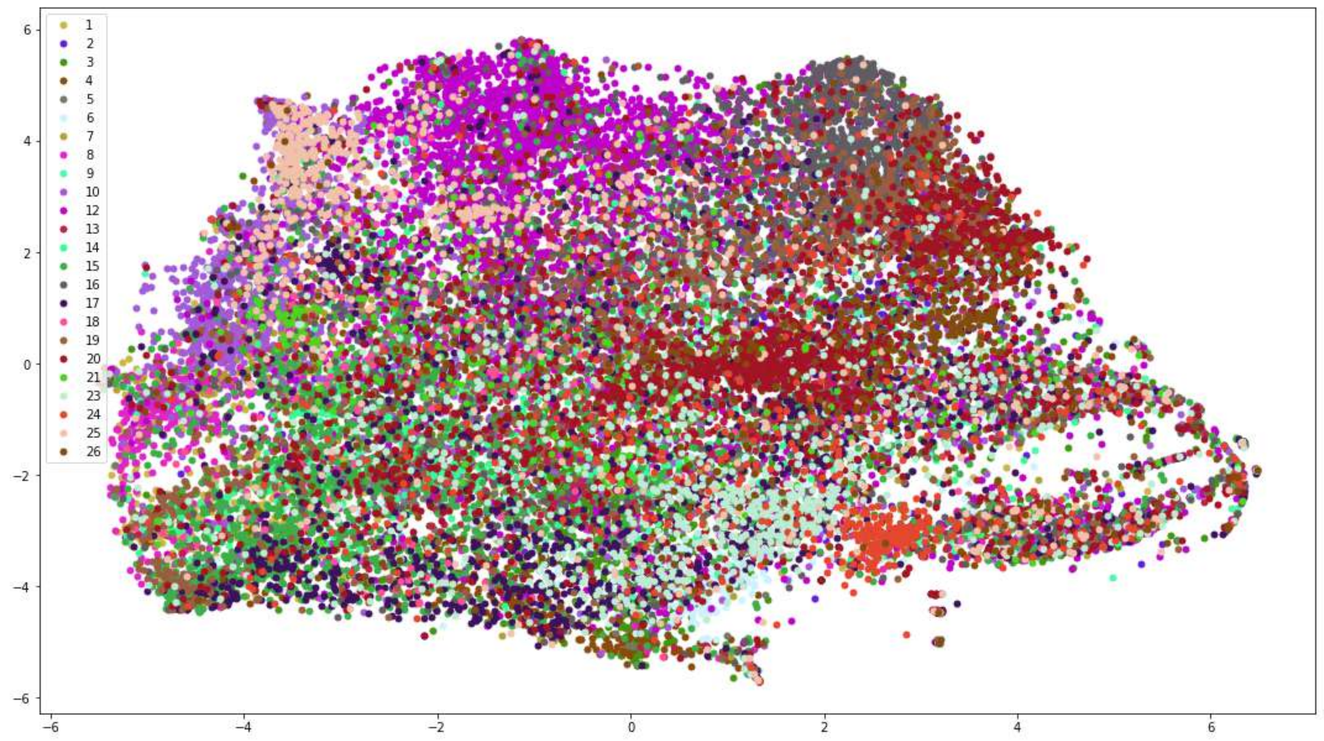

Illustration 11: FastText document mean TSNE

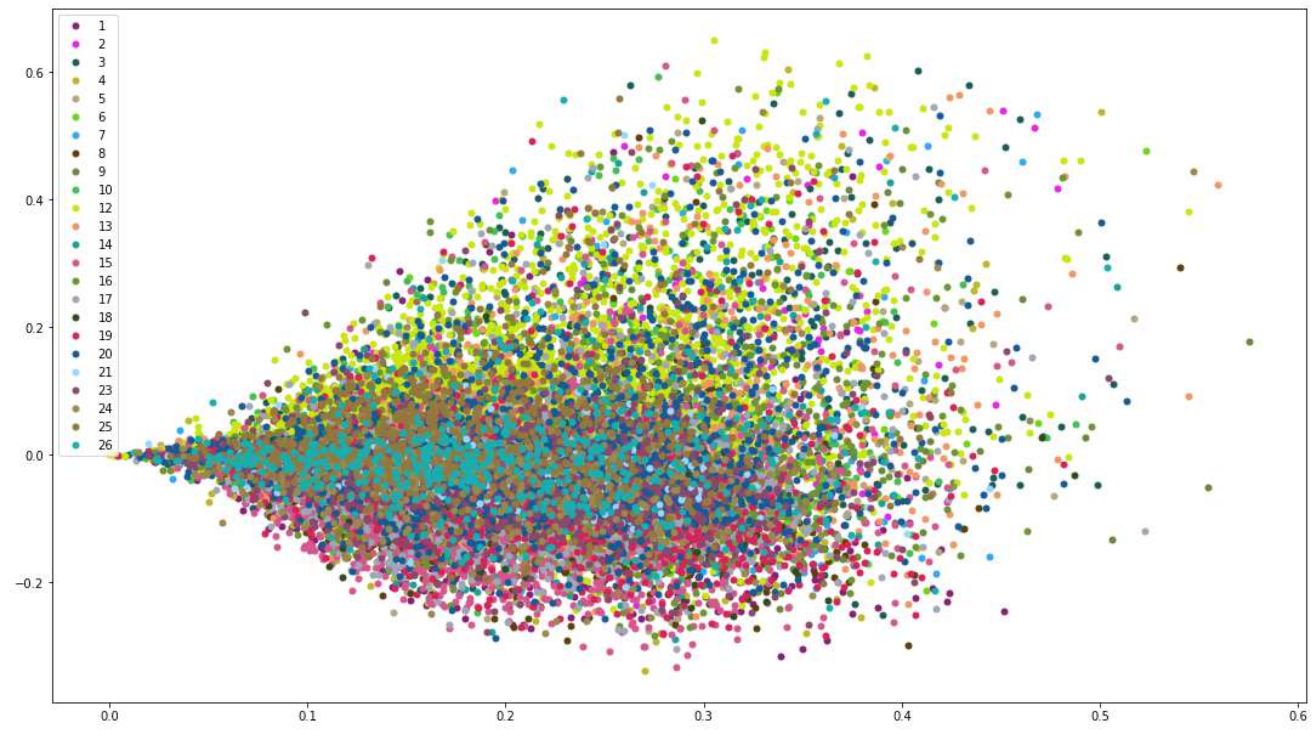

Illustration 12: TFIDF TruncatedSVD document distribution

As illustrated above we can safely say that we have a very difficult task ahead. We are battling both low discrimination between classes and a very unbalanced dataset with rather few training datapoints for the majority of classes.

Experiments

In this section I will give an account for how I have implemented and strategised in solving the classification problem.

Train- test splitting

To be able to measure and compare models we have to do some sort of train and test splitting of the training data. I have choose to perform just a simple holdout strategy where I have split the training data in a train and validation split, leaving 33% of the data for the validation. This is not as robust as doing cross-validation, but it is easier and faster to implement and it gives a good indication of how well a model performs.

For the splitting I have used scikit-learns train_test_split module with the following parameters:

Hyper-params	**Values**
test_size	0.33
random_state	1
stratify	target class (label)

Table 3: Train_test_split hyper-params

Corpus

As mentioned in the previous section we have an issue with dual language representations in norwegian; nynorsk and bokmål. Since the majority of the corpus is in bokmål we will run classification with and without nynorsk articles to see if removing them will improve performance.

To do this we create a simple "nynorsk-bokmål" classifier first. This can be done by creating two corresponding sets of words that differ between nynorsk and bokmål, use some common words. Then we can iterate over the corpus by making each article a set of words, take the intersection with each of the nynorsk and bokmål sets and if nynorsk has a greater intersection of words then bokmål we classify the article to be nynorsk. If equal then keep it as bokmål, so that our classifier will be as conservative as possible.

We will use MultinomialNB (NaiveBayes) classifier with 'alpha' = 0.0015 and scikit-learn TfidfVectorizer with best performing hyper-params from section below.

Language representations	Accuracy
Bokmål+nynorsk	0.4767
Bokmål	**0.4869**

Table 4: Accuracy with different language representations

As we can see removing nynorsk articles improves score. In the following sections we will only use bokmål articles.

Vectorizers

I have tried three of scikit-learns vectorizers, CountVectorizer, TfidfVectorizer and HashingVectorizer. They all basically work in a similar matter of converting the corpus documents into token occurences represented by a scipy.sparse matrix.

The HashingVectorizer has a couple of advantages and disadvantages compared to the Count- and TfidfVectorizers. An advantage is that it is low on memory as it does not have to keep vocabulary in memory. Disadvantages is that there is no way to perform a inverse transform and there is no IDF weighting.

All the vectorizers have been tuned using scikit-learns MultinomialNB (NaiveBayes) classifier with 'alpha' = 0.0015 and only using text features (only text from corpus).

CountVectorizer

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
strip_accents	['ascii', 'unicode', None]
max_df	[round(0.1*x, 2) for x in range(3,11,2)] + [1]
max_features	[1000*x for x in range(5,50,20)] + [None]

Table 5: CountVectorizer hyper-parameter value ranges

Best performing hyper-parameters:

Hyper-param	Values
strip_accents	'ascii'
max_df	0.3
max_features	None

Table 6: CountVectorizer best hyper-params

Accuracy: **0.4782**

TfidfVectorizer

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
strip_accents	['ascii', 'unicode', None]
max_df	[round(0.1*x, 2) for x in range(3,11,2)] + [1]
max_features	[1000*x for x in range(5,50,20)] + [None]
norm	['l2', None]
use_idf	[True, False]

Table 7: TfidfVectorizer hyper-parameter value ranges

Best performing hyper-parameters:

Hyper-param	Values
strip_accents	'ascii'
max_df	0.3
max_features	None
norm	'l2'
use_idf	False

Table 8: TfidfVectorizer best hyper-params

Accuracy: **0.4869**

HashingVectorizer

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
strip_accents	['ascii', 'unicode', None]
n_features	[2**x for x in range(15,25,2)]
norm	['l2']
alternate_sign	[False]

Table 9: HashingVectorizer hyper-parameter value ranges

Best performing hyper-parameters:

Hyper-param	Values
strip_accents	'unicode'
n_features	131072

Table 10: HashingVectorizer best hyper-params

Accuracy: **0.4641**

Conclusion

As expected TfidfVectorizer performed best, but perhaps unexpected it was best without IDF weighting.

For further gridsearch on other modules we will use the best TfidfVectorizer settings.

Feature selectors

To make our classifier more robust and more discriminatory we can perform feature selection before we give it the feature vectors. This is dimensionality reducing and decrease the complexity of our model and should help improve accuracy for our model. To do this feature selection we need to use some metric to score our features in a way so that we can select the most discriminatory features.

For our model we choose to do univariate feature selection and we will use SelectKBest from scikit-learn package.

SelectKBest choose the k-best features according to some scoring method. We will try using Chi2 and f_classif (ANOVA F-value) as scoring methods.

We will tune on text only features with the best performing TfidfVectorizer and using scikit-learns MultinomialNB (NaiveBayes) classifier with 'alpha' = 0.0015.

SelectKBest

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
score_func	[chi2, f_classif]
k	[25000, 30000, 35000, 40000, 50000]

Table 11: SelectKBest hyper-param ranges

Best performing hyper-parameters:

Hyper-param	Values
score_func	f_classif
k	35000

Table 12: SelectKBest best hyper-params

Accuracy: **0.4887**

Conclusion

We see that by using feature selection we increased our accuracy by 0.0018 points or 0.37%. It is not much, but we might acheieve better score by finetuning.

In further experiments we will use SelectKBest with best parameters.

Samplers

To battle our imbalance we can try to compensate by doing different sampling strategies. The two main routes of sampling is either oversampling the minority classes or undersampling the majority classes. For our sampling we will use a selection of samplers from the imbalanced-learn package [3].

- ClusterCentroids
- RandomUnderSampler
- TomekLinks
- ADASYN
- SMOTE
- RandomOverSampler

I will explain further each samplers characteristics in the sections below. We perform feature selection before we apply sampling.

ClusterCentroids

This is a undersampling technique that also generates new synthetic samples. It generates centroids using KMeans clustering and use these centroids as the new samples. It can be used with different strategies, but we will focus on 'majority', 'not minority' and 'all'.

When using 'majority' strategy it will only resample the majority class and in effect undersample it.

When using 'not minority' it will resample all classes except the minority class.

And 'all' will resample all classes.

It is a rather simple and sound undersampling technique with only the sampling strategy as hyper-parameter to tune.

A downside is that it is very heavy and slow to use as it use a lot of resources when it performs the KMeans.

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
sampling_strategy	['majority', 'not minority', 'all']

Table 13: ClusterCentroids hyper-param ranges

Best performing hyper-parameter:

Hyper-param	Value
sampling_strategy	'majority'

Table 14: ClusterCentroids best hyper-param

Accuracy: **0.4229**

RandomUnderSampler

This the most basic undersampling technique. It performs undersampling by randomly picking samples from the different classes. Here there is two hyper-parameters to tune, 'sampling_strategy' and 'replacement'.

Sampling strategy 'majority' will only undersample the majority class.

Sampling strategy 'auto' is equal to 'not minority' wich means it will undersample all classes but the minority class.

Replacement true or false means what it means. If true a sample can be sampled several times. This might sound counterintuitive when we talk about undersampling, but it just means that we use the entire class subset as samplepool and then we sample from this until a desired number of samples is picked.

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
sampling_strategy	['majority', 'auto']
replacement	[True, False]

Table 15: RandomUnderSampler hyper-param ranges

Best performing hyper-parameter:

Hyper-param	Value
sampling_strategy	'majority'
replacement	FALSE

Table 16: RandomUnderSampler best hyper-param

Accuracy: **0.4347**

TomekLinks

A Tomek's link means that two samples of different classes are closest neighbours measured by some distance metric.

TomekLinks undersampling works by removing either one or both of these samples according to what sampling strategy is selected. With 'majority' as strategy the majority class sample will be removed. With 'auto' as strategy, equal to 'not minority', means that both will be removed unless one is in minority class.

TomekLinks have only one hyper-parameter to tune wich is sampling_strategy.

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
sampling_strategy	['majority', 'auto']

Table 17: TomekLinks hyper-param ranges

Best performing hyper-parameter:

Hyper-param	Value
sampling_strategy	'majority'

Table 18: TomekLinks best hyper-param

Accuracy: **0.4883**

ADASYN

Short for Adaptive Synthetic sampling method. This is a oversampling method. It use k-nearest neighbours to calculate a density distribution for a given point (sample) and based on this density distribution it generates new synthetic samples until a set imbalance threshold is met.

There are two hyper-parameters to tune; sampling_strategy and n_neighbors.

Sampling strategy is similar to above mentioned samplers.

n_neighbors is number of neighbour samples used for calculating the density used for the creation of new samples.

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
sampling_strategy	['minority', 'auto']
n_neighbors	[3,5,7,13]

Table 19: ADASYN hyper-param ranges

Best performing hyper-parameters:

Hyper-param	Values
sampling_strategy	'minority'
n_neighbors	7

Table 20: ADASYN best hyper-params

Accuracy: **0.4891**

SMOTE

Short for Synthetic Minority Oversampling Technique, similar to ADASYN, but differ in that it does not use the density function when creating synthetic samples, but instead a uniform distribution for minority samples. It also use k-nearest neighbours, but here to consider which samples to include in a sort of grid between the minority class samples and new synthetic samples are distributed on this grid until some imbalance threshold is met.

Hyper parameters are the same as for ADASYN.

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
sampling_strategy	['minority', 'auto']
k_neighbors	[3,5,7,13]

Table 21: SMOTE hyper-param ranges

Best performing hyper-parameters:

Hyper-param	Values
sampling_strategy	'minority'
k_neighbors	7

Table 22: SMOTE best hyper-params

Accuracy: **0.4891**

RandomOverSampler

This is the basic oversampling method, it creates new synthetic samples by randomly sampling with replacement from the minority classes.

It has one hyper-parameter; sampling_strategy, wich is explained in above samplers.

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
sampling_strategy	['minority', 'auto']

Table 23: RandomOverSampler hyper-param ranges

Best performing hyper-parameters:

Hyper-param	Value
sampling_strategy	'minority'

Table 24: RandomOverSampler best hyper-params

Accuracy: **0.4889**

Conclusion

Of the undersamplers only TomekLinks is able to keep almost the same accuracy as with no undersampling. It seems as undersampling is the worst sampling strategy and that oversampling is the way to go. Of the oversamplers ADASYN and SMOTE performed best with same accuracy and same hyper-parameters indicating how similar these two methods are. They gave an increase of accuracy by 0.0004 points or 0.00082%.

Perhaps the best "bang for buck" here was the RandomOverSampler, the most naive implementation of them only with a 0.0002 points difference in accuracy.

But these are very small numbers and perhaps not even significant to even call it improvements.

We will use SMOTE with best params for further experiments.

Classifiers

For this project our main classifier to benchmark with is the MultinomialNB (Naive Bayes) classifier from the scikit-learn package. This because the corpus has been classified before with an accuracy of 65% and therefore my initial goal has been to reach this accuracy. Also then my experiments will be comparable to those previous results.

We will also try some other classifiers to see if they might perform better, they are;

- LinearSVC
- ComplementNB
- SGDClassifier

The reasoning for why I choose these classifiers I refer to the plot at [4] where these classifiers show promising results on text classification.

More about each classifier in the sections below.

MultinomialNB

This variation of the Naive Bayes classifier assumes a multinomial distribution of features wich is well suited for our task where we have feature vectors consisting of word counts (or frequencies).

It is a easy and fast classifier to work with and there is only one hyper-parameter to tune, the alpha smoothing parameter (Laplace/Lidstone).

I have used the best performing settings from the previous experiments while tuning. For sampling I choose SMOTE as there was a tie in performance with ADASYN.

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
alpha	[0.01, 0.0015, 0.001, 0.0005, 0.0001]

Table 25: MultinomialNB hyper-param value ranges

Best performing hyper-parameters:

Hyper-param	Value
alpha	0.001

Table 26: MultinomialNB best hyper-param

Accuracy: **0.4892**

LinearSVC

This is a linear implementation (linear kernel) of Support Vector Classification and it resolves multiclass by one-vs-rest. It can also handle sparse representations wich is important for us.

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
loss	['hinge', 'squared_hinge']
tol	[0.01, 0.001, 0.0001, 0.00001]
C	[0.001, 0.01, 0.1, 1, 10]

class_weight	['balanced', None]

Table 27: LinearSVC hyper-param value ranges

Best performing hyper-parameters:

Hyper-param	Value
loss	'squared_hinge'
tol	0.0001
C	0.1
class_weight	'balanced'

Table 28: LinearSVC best hyper-params

Accuracy: **0.4775**

ComplementNB

The Complement Naive Bayes version of Naive Bayes is particulary suited for imbalanced data so it should be a good contender for classifying our dataset. It has a reputation of outperforming Multinomial Naive Bayes specifically on text classification tasks. Particulary it use a better normalization for longer documents.

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
alpha	[0.01,0.001,0.0001]
norm	[True, False]

Table 29: ComplementNB hyper-param value ranges

Best performing hyper-parameters:

Hyper-param	Values
alpha	0.01
norm	False

Table 30: ComplementNB best hyper-params

Accuracy: **0.4842**

SGDClassifier

This is a Stochastic Gradient Descent wrapper around in our case a linear SVM classifier. It gives us the benefits of SGD for training our classifier. A disadvantage is that we get a lot of hyper-parameters to tune. We have decided to go with mostly default values to see in what ballpark this classifier will land.

To tune I have gridsearched the following hyper-parameters:

Hyper-param	Value ranges
loss	['hinge', 'log', 'modified_huber', 'squared_hinge']
alpha	[0.1,0.01,0.001,0.0001]

Table 31: SGDClassifier hyper-param value ranges

Best performing hyper-parameters:

Hyper-param	Values
loss	'modified_huber'
alpha	0.0001

Table 32: SGDClassifier best hyper-params

Accuracy: **0.4828**

Conclusion

As we can see none of the additional classifiers outperformed MultinomialNB. Perhaps in regards to SGDClassifier we got a hyper-param on the boundary of the grid-search, so additional grid-searching in this hyper-param would be adviced, aswell as all the other SGD related hyper-params we have not touched upon. So there might be some unexplored potential there. But it does seem that all in all the MultinomialNB classifier is performing at top of what is possible to acheive on this dataset.

Feature engineering

Document length & word count

The two first and easiest features I have engineered is document length (raw document before any preprocessing) and word count. As illustrated above these metrics have little promise of any positive contribution to any models.

File	Columns
words.tab	word_id, word, pos_tag
synsets.tab	synset_id, ontological tags
wordsenses.tab	word_id, synset_id
relations.tab	synset_id, name2 (EuroWordNet), target_synset_id

Table 33: Norwegian wordnet bokmål interesting files

POS tagging

POS tagging is also a way to do feature engineering for text. POS taggers are easily available for english language, but not so much for norwegian. The way I have solved this in this project is to use the raw files of a wordnet called "The norwegian wordnet bokmål v.1.1.2" [5] distributed by Språkbanken. It contains an extensive wordlist of 189 525 words coupled with POS tags (words.tab). From this list I performed basic lookup and counting and aggregated results over documents.

Word ontology tagging

In the same wordnet aforementioned there was a file of synsets (synsets.tab), coupled up with the wordsenses.tab file I could get the ontologies out from here and did a similar lookup, counting and aggregation of results over documents.

Performance

To test if these additional features gives any additional perfomance we try to do classification with and without the addional features and measure the accuracy. For scaling of the additional features we have used scikit-learns MinMaxScaler (scaled to 0, 1).

Our best model is so far, TfidfVectorizer, MultinomialNB, SMOTE and SelectKBest with hyper-params set according to best results in sections above.

The results are as follow;

Features	Accuracy
Text only	0.4892
Text + POS	**0.4893**
Text + ONT	0.4868
Text + Document length + Word count	**0.4893**
All	0.4865

Table 34: Feature sets accuracy performance

Conclusion

As we can see we get a minimal increase in accuracy by using some of the extra features, while the ontological features actually diminishes our performance.

We will use Text + Document length + Word counts features in further experiments.

Word embedding

So far we have only worked with discrete representations of our corpus in local vector space. These vectors do not capture any relationships of meaning between words.

Word embeddings try to capture meanings between words by training a neural net looking at the context each word occurs in some traing data and project these words into a N-dimensional space.

It is very resource demanding to train your own word embeddings, but luckily it is possible to use transfer learning here. This means we can take a embedding model trained on a different dataset, preferably related context and of course in same language and use that model on our dataset without doing any training. It is possible to also do finetuning to better adapt an existing model to our dataset, but we will just use the plain model here.

Luckily the Language Technology Group at University of Oslo have created a repository of such embedded models [6] and we will use model 81, a fastText wordembedding model.

This model is trained on a bokmål corpus consisting of Norsk Aviskorpus, NoWaC and NBDigital corpuses. It has a vector size of 100, vocabulary size of 4 428 648 words and it is not lemmatized.

Since we do not know the exact lemmatization of the model I think it is better to go with just raw words. Though a lemmatize model might be more generalized and regulatory to our model and fastTexts [7] ability to handle out of vocabular words in the manner that it split up the words in character n-grams (ie n=3, 'difference' becomes 'dif', 'iff', 'ffe', ... and so on) would perhaps negate any difference in lemmatization. This would be for further investigation.

Implementation

To convert our corpus to the embedded space we first need to identify the corpus vocabulary. An easy way to do this is to use scikit-learns CountVectorizer to vectorize the corpus and then extract the vocabulary from there. We can then iterate over the vocabulary and query the words from the fastText model to retrieve the embedded vector for that word. When done we have a matrix representation of our vocabulary.

Further we need to get document representations using this embedded matrix representation of our vocabulary. To do this I have used two different implementations.

Naive document representation

Here we simply represent our documents as the mean values over the documents word embedded vectors. To get the words for each document we extract the document vector from the CountVectorizer object and get each words indices from there, these indicies correspond to the vocabulary and we can then get the embedded word vectors from our matrix representation using the same indices. Then we have a subset of the matrix representing the document. We reduce it by taking the mean over each dimension and that is now our document representation in embedded space.

Tfidf weighted document representation

A slightly more sophisticated representation where we also keep the word frequency information.

Here instead of just taking the mean we take the weighted sum of the documents word representations. The implementation is similar to the naive representation, but we take the dot product of the Tfidf weights and the embedded matrix representation of the document. Depending on the Tfidf implementation we need to regularize by also dividing on the sum of the weights if the weights are not summed to 1.

Performance

We have an issue with our current best model, it being MultivariateNB and since it assumes multinomial distribution it can not tolerate negative values, which we have in our embedded word vectors. We can circumvent this problem by applying min-max scaling to our vectors, but we do not know if this will affect performance of the embedded vector representations so first we will try the vectors with a different classifier and try with and without scaling to see if min-max has any effect.

I will try using a logistic regression classifier for this task, sticking with scikit-learn package we use the LogisticRegression classifier. I will use the following hyper-parameters:

Hyper-params	Values
solver	'lbfgs'
multi_class	'multi_nomial'

Table 35: LogisticRegression hyper-param values

Before measuring scaling we check whether to use the naive or Tfidf-weighted representations. I use no scaling and the LogisticRegression classifier for this. We have no SelectKBest, sampling or extra features for these experiments. Just pure embedded vectors.

Representation	Accuracy
Naive	0.4796
Tfidf-weighted	**0.4995**

As we can see Tfidf-weighted performs better and we will use that in further experiments.

For scaling we use the MinMaxScaler from scikit-learn. It scales to values in the range 0 to 1.

Scaling	Accuracy
No scaling	**0.4995**
MinMaxScaler	0.4958

Table 36: Accuracy with and without min-max scaling

The experiments shows scaling has a negative effect on the result. But we also see that we suddenly have a much better accuracy then our previous best model, and this is right out the box. So either our word embeddings are doing the work or it is logistic regression that is a better classifier. Let us try the scaled embedding vectors with our previous best model and see how it performs.

Model	Accuracy
Text+Document length+Word count, TfidfVectorizer, SelectKBest, MinMaxScaling, SMOTE, MultinomialNB + Tfidf-weighted embedding (with MinMaxScaler)	**0.4886**

Table 37: Accuracy for previous best model with extra embedding vectors

As we can see it actually reduced performance in our previous model.

Logistic regression

Lets try to just swap MultinomialNB with LogisticRegression and no MinMaxScaling on the embedding vectors.

Model	Accuracy
Text+Document length+Word count, TfidfVectorizer, SelectKBest, MinMaxScaling, SMOTE, LogisticRegression + Tfidf-weighted embedding (without MinMaxScaler)	**0.5103**

Table 38: Accuracy for LogisticRegression full model

We clearly have a better classifier in LogisticRegression.

Conclusion

We have proved that word embeddings improve our accuracy and we have also stumbled upon a much better classifier in LogisticRegression something we can take with us in further experiments.

Additional experiments

As mentioned in the introduction we wanted to take a look at a novel modeling technique presented by Anshumali Shrivastava at the High Performance computing Conference at Rice Ken Kennedy Institute for Information Technology. This was a lecture mentioned to me by Vinay Setty to further investigtate.

I also have one more ensemble-variant model that I have implemented after discussions with Vinay Setty.

I will describe them in the following sections.

BucketClassifier

In his [Anshumali Shrivastava] presentation he outlines a modeling technique using repeated randomized aggregation of classes into buckets as a means of regularizing the dataset. Then separate classifiers are trained to predict samples to their respective family of buckets. The results are then aggregated and intersection of classes in the predicted buckets is then predicting the most probable original class. See illustration below.

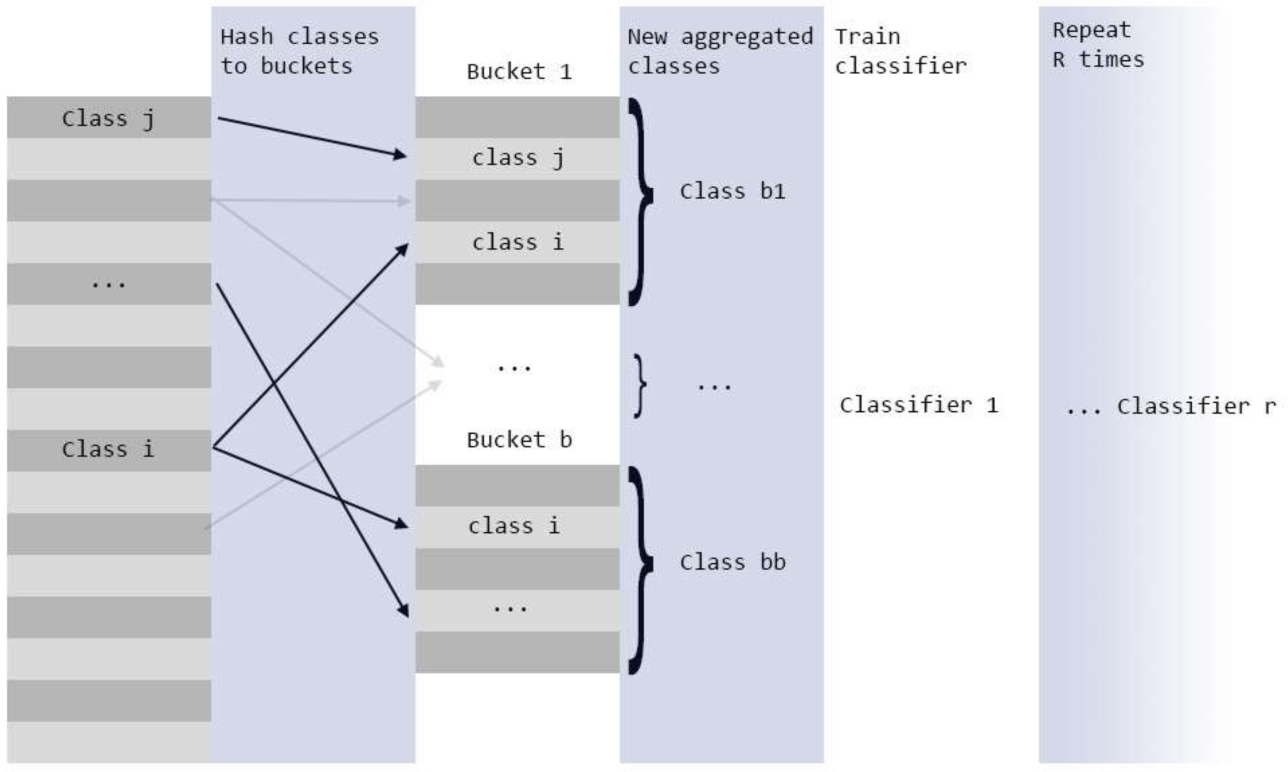

Illustration 13: Bucket classifier; aggregation of classes and training of separate classifiers

From the illustration we see that we hash classes to b buckets, r number of times with r numbers of classifiers. We use r families of hashfunctions so that for each step classes are randomly spread. If we take the intersection of classes in the r number of buckets one sample is assigned to by each classifier we get the high probability classes for that sample.

The idea is straightforward, but it is originally intended for really high number of classes, in the presentation it is mentioned with a 1 million class problem. So we might not have enough classes to get r number of classifiers high enough as r is intended to be equal to log of number of classes. In our case that would be less than 2 and there would be not enough buckets to get any reasonable intersections per sample.

But we can disregard this limitation and just try and tune the parameter r and see if we get any reasonable results.

Implementation

Since we have been working mostly with scikit-learn packages I will use their framwork for making compatible estimators that work in their Pipeline system. This means making a class with a fit and predict method.

Below is a snippet of the method for performing the bucketing of classes;

```
for n in range(self.num_classifiers):
  bucketed.append([self.hashfunc.hash(str(yi+n))%self.num_buckets for yi in self.y_])
```

```
for yu in self.classes_:
    b = self.hashfunc.hash(str(yu+n))%self.num_buckets
    self.buckets[b].append(yu)
```

Here we iterate over the number of classes (r in above explanation) and for each classifier we append a new list of classes generated by hashing the original class + number of classifier (this to create different families of hash functions) and we modulus with number of buckets. Then we create a bookeeping of what buckets each class is assigned to.

To do prediction we can either do a count of each of the classes in the buckets predicted and assign the majority class as the final prediction. Or we can train a classifier when we fit to predict the final class using the buckets assigned as train data. This method proved to give better results over just counting.

I refer to github for implementation details.

Results

Hyper-param	Values
hashfunction	murmuhash
number of classifiers	10
bucket classifiers	LogisticRegression
number of buckets	8
final predictor classifier	MultinomialNB

Table 39: BucketClassifier hyper-param values

Accuracy: **0.4295**

Conclusion

As we can see the BucketClassifier did not perform very well at our task. I have done only superficial testing of hyper-parameters for this one so there might be unexplored potential, but I think that the ideas original purpose of extremely large datasets with extreme amounts of classes is perhaps where its best potential lies.

EnsembleClassifier

The idea behind this classifier was to try to deal with the imbalance by creating a sequence of binary classifiers that for each iteration split the corpus in half. If we call it a left and right class we can say that if it classify to the left, the sample is sent to a classifier that is trained on the classes in the left bucket of classes. If the sample is classified to the right it is sent to the next binary classifier, splitting the remaing corpus in half and the procedure repeats itself until the sample is either classified left or there is only one class left and no more binary splits and it is classified as such.

Implementation

I still keep with the structure of scikit-learns classifiers and created a class with a fit and predict method. Here it is a little more complex with helper functions for calculating the size of the classes and aggregating them in binary splits such that the corpus is always as close to half and half as possible.

Again I refer to github for implementation details.

Results

Hyper-param	Value
binary classifier	LogisticRegression
bucket/left classifier	MultinomialNB

Table 40: EnsembleClassifier hyper-param values

Accuracy: **0.4437**

Conclusion

Performance here is still sub-par regular classifiers, but here aswell only superficial hyper-parameter tuning has been performed. One could also extend this classifier to use different classifiers at each step and one could also play with the objective of the split, here is was to balance in regard to number of samples, but one could perhaps balance in regard to class separation (classes most separate are bundled together, while similar are bundled in "the rest"). I leave this for further work.

TF-IDF analysis

This part of the project should have been applied before I did all the other experiments, but better late then sorry.

We will look at the distribution of important words for entire corpus and also for separate classes and we will see if there is anything we can infere about the corpus and perhaps get an indication of why we have so poor performance.

To gather statistics about important words I have used two methods, first I have aggregated the tfidf-weights and just sorted them to see which words are the most weighted. But since tfidf weights are L2-normalized this procedure might not give an accurate result when aggregated. So I also implemented a version that counts the top 10 words in each single documents and aggregate the counts over the corpus. We will see if the two methods give similar results.

I have used the same test, train split and same TfidfVectorizer hyper-parameters as in previous best models. I am doing inference on train split, because I do not want any decisions I make leak into the test data.

Top 25 words corpus, weights aggregated	Top 25 words corpus, top 10 counts aggregated

'hun', 'norsk', 'norge', 'du', 'krone', 'mene', 'prosent', 'land', 'man', 'gar', 'liten', 'oslo', 'slik', 'mann', 'tid', 'denne', 'der', 'vise', 'million', 'ntb', 'gang', 'politi', 'sak', 'hva', 'sta'	'hun', 'norge', 'norsk', 'du', 'krone', 'prosent', 'man', 'land', 'oslo', 'politi', 'mann', 'gar', 'mene', 'million', 'sak', 'selskap', 'vg', 'barn', 'kvinne', 'liten', 'slik', 'vise', 'gammel', 'usa', 'der'

Table 41: Top 25 words for entire (train) corpus

As we can see there is not much difference in the two methods and if we collect the top 100 words and do an intersection we get that the two methods overlap with 77%.

What can we infere from these lists of top words? We have to also remember that words with document frequency above 30% is removed, so it is perhaps surprising that the word 'hun' and 'du' is so high. Of the 25 words in this list, perhaps we can say that; 'hun', 'du', 'man', 'slik', 'denne', 'der', 'hva', 'slik' are words that carry little information value. So I think we can conclude that there is probably a lot of noise in this data.

Let us also take a look at the top words for a selection of the classes, I choose class 12, 19 and 18 because they are at the boundaries and middle of the distribution of classes.

Since there was little difference in the methods for aggregating we go further with just tfidf-weight aggregation as this method is much quicker.

Top 25 class 12, *'Crime and justice'*	**Top 25 class 19,** *'Foreign affairs, development aid and international economy'*	**Top 25 class 18,** *'Foreign trade'*

'politi', 'mann', 'hun', 'sak', 'aring', 'gar', 'gammel', 'finne', 'oslo', 'vg', 'mene', 'kvinne', 'ntb', 'tre', 'rett', 'fengsel', 'krone', 'dmme', 'tid', 'norsk', 'der', 'norge', 'iflge', 'gang', 'sikte'	'eu', 'land', 'norge', 'norsk', 'usa', 'gar', 'prosent', 'hun', 'mene', 'mellom', 'verden', 'gang', 'slik', 'mte', 'denne', 'kina', 'ntb', 'der', 'liten', 'krone', 'siden', 'tid', 'man', 'under', 'sta'	'norsk', 'norge', 'land', 'prosent', 'krone', 'ke', 'usa', 'sverige', 'mene', 'du', 'svensk', 'ntb', 'verden', 'hun', 'million', 'vare', 'eu', 'handle', 'aftenposten', 'mellom', 'slik', 'toll', 'man', 'nordmann', 'sist'

Table 42: Top 25 words single classes

Here we can see that the the noisy words at top of corpus is present in each of these three classes. We can also see a slight theme corresponding to the topics for each class.

Let us do some more investigation in how great the intersection in each class is with the entire corpus top 100 words.

Class	Intersection %	Class % of (train) corpus
1	73	3.47
2	83	2.28
3	82	7.43
4	80	3.74
5	73	2.39
6	72	2.95
7	79	2.67

8	76	3.16
9	84	2.23
10	77	5.70
12	77	13.44
13	82	3.61
14	75	1.93
15	78	8.11
16	79	8.10
17	82	5.51
18	65	0.56
19	76	4.79
20	77	7.60
21	74	0.72
23	83	3.20
24	72	2.00
25	61	2.67
26	81	1.75

Table 43: Class, corpus top 100 intersections

As we can see the intersection is quite large and stable across classes. This could be natural, most popular words in entire corpus would naturally also be popular in its subsets, but if we know there is a lot of noise in the corpus top 100 then we can probably assume that the noise is also distributed among the subsets of classes.

Custom vocabulary

To further investigate this issue I will attempt a technique mentioned to me by Erik De Vries. Let us take the top 100 most discriminating words for each class using for example Chi-squared. To do this I have applied the SelectKBest module from scikit-learn in iterations. For each iteration I recode the corpus to single class vs rest, select the 100 most discriminating words and append those to a set. When done iterating we have a vocabulary of the most discriminating words.

Using this vocabulary I initialize a vectorizer and transform the corpus using only words from the custom vocabulary.

Applying MultinomialNB with best performing hyper-parameters we end up with an accuracy of **0.4817**

So we do not get any much improvement in accuracy by removing noise. A interesting result here is that using only 20 words from each class we are able to acheieve an accuracy of 41%.

Conclusion

We have done a lot of experiments with this data, but none that were able to achieve the proposed 67% accuracy. So in conclusion this must mean that this is then atleast what did not work and I have tried to document my steps as thorough as possible for any future references.

The codebase is available at github. [8]

Future work

There could be potential in using the unlabeled data using semi-supervised methods that could improve accuracy. Also using more complex word embeddings like BERT, and perhaps do custom tuning or fine-tuning of existing models on the entire corpus could prove beneficial.

Refrerences

1) High Performance Computing Conference - Anshumali Shrivastava

https://www.youtube.com/watch?v=-YcSG4taFUM

2) scikit-learn

https://scikit-learn.org/

3) imbalanced-learn

https://imbalanced-learn.readthedocs.io/

4) Classification of text documents using sparse features

https://scikit-learn.org/stable/auto_examples/text/plot_document_classification_20newsgroups.html

5) THE NORWEGIAN WORDNET BOKMÅL v.1.1.2

https://www.nb.no/sprakbanken/show?serial=oai:nb.no:sbr-27&lang=

6) NLPL word embeddings repository by Language Technology Group at the University of Oslo

http://vectors.nlpl.eu/repository/

7) fastText

https://fasttext.cc/

8) DAT620 Classifying nordic language news articles into hierarchical topics

https://github.com/nmfoss/DAT620_delivered