Assignment 01: *BoW document classification*

Victor Alexander Steen-Olsen
Magnus Pettersen Nytun
https://github.uio.no/Victoste/magnuspn_victoste

ABSTRACT

In this assignment we are using a subset of the Signal Media OneMillion News Articles Dataset , with the intention of training a feed-forward neural network and classify the source of a text by using BoW features. The data we are using have already been split into a training set and a test set by the course supervisors, and the test set is unavailable to us until after the delivery of the assignment. The training data has been lemmatized, punctuation and stopwords have been removed and the tokens have been tagged with parts of speech.
Code for this assignment is stored in the GitHub-repository.

1

1.1 DATA PROCESSING

The code and a more experiments can be found in the Assignment01/notebooks/Data_exploration.ipynb and Assignment01/notebooks/magnuspn_data_exploration.ipynb files in the git-repository).

DATA EXPLORATION We observe an uneven split between the source-classes. The most frequent class 'MyInforms' accounts for 22.9% of our data. While most classes (17 of 20) account for between 3% and 5% of our data each. This knowledge gives us some important insights:

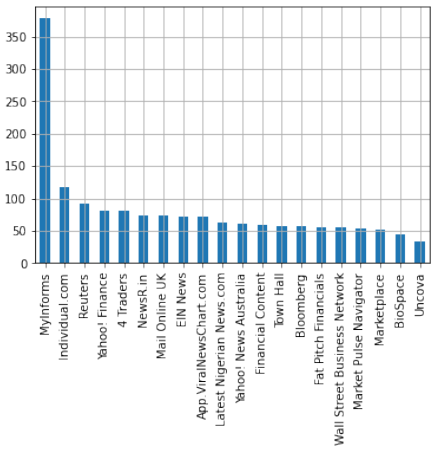

Figure 1.1: Distribution of source-labels in the training set.

It helps us establish a baseline. A majority classifier, that categorises everything as belonging to the majority class, here 'MyInforms', will have an accuracy of 22.9%. This then serves as a baseline for our own classifiers.

We need to be careful when splitting our data so that our training and development sets contains similar, ideally equal, proportions of each class.

We also examined the data for special characters, and found that our data contains email-addresses and URLs. It is uncertain if these tokens are informative, or not. They could be highly informative if these email addresses and urls have a real world connection to our source, but at the same time they are unlikely to be generalizable. We have so far decided against removing these from the data-set since we handle generalization in different ways. By using dropout, batch normalization in the feed forward neural network (FNN), and trying different vectorizes that excludes infrequent words.

Another discovery of note. Text length seems to be a significant feature. Average text length varies significantly between sources. There are some texts from 'Latest Nigerian News.com' that only contains a single token. These and other very short texts will likely be very hard to categorize unless we have the right vectorizer.

SPLITTING THE DATA We have decided to use a 85/15 train/dev-set split. We have looked at some research on what the optimal split is, and found that the main concern is to balance the variance in the training set and the variance in the test and dev set. It is therefore recommended to do cross validation if the data set is small, but this is not the case in this assignment. We have 75141 samples, and for data sets of this size somewhere between a 80/20 and a 90/10 split is generally recommended. We have therefor decided on a 85/15 split.

In order to make sure that our training set and development set have a similar distribution of classes, we used the train_test_split function from sklearn's model_selection module. We make use of the 'stratify' parameter to specify that the function should make sure that the categories in the 'source' column have, as far as it is possible, the same distribution in both the train and dev set.

1.2 Training the classifier

Our code pipeline For reference, see the __main__.py file in the Assignment01 folder.

First we read in the provided training data as a .tsv.gz file, and store this in a pandas dataframe. We then split our data into train and validation sets. The code for this can be found in data/datasplit.py

We then create a list of vectorizers that we want to test. Most of these vectorizers are based on the sklearn.feature_extraction.text CountVectorizer and TfidfVectorizer. We have created some custom preprosessors that can be found in the customVectorizers/preprosessor.py file. Commented out, you can also find most of the vectorizers that we have tried in our experiments, more on this in section 1.3. We then create a list of our FNN-models, which can be found in the models-folder.

For each vectorizer, we fit our vectorizier (extract vocabulary) on the train set, and create our train and validation datasets using a custom Dataset class that inherits from torch.utils.data.Dataset. Code for this can be found in data/dataset.py. This dataset class also maps the 'source' labels into numerical values, necessary since pyTorch and most other ML and deep-learning modules require exclusively numerical input and output features.

We then create our dataloader, and extract the number of input and output features for our model.

For each model in our model-list. We set the input and output size, create an AdamW optimizer, set learning rate and other parameters. We also use a Exponential learning rate scheduler. We then start the timer, and train the model for 100 epochs. This is handled by the train_loop.py function where we have also implemented early stopping. The file containing the code for early stopping is eary_stopping.py. The principle of early stopping is that we terminate the training early if we do not see any f1-scores (could be other performance measures) within delta of the best result within a fixed number of epochs.

We keep track of the best model, and save our best model and vectorizer at the end of the run.

1.3 FEATURE TUNING

We have experimented with many different BoW features. A complete list can be found in the __main__.py file where they are commented out. In section 1.4 and 1.5 we focus on the vectorizers in the list called best_vectorizers. The preprosessors used for some of these vectorizers can be found in the file customVectorizers/preprosessor.py.

We used a evolutionary-algorithm inspired approach to find the optimal model/vectoriser. For each new run, we ran all vectorizers on all models. After each run we evaluated the results, removed underperforming vectorizers, and used what we had learned as inspiration to create new ones. We started with models with few hidden nodes in each layer, but we gradually increased the number of these nodes as our models became more complex.

From our data analysis (see section 1.0) we found that the length of the text, would likely be a very informative feature. We have tested several of the vectorizers with and without length feature added. For unigrams the f1 score for the vectorizers with the length feature, was higher than for those that did not use this feature, but as the models grow in size, and we use more features we do not observe the same benefit from adding the length feature.

We have tried vectorizer that uses unigram, bigram and trigrams as features. Larger n-grams perform better, and it would be interesting to see if this tendency continues if we use 4-grams, 5-grams or even 10-grams, but we encountered some problems when using n-grams. The size of our vectorizer became too large to push to git. This is to be expected, since the amount of possible features increases exponentially as n becomes larger.

Also from our data exploration stage, we looked at converting all tokens with the _NUM pos-tag, to be the same token - 'NUM'. This gave good results, but bigrams and trigrams, performed better.

We also tried using the sklearn TfidfVectorizer. From observation, this vectorizer gave faster convergence, but worse results. Tfidf gives features that occur in few documents higher weights, this is often useful for document classification using statistical methods, but when using a FNN this weighting is learned and is a part of Θ, and is therefore unnecessary.

We tried using a subset of the POS tags as features. We created vetorizers that only extracted content words (NOUN, VERB, ADJ). This was based on the assumption that topic of the sources is significantly different from each other, and that most of the semantic information related to the topic at hand, is contained in topic words. This is the same reason why one removes stop words, but this gave unsatisfactory results. This approach did not perform well, frankly it performed horribly.

We also tried counting number of occurrences of each POS tag in a text and adding this as a feature, but we did not have time to test this sufficiently, so we cannot conclude on the performance of adding these features.

We tried removing extremely rare, and extremely common tokens, but found that both of these were relevant for our classification task. We also experimented with using different amounts of input-features for our models. More features performed significantly better, but took much longer to train. So we settled for 7500 features for our largest models.

CONCLUSION: More features and features that capture more semantic context (bigrams and trigrams) performed better. Adding the length of the text, and the count of each POS tag, showed promise, but we have too little information to conclude in regards to these features.

1.4 TIME EFFICIENCY

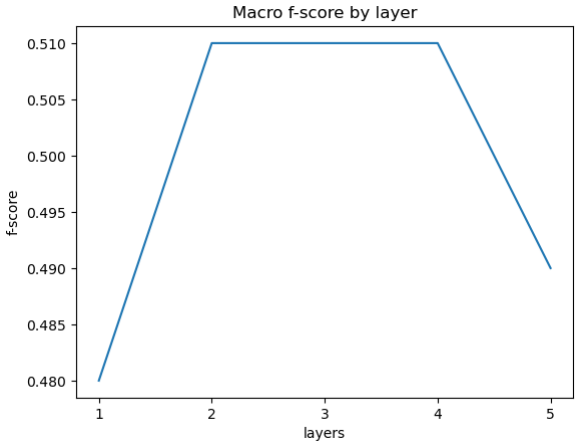

Figure 1.2: macro f1-score for our best model: two_hidden_layers_test, and trigram_len_vec vectorizer

The code to generate these plots can be found in notebooks/make_plot.ipyb

We see that the f1: score is highest for the models with 2-4 hidden layers, and a drop in performance for 5 hidden layers, this is something we have also observed during testing, and tuning of vectorizers. This is likely due to the increase in complexity of the search landscape for Θ when we have more layers, and that our training algorithm has problems finding an optimal Θ

Training time seems to increase linearly in relation to the number of hidden layers, but the training time for 3 hidden layers is lower than expected. This is likely do to the random aspects of the training algorithm, like weight initialization, combined with our use of early stopping.

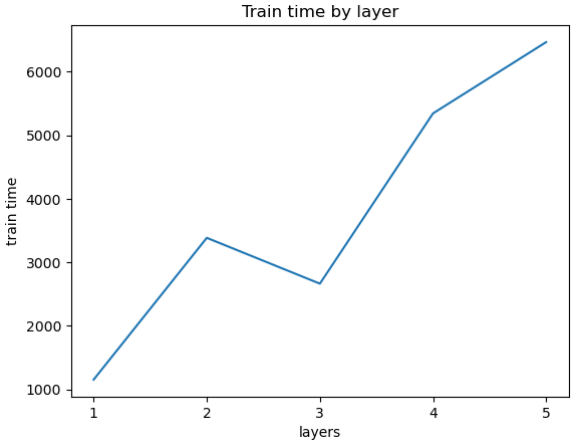

Figure 1.3: Train time for our best model: two_hidden_layers_test, and trigram_len_vec vectorizer

1.5 Evaluation

We ran over 150 models in total, we will only include some of our latest and best models here. Statistics related to most (some where unfortunately lost or overwritten by other data) of our runs can be found in the slurm_outfiles and the logfiles folders. The f1, score for each vectorizer model pair, can also be seen in the models that end with _test, in the model folder The moddellogs folder is quite unorganized so we have also created a modellogs_best folder where the results from some of our best runs can be found.

Model name -> *Vectorizer name*	**one_hidden_layers_test bigram_len_vec**	**two_hidden_layers_test bigram_len_vec**	**three_hidden_layers_test bigram_len_vec**
f1:	0.4772	0.4925	0.4965
accuracy:	96.564	96.413	96.451
precission:	0.5893	0.5703	0.5513
recall:	0.4144	0.4521	0.4622

Model name -> *Vectorizer name*	**four_hidden_layers_test bigram_len_vec**	**five_hidden_layers_test bigram_len_vec**
f1:	0.5044	0.4805
accuracy:	96.465	96.739
precission:	0.5540	0.5832
recall:	0.4764	0.4477

Model name -> *Vectorizer name*	**one_hidden_layers_test** **trgram_pos_vec**	**two_hidden_layers_test** **trgram_pos_vec**	**three_hidden_layers_test** **trgram_pos_vec**
f1:	0.4946	0.5095	0.5115
accuracy:	96.591	96.413	96.508
precission:	0.5853	0.5460	0.5619
recall:	0.4411	0.4845	0.4793

Model name -> *Vectorizer name*	**four_hidden_layers_test** **trgram_pos_vec**	**five_hidden_layers_test** **trgram_pos_vec**
f1:	0.5060	0.4952
accuracy:	96.647	96.713
precission:	0.5833	0.5401
recall:	0.4711	0.4656

Model name -> *Vectorizer name*	**one_hidden_layers_test** **trigram_len_vec**	**two_hidden_layers_test** **trigram_len_vec**	**three_hidden_layers_test** **trigram_len_vec**
f1:	0.4799	0.5137	0.5078
accuracy:	96.581	96.433	96.447
precission:	0.6129	0.5518	0.5683
recall:	0.4189	0.4883	0.4732

Model name -> *Vectorizer name*	**four_hidden_layers_test** **trigram_len_vec**	**five_hidden_layers_test** **trigram_len_vec**
f1:	0.5091	0.4937
accuracy:	96.489	96.654
precission:	0.5518	0.5550
recall:	0.4829	0.4633

STATISTICS FOR OUR BEST MODEL using two_hidden_layers_test model, and trigram_len_vec vectorizer:

	mean	**std**
f1:	0.50998	0.00141
accuracy:	96.367	0.01141
precission:	0.54720	0.00243
recall:	0.48457	0.00124

When training the algorithm three times, we observe that the average for all performance measures are slightly lower than for our initial run, and in particular the f1 score is significantly lower. This makes me reconsider whether this was actually our best model, or if our trgram_pos_vec vectorizer would have performed better.

STATISTICS FOR OUR BEST MODEL (REVISED) using three_hidden_layers_test model, and trigram_pos_vec vectorizer:

	mean	**std**
f1	0.51078	0.00131
accuracy:	96.46483	0.01542
precission:	0.55789	0.00098
recall:	0.48069	0.00163

Looking at the results after tree runs, we see that the precission, accuracy and f1, score is slightly better for the model three_hidden_layers_test model, and trigram_pos_vec vectorizer, while the recall is slightly worse. Since we are primarily concerned with f1 score in this assignment we choose to use the hree_hidden_layers_test model, and trigram_pos_vec vectorizer combination in eval_on_test.py

Other avenues to research that we would have like to examine are: 1. Using an Genetic algorithm to find the optimal number of nodes in each hidden layer, and the optimal vectorizer-model combination. 2. look at how our model performs with larger n-grams. 3. create an ensemble model, that combines several of our models into a single model which hopefully performs better than the models individually. 4. using embeddings as input features.

EVAL_ON_TEST.PY SCRIPT It takes as input a path to a saved model and a test dataset (presumed to be a tsv file), and return accuracy, precision, recall, and F1 scores for each class in the test set, as well as their macro averaged values
We encountered some problems with file-sizes, so we have saved our best model to fox in the directory:
fp/projects01/ec30/magvic_large_files/output/three_hidden_layers_1024-trigram_pos_vec-0.51.bin

See the test_eval_on_test.slurm for an example of how to run the eval_on_test.py script.