UiO - V21: IN5550 - Project1

Fábio Rodrigues Pereira - fabior@uio.no
Per Morten Halvorsen - pmhalvor@uio.no
Eivind Grønlie Guren - eivindgg@ifi.uio.no
UiO GitHub repository: https://github.uio.no/fabior/IN5550/tree/master/Oblig1

Feb 7th, 2021

Contents

1 Introduction

The purpose of this project was to implement a feed-forward neural network, using bag-of-words model features, to solve a multi-label classification problem, predicting the source of a given news article.

The data provided for the task contained around 75,000 entries (samples) from 20 different sources (classes). The texts included in these data were already lemmatized and part-of-speech tagged, with the stop words and punctuation removed. This scenario makes NLP tasks slightly easier.

A feed-forward neural network has many variables that can be tuned to find the optimal architecture for the task at hand. The order in which these variables are tuned could also affect the final performance of the model.

1.1 Source code

All code made for this project is written in Python v.3.9, and found in the GitHub repository at https://github.uio.no/fabior/IN5550/tree/master/Oblig1. The repository contains:

- `data/` - Directory containing the data utilized in this project.
- `output/` - Directory containing the collection of tuning results.
- `package/` - Directory containing the collection of Python methods utilized in the project.
- `package/ann_models.py` - Python file containing the neural network model.
- `package/studies.py` - Python file containing all the studies used for tuning hyper-parameters.
- `package/preprocessing.py` - Python file containing classes that load, split, and preprocess the data.
- `report/` - Directory containing the written report in latex and pdf.
- `eval_on_test.py` - Python file for testing the optimally trained model.
- `evaluation.py` - Python file used for tuning the hyper-parameters.
- `plot.py` - Python file used for plotting the results found in tuning step.

2 Discussion

2.1 Data processing

This project's first step was to load the data and split it between a training set and a validation set. Since there are a few more necessary preprocessing steps before the classifier can be trained, this split is taken care of in the `BOW` class, found in `preprocessing.py`. This split must be done before counting word occurrences to ensure the counter has not seen any of the data used to tune the model. Another critical aspect of the splitting is to ensure the distribution between the labels is preserved in both sets.

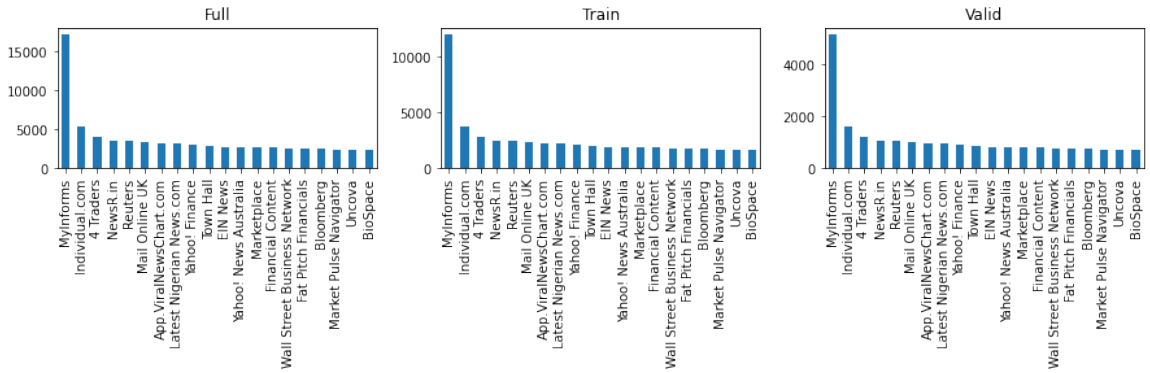


Figure 1: Source distributions

This distribution was maintained by using the `sklearn.utils.resample` method and providing the source column for the `stratify` parameter. The ratio between entries in the training set and validation set was set to 0.7, in the `train_size` parameter when initializing a `BOW` class object.

The `resample` method returns a subset of size `n_samples` of the original data, as long as the `resample` parameter is set to false. The number of samples was found multiplying the provided

proportion with the length of the original data. The output from `resample` was stored as the training set.

The validation set was then found by pulling all the remaining data points from the original data set, by filtering on the indexes *not* in the train set. A code snippet of this process is shown below.

```
n = len(df) * train_prop

train = sklearn.utils.resample(
    df,
    replace=False,
    stratify=df.source,
    n_samples=n
)

test = df[~df.index.isin(train.index)]
```

2.2 Training a classifier

A precise formulation of the task at hand can be as follows:

> Given a list of words in a document, **predict the source** of the document.

The algorithm needed to define a common vocabulary for the input data and convert the documents into bag-of-words vectors to later feed into the classifier.

A b.o.w. transformation converts raw text documents to count vectors, disregarding grammar, stop words, and any other irrelevant characters. The counts that replaced each word correspond to the number of times that word is found in the training set, as long as it is part of the common vocabulary. Note that since the original data was split into a training and validation set, word occurrences in the *validation* set are *not* included in this counting.
Specifically, the assignment asked to test three different types of bag-of-words techniques for this representation. The chosen types for this project were:

- `counter` These b.o.w. vectors are the actual counts of that word in the common vocabulary.
- `binary`: Instead of the numeric counts, a word is represented as a binary value, depicting whether or not that word is a part of the common vocabulary.
- `tfidf`: Like the `counter` representation, this b.o.w. vector type replaces the word with a numeric value. The difference here is that the word counts are normalized according to how many other documents this same word occurs. The idea is to weed out ubiquitous words that are not predefined as stop words.

The size of the common vocabulary was determined by a tunable hyper-parameter `vocab_size`, which is taken in as a parameter for new `BOW` instances. The definition of this parameter already in the initialization of `BOW` objects allowed for hard definitions of tensor shapes, avoiding expensive and unnecessary computations. More on this parameter in the feature tuning section.

Before actually training and tuning the classifier, the last step was to extract the gold source labels for the input data. These are the true labels for each data point, and are found in the attribute `y_train` for `BOW` objects.

Discussion around the model's parameters along the model's best performance can be found in the upcoming sections.

2.3 Feature tuning

Table 1 shows the initial default values chosen for the classifier:

Hyper-parameters				
Hidden Layers	**Units**	**Epoch**	**Mini-batch**	**Dropout**
1	25	20	32	0.2
Bias	**Learning Rate**	**Momentum**	**Loss-Function**	
0.1	0.01	0.9	Cross-Entropy	

Table 1: Initial model parameters

The values for these initial parameters were not entirely random. Arguments for the choice of default values are given below:

- `Bias`, `learning rate` and `momentum` were set as their default value.
- `Batch size` was initialized to 32 as per instructed.
- `Loss-function` was initialized to cross-entropy because it is typically used for multi-label classifications.
- `Hidden layers` was initialized to 1 to make the simplest initial model.
- `Epochs` was initialized to 20 after trying out several different epochs and seeing no improvement after about 20.
- `Units`
- `Dropout` was initialized to 0.2 after seeing a lot of variance without regularization.

Tuning the classifier was done semi-automatically with the help of the `packages/studies.py` file. This module consists of multiple different `Study` classes for each of the different parameter pairs this group decided to test. The encapsulation of the classes helped maintain efficient memory usage and made for straightforward and understandable tuning.

An `packages/evaluation.py` script created executed the studies for each of the tuning steps. It was this file that generated the results presented below. These results are stored in the `output` directory.

The plots were produced using the `plot.py` script, along with the header of the file to plot as a command-line argument.

2.3.1 Activation functions

The first parameters tuned were the activation functions, both in the hidden-layers and in the output-layer. Four different activation functions were tested for each of these, giving a total of 16 possible combinations. These were `["sigmoid", "tanh", "relu", "softmax"]`.

The metrics used to decide which combinations of activation functions were optimal for each position were the times, accuracy, and F1 scores of each model. As shown in Figure 2, the best candidate gave the lowest possible time and highest possible accuracy and F1-score.

Here, it is essential to note that the F1-score is a more reliable metric than the accuracy since the distribution between classes was not entirely balanced. While accuracy measures the percentage of correct classifications, the F1-score gives a measurement of the combination of precision (number of correctly classified over the total number of samples) and recall (number of correctly classified over the total number that should have been classified with that label). For this task, a macro F1-score was needed since there were multiple labels to classify. The formula for calculating a macro F1 score is given below, where N represents the number of classified labels.

$$\text{Macro } F_1 = \frac{1}{N}\sum_{i=0}^{N}\frac{\text{precision}_i \cdot \text{recall}_i}{\text{precision}_i + \text{recall}_i}$$

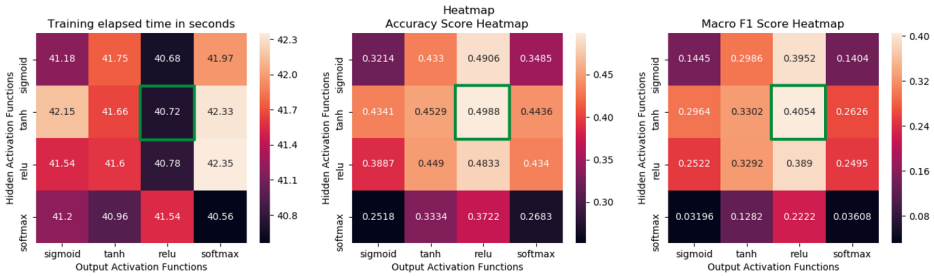


Figure 2: Heat maps for activation function tuning

The optimal combination of activation functions was the hyperbolic tangent on the hidden layers, with relu on the output layer. This decision was reasonably straight-forward since for each of the heat maps shown in Figure 2, this combination gave the best score according to the criterion previously mentioned.

2.3.2 Bag of Words Type and Vocabulary Size

The second set of parameters tuned was the vocabulary sizes and the b.o.w types. We tested a range of $\in [50, 30000]$ as vocabulary sizes and three variations of b.o.w: `binary`, `tfidf`, and `counter`. These were performed using the function `BOWStudy` in `packages/studies.py`.

As with tuning the activation functions, the metrics in focus when tuning the optimal bag-of-words parameters were times, accuracy, and F1-scores for each model. The training time is increasing linearly with the size of the input vectors after vocab_size

≈ 5000

As displayed in 3 the best candidate is the values that yielded the best F1-score and accuracy compared to the training time.

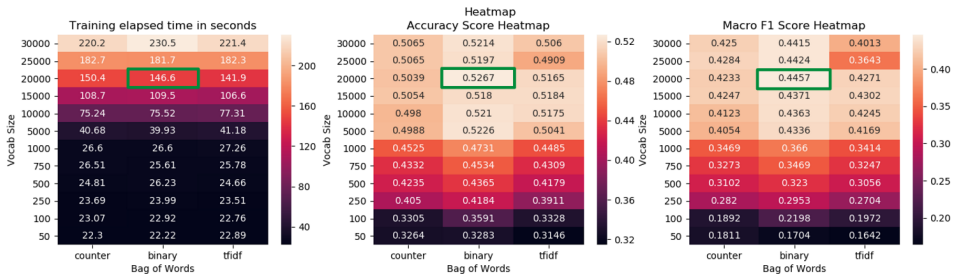


Figure 3: Heat maps for vocabulary sizes and bag-of-words types.

Looking at 3 it is pretty clear that the optimal settings found from this study were a vocabulary size of 20000 with the binary counter for the CountVectorizer as this setting yielded the best accuracy *and* F1-scores while still training reasonably fast compared to the larger vocabulary sizes.

2.3.3 Part of Speech (PoS) analysis

One typical technique applied in NLP for feature tuning is POS (Part of speech), which considers the types of the words and how much information these groups can bring for training the artificial

neuron network models. We filtered the entire vocabulary for capturing a specific given word type, or combination of that, per time and then performed the model's training and evaluation as follows:

Part of Speech (PoS) Analysis		
Vocabulary	Acc. score	F1-score
all PoS together	0.5267	0.4457
only VERB	0.3938	0.2966
only ADJ	0.3700	0.2819
only PROPN	0.4057	0.3290
only NOUN	0.4420	0.3568
only ADV	0.3316	0.2123
NOUN and PROPN	0.4820	0.4026
NOUN and PROPN and VERB	0.5110	0.4319
NOUN and PROPN and VERB and ADJ	0.5032	0.4265
NOUN and PROPN and VERB and NUM	0.5131	0.4298

Table 2: Results with the most common 20.000 words and Binary Bag of Words (BoW) type for a given PoS vocabulary.

It is important to mention that the word type `NOUN` had the highest individual score, showing that this category brings more relevant information than the others. However, as shown in Table 2, the best PoS combination is still the vocabulary containing the most common 20.000 words for all PoS together.

2.4 Measuring time efficiency

2.4.1 Epochs and Batch Size

The next step after finding the ideal shape of input vectors was to find approximately how many epochs were necessary for training the neural network. This step was decided to be done here to save time and avoid the hassle of training unnecessary epochs for the remaining optimization. The fewer epochs needed, the faster future models could be built, meaning more efficient parameter tuning.

The batch size was tested together with epochs since they are used to measure time efficiency. A range of $\in [32, 60]$ was checked here, along with an outlying 500 as a control.

Again, heat maps were used to help visualize these statistics:

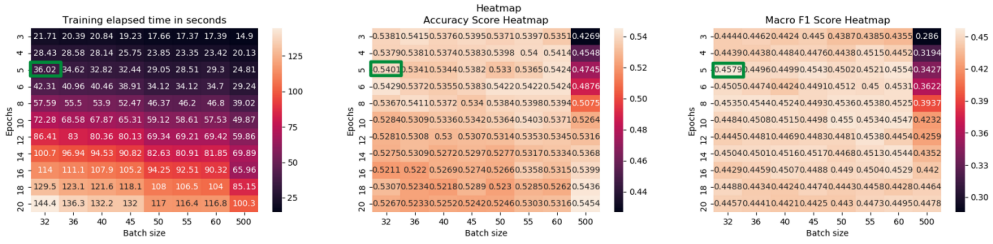


Figure 4: Heat maps for epoch and batch size optimization

It was found that the model converged quickly, giving good enough scores already after only five epochs. The smaller batch sizes were preferred over more extensive, although the variation between (most of) the batch sizes analyzed in this step proved to be relatively small. The control size of 500 was used to help highlight that smaller sizes were preferred.

After these results and deductions, we decided to stick with five epochs and a batch size of 32 for all future models.

2.4.2 Hidden Layers and Units per Layer

After most of the heavy and computationally expensive parameters were tuned, it was time to optimize the number of hidden layers. The number of units per layer was decided as an excellent match to optimize with this parameter since they both deal with the network's inner architecture.

Given that the assignment specifically asked to test at least five different numbers of hidden-layers, the group decided to test hidden-layer values in the range $\in [1, 6]$. For the number of units, the range of $\in [10, 2000]$ was decided to encompass as many candidates as possible. The capability to check up to 2000 units highlights how beneficial it was to tune epochs *before* hidden layers. Especially taking into account that the time exploded as the values for each parameter combo grew, as seen in Figure 5.

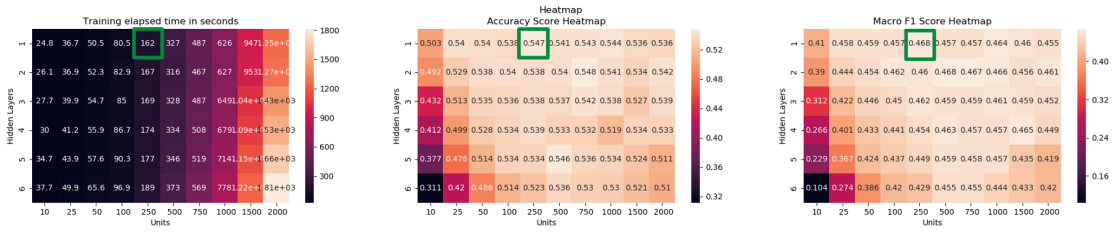


Figure 5: Heat maps for number of hidden layer and units optimization

Surprisingly enough, the F1-score and accuracies did not vary much as the number of hidden layers increased. Instead, it looks as if the best accuracy was achieved already with only 1-2 hidden layers. As expected, computation time increased almost linearly with the number of units. This increase occurred because more units per hidden-layer mean more computations both in forward and backward propagation. Figure 6 show the relation between the number of hidden layers, computation time, and F1-score.

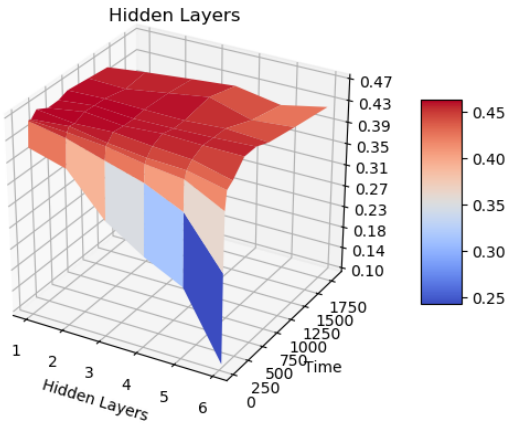
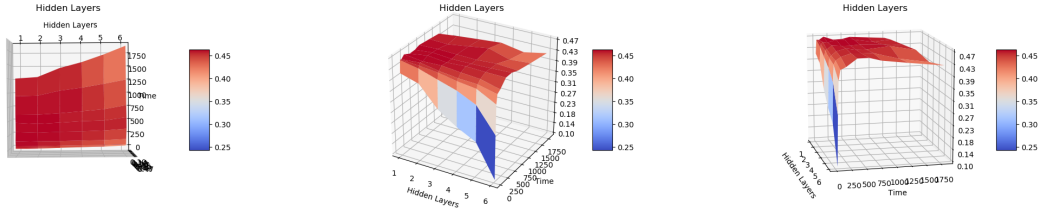


Figure 6: X: Number of hidden layers, Y: Time, Z: F1-Score

This part of the assignment text specifically asked to analyze how the number of hidden-layers affects computation time. Some subplots showing different views of the 3D plot from Figure 6 are shown in Figure 7.

Specifically, Figure 7a shows the requested relation between hidden-layers and computation times. Here, the different levels on the graph specify the different number of units. This information needs to be implied from looking at the other plots these data generate (when running `plot.py`).



(a) H.l. vs time (b) H.l. vs time vs F1-score (c) H.l. vs time vs F1-score

Figure 7: Different views of 3D plot above

As shown, the general trend of increased computation time is still prevalent as the number of hidden layers increases. The rate of change increases as the number of units per layer grows. This circumstance makes intuitive sense since higher values for both units and hidden-layers directly mean more calculations per epoch.

2.5 Model evaluations

2.5.1 Learning rate and momentum

After finding the optimal *structure* of the neural network, the time had come to evaluate some of the standardized parameters for the model, in hopes to achieve even better model performance. Specifically learning rate and momentum were tuned here. Learning rate was tested with values in the range $\in [0.9, 0.001]$ and momentum in the range $\in [0, 0.9]$. Both of the ranges were chosen to pillow the default values for the respective parameters.

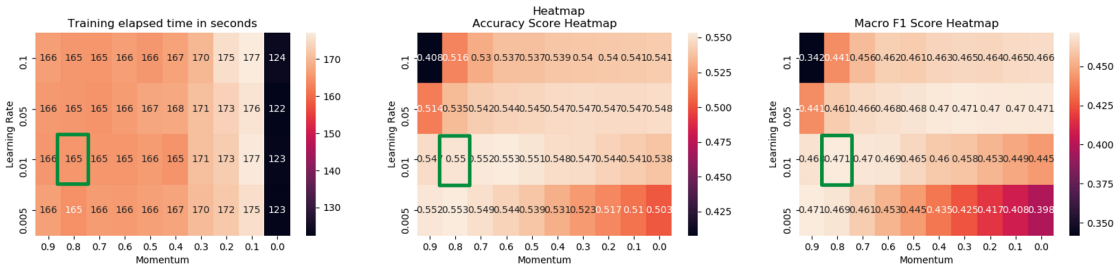


Figure 8: Learning rate and Momentum heat maps

This evaluation gave no clear winner for either of these parameters, although a trend between their relationship can be observed. It appears that the higher the learning rate is, the lower the momentum needs to be, and vice-versa. To be safe, values within this range that were also close to their defaults were chosen.

And with this step, the tuning of the model was complete and the final hyper-parameters are:

Hyper-parameters					
Hidden Layers	**Units**	**Epoch**	**Mini-batch**	**Dropout**	**Bias**
1	250	5	32	0.2	0.1
Learning Rate	**Momentum**	**Loss-Function**	**HL-Act. Funct.**	**Out-Act. Funct.**	
0.01	0.8	Cross-Entropy	tanh	relu	

Table 3: Optimized model parameters

3 Conclusion and future development

This project introduces the bag-of-words (BOW) features processing technique for predicting multi-label classes using Multi-Layer Perceptron (MLP) artificial neural networks. The MLP model predicts the source (output) of a given transformed text (input). This research utilizes three different types of BoW (counter, binary, and tfidf) for input transformation, and the metrics used for assessing the predictions were accuracy-score, macro precision-, macro recall- and macro F1-score. It is essential to say that the sourcing's balancing is not applied because we want to have a model capable of predicting unbalanced data. Regarding the hyper-parameters tuning's steps, five pairs of parameters were defined and employed iteratively: Hidden-layers activation functions X output-layer activation function, vocabulary size X b.o.w.'s type, epoch X batch size, hidden-layer X units, learning rate X momentum.

Our optimal model's best results were approximately 12% better than our benchmark scores 0.49 and 0.40, respectively. These results were achieved in the first study (Hidden-layers activation functions X output-layer activation function), confirming that the tuning procedures, especially the vocabulary size X b.o.w.'s type, had a significant impact on the final score.

Score type	**Run 1**	**Run 2**	**Run 3**
F1-score	47.39%	46.54%	47.12%
Accuracy	55.81%	55.42%	56.04%
Precision	46.67%	45.61%	46.11%
Recall	49.49%	48.87%	48.97%

Table 4: Scores for three different builds

Table 4 shows the optimal model's final performance for 3 different builds, using 3 different random states. For this task, metrics as high as 0.56 (accuracy score) and 0.46 (f1-score), are assumed to be quite good, especially using the vanilla MLP model.

As a future research objective, it would be relevant to explore more Part of Speech (PoS) filtering techniques in the preprocessing stage. Another natural next step could be to map the raw text documents to pre-trained word-embedding space instead of count vectors as the classifier's input features. This mapping would allow the model to retain more contextual meaning of the documents, allowing the network to pick up on even smaller nuances between the different sources. However, such a drastic change would require a new round of optimization and tuning since the input vectors would be entirely new for the model. That would, however, be outside the scope of this project.

4 Bibliography

[1] T. Hastie, R. Tibshirani, J. Friedman. *The Elements of Statistical Learning data mining, inference, and prediction*; 2nd edition; Springer; New York, USA. 2017.

[2] Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning*; 2nd edition; Packt; Birmingham, UK. 2017.

[3] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*; O'Reilly Media; Sebastopol, CA. 2017.