Assignment 1

Group: Anette Fredriksen (anettfre), Anthi Papadopoulou (anthip), Nick Walker (nichow)

February 2021

Code: https://github.uio.no/anthip/IN9550-IN5550/tree/master/obligatory1

1 Data processing

We read the data file using the Pandas[1] framework, and then to split the data into train and test sets we used sklearn's[2] train_test_split. Our test size was 0.2 and we also used the stratify option, so as to keep the distribution of labels as equal as possible in our two datasets. The stratify parameter performs a split in such a way, so that the proportion of values in the sample is the same as the proportion of values given to stratify. We then plotted the distribution of the train and test set we created using the Matplotlib library[3], as visible in Figure 1.

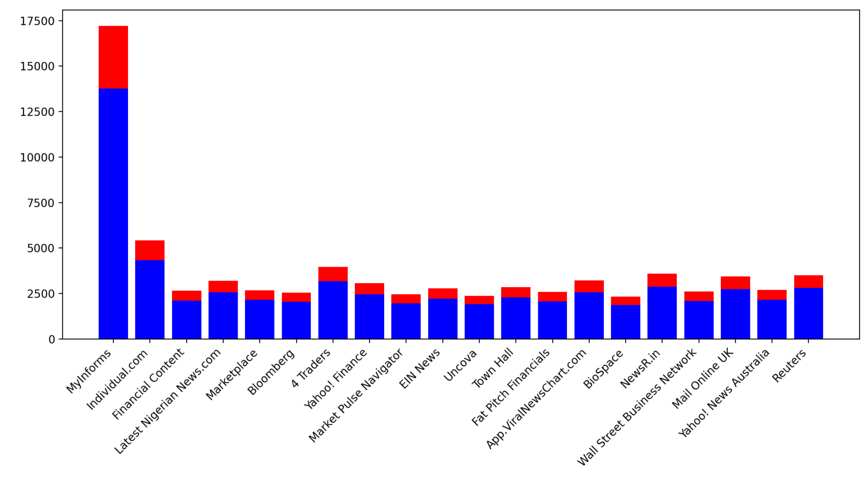

Figure 1: Distribution of classes in the train and test set

2 Training a classifier and feature tuning

Before splitting the data, we used sklearn.feature_extraction.text's CountVectorizer and LabelEncoder on the source and text columns of our dataframe. We decided on a vocabulary of size 2000, and used Bag of Words as the initial feature extractor. After splitting, we transformed our train and test features and labels to float and long tensors respectively. Finally, we utilised Pytorch's[4] TensorDataset and then DataLoader to create batches out of our training and test data. The whole procedure takes around 1 minute to run.

We then implemented our model as a class inheriting from Pytorch nn.Module. In the initialization function we implemented an input layer whose size was equal to the size of the input features (2000), leading to one hidden layer of 64 nodes, which then led to the output layer of size 20, as is the number of output classes. All the layers were linear ones.

For the forward function which follows the initialization, we passed the input through each layer and through a rectified linear unit activation, returning x at the end after all the passes. We chose cross entropy loss for our loss function as it works well for the task of multi-class classification, and finally we chose SGD optimizer, a learning rate of 0.001 and a Nesterov's momentum value of 0.9 since it is said to improve convergence and speed up training when one uses SGD.

We also applied dropout regularization with a p value of 0.5. We used a batch size of 8 and we trained the model for 50 epochs. The results of this can be found in the following section's table.

3 Feature tuning

We experimented with three feature variations, BoW unigrams and bigrams, BoW bigrams and TF-IDF unigrams feature extractor. To achieve the first two we set the parameter ngram_range of the vectorizer to (1,2) and (2,2) respectively, while for TF-IDF we utilised the TfidfVectorizer with an ngram_range of (1,1). Table 1 shows the performance of the model using each feature variation.

Feature Variation	Accuracy	Precision	Recall	F1-score
BoW	**0.4911**	**0.4161**	**0.4014**	**0.4018**
BoW Bigrams	0.4870	0.4490	0.3786	0.3992
TF-IDF	0.4901	0.4142	0.3753	0.3931

Table 1: Feature variation experimentation and scores

We selected BoW unigrams and bigrams as our feature extractor seeing how even it showed not only the biggest accuracy but also F1 score, which is more relevant for the class for multi class text classification.

4 Time efficiency

We then moved on the experimenting with the number of hidden layers and nodes. We tried a number of hidden layers from 1 to 5, and combinations of 64 and 32 nodes. Table 2 bellow shows the performance and the training time in respect to each added layer.

Layers	**Size**	**Time**	**Accuracy**	**Precision**	**Recall**	**F1 Score**
1	64	7’	0.4911	0.4161	0.4014	0.4018
2	**64-64**	**9’**	**0.5135**	**0.4438**	**0.4337**	**0.4302**
3	64-64-32	11’	0.5002	0.4205	0.4209	0.4131
4	64-64-32-32	12’	0.4794	0.4222	0.4031	0.4001
5	64-64-32-32-32	14’	0.4747	0.4041	0.3994	0.3899

Table 2: Hidden layer experimentation and scores

We then chose to move forward with two hidden layers of size 64, seeing how they yielded good results compared to the other combinations and also trained in a reasonable time window. The following plot shows the relation between the 4 evaluation metrics and the time it took of each layer combination to run.

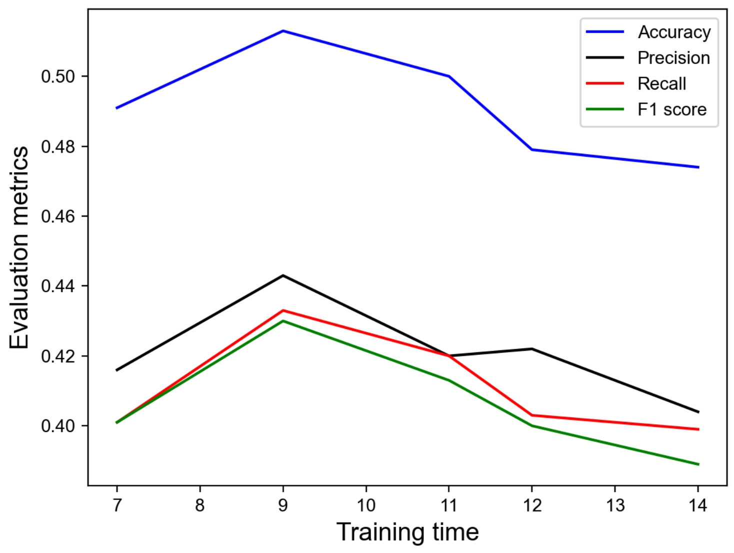

Figure 2: Time vs Metric for layer added

5 Evaluation

Having then chosen our best model we proceeded to train it 3 times so as to see the results it would output when initialized differently each time.

A small summary of our best model parameters:
→ Two hidden layers of size 64
→ Relu activation function
→ Dropout regularization, p=0.5
→ CrossEntropyLoss as a loss function
→ SGD for optimizer and momentum=0.9
→ Learning rate with a value of 0.001
→ Bow feature extractor and vocabulary size of 2000
→ Train for 50 epochs with a batch size value of 8

We kept track of the scores (Table 3), and then we calculated the average and the standard deviation of the scores (Table 4).

Training Run	**Accuracy**	**Precision**	**Recall**	**F1-score**
1	0.5056	0.4348	0.4325	0.4277
2	0.5091	0.4365	0.4280	0.4307
3	0.5009	0.4321	0.4249	0.4263

Table 3: Scores after each training run

	Accuracy	**Precision**	**Recall**	**Fscore**
Average	0.5051	0.4344	0.4284	0.4282
Standard Deviation	0.004	0.002	0.003	0.002

Table 4: Average and standard deviation for all runs

6 Discussion

6.1 Feature variation selection

From the three feature variations we chose to test, BoW unigrams and bigrams had the best performance which was not expected, since we believed that TF-IDF would be the best option due to its more sophisticated nature and the weights it assigns to the words depending on their frequency. It is obvious by looking at Table 1 that their difference was not vastly different, so we could argue that BoW performed better due to the ngram_range = (1,2) we provided it with, which allowed for bigrams to also be a part of the features along with unigrams.

6.2 Number of hidden layers and nodes

We initialised our model with one hidden layer, knowing that, generally, for most problems one could use one to two hidden layers and get a decent performance. Neural networks with one hidden layer can approximate any function that contains a continuous mapping from one finite space to another, while neural networks with two hidden layers can thus represent functions with any kind of shape [5].

Regarding the size of the hidden layers, as a general rule it is believed that the it should be between the size of the input layer and the size of the output layer. We chose 64 nodes for both of our layers since it affected the performance positively and it also did not impact the training time, which we made an effort to be around 10 minutes maximum. We also tried a model with one hidden layer of 500 nodes (and a vocabulary size of 6000 words). The training time was longer than an hour, which was not optimal for this assignment, and it also did not improve accuracy in any way, so we decided to proceed with that number of nodes, which seemed reasonable.

Finally, we would like to acknowledge that *the computations/simulations/[SIMILAR] were performed on resources provided by UNINETT Sigma2 - the National Infrastructure for High Performance Computing and Data Storage in Norway.*

References

[1] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference.* Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51–56.

[2] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project". In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning.* 2013, pp. 108–122.

[3] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[4] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32.* Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[5] Jeff Heaton. *Introduction to Neural Networks for Java, 2nd Edition.* 2nd. Heaton Research, Inc., 2008. ISBN: 1604390085.