Multi-label classification on the Reuters RCV1 corpus

Olli-Pekka Kinnunen
student number: 014729270
olli-pekka.kinnunen@helsinki.fi

Juhani Kivimäki
student number: 011835523
juhani.kivimaki@helsinki.fi

Jaakko Kuurne
student number: 013051509
jaakko.e.kuurne@helsinki.fi

1 Introduction

In this project we experimented with two different deep learning architectures in an effort to do multi-label classification on the Reuters corpus. The corpus consists of news stories from year 1997. Each news story is tagged with one or more codes, which correspond with certain news categories. Our task is to generate a deep learning algorithm, which learns to tag news stories given only the headline and text of those news stories.

2 Electronically-available resources

All of the code used in this project is publicly available at our GitHub repository

https://github.com/jjaakko/IDL_project

3 Models used

We built two models to tackle the classification challenge. One of these was a BERT model and the other an LSTM model.

3.1 BERT

BERT stands for Bidirectional Encoder Representations from Transformers (Devlin et al., 2018). BERT is a language representation model based on transformer architecture. BERT is using masked language model objective, that is, BERT is trained to predict randomly masked input tokens based on the context of the tokens. In addition, BERT is trained with the next sentence prediction task.

There are various flavors of pre-trained versions of BERT available. Pre-trained BERT models act as a feature extractors for language data, designed to be fine-tuned for downstream tasks.

BERT enables the model to differentiate between the possible multiple meanings of the same words in different contexts, in contrary to fixed embeddings such as Word2vec. BERT has a fixed size vocabulary (the size depends of the variant of BERT). Out of vocabulary words are handled via WordPiece embeddings, where the words are divided into subwords (Devlin et al., 2018).

3.2 LSTM

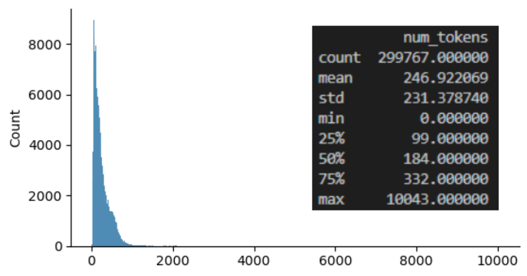
LSTM (Long Short-Term Memory) is a model that extends the vanilla RNN (Recurrent Neural Network) model by adding a long-term memory component to the model. This gives it the ability to take advantage of dependencies between observations in a sequence even if there is a long gap between the observations. LSTM also helps the vanishing gradient problem in RNNs, where each previous observation in the sequence has less and less impact on the parameters and thus most of the informational value comes from the last few observations in the sequence.

To achieve the "long-term memory", LSTM adds a few components to the vanilla RNN: a cell, an input gate, a forget gate and an output gate. The cell holds the long-term memory component and the different gates adjust the cell: input gate decides how the cell is updated, forget gate decides how much memory is retained, and output gate decides how the hidden layer is updated. All the gates are separate networks and use sigmoid activations to provide a multiplier between 0 to 1 to decide how much information is passed through the gates.

4 Exploratory data analysis and pre-processing

The corpus consisted of 299,773 Reuters news stories ranging from 1997-04-01 to 1997-08-19. Each news story was given as a single `.xml` file with headline, text, labels and other metadata. At first we created a script to go through each file and parse the relevant content into a `pandas DataFrame`. We used functionalities provided by the `lxml` library to do the parsing. Our parser gathered each news story as a list of sentences, with the headline as the first sentence in the list. These sentences form the input of our BERT model. We also performed tokenization, using `spaCy` library for this. These tokens were saved as lists to be used with the LSTM model. The parser also gathered the labels as lists. These lists were then processed into a vectorized (multi-label binarized) form.

Next, we took a look at the distribution of news lengths (number of tokens) and saw Zipf's law in action. This distribution along with some statistics is shown on figure 1. Since our BERT model can only process strings with at most 512 tokens, this information was crucial. About 90 % of all news were estimated to fall below this limit. Furthermore, we estimated that reasonable predictions could be made by taking only the first 512 tokens of those news, which over exceeded this limit.



```
       num_tokens
count  299767.000000
mean      246.922069
std       231.378740
min         0.000000
25%        99.000000
50%       184.000000
75%       332.000000
max     10043.000000
```

Figure 1: Distribution of the number of tokens

We also inspected the distribution of labels and found out that it was highly uneven. The distribution of labels is shown on figure 2. The highest frequency was with the label `CCAT` ($f = 137, 531$) and lowest with the label `GMIL` ($f = 4$). Furthermore, there were a total of 23 labels with a frequency of zero. We assumed that having such an uneven distribution would cause difficulties for our classifiers and tried to figure out ways to deal with this potential problem.

Our first intuition was to come up with ways to even out these differences. We discussed the possibility of using different techniques to oversample and/or undersample data or to augment or synthesize new data. Since we didn't have a clear understanding of how to do these in practice with multi-label data, we decided to study what kinds of structural dependencies and correlations the labels might have.

Further inquiry revealed a hierarchical structure between the labels. For instance, label `C15` (performance) denoted a subcategory of `CCAT` (corporate/industrial) and held subcategories `C151` (accounts/earnings) and `C152` (comment/forecast), with `C151` having yet another subcategory of it's own: `C1511` (annual results). On the highest level of this hierarchy were categorical labels `CCAT`, `ECAT` (economics, `GCAT` (government/social) and `MCAT` (markets). Comparing the frequencies of different labels revealed that these hierarchical category boundaries were neither strict nor mutually exclusive. For example, a news story labeled with `C15` might not have label `CCAT` and another news story might bear both the label `CCAT` and `ECAT`.

To make matters even more complicated, there were strong correlations between some of the labels from two different categories, most notably `E41` (employment/labour) and `GJOB` (labour issues). Some of these correlations are depicted in figure 3. All of these dependencies implied that it would be hard to target oversampling to only those labels with low frequencies, so we decided to forego that option. We also thought about augmenting the data by taking news stories with rare labels and producing new news stories by changing some words within those stories with their synonyms, but this approach was abandoned for the same reason as the oversampling approach.

We found a promising tool called MLSMOTE (Charte et al., 2015) to synthesize new data within a multi-label setting (an implementation of this algorithm can be found at: `https://github.com/niteshsukhwani/MLSMOTE`). In our case we could not directly apply this technique to our data, since there is no sense in interpolating new tokens between existing tokens. We assessed that we could, in theory, apply this technique after we had embedded our tokens (essentially creating new vectors in the embedding space with no real counterparts in the token space), but this would have been too complicated of a strategy to pull off within the scope of this project, so we decided not to implement it either.

One possible solution would have been to use some kind of a hierarchical model ensemble, such as the Binary Relevance method, the Label Powerset method or Classifier Chains. But even with all the redundant labels removed, our data still held 103 labels. This meant that we would have had to train 103 binary classifiers at minimum and we didn't have either the time or the resources to do that.

The final solution to alleviate the problem of the uneven distribution of the tokens, was to use the `pos_weights` parameter of the `BCEWithLogitsLoss()` function. The docu-

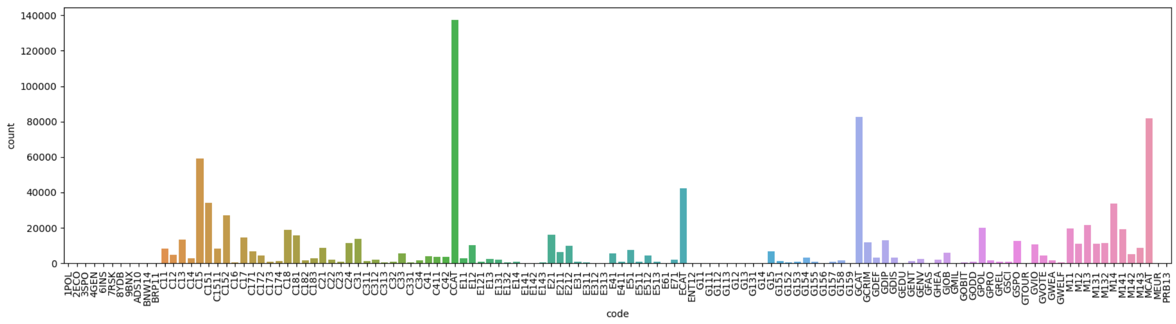


Figure 2: Distribution of the labels

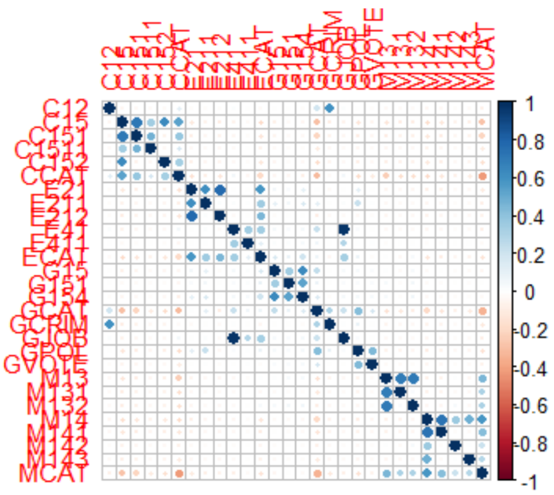


Figure 3: Correlations between selected labels

mentation for this function recommends using the number of negative samples divided by the number of positive samples as the weight of each sample. This way the algorithm gets punished more when making false predictions for rare labels than for common labels. These ”default” weights were also calculated in the pre-processing step to be used later when tuning our models.

The uneven distribution of labels also presented a problem on how to split the data into training, validation and test sets in a stratified manner. After some research we found an algorithm developed just for this purpose, called Iterative Stratification (Sechidis et al., 2011). This approach has been further developed (Szymański and Kajdanowicz, 2017) to take into account second-order relationships between labels. Both approaches are implemented in the `IterativeStratification` class of the `skmultilearn.model_selection` library. We used the second-order variant of this algorithm to split our data into training, validation and test sets with roughly 80 %, 10 % and 10 % of the samples in each set respectively and were positively surprised how even the distribution of labels was after the split. The relative frequencies of labels in each of the three sets can be seen in figure 4.

5 Building the models

In this part we will discuss how each model was built and how the hyperparameters were tuned.

5.1 BERT

We used BERT implementation from Hugginface. We chose the uncased BERT base model `https://huggingface.co/bert-base-uncased`, with a vocabulary size of 30 k tokens, for the multi-label classification task that was the objective of this project. The BASE model has twelve Transformer Blocks, twelve attention heads and the hidden size is 768 (Devlin et al., 2018).

Our model’s architecture is simple. We concatenate the news article title and body. BERT tokenizes the input, truncating to 512 tokens, and produces a vector of size 768, which is then fed in to a linear layer to obtain the classification result. We also looked into document BERT, which could handle longer sequences of texts but thought that 500 tokens should be enough for the model to identify the topics the news article is about. The fact that most of the news articles consist of less than 500 tokens further encouraged us to stick with basic BERT instead of document BERT.

We started the training with an approach where we calculated metrics on a small part of the validation set often. With this very first attempt we already got promising results.

However, we had some inefficiencies that made the training overall very time consuming. We increased batch size, made some parts of our code a bit more optimized performance-wise and also increased the interval of steps after which perfor-

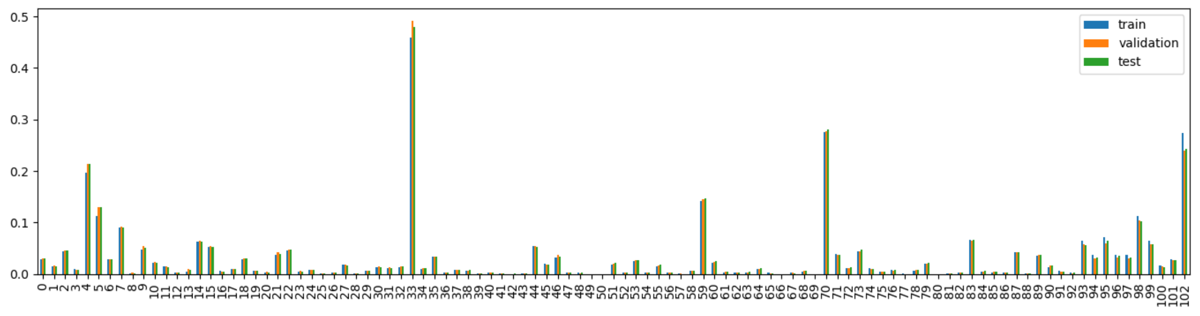


Figure 4: Relative frequencies of the labels in train, validation and test set

mance on the validation set was calculated. We settled for a validation set size if 6000. The intuition here is that 6000 samples may be enough as an indicator of the validation set performance and thus we have more data available for training.

Often times the best accuracy with the validation set was not at the end of each epoch. Thus we thought it would make sense to calculate validation set accuracy twice per epoch: approximately at the middle of the epoch and at the end of the epoch.

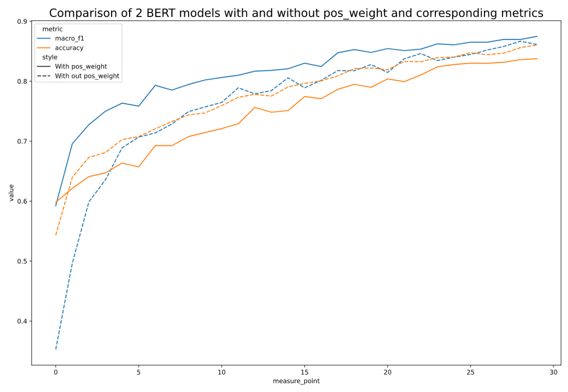


Figure 5: Comparison of two BERT models and corresponding two metrics

To mitigate the problems caused by very unbalanced distribution of labels we experimented with different approaches for the `pos_weight` parameter of `torch.nn.BCEWithLogitsLoss` function. The default recommendation by the function documentation did not work in our case. However, a more modest approach of using `torch.log1p` to decrease the weights yielded to a notable performance boost in terms of macro f1 score. This can be seen in Figure 5.

5.2 LSTM

5.2.1 Word embeddings

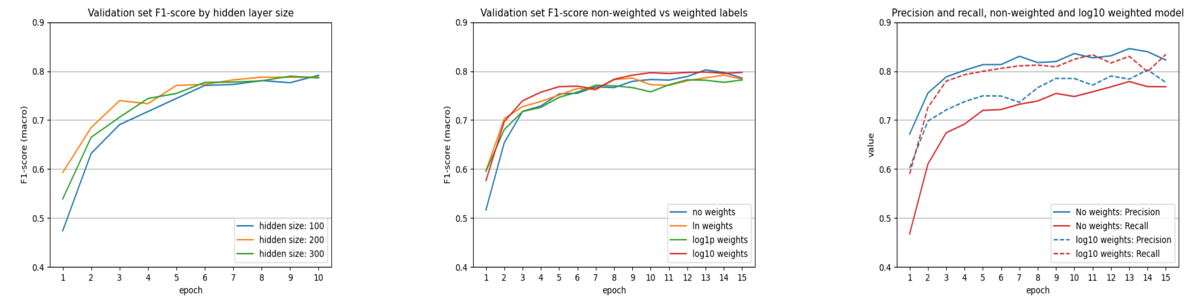
As LSTM requires a numerical representation of words, we decided to use word embeddings as our input layer to our LSTM model. Since the target data consists of news articles, we used the Google News Word2vec word embeddings that were trained on about 100 billion words in Google News. The embeddings can be found at `https://code.google.com/archive/p/word2vec/`. The pre-trained embeddings have a vocabulary size of 3 million words and phrases, which are embedded into a 300-dimensional real space.

Since the pre-trained embeddings covered only around 32% of the vocabulary in our data, we tried two different strategies with the embeddings:

- Drop all unknown words
- Set a common numeric tag to numbers and set a random vector for unknown words, sampled from a uniform random distribution with values between -1 and 1

After some testing, dropping the unknown words not only made the training faster, but also gave better results so we decided to go forward with the first method of word embedding.

We also made the observation that there were a significant amount of typos in the news stories, which clearly had a contribution for the number of Out of Vocabulary (OOV) tokens. We investigated whether it would be possible to do spelling correction for these tokens to decrease the number of unknown tokens in the inputs. First we tried to achieve this with a simple library `pyspellchecker`, which uses Levenshtein distance to find permutations within an edit distance of 2 from the OOV token. It then picks the most probable word to replace the OOV token with. This solution provided very poor results, since the vocabulary it used was rather narrow and it made no use of the word context. For example, it tried to correct almost all proper nouns, such as 'tobago' $\rightarrow$ 'tobacco' and 'guyana' $\rightarrow$ 'iguana'. Furthermore, it



(a) Validation set F1-score by hidden layer size

(b) Validation set F1-score for non-weighted and weighted labels

(c) Precision and recall for non-weighted and log10 weighted model

Figure 6: Hyperparameter tuning for LSTM

took quite a long time to run; doing spelling correction for a sample of 100 news stories took almost two minutes.

Our second try on spelling correction was with the library `contextualSpellCheck`. It utilizes a BERT model to do the spelling correction, which means it is able to take the context of the unknown word into consideration. It is used by adding it to a `spaCy` pipeline. That means it can utilize the Named Entity Recognition module of the pipeline. Although, this solution provided better results, it was painstakingly slow to run. Processing the same sample of 100 news stories took almost 10 minutes. Processing the whole corpus would thus have taken almost 500 hours. This was clearly an infeasible solution, so we just accepted the large number of OOV tokens.

5.2.2 Model structure

The starting point for our model structure was a bidirectional LSTM with 2 layers of 200 hidden nodes. Due to the long training times of the models (depending on the structure, from 15 minutes to over an hour per epoch), we restricted our structure search to a few parameters: hidden layer size and dropout.

At hidden layer size 400 the memory of our LSTM-dedicated GPU ran out, so we took test data from sizes 100, 200 and 300. The results are very similar on all sizes (figure 6a), so we opted for a hidden layer size of 200 to account for potentially adding dropout later. We experimented with dropout values of 0, 0.2, 0.4 and 0.6, where 0.4 seemed to get the best results in the long run. Thus, we set the dropout parameters as 0.4 for the final model.

5.2.3 Optimization

We started by using momentum SGD as our initial optimizer. It took a long time for the model to start improving the results, so we switched to AdamW (with learning rate 0.001 and weight decay 0.01), with which the model improved much faster. With SGD the F1-score was close to zero after many epochs, while with AdamW the F1-score tends to go over 0.5 already after the first epoch. We did some grid search for the learning rate and weight decay of AdamW, but in the end the “default” parameters of AdamW seemed to get at least as good or better results as any other combination we tried.

We found that batch size 16 worked best for our model training, so all the other hyperparameter tunings are made with a batch size of 16. The validation set F1-score tended to increase until around epoch 30-35 and started decreasing after. The final model was trained by taking all the data and training for 35 epochs to take into account the amount of added data.

5.2.4 Label weighting

In order to balance our model’s precision and recall, we tried different label weighting methods (figure 6b). At first we found that the natural logarithm and log1p (natural logarithm of 1 + given number) didn’t give better F1-score results than the non-weighted model. However, a base-10 logarithm ended up providing the best balance of precision and recall (figure 6c), while also consistently achieving somewhat better F1-scores than the unweighted model in a longer, 30 epoch training run.

6 Final model

After selecting the best BERT and LSTM models, we made predictions on our own test set. LSTM achieved a macro F1-score of 0.805, which BERT beat easily with a macro F1-score of 0.855. For this reason, we decided to use BERT as our prediction model for the competition test set with hidden labels.

We did make predictions on the competition test set with LSTM as well to make a sanity check of our predictions. We compared the predictions between BERT and LSTM: the predictions are quite similar in general, although the total agreement rate (where all of the labels for an article are exactly the same for both BERT and LSTM) was only around 67%. Even though the total agreement rate is not especially high, over 99% of the articles had at least one same label for both models and over 93% had at least two same labels.

7 Discussion

Had we have more time, we would've considered combining the two models. We might have experimented to use BERT just for tokenization or to generate the embeddings. This way, we could have made use of the contextualized way BERT generates these representations. These approaches were however out of the scope of this project.

We also considered creating a consensus model, which consists of multiple different models that together vote for the labels on each observation. The labels would then be decided by majority vote of all models. Since we only had two main models and the performance difference between BERT and LSTM was quite large, we decided to instead only use the stronger model for our final predictions.

References

F. Charte, Antonio Rivera Rivas, María José Del Jesus, and Francisco Herrera. 2015. MLSMOTE: Approaching imbalanced multilabel learning through synthetic instance generation. *Knowledge-Based Systems*, 89:385–397.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

K. Sechidis, G. Tsoumakas, and I. Vlahavas. 2011. On the stratification of multi-label data. In D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, editors, *Machine Learning and Knowledge Discovery in Databases. ECML PKDD 2011. Proceedings, Part III*, volume 6913. Springer, Berlin, Heidelberg.

Piotr Szymański and Tomasz Kajdanowicz. 2017. A network perspective on stratification of multi-label data. In *Proceedings of the First International Workshop on Learning with Imbalanced Domains: Theory and Applications*, volume 74 of *Proceedings of Machine Learning Research*, pages 22–35, ECML-PKDD, Skopje, Macedonia. PMLR.