Amarandei Alexandru, Mosor Andrei, Pelin Ioana, Smău Adrian

Team-Leader: *Cușmuliuc Ciprian Gabriel*

May 2021

CLEF2021 - CheckThat! Lab

Week 4 - Final Report

Introduction

This team was assigned with the task of implementing an algorithm for detecting Fake News in Python. During the last week, the 3Layer Team's work was focused on implementing an approach composed of TFIDF Vectorizations on multiple representations, each with its own weight, whilst the BERT Team's main aim was to research as much as possible into BERT Technology and prepare all the things necessary for the final version of the algorithm which is to be implemented during this week's run.

Methodology

Overview

For the remainder of the document the first sub-team will be referred to as the 3Layer Team and the second one as the BERT Team.

The 3Layer Team's main purpose was to improve upon the Three-Layered Approach using Hyper-Parameter Tuning, testing out a new Classifier, balancing the datasets, all while focusing on raising the *Macro Average* score, the contest's hierarchy being based on it.

The BERT Team's main purpose was to overcome the difficulties encountered during the Week 3 Run by labelling the outputs in a four-way manner, not in a binary manner, while also optimising the output of the algorithm by fine-tuning and taking advantage of the BERT predefined tokenizer and sequence classifier.

3Layer Team

Data Preparation

The Data Preparation stage consisted of:

- removing the **public_id** column from the CSV files received
- combining the two datasets
- combining the **title** and **text** columns
- removal of punctuation signs
- removal of stop-words
- removal of dashes and underscores
- lowercasing the text
- lemmatization of text

And, additionally, the random under-sampling of the False texts in the resulted combined dataset.

TFIDF Vectorization of the Three Layers

As during the last week's run, the three representations (each stored in their corresponding columns), namely: Clean Text (**clean_text**), POS Tags (**POS_text**) and Semantic Groups (**semantics_text**) were applied a TFIDF Vectorizer with n-grams ranging from 1 to 3 for the Clean Text and POS Tagging and unigrams for the Semantic Analysis.

Models

The following classifiers were used: NB (Naive-Bayes), KNN (K-Nearest Neighbours), RF (Random Forest) and GB (Gradient Boosting).

Similarly with the results of the last week, the Accuracy was good for each of the individual classifiers on each of the representations and will be shown in the **Results** section and elaborated on in the **Discussion** section below.

As expected though, the ensemble of the three yielded the best Accuracy-to-Macro-Average Ratio and this sub-team is content with the performance of the final model.

Hyper-Parameter Tuning

All the models, the only exception being the GB Classifier (due to its best performance being on the default parameters), were tuned and had their respective performances improved. The exact parameters will be listed in the Results Tables in the following section.

BERT Team

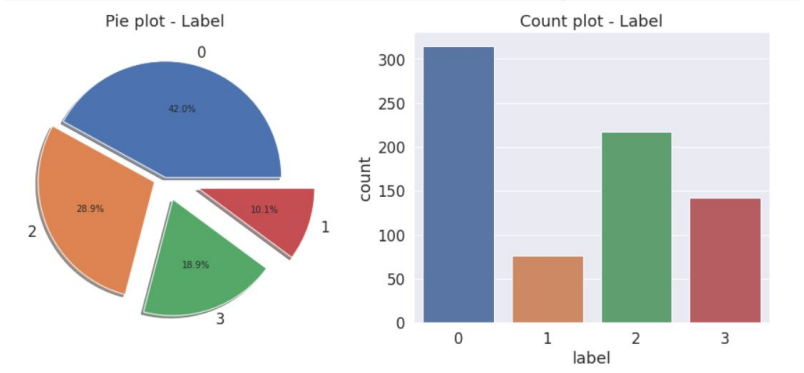
Data Preparation

After the necessary imports were made and the device was set, the team began data merging, shuffling and splitting. The steps taken are listed below:

- from the second batch given for training, the *False* values were eliminated (since the overall number of *False* values were overwhelming and the risk of overfitting the algorithm was encountered)
- the data from the second batch was then shuffled and then concatenated with the first batch, this way obtaining the training set
- the data was cleansed
- using the LabelEncoder, we encoded the values of the training set (0 – *False*, 1 – *Other*, 2 – *Partially False*, 3 - *True*), so that the values were eligible as parameters for the functions

Data Exploration

The process consisted of generating a pie plot and a count plot for the entire training set, as well as generating four different word-clouds, each for a different label.



Preprocessing

- merged the text and title columns
- eliminated the **public_id**
- defined a set of functions, including: float-to-str converter, punctuation signs removal, thorough text cleaner (including lemmatization) and string to word array
- applied the above mentioned functions to the text

Modelling, Tokenizing, Mapping and Masking

- imported the tokenizer (*bert-large-uncased* version provided by BERT – 24-layer, 1024 hidden dimensions, 16 attention heads, 336M parameters)
- formatted the train and test datasets in order to avoid a size mismatch when creating the TensorDataset
- used the **encode_plus** function provided by the tokenizer, to encode the text, truncate/pad it (which required some trial and error, since the platform's resources were limited – if the imported model supported up to 512 as max-length of the **input_ids**, we were only able to get 72, which was definitely a setback), added special tokens to it, and returned attention masks and tensors
- created the TensorDataset based on the training inputs

Fine-Tuning

The fine-tuning part of the algorithm required, by far, the most effort in getting the best results (or, rather, getting closer to those).

- started by setting a batch size for training
- created the training and validation data-loaders; while the train data-loader was obtained by sampling the training data-frame in random order, the validation data-loader was read sequentially (using SequentialSampler), since the order doesn't matter. (this generated **prediction_dataloader**)
- loaded a pre-trained BERT model (opted for the *bert-large-uncased* model) and we set the label number to 4

- set up the AdamW Optimizer (fine-tuned the learning rate as well as possible, 6e-6 yielding the best results)
- chose to go with 3 epochs
- taking all of these into account, the team obtained the scheduler (linear, without warmup steps), and set up a utility functions to keep track of the accuracy, F1 and elapsed time for the validation phase

Training

Everything being set up, the team could now begin training and evaluating. Within the Training phase, the team kept track of the loss of each epoch, the elapsed time, whilst the Validation phase showed us the accuracy, F1, validation loss and validation elapsed time.

For the training phase, the steps were as follows:

- for each epoch, in the training phase, the training batch was unpacked from the data-loader, whilst also copying each tensor to the device using the **to** method
- each batch contains three **pytorch** tensors: [0] – **input_ids**, [1] – **attention_masks**, [2] – **labels**
- as **pytorch** doesn't automatically clear the previously calculated gradients, it was done using **model.zero_grad()**
- furthermore, a forward pass is performed (models are evaluated on the current training batch); this returns a varying number of parameters depending on what arguments are given and what flags are set
- used it to get the loss and the logits (the model outputs prior to activation)
- afterwards, the training loss over all the batches is accumulated so that it is possible to get an average at the end (the 'loss' gotten earlier is a tensor containing a single value, so the **item()** function just returns the Python value within the tensor
- moreover, a backward pass is performed in order to calculate the gradients.

- the norm of the gradients is clipped to 1.0 – this prevents the 'exploding gradients' problem
- the parameters are updated before taking a step using the computed gradient – the optimizer dictates how the parameters are modified based on their gradients, the learning rate, etc.
- updating the learning rate using the scheduler
- computed the average loss over all the batches and calculated the elapsed time

Validation

For the validation phase, the steps were as follows:

- after the completion of each training epoch, the performance was measured on the validation set
- unpacked the training batch from the data-loader, as well as copied each tensor to the device using the **to** method (batch contains the same three **pytorch** tensors mentioned above)
- by specifying *with* ***torch.no_grad()***, it is settled that, going further with the forward pass, the algorithm does not bother with constructing the complete graph, since it is unnecessary
- **token_type_ids** are the same as the **segment_ids**, which differentiates sentence 1 and 2 in 2-sentence tasks
- during the forward pass, we calculate logit predictions (we get the **logits** output from our model)
- accumulated the validation loss
- moved the logits and labels to CPU
- computed the accuracy for the current batch of test sentences and accumulate them over all batches

Results

BERT Team

epoch	Training Loss	Valid. Loss	Valid. Accur.	Val_F1	Training Time	Validation Time
1	1.311	1.260	0.500	0.500	0:00:44	0:00:02
2	1.295	1.253	0.488	0.488	0:00:47	0:00:03
3	1.256	1.244	0.500	0.500	0:00:50	0:00:03

3Layer

TFIDF Vectorization on Cleaned Text

Classifier	Parameters	Accuracy	Macro Average
Multinomial Naive-Bayes	alpha = 0.0	0.57	0.48
K-Nearest Neighbours	p = 2, n_neighbors = 29, leaf_size = 45	0.61	0.41
Random Forest	n_estimators = 1000, max_features = 'sqrt', max_depth = 50, min_samples_split = 2, min_samples_leaf = 2	0.47	0.25
Gradient Boosting	n_estimators = 200	0.57	0.43
Note: Although the Accuracy is lower than previously, the Macro Average increased as desired.			

TFIDF Vectorization on POS Tags

Classifier	Parameters	Accuracy	Macro Average
Multinomial Naive-Bayes	alpha = 0.0	0.48	0.23
K-Nearest Neighbours	p = 1, n_neighbors = 25, leaf_size = 35	0.52	0.37
Random Forest	n_estimators = 400, max_features = 'sqrt', max_depth = 30, min_samples_split = 10, min_samples_leaf = 4	0.54	0.35
Gradient Boosting	n_estimators = 200	0.58	0.44

Note: This time around, the performance on the POS Tags representation has slightly decreased.

TFIDF Vectorization on Semantic Tags

Classifier	Parameters	Accuracy	Macro Average
Multinomial Naive-Bayes	alpha = 0.1	0.49	0.29
K-Nearest Neighbours	p = 2, n_neighbors = 27, leaf_size = 12	0.35	0.24
Random Forest	n_estimators = 200, max_features = 'sqrt', max_depth = 30, min_samples_split = 10, min_samples_leaf = 1	0.52	0.32
Gradient Boosting	n_estimators = 200	0.52	0.42

Note: The Semantic Tags yielded slightly better results than the POS Tags.

TFIDF Vectorization on All Three Representations, using a sparse matrix form

Classifier	Parameters	Accuracy	Macro Average
Multinomial Naive-Bayes	alpha = 0.0	0.62	0.45
K-Nearest Neighbours	p = 2, n_neighbors = 19, leaf_size = 6	0.51	0.34
Random Forest	n_estimators = 1000, max_features = 'auto', max_depth = 30, min_samples_split = 10, min_samples_leaf = 2	0.57	0.39
Gradient Boosting	n_estimators = 200	0.59	0.48

Note: The Gradient Boosting is the best performer amongst all classifiers.

Discussion

3Layer

The combination of the three representations, along with the tuned classifiers, yielded great results and has now concluded the work of this sub-team.

By degree of objectivity, the percentages (i.e. weights) used for the representations remained the same as last week's:

Weights

Representation	Weight
Clean Text TFIDF	0.5
POS Tagging	0.15
Semantic Analysis	0.35

Note: Multiple distributions were tried, however, the one above performed the best amongst all.

The discrepancy between the False texts and all the other labels was partly dealt-with, but still remained an important factor in the grand scheme of things.

The final model is using the Three-Layer approach, along with the Gradient Boosting Classifier and is to be used for predicting the test samples and then serialized for integration in the Fake News Platform that our faculty's team is working on developing.

BERT Team

Along the way of coming up with this final product, the team faced many challenges and learned a significant amount of new – otherwise, seemingly overwhelming at first – information, but the members were able to prevail and come up with an algorithm which yielded a great percentage, according to our predictions – something in the range of 40-50% accuracy.

On a 4-labeled output scheme (BERT does really well on binary labels – somewhere along the lines of 90-95% accuracy), the team considers that it is better than expected initially.

Serialization

Pickle

The serialization was done using Python's built-in persistence model, *Pickle*.

Integration with the other Models

A general *Model* class (from which all the specific model objects will be derived) has been created, containing the following methods:

- __init__(self, text) -> *constructor taking as parameter the text to be classified*
- download_dependencies(self) -> *downloads all the libraries and packages needed if they do not already exist on the system*
- process_text(self) -> *the same processing of text used during the training phase*
- load_model() -> *loads the pickled trained model from memory*
- predict(self) -> *classifies the text received as parameter during the instantiation of the object*

Max-Voting

The final classification is established using a max-voting approach, taking all five models' (from the other teams aside 3Layer and Bert) results and choosing the most common occurrence amongst the results array.