Exploring Romanian NER

Testing fine-tuned Spacy models vs custom trained BERT models

Iancu-Gabriel Onescu, Mihail Feraru,

I. Introduction

Extracting the main entities from a text helps us sort the unstructured data and detect the important information, which is crucial in most NLP tasks. Being of such a critical importance, the hackathon organized by the team Nitro Language Processing [1] had NER as its main theme.

We were given a dataset consisting of sentences in Romanian language and we had to predict the corresponding entity for each token in each sentence. The comparison performance metric was a weighted accuracy, so instead of computing $\frac{\text{correct predictions}}{\text{total examples}}$, it was:

$$\frac{\sum \frac{\text{correct predictions in class } i}{\text{total examples in class } i}}{\text{classes}}$$

In this document we will describe our approach for solving the challenge. In section II we will discuss about the content of the dataset and pre-processing steps we took, in section III will be presented a simple SpaCy network used as a base line for evaluation, section IV will describe our best method for predicting entity labels using a BERT model and in section V we will draw a few conclusions and lessons learned.

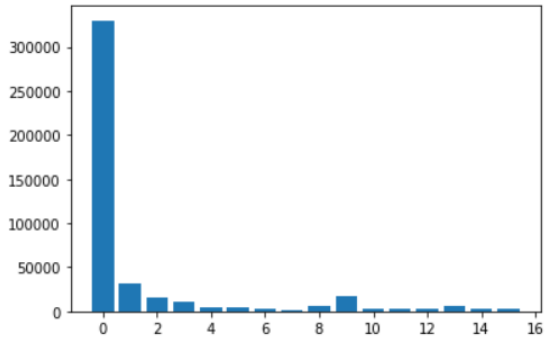
II. Dataset

The dataset consists of pre-tokenized sentences from multiple sources. We have no knowledge of what method was used to extract the tokens. For each token an entity label was provided. There were 16 unique labels present in the dataset corresponding to the ones presented in the Romanian Named Entity Corpus (RONEC) [1].

Data cleaning is an important step in any artificial intelligence project. Given so, we analyzed our input data and identified the following problems which could impact the accuracy of our models: inconsistent tokenization of certain words, UTF-8 characters not specific to Romanian and a highly unbalanced distribution of entity labels.

The inconsistent tokenization had no easily identifiable pattern, it occurred in seemingly random cases. For example, the word *Romani* resulted in two tokens: *r* and *omani*. This might have been caused by an inconsistent tokenizer that also processed sub-words, but it is just an assumption. Given the number of sentences containing invalid tokenizations weights less than 1% of the total, we decided to drop them entirely for simplicity.

In Romanian there is a set of characters which can be altered with diacritics (eg. *s* becomes *ș*). UTF-8 allows a wide range of characters from multiple languages and sometimes the Romanian character S-comma (*ș*) is confused with the Turkic S-cedilla (*ş*). This is the case in the dataset we received too, so we had to replace the invalid occurrences.



Looking at the tag distribution we notice that the set is severely unbalanced. Since tokens labels can be decided only given a context (ie. the sentence) stratified sampling is not an option. We concluded this problem cannot be addressed in the pre-processing step, thus a solution will be proposed in the next sections.

III. SpaCy Model

SpaCy [2] is a framework for NLP tasks supporting a plethora of languages and pre-trained models. We decided to use their pre-trained NER identification model as our base line evaluation. The model was trained on RONEC which we mentioned earlier.

The default model for NER tasks is a Transition-Based Parser (TBP) [2]. We did not dive into the implementation details, but presenting it briefly, a TBP is an approach for predicting structures using a series of state transitions. In our case, the structure of the sentence (ie. relations between entities) determines the entity label of each token. The model will try learn the best possible strategy (ie. a set of state transitions) which will provide correct labels to the tokens.

We compared the performance of the default model, the model retrained on our training set and the model retrained with dropout. We retrained the model for 10 epochs. We did not perform any further fine-tuning, because customizing SpaCy models is not very convenient. In Table I you can observe the comparison of the results on the test set.

IV. BERT Model

Since it's release in 2019 BERT has been the state of the art tokenizer for most NLP tasks, and Named Entity Recognition makes no exception. Thankfully we were lucky enough to find a model already pretrained on a Romanian corpus.[3]

[1] You can check the hackathon's page at https://www.nitronlp.rocks/home

[2] You can check the framework at https://spacy.io/.

[3] You can check the model at https://huggingface.co/dumitrescustefan/bert-base-romanian-cased-v1.

Model	Accuracy
TBP Pre-trained	50.17%
TBP Re-trained	66.75%
TBP Re-trained 0.2 dropout	67.13%

Table I: Comparison of SpaCy models results on the test set

A. Matching default tokenization with BERT tokens

One of the problems we faced was matching the already given tokenization with the tokens produced by BERT. Since irrelevant changes in punctuation are correctly ignored by BERT we were left to deal with the cases where the word was broken into multiple sub-tokens each with its own information. Moreover, BERT recognizes the correct entity for sub-token but uses a different tag that shows if the token is at the beginning, center or end of the named entity. For example, BERT tokenizer would split words like *românul* into two tokens: *român* (the lexeme) and *-ul* (the article), both considered to be a *person* entity.

To match the BERT output with our tokezenization and labeling we had to introduce a pre-processing step which expands the labels to all sub-tokens and a post-processing step which contracts them back. As in the previous example, if *românul* was marked as a *person* in our dataset, we would mark its two BERT sub-tokens as *person* too, and store a *reconstruction array*, meaning a mapping between tokens and the number of sub-tokens they generated. If we are not interested in labels because we are evaluating and not training, we would build only the reconstruction array. After collecting BERT predicitions we would use reconstruction array to match the outputs with the original tokens. The algorithms can be viewed in Listings 1 and 2.

Listing 1: Match input tokens (it) to BERT tokens (bt) and build a reconstruction array (reca). Propagate labels if they are available too (outlb).

```
expand(bt, it, lb?):
    i, j, outlb, reca = 1, 0, [...], [...]
    while i < bt.len and j < it.len:
        tk, eat, exp = it[j], 0, 0

        while eat < tk.len:
            if lb:
                outlb[i] = lb[j]
            eat += bt[i]
            i += 1
            exp += 1

        reca[j] = exp

    return outlb, reca
```

Listing 2: Recover the labels in the original format (outlb) from a reconstruction array (reca) and some BERT label predicitions (lb).

Model - trainable layers	Accuracy
BERT - final layer	67.638%
BERT - final 4 layers	70.245%
BERT - final 5 layers	70.155%
BERT - final 6 layers	70.330%
Best anterior with custom loss function	74.650%

Table II: Comparison of BERT models results on the test set

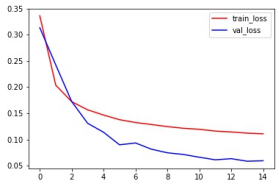


Figure 1: Loss for BERT with last 4 layers unfrozen

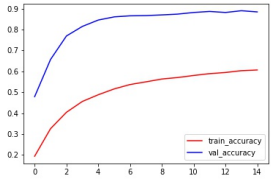


Figure 2: Accuracy for BERT with last 4 layers unfrozen

```
contract(lb, reca):
    outlb, i = [], 1

    for v in reca:
        if v == 0 or i >= lb.len:
            break

        outlb.append(lb[i])
        i += v

    return outlb
```

B. Model training

Since BERT comes with such a generous architecture it is easy to overfit the little training ammount of data that we have. So part of the fine tunning is discovering how many hidden layers should be retrained. Another important factor is discovering the sweet spot for the learning rate, and considering the regularization approaches to prevent the overfit. After much testing we decided that a low learning rate (1e-05 or 2e-05) together with decay (1e-06) and a consistent dropout(0.5) yields the best results.

One major breaktrough was changing the loss function which considers the classes to be balanced to a custom loss function for which we computed our own weights. This results in a 3-4% improvement in accuracy and a more stable model.

For determining the weights we computed the number of entries for the dominant class and divided it with the number of entries for each class, the results being the corresponding weights for each class. This way we also took into calculation the padding we added for the shorter samples and set it's weight to 0, intuitively telling the model that there is nothing to learn from those tokens.

In Table II you will see the results as computed onto the final test set and some of the graphs on the training set.

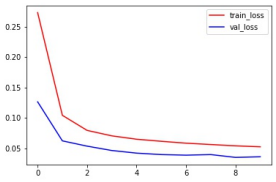


Figure 3: Loss for BERT with last 5 layers unfrozen

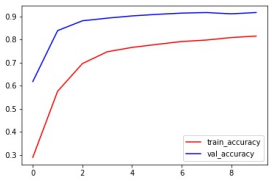


Figure 4: Accuracy for BERT with last 5 layers unfrozen

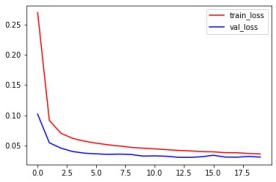


Figure 5: Loss for BERT with last 6 layers unfrozen

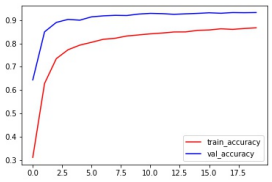


Figure 6: Accuracy for BERT with last 6 layers unfrozen

From the figures we see that the validation dataset was a little bit sparse, and easier to learn.

V. Conclusions and further improvements

In conclusion, the analysis conducted showed once again that BERT possesses a strong architecture. The challenge comes from finding a good pretrained model, if there is one available, and from fine tunning. We also learned that the computations require a lot of processing power and time. Interpreting the results of the models and deciding onto following steps is an art still in development.

For future improvements a wider range of hyperparameters can be tried and also different methods for computing the custom weights.

References

[1] Stefan Daniel Dumitrescu and Andrei-Marius Avram. *Introducing RONEC – the Romanian Named Entity Corpus*. 2019. DOI: 10.48550/ARXIV.1909.01247. URL: https://arxiv.org/abs/1909.01247.

[2] Guillaume Lample et al. *Neural Architectures for Named Entity Recognition*. 2016. DOI: 10.48550/ARXIV.1603.01360. URL: https://arxiv.org/abs/1603.01360.