NLP Homework 1: Event Detection
Keep it simple!

Leonardo Lavalle
1838492
Sapienza, University of Rome
lavalle.1838492@studenti.uniroma1.it

1 Introduction

The first homework for the 2023 NLP course is about Event Detection (ED). In particular we want to locate *event triggers* and classifying them using the BIO format. The task has a lot of possible applications, including Narrative Understanding where recognizing events is essential to comprehend the plot of a story or text. The dataset, as expected, is unbalanced (Figure 1) and for this reason a model that can analyze sequences and make predictions based on the context is necessary. All the choices made for the homework have been guided by the *unbalanced* and the *low-context* (short sentences) nature of the problem.

2 Methods

I've tried to follow all the best practices in Deep Learning and NLP (e.g. pre-trained word embeddings) and employed as a starting architecture the adviced one formed by an *embedding layer* (section 2.2), a *sequence* encoder (section 2.3) and a *classifier* on top (section 2.4). But first of all, let's dive into the preprocessing part, maybe the most time consuming step in a NLP pipeline, where the input tokens are transformed to be fed into the prediction system.

2.1 Preprocessing

Look at the data! Without a quality dataset there's no way of achieving good results. Since the context setting is low and for each sentence there are no many words to help the model to predict the events, we need to primarly *clean* the tokens from any dirtiness and create new and smart ones, and secondly to build a *vocabulary* by generating as few OOV words as possible because any word is precious in a task like this. The dataset was quite clean, but some cleaning operations were required. Basically at this step I had two different aims: simple cleaning of tokens from weird symbols and/or punctuations; definition of new tokens (i.e. IGNORE and NUMBER) hoping to help the model to make good predictions based on their position and context. I choose GloVe [3] as pre-trained word embedding, specifically the uncased version with vocabulary size of 400K. The 300 word embedding size performed better than 50, 100 and 200 on all my experiments. Therefore, the starting vocabulary will be the one implicitly defined by GloVe. But given that I introduced two new tokens and I didn't want to lose any information that could be used to benefit events prediction, I extended the vocabulary with them plus the most frequent OOV words from the training set. Finally, I decided to filter out some sentences. As we can see in Figure 2 on the left, there are some "sentence outliers" which deviates from the majority of data. This kind of training samples have to be avoided in order to not make the model learn other type of distribution. I filtered using lower and upper length bounds and by ignoring sentences with no labeled events. In the right portion of Figure 2 it can be seen how good the histogram shape is looking after the filtering.
Following the same *sliding window* approach seen in class [1], after looking at the plot in Figure 2, I set the default window size to 40. At the same time, I encoded the inputs according to the vocabulary's keys. Thanks to them the model is able to assign to each token its actual embedding.

2.2 Embedding Layer

The embedding layer is merely a lookup table for word embeddings, nothing more. I added the possibility of splitting the embedding layer into two: one related to the GloVe embedding and the other to the tokens with which I extended the vocabulary. In this way, at a certain epoch, I can stop the training for the pretrained embeddings while letting the others (that need more time to be learned) continue

the training. I used this strategy throughout all my experiments.

2.3 Sequence Encoder

LSTM architecture [2] was the immediate choice. I didn't try other RNN-like architectures as GRU for example because honestly since the beginning I achieved good results and therefore I decided to focus on hyperparameters tuning by employing the *Sweep* feature of WandB (Figure 3). Without LSTM, it would have been impossible to encode information about the context of words. With respect to vanilla RNNs, it overcomes the vanishing gradient problem and it's most robust to long term dependencies between words. One of its greatest power is the possibility of encoding the sequence both from left-to-right and from right-to-left to further enhance contextualization (Bi-LSTMs). To increase the performance I played with its hidden dimension and its number of stacked layers.

2.4 Classifier

A unique linear layer has been used as classifier. I also implemented a MLP classifier with three layers and ReLu activation functions, but with poor results. Therefore I preferred the simplest single layer version which performed better and has moreover less weights to train.

2.5 Training Setup

I've used WandB as logging tool, which allowed me to generate plots of performance metrics for free. I trained all my models from scratch utilizing *Adam* as optimizer and *Cross-Entropy Loss* as loss function. To help the optimization procedure, I employed the *ReduceLROnPlateau* learning rate scheduler which reduces it when the validation loss is not decreasing anymore. Techniques as weight decay and early stopping have been exploited to avoid overfitting. Aside from the "separate" fine-tuning strategy of embeddings explained in 2.2, I decided to implement a *"mix windows strategy"*. Instead of having a fixed pair {window size, window shift}, I change (each 2 epochs) at training time their values producing a complete different dataset version each time. A greater generalization power is transferred to the model. Thinking about that, is not only a matter of words dependencies length, but it is moreover a matter of the different and heterogeneous contexts (w.r.t. a token) the system sees each training step to make predictions.

3 Results

Since the beginning I reached very good results with a quite basic architecture. Maybe it was the preprocessing or the training pipeline. The fact is that the *baseline* from which I started had no pretrained embeddings, only a single LSTM layer, not bidirectional and with no dropout at all. And it reaches 62.5% of macro F1-score (Table 1). It's undeniable the positive impact of the addition of word embeddings to the baseline (+7%). The plots in Figure 4 show how the learning process of the instance with 5 LSTM layers is slower than the others (more weights) but also more effective (71.6%). Starting from these basic and successful ingredients I engaged an hyperparameters search from which I selected 537 as random seed (a convenient weights initialization can increase performances) and 0.4 as dropout value. Indeed, in Table 2, *BEST base* begins to have an excellent F1-score (72%). Reducing the LSTM hidden dimensions from 512 to 440 not only reduces the model weights but also improves the performance. Taking a look to the outcomes of the my first experiments I noticed a difficulty of the system to correctly predict the "I" type labels. My intuitions were two: (i) sum *positional encodings*, as it happens in Transformer models [4], to word embeddings in order to convey some sort of knowledge about token's position in the sequence; (ii) leverage somehow *POS tags* information because I noticed that most of the time the event triggers are verbs (VERB). Unfortunately, both these ideas didn't work as expected. As we can see in the plots of Figure 5, the two solutions both overfits fastly and don't over-perform the other models. I tried different configurations aiming to improve the POS embedding version (Figure 6), but I couldn't. **Keep it simple!** All the simplest models worked better. For this reason I decided to keep the model as it was and tuning the last hyperparameters. I ended up with a batch size of 512, a dropout value of 0.5 and increasing the learning rate from 2e-4 to 1e-3. The achieved macro F1-score is **72.42%**. Its confusion matrix (Figure 7) shows us that when an event is mispredicted is often predicted as an "O" label and also how it badly behaves with "I-SENTIMENT" and "I-POSSESSION" with respect to the rest. It's quite clear how the primary cause of this discrepancy lies in the unbalanced train set (only 2 and 1 tokens are labeled with these types). Coincidentally (Figure 1), the two least represented classes are the ones that exhibit the poorest performance.

Images and Tables

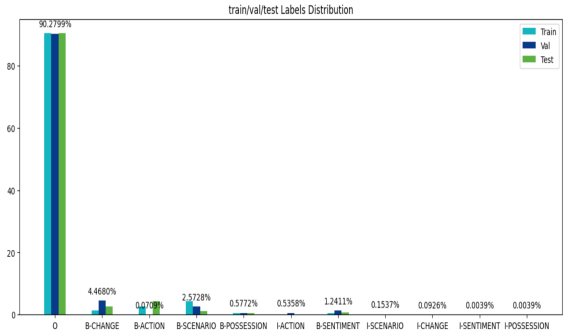

Figure 1: Plot of labels distribution of *train/val/test* datasets. All the three splits are equally distributed: the "O" labels, that indicates *"no event"*, are the most numerous.

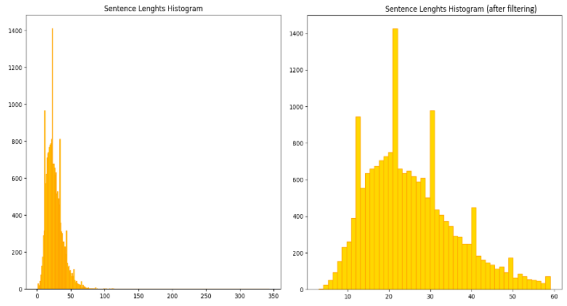

Figure 2: Histogram of training sentences length. On the left the one before the sentence filtering and on the right the one after. The "sentence outliers" are the points which contribute to the long and thin tail of the left histogram. In fact, the mean value is 25, but we have the maximum sentence length that reaches 343. After the filtering, only 1778 sentences (9%) are excluded.

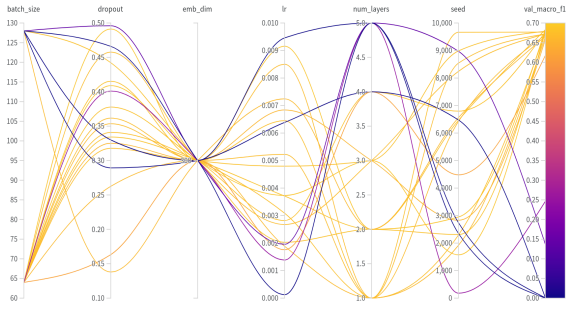

Figure 3: WandB Sweep example run where I was searching for the best configuration of some hyperparameters. It is possible to see how eye-catching this visualization tool can be.

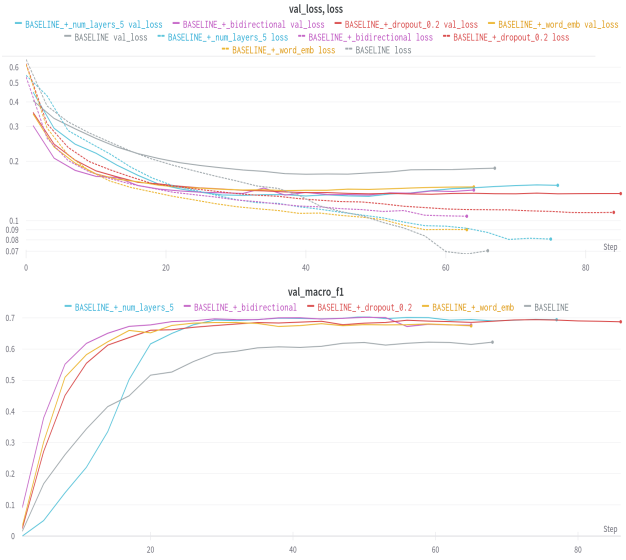

Figure 4: Training and validation losses (on top) and validation macro F1-scores (on bottom) plots of the incremental *baseline* improvements. The plots are taken from WandB. On the *x* axis there are the number of training steps. The *y* scale for the losses is logarithmic.

Figure 5: Plots of train/val loss (on the left) and val macro F1-score (on the right) of my *best* performing model and the two trial versions with the positional encoding and the POS tags additions.

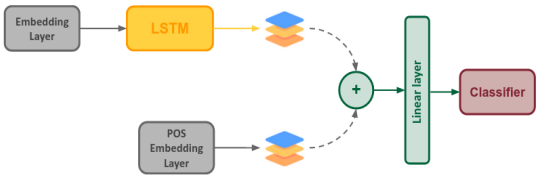

Figure 6: Qualitative representation of the *POS embedding version* architecture. It can be observed how the POS informations are leveraged by the model. POS tags embeddings are computed and then summed to the output of the LSTM layer. Finally the result is fed it to a Linear layer which learns the best way of combining them.

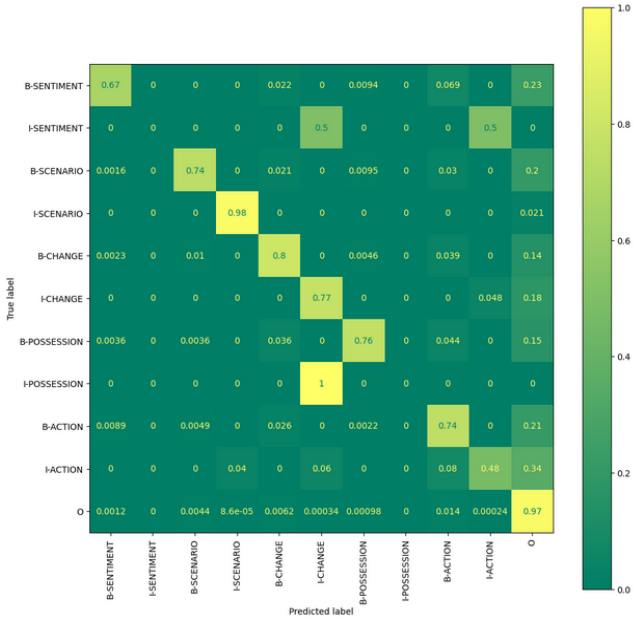

Figure 7: Confusion Matrix of my *best* model computed on *test* set.

Model	macro F1-score
Baseline	0.6256
+ word embeddings	0.6941 *(+0.0685)*
+ 0.2 dropout	0.7086 *(+0.0145)*
+ bidirectional	0.6963 *(-0.0123)*
+ 5 LSTM layers	**0.7158** *(+0.0195)*

Table 1: Improvements of *macro F1-score* after each change of architectures with respect to the *baseline* on the *test* set. There's an overall improvement of almost 10%.

Model	macro F1-score
BEST base	0.7202
BEST (440 hidden dim)	0.7215
BEST (positional encode)	0.7188
BEST (POS embed)	0.7198
BEST	**0.7242**

Table 2: Results of *macro F1-score* of my *best* models on *test* set. The last model is the one which has performed better over all my experiments and indeed the one choosen to be evaluated for the homework.

References

[1] Sapienza NLP research group. 2023. Notebook 4: Pos tagging.

[2] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9:1735–80.

[3] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need.