Named Entity Recognition (NER)

Delivery 2

Natural Language Processing

Authors

Alejandro Astruc

Joel Dieguez

Alba García

Clàudia Valverde

June 2024

Contents

1 Introduction

In the vast domain of Natural Language Processing (NLP), Named Entity Recognition (NER) stands out as a crucial sub-task that focuses on identifying and categorizing specific entities in a given text into predefined classes. These entities can range from names of people and organizations to locations, dates, percentages, and more. It also plays a pivotal role in information extraction, a process that transforms unstructured data into a structured form, suitable for further analysis or feeding into databases.

In this work we are asked to perform NER on a given dataset (section 2) using three different approaches. i) implementing the provided code for a Structured Perceptron (subsection 3.1) ii) enhancing the baseline perceptron model by adding features (subsubsection 3.1.3), and iii) exploring a deep learning approach (subsection 3.2), which in our case we have investigated Bi-LSTM architecture and a pre-trained BERT model for the NER task. Our findings are summarised in section 4.

The group members are Alejandro, Joel, Alba, and Clàudia. Alba focused on data analysis and explained the theoretical framework of the Structured Perceptron. Alejandro trained both Structured Perceptron approaches and developed the evaluation metrics function for them. Clàudia created the extended features for the second approach of the Structured Perceptron and trained the Bi-LSTM model. Lastly, Joel concentrated on leveraging a pre-trained BERT model for the NER task.

Our code can be found in the following github repository: `https://github.com/45truc/NER_task.git`.

2 Data

Let us start by looking at our data, we have three data sets: `train_data_ner`, `test_data_ner` and `tiny_test`, which are the train dataset, the test dataset and a small dataset, respectively. The train dataset contains 38,366 sentences, the test dataset has 38,367 sentences and 13 sentences for the tiny test. We will train and evaluate our models with the train and test datasets, and we will see how the different predictions of the models for the tiny test dataset.

All three dataset come with the tag for their entities. As we have explained entities can range from names of people and organizations to locations, dates, percentages, and more. In particular, our train and test data both contain eight different entities defined in Table 1

Entity Tag	**Description**
geo	Indicates a geographic location (e.g., mountains, rivers, regions)
gpe	Indicates a geopolitical entity (e.g., countries, cities)
tim	Indicates a time expression (e.g., dates, times)
org	Indicates an organization (e.g., companies, institutions)
per	Indicates a person's name
art	Indicates an artifact (e.g., titles of books, movies)
nat	Indicates a natural object (e.g., species, plants)
eve	Indicates an event (e.g., festivals, wars)

Table 1: Entity Tags and their descriptions

The way the tags are formatted in the data is using the IOB format (Inside-Outside-

Beginning). This is a common tagging scheme used for identifying the boundaries and labels of entities within a text. It provides a clear and unambiguous way to annotate sequences by assigning to each token in the sentence one of the following labels:

- **B-<tag>**: Indicates the beginning of a named entity, it is used for the first word of the entity. For instance, in "Southern California", "Southern" would be tagged as B-geo.
- **I-<tag>**: Indicates that the word is inside a named entity, it is used for words that are part of an entity but are not the first word. For instance, "California" in "Southern California" would be tagged as I-geo.
- **O**: Indicates that the word does not belong to any named entity. Most words in a sentence typically fall under this category.

We can take an example from the tiny test data set to show how this system works, see table 2.

Tokens	Robin	does	not	want	to	go	to	Saudi	Arabia	.
Tag	B-per	O	O	O	O	O	O	B-geo	I-geo	O

Table 2: Example sentence in IOB format

Finally, to complete our analysis, we have studied what are the most common tags in our datasets (train and test), see Figure 1. In general, we can observe that the most common tags are the ones for geographic locations (geo), for time expressions (tim), for organizations (org) and for peoples' names (per). In general, observe that both train and test data follow the same distribution of tags.

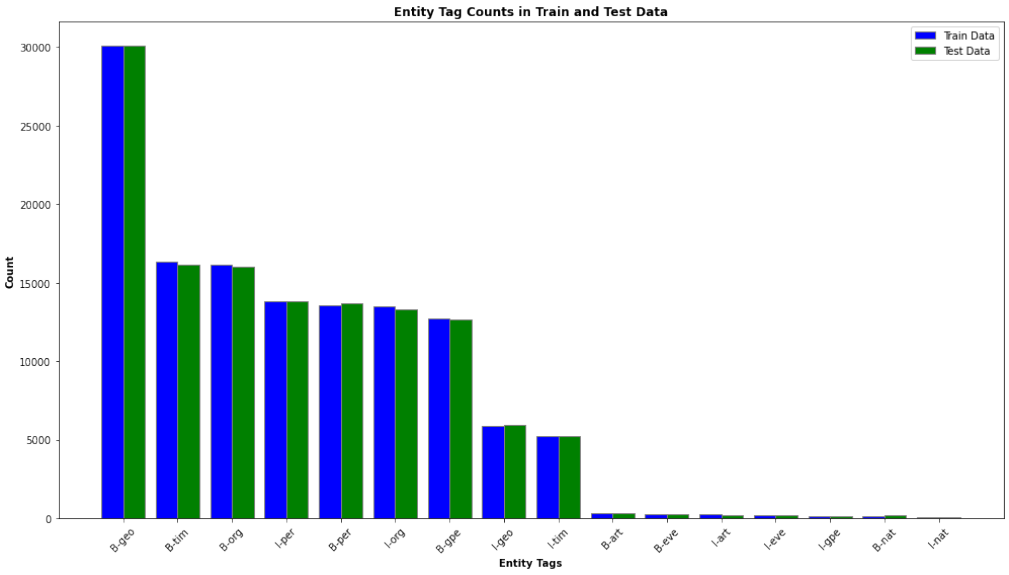


Figure 1: Frequency of entity tags

3 Models

To evaluate our models, we divided our work between machine learning (ML) and deep learning (DL) approaches. The ML approach consists of a structured perceptron with handcrafted features. We created two different models: one trained with fewer, simpler features which replicate the same features as the HMM and another with an extended set of handcrafted features. For the DL-based approach we have also investigated 2 different models i) a bi-LSTM architecture and ii) a pre-trained BERT.

3.1 Structured Perceptron

3.1.1 Theoretical model explanation

Let us start by explaining the theoretical foundations of the Structured Perceptron algorithm, which are the probabilistic Hidden Markov Models (HMM).

Hidden Markov Models

Hidden Markov Models are Markov models in which the observations are dependent on a latent (or "hidden") Markov process X. An HMM requires that there be an observable process Y whose outcomes depend on the outcomes of X in a known way. Since X cannot be observed directly, the goal is to learn about state of X by observing Y, that is inferring the most likely sequence of hidden states given the observed sequence of outcomes [1]. These probabilities also depend on the initial state and can be written as follows:

$$\begin{aligned} P(X,Y) = P_{\text{init}}(y_1 \mid \text{start}) \cdot \left(\prod_{i=1}^{N-1} P_{\text{trans}}(y_{i+1} \mid y_i) \right) \\ \cdot P_{\text{final}}(\text{stop} \mid y_N) \cdot \left(\prod_{i=1}^{N} P_{\text{emiss}}(x_i \mid y_i) \right) \end{aligned} \tag{1}$$

where $X = x_1 \dots x_N$ and $Y = y_1, \dots y_N$.

The previous formula tells us how to compute the probability of a pair if we have the initial, transition, final and emission probabilities, but how do we find such probabilities? First, let us explain what these probabilities mean:

- Initial probability (P_{init}): is the probability of starting in a particular state.
- Transition probability (P_{trans}): is the probability of transitioning from one state to another state ($P_{\text{trans}}(y_{i+1} \mid y_i)$).
- Final probability (P_{final}): is the probability of transitioning from the last state to the end of the sequence (the stop state).
- Emission probability (P_{emiss}): is the probability of an observed output being generated from a hidden state ($P_{\text{emiss}}(x_i \mid y_i)$).

In our case, we are dealing with a problem of sequential labelling, that is finding entities in a sentence, for that we are working with the following set up:

- A set of words $\Sigma := \{w_1, \ldots, w_J\}$.
- A set of labels $\Lambda := \{c_1, \ldots, c_K\}$.

In this context, a sentence is an element from Σ^*, which is the Kleene closure of Σ. The Kleene closure of a set Σ is defined as the set containing all possible strings of arbitrary length made up with elements in Σ.

Now, the previous probabilities are defined as in Table 3.

Parameter	**#Floats**
$P_{\text{init}}(c \mid \text{start})$	$\lvert\Lambda\rvert$
$P_{\text{trans}}(c \mid \hat{c})$	$\lvert\Lambda\rvert^2$
$P_{\text{final}}(\text{stop} \mid c)$	$\lvert\Lambda\rvert$
$P_{\text{emiss}}(w \mid c)$	$\lvert\Sigma\rvert \cdot \lvert\Lambda\rvert$

Table 3: HMM Parameters

It is important to notice that HMM are based on the following assumptions:

- **Independence of Previous States:** The probability of being in a given state at position i only depends on the state of the previous position $i-1$ (this defines a first-order Markov chain).

$$P(Y_i = y_i \mid Y_{i-1} = y_{i-1}, Y_{i-2} = y_{i-2}, \ldots, Y_1 = y_1) \approx P(Y_i = y_i \mid Y_{i-1} = y_{i-1})$$

- **Homogeneous Transition:** The probability of making a transition from state c_l to state c_k is independent of the particular sequence position.

$$P(Y_i = c_k \mid Y_{i-1} = c_l) \approx P(Y_t = c_k \mid Y_{t-1} = c_l)$$

- **Observation Independence:** The probability of observing $X_i = x_i$ at position i is fully determined by the state Y_i at that position. The probability is also independent of the particular position.

$$P(X_i = x_i \mid Y_1 = y_1, \ldots, Y_i = y_i, \ldots, Y_N = y_N) \approx P(X_i = x_i \mid Y_i = y_i)$$

$$P(X_i = w_j \mid Y_i = c_k) \approx P(X_t = w_j \mid Y_t = c_k)$$

Given a HMM we can use other algorithm to decode its information, as we have seen in class, two of these algorithms are the Viterbi algorithm and the Forward-Backward algorithm. The Viterbi algorithm is used for finding the most likely sequence of hidden states that best explains a sequence of observations. The forward-backward algorithm is used for estimating the posterior probabilities of hidden states given observations. Next, we will explain the Viterbi algorithm since it is the one used in the practical implementation.

The Viterbi algorithm

As we have explained, the Viterbi algorithm is a dynamic programming algorithm for obtaining the maximum a posteriori probability estimate of the most likely sequence of hidden states—called the Viterbi path—that results in a sequence of observed events

[2]. After estimating model probabilities (as seen in 1), sequence labels can be obtained using the Viterbi decoding:

$$\mathbf{y}^* = \arg\max_{\mathbf{y}\in\Lambda^N} P(Y_1 = y_1, \ldots, Y_N = y_N \mid X_1 = x_1, \ldots, X_N = x_N)$$

The steps of the Viterbi algorithm simplified are:

- Initialization: Set up initial probabilities.
- Recursion: Update probabilities based on transitions and emissions.
- Termination: Determine the most probable final state.
- Backtracking: Trace back to find the most probable sequence of states.

The Structured Perceptron

Finally, with all this information, we can build the Structured Perceptron algorithm.

Let us go back to how the probabilities of a HMM are defined. Given the equation 1 we can defined the *log*-probability by taking the logarithm of the expression, with that, we obtain the following:

$$\begin{aligned}\log P(X,Y) = \log P_{\text{init}}(y_1 \mid \text{start}) + \sum_{i=1}^{N-1} \log P_{\text{trans}}(y_{i+1} \mid y_i) \\ + \log P_{\text{final}}(\text{stop} \mid y_N) + \sum_{i=1}^{N} \log P_{\text{emiss}}(x_i \mid y_i)\end{aligned} \tag{2}$$

Now, introducing indicator functions ($\mathbb{I}_{[e]}$) and rearranging the terms, we obtain the following expression:

$$\begin{aligned}\log P(X,Y) = &\sum_y \log P_{\text{init}}(y \mid \text{start}) \mathbb{I}_{[y_1=y]} \\ &+ \sum_y \sum_{y'} \log P_{\text{trans}}(y' \mid y) \sum_i \mathbb{I}_{[y_i=y, y_{i+1}=y']} \\ &+ \sum_y \log P_{\text{final}}(\text{stop} \mid y) \mathbb{I}_{[y_N=y]} \\ &+ \sum_x \sum_y \log P_{\text{emiss}}(x \mid y) \sum_i \mathbb{I}_{[x_i=x, y_i=y]}\end{aligned} \tag{3}$$

Observe that some terms can be interpreted as counts:

- $\sum_i \mathbb{I}_{[y_i=y, y_{i+1}=y']}$: Number of times transition $y \to y'$ occurs in Y.
- $\sum_i \mathbb{I}_{[x_i=x, y_i=y]}$: Number of times x is emitted from state y in (X, Y).

Resulting in the following expression:

$$\begin{aligned}\log P(X,Y) = &\sum_y \log P_{\text{init}}(y \mid \text{start}) \mathbb{I}_{[y_1=y]} \\ &+ \sum_y \sum_{y'} \log P_{\text{trans}}(y' \mid y) \cdot C_Y(y,y') \\ &+ \sum_y \log P_{\text{final}}(\text{stop} \mid y) \mathbb{I}_{[y_N=y]} \\ &+ \sum_x \sum_y \log P_{\text{emiss}}(x \mid y) \cdot C_{X,Y}(x,y)\end{aligned} \tag{4}$$

With that we have the following matrix expression:

$$\log P(X,Y) = \begin{bmatrix} \log P_{\text{init}}(\cdot) \\ \text{vec}(\log P_{\text{trans}}(\cdot)) \\ \text{vec}(\log P_{\text{emiss}}(\cdot)) \\ \log P_{\text{final}}(\cdot) \end{bmatrix}^\top \begin{bmatrix} \mathbf{e}_{y_1} \\ \text{vec}(\mathbf{C}_Y) \\ \text{vec}(\mathbf{C}_{X,Y}) \\ \mathbf{e}_{y_N} \end{bmatrix} \tag{5}$$

where $\mathbf{e}_{y_i}$ represents the y_i-th standard basis vector and vec concatenates matrix columns into a single vector.

From equation 5 observe that the left vector contains the model parameters and does not depend on X or Y. The right vector describes X and Y and knows nothing about the model. With these observation we can reinterpret the left vector as a weight vector W and the right vector as a feature vector $\Phi(X,Y)$, resulting a linear model for the *log*-probability.

$$logP(X,Y) = W^T \Phi(X,Y) \tag{6}$$

In this new setting, we can still use the Viterbi algorithm as long as we can map the feature vector onto each edge. In the same way as these features are defined, we can also defined more general features beyond those used in HMMs. The Structured Perceptron algorithm is a way of learning the weights.

The Structured Perceptron algorithm is an algorithm based on the Perceptron algorithm that works with structured data (e.g. sequences) opposed to discrete or real values. In this setting, we define X and Y to be the sequences of words and labels, respectively; and $\Phi(X,Y)$ to be a sum of HMM-like features, $\Phi(y,y',X,i)$. The way of mapping the HMM distributions (Equation 1) to the features can be seen in Table 4.

Conditions	**Features**
$y = \text{start}, y' = c, i = 1$	Initial features
$y = c, y' = \tilde{c}$	Transition features
$y' = c, i = N$	Final features
$y' = c, X = w$	Emission features

Table 4: Features for Structured Perceptron

With this construction, we have built the Structured Perceptron algorithm.

Let us recap on the **overall process of the algorithm**. The Structured Perceptron is a discriminative learning algorithm used for structured prediction task, where the output is a structured object like a sequence. The algorithm can be seen as a linear classification

algorithm that learns a weight vector to make predictions based on a feature vector. The linear model is based in Hidden Markov Models and the way the probabilities for each state are defined. The HMM also defines the way the features are considered, but we have seen that adding more general, custom features is possible. Like in a regular Perceptron algorithm, the current weights are used to predict the most likely structure, and are then adjusted and updated depending of the error between the prediction and the real structure.

Finally, we want to consider what would happen if the model **encounters a word during testing that was not present in the training data**. In this case, the model would not have learned specific parameters (emission probabilities) for that word and would assigns probabilities based on a default or generic behavior. This can have an impact of prediction, given that the model does not know the unknown word relates to the labels, affecting the performance of the model specially if the distribution of unknown words is significantly different from the distribution of known words. To mitigate this impact, we can introduce new features to help classifying correctly the unseen words.

3.1.2 Baseline Structured Perceptron

In this section we have focused on reproducing the Structured Perceptron code provided by the professor. In this case we are applying it in a different dataset thus, in order to be able to correctly read the data we introduced a new class `NerCorpus()`, defined in `utils.py`. This class has a method for reading the data and incorporating new items into the corpus.

We will use the given perceptron and features as baseline for comparison between the models we will study.

3.1.3 Structured Perceptron with Extended Features

As noted by the teacher, the easier way to add features is by redefining the `add_emission_features()` method from the `ID_features()` class.

These additional features should enhance the Structured Perceptron's ability to correctly identify the entities in the sequence. We decided to add new features based on different types i) orthographic and contextual features, ii) word position features and iii) common prefixes and suffixes.

1. **Orthographic and contextual features**

    - *Orthographic Features:* Adds features based on the case of the word (title case, upper case, lower case) and whether the word is a digit.
    - *Stopwords:* Adds a feature if the word is in the predefined set of stopwords.
    - *Prepositions:* Adds a feature if the word is in the predefined set of prepositions.
    - *Non-ASCII Characters:* Adds a feature if the word contains any non-ASCII characters.
    - *Punctuation:* Adds a feature if the word contains any punctuation characters.
    - *Hyphens:* Adds a feature if the word contains hyphens.
    - *Word Length:* The length of the word can sometimes be indicative of certain types of entities (e.g., organizations often have longer names).

2. **Word position features**

   This indicates whether the word is at the beginning, middle, or end of the sequence.

3. **Common prefixes and suffixes**

   Each entity type has its own set of prefixes and suffixes. When the word in the sequence starts with any of the specified prefixes or ends with any of the specified suffixes for that entity type, a corresponding feature is added. In table 1 it can be seen the prefixes and suffixes selected for each entity.

Entity Type	**Common Prefixes**	**Common Suffixes**
Geographic Location (geo)	Mount, Lake, River, San, New	land, ville, city, town, berg
Geopolitical Entity (gpe)	Republ, King, United, Sultan, Comm	land, stan, ica, nia, tan
Time Expression (tim)	Jan, Feb, Mar, Apr, Mon, Tues	day, month, year, week, hour
Organization (org)	Corp, Univ, Assoc, Org, Comp	Inc, Ltd, Corp, Group, Co
Person's Name (per)	Mr, Ms, Dr, Prof, Sir	son, man, berg, stein, ski
Artifact (art)	The, An, A, Book, Film	Book, Film, Story, Piece, Work
Natural Object (nat)	Canis, Felis, Homo, Pan, Rosa	sapiens, lupus, domesticus, tigris, rubra
Event (eve)	World, Olymp, Conf, Expo, Summ	Con, Meet, Expo, Fest, Summit

Table 5: Common Patterns Associated with Each Entity Type

3.2 Deep Learning Approach

3.2.1 Bi-LSTM

For the Deep Learning model we have chosen the Bidirectional Long Short Term Memory (Bi-LSTM) [3] architecture, from the family of RNN. Is a type of neural network architecture specifically designed for sequential data processing tasks such as NLP tasks like NER or sentiment analysis. It is an extension of the LSTM model, enhancing its capability to capture dependencies in both forward and backward directions within a sequence.

The Bi-LSTM architecture stands out in sequential data modeling due to its ability to capture contextual information from both past and future elements in a sequence. By processing inputs bidirectionally, Bi-LSTMs effectively address the challenge of capturing long-term dependencies. This bidirectional approach allows the model to predict outputs based on comprehensive context information, leading to more accurate and nuanced predictions compared to traditional unidirectional models.

Built upon the foundation of Recurrent Neural Networks (RNNs), Bi-LSTMs mitigate the vanishing gradient problem through the use of LSTM cells. Moreover, in multilayer architectures, Bi-LSTMs stack multiple layers of these bidirectional units, each layer extracting progressively abstract features from the input sequence. This hierarchical feature extraction empowers the model to learn intricate patterns and relationships within sequential data, enhancing its performance on complex tasks requiring deep understanding of context and dependencies.

We have done an implementation of a Bi-LSTM model for Named Entity Recognition (NER) in Pytorch by scratch. As it is out of our knowledge we have had some trouble in different technical steps. Nevertheless, we have tried to follow the preprocessing steps to prepare the data for training and evaluation. Firstly we convert the sentences and tags into sequences of tokens and labels. We then create vocabularies for both words and tags. We ensure all sequences in a batch have the same length by padding and finally we encode the words and tags into integer indices.

Regarding our Bi-LSTM model for NER it includes an embedding layer to convert the encoded words into dense vectors, followed by a bidirectional LSTM layer that processes sequences in both forward and backward directions, capturing comprehensive context. A dropout layer is added to prevent overfitting, and a linear layer maps the LSTM outputs to tag scores, which are then normalized using log softmax. The model is trained with hyperparameters including an embedding dimension of 100, a hidden dimension of 256, a dropout rate of 0.5, a learning rate of 0.001, and weight decay of 1e-5 for regularization. To handle class imbalance, we compute class weights inversely proportional to the frequency of each tag, ensuring that rare tags are given higher importance in the loss function, which improves model performance on infrequent entities (`create_class_weights()`).

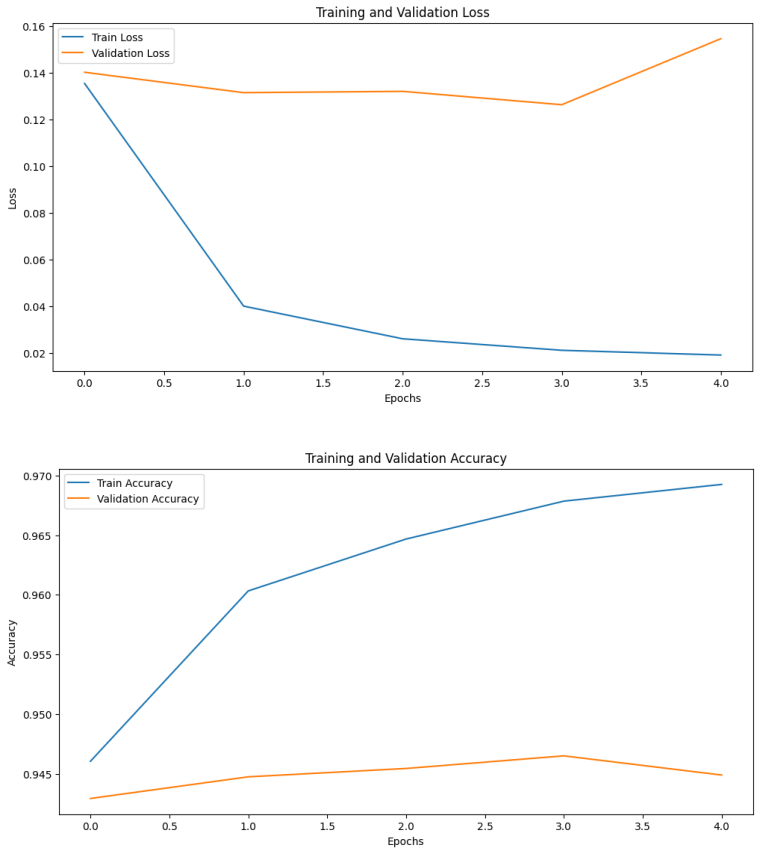


Figure 2: Train and Validation Metrics for Bi-LSTM

In the plots 2 we can see the training routine of Bi-LSTM for 5 epochs, which has lasted 30min on CPU from google Colab. As seen from the training behaviour on

train data the model performs quite well (loss reduces and accuracy is quite good and augments). Regarding validation, loss does not reduce a lot, we suspect it might be in fact overfitting. We will be able to know more about it in the external validation with the tiny_test set 4.4.

3.2.2 BERT

We also decided to try a Transformer approach. Transformers tend to thrive in Natural Language Processing tasks so we wanted to try it. In the beginning we tried text-to-text approaches as the problem seemed similar to a translation task. However, we soon saw on the literature that it was usually tackled using BERT Transformers. The Bidirectional Encoder Representations from Transformers (BERT)[4] is a model that reads text in both directions. It is widely used in many NLP tasks like NER. Its most important advantage is that there are many versions of it pretrained on large datasets. Transformers are famous for needing a lot of data and resources to train, and we don't have any of those, so this is great help. We decided to use one pretrained model from the Transformers library from Hugging Face.

As we want to collect data from the context of the words, we have to use full sentences. We can do it by grouping the given words into sentences by their sentence_id. Also, the tags are transformed into numbers from 0 to 16 for the model to understand.

Then, we are using a tokenizer, also extracted from a pretrained BERT, to transform the words into tokens. Not only one for each, but some words are split into different tokens if they contain relevant information. These extra tokens will have the same tag as the original word except in the case that it was a "B-xxx", in which case it will be transformed to "I-xxx" as only the first token of the named entity should be marked as a beginning. Moreover, some extra tokens will also be added to represent the start and end of a sentence and the empty spaces, to make sure all have the same length. These extra tokens will be assigned the tag -100, which will be ignored in the metrics calculation.

Once we have all the data ready, we feed it to our pretrained BERT, to finetune it.

We also tried adding a custom loss to the model based on the support of each class to try to improve the accuracy on the important classes. The results are in the Results section.

Finally, the task to assign the label to each real word and not to the tokens (i.e. "detokenize") is not quite trivial. For this task, we created a function that assigns to each word the first label from the predictions given to the tokens that correspond to said word. This way, we don't lose the begging of named entities. Then, we remove all additional labels for the extra tokens, like the ones for start or end of sentence.

4 Results

For each different approach explained in 3 we are going to report different metrics of their behaviour and performance.

4.1 Metrics used

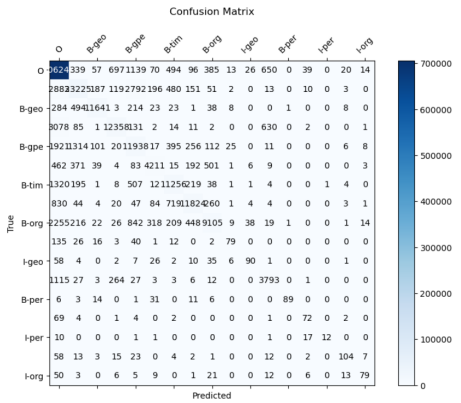
We are reporting the following metrics:

- Confusion Matrix: we show the confusion matrix for train, test and tiny_test sets. With this plot we are able to identify patterns and understand how well the model is performing in terms of correctly and incorrectly classifying for each class
- Accuracy Accuracy on the train set and test set but taking into account only those predictions where the ground truth is not "O".
- F1-score (weighted version from sklearn)

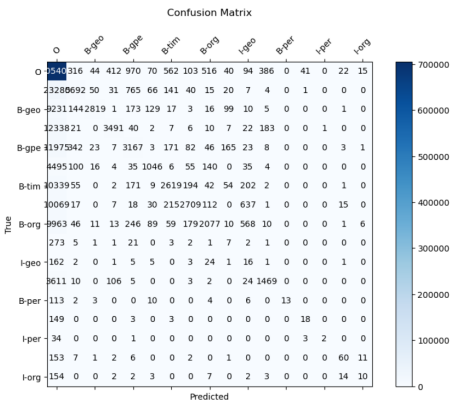
For comparison purposes and due to time and computer resources limitations we have decided to train all models on 5 epochs.

4.2 Baseline Structured Perceptron

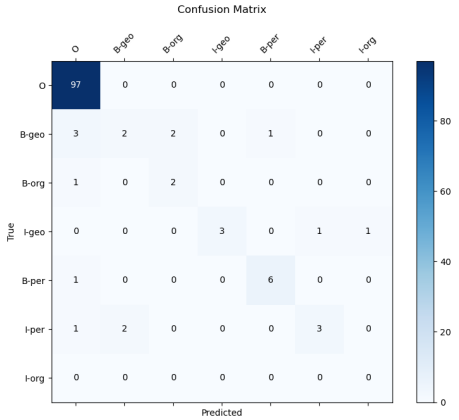
We will take the results from the unmodified features in the code provided as a baseline for further comparison. They are gathered on Table 6. The confusion matrices show how the unbalanced size of each class has affected the training and thus predictive capacities of the model (Figure 4).



(a) Train



(b) Test



(c) Tiny_Test

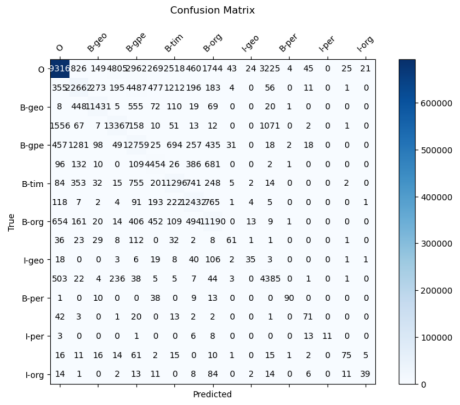
Figure 3: Confusion matrices for baseline Structured Perceptron

Metric	Train	Test	Tiny_Test
Accuracy	0.7758	0.1967	0.4138
Accuracy without 'O'	0.8106	0.4169	0.4138
F1 Score	0.9597	0.8385	0.8454

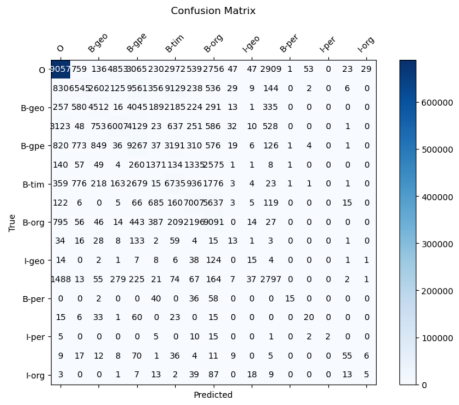
Table 6: Evaluation Metrics for Baseline Structured Perceptron

4.3 Structure Perceptron with Extended Features

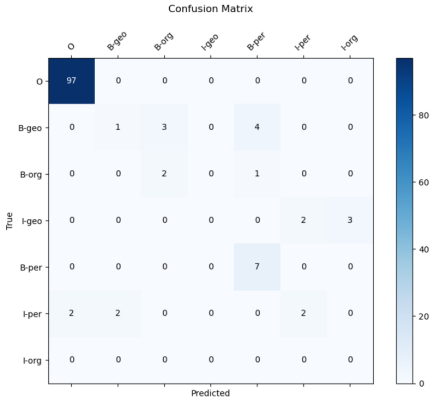
The addition of extra features to the previous perceptron results in a slight improvement in the F1 score on the test set as represented on Table 7. Probably with more epochs in training, the difference would become starker. In both with and without extended features the big difference between accuracy and F1 score signals that the precision i.e., true negative rate, of the two models is quite low. This comes as no surprise in classification problem with so many unbalanced classes.



(a) Train



(b) Test



(c) Tiny_Test

Figure 4: Confusion matrices for Structured Perceptron Extended

4.4 Bi-LSTM

Regarding the Bi-LSTM results, our suspicions about overfitting were confirmed by the behavior of the loss metrics. Table 8 reveals that while both Accuracy and F1 met-

Metric	Train	Test	Tiny_Test
Accuracy	0.8106	0.4169	0.4138
Accuracy without 'O'	0.8106	0.4169	0.4138
F1 Score	0.9529	0.8914	0.8454

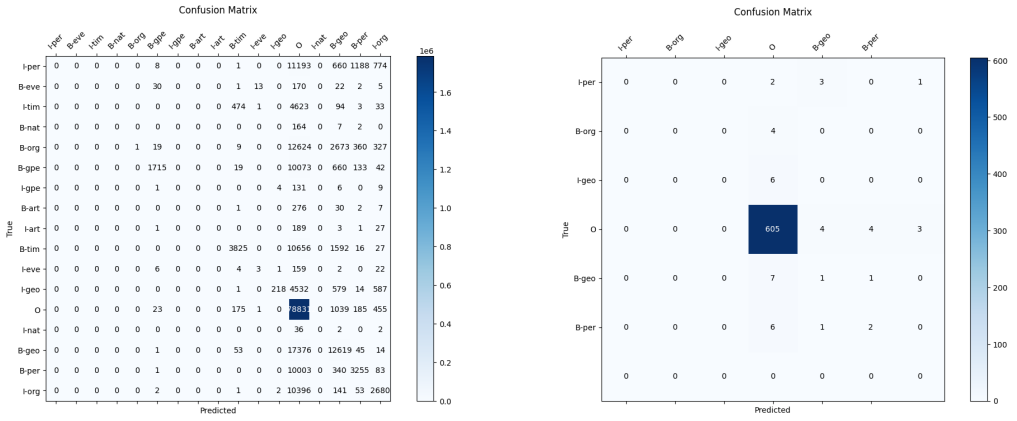
Table 7: Evaluation Metrics for Structured Perceptron Extended

rics show improvement, the Accuracy excluding the 'O' tag significantly deteriorates. This discrepancy suggests that the Bi-LSTM model has indeed overfit to the training and validation sets, achieving high scores primarily due to its performance on the overwhelmingly common 'O' tag.

However, it is noteworthy that during various model trainings, significant enhancement was observed by applying weighted losses inversely proportional to tag frequencies. Notably, this method only accounted for tags in the 'train' and 'test' sets, neglecting those in the external 'tiny_test' set which the model should not have access to.

Metric	Train	Test	Tiny_Test
Accuracy	0.9711	0.9449	0.9670
Accuracy without 'O'	0.5842	0.1915	0.3530
F1 Score	0.9621	0.9264	0.9570

Table 8: Evaluation Scores for Train, Test, and Tiny-Test Sets of Bi-LSTM



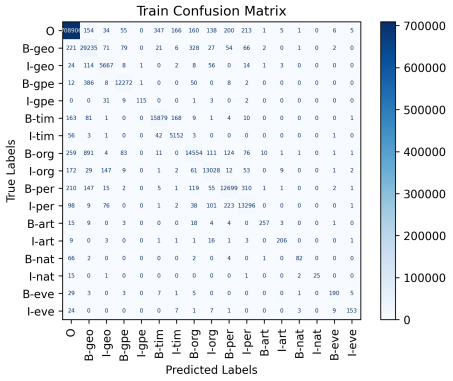
(a) Test (b) Tiny_Test

Figure 5: Confusion matrices for Bi-LSTM

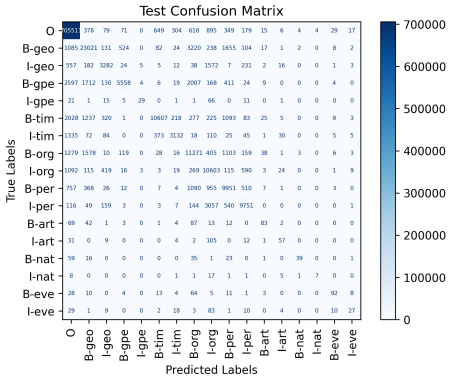
4.5 BERT

Metric	Train	Test	Tiny_Test
Accuracy	0.9911	0.9471	0.9658
Accuracy without 'O'	0.9538	0.6824	0.8529
F1 Score	0.9911	0.9457	0.9635

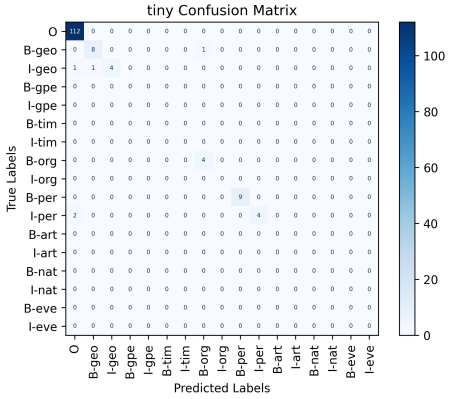
Table 9: Evaluation Scores for Train, Test, and Tiny-Test Sets of BERT.



(a) Train



(b) Test



(c) Tiny_Test

Figure 6: Confusion matrices for Pretrained BERT.

As one can see in Table 9 and Figure 6 the results for this model are very good, the best we got. But we think that was expected as a Transformer with a BERT tokenizer is very good at capturing the significance of the words and are not easily fooled with out of distribution samples.

However, we think it is important to note that it takes approximately 25 minutes to train for 3 epochs using GPUs from Google Colab.

We also tried training it with a Custom Loss to balance the dataset, however we were not able to improve the results so we left the original approach. In the files train_models.ipynb and reproduce_results.ipynb, there is the training and results for the tiny problem. Moreover, we decided to leave the tests done with the custom loss transformer in transformer_tests.ipynb as well as the calculation for the results on the train and test datasets.

5 Conclusions

To conclude, let us recap the work done and the result that we have obtained.

We started by reviewing our data and familiarizing ourselves with the IOB format for sequences and sentences.

Next, we delved into the Structured Perceptron algorithm, outlining its theoretical underpinnings as a linear classifier rooted in Hidden Markov Models. Our discussion extended to the practical implementation of this model. Despite our efforts, the results with extended features were not as promising as anticipated, possibly because we did not adequately select or incorporate sufficient high-quality additional features.

Regarding our initial Deep Learning approach with bi-LSTM, we observed impressive performance during training and validation, yet it clearly exhibited signs of overfitting. While implementing a weighted loss based on tag frequencies proved beneficial, we believe additional preliminary analyses should have been conducted beforehand.

The pre-trained BERT model delivered the best results, achieving remarkably high accuracy even when excluding class 'O'. This reinforces the observation that Transformers excel in NLP tasks.

References

[1] W. authors, "Hidden markov model," 2024.

[2] W. authors, "Viterbi algorithm," 2024.

[3] D. Zhang, L. Tian, M. Hong, F. Han, Y. Ren, and Y. Chen, "Combining convolution neural network and bidirectional gated recurrent unit for sentence semantic classification," *IEEE Access*, vol. 6, pp. 73750–73759, 2018. [Online; accessed 2023-05-25].

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.