UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

Natural Language Processing

Named Entity Recognition (NER)

Brandon Jersaí Alfaro Checa
David Íñiguez Gómez
Jaime Leonardo Sánchez Salazar
Alejandro Vara Mira

Barcelona, June 20, 2024

Contents

1 Introduction

Named Entity Recognition (NER) is a crucial task of information extraction that aims to identify and classify named entities mentioned in unstructured text into predefined categories such as the names of persons, organizations, locations, expressions of times, etc. For example, in the sentence "Barack Obama was born in Hawaii." a NER system should identify "Barack Obama" as a person and "Hawaii" as a location.

NER is a key component in various natural language processing (NLP) applications, including:

- **Information Retrieval:** Enhancing search engine accuracy by tagging named entities.
- **Question Answering:** Improving the extraction of specific information in response to queries.
- **Machine Translation:** Ensuring accurate translation of named entities between languages.
- **Summarization:** Identifying key elements in texts for more effective summarization.

There are various approaches to implementing NER systems, which can be broadly categorized into:

- **Rule-based Approaches:** These rely on handcrafted rules and patterns to identify entities. While straightforward, they require extensive domain knowledge and are not easily adaptable to new domains.
- **Machine Learning Approaches:** These include supervised learning techniques where models are trained on labeled data. Common algorithms include Hidden Markov Models (HMMs), Conditional Random Fields (CRFs), and structured perceptrons.
- **Deep Learning Approaches:** These use neural networks, such as Recurrent Neural Networks (RNNs) and Transformer models, to automatically learn features from large amounts of data, often resulting in state-of-the-art performance.

This project explores both classic machine learning and modern deep learning approaches to NER. The classic approach utilizes handcrafted features, while the modern approach leverages automatic feature extraction through neural networks. We also enhance the structured perceptron model by integrating Cython to improve performance and efficiency.

In particular, we will distinguish between the following entities, with their corresponding tags:

- **geo**: Geographical Entity
- **org**: Organization
- **per**: Person
- **gpe**: Geopolitical Entity
- **tim**: Time indicator
- **art**: Artifact
- **eve**: Event
- **nat**: Natural Phenomenon
- **O**: No chunk token

These tags are also assigned following the IOB (Inside, Outside, Beginning) Tagging system, where:

- **B-**{tag}: Word at the beginning of an entity
- **I-**{tag}: Word inside an entity
- **O**: Outside any entity

This can be easily understood with the previous example. In the sentence "Barack Obama was born in Hawaii.", the tags would be ["B-per", "I-per", "O", "O", "O", "B-geo"].

After introducing the topic, we are going to start with a detailed explanation of how the Structured Perceptron operates. Next, we will cover the different implementations we employed to address the issue at hand. Finally, we will present our findings and draw conclusions based on the results obtained.

2 Structured Perceptron

The Structured Perceptron is an extension of the perceptron algorithm designed for structured prediction tasks, where the output is a structured object, such as a sequence or a tree, rather than a single label. This algorithm is particularly effective for tasks like Named Entity Recognition (NER), where we need to predict a sequence of labels for a given input sequence.

2.1 Hidden Markov Models

Hidden Markov Models (HMMs) are a type of statistical model used to represent systems where the observations are dependent on a sequence of hidden states. An HMM is defined by the following components:

- **States** $\{y_1, y_2, \ldots, y_N\}$: A set of hidden states that are not directly observable.
- **Observations** $\{x_1, x_2, \ldots, x_N\}$: A sequence of observations that are dependent on the hidden states.
- **Initial State Probabilities** $P_{\text{init}}(y_1|\text{start})$ is the initial state probability, representing the probability of starting in state y_1.
- **Final State Probabilities** $P_{\text{final}}(\text{stop}|y_N)$ is the probability of ending in the state y_N.
- **Transition Probabilities** $P_{\text{trans}}(y_{i+1}|y_i)$: The probability of transitioning from state y_i to state y_{i+1}.
- **Emission Probabilities** $P_{\text{emiss}}(x_i|y_i)$: The probability of observing x_i given the state y_i.

An HMM defines a probability distribution over sequences of input and output pairs as follows:

$$P(X_1 = x_1, \ldots, X_N = x_N, Y_1 = y_1, \ldots, Y_N = y_N) = P_{\text{init}}(y_1|\text{start}) \cdot \left(\prod_{i=1}^{N-1} P_{\text{trans}}(y_{i+1}|y_i) \right) \cdot P_{\text{final}}(\text{stop}|y_N) \cdot \left(\prod_{i=1}^{N} P_{\text{emiss}}(x_i|y_i) \right)$$

Key Assumptions of Hidden Markov Models:

1. **Independence of Previous States:** The probability of being in a given state at position i only depends on the state at position $i-1$. This is known as the first-order Markov property.

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

2. **Homogeneous Transition:** The probability of transitioning from one state to another is constant and does not depend on the position in the sequence.
$$P(Y_i = c_k | Y_{i-1} = c_l) \approx P(Y_t = c_k | Y_{t-1} = c_l)$$

3. **Observation Independence:** The probability of observing a particular output at position i is fully determined by the state at that position and is independent of other states and observations.
$$P(X_i = x_i | Y_1 = y_1, \ldots, Y_i = y_i, \ldots, Y_N = y_N) \approx P(X_i = x_i | Y_i = y_i)$$
$$P(X_i = w_j | Y_i = c_k) \approx P(X_t = w_j | Y_t = c_k)$$

The goal of an HMM is to find the most likely sequence of hidden states that could have produced a given sequence of observations. This is can be solved using the Viterbi algorithm, in which we will delve next.

2.2 Viterbi Algorithm

The Viterbi algorithm is a technique employed to determine the most likely sequence of hidden states in a Hidden Markov Model (HMM), given a sequence of observed events. It systematically evaluates all possible state sequences and selects the one with the highest probability, ensuring the optimal path through the states based on the observations and model parameters The steps of the Viterbi algorithm are as follows:

1. **Initialization**: For each state y, calculate the initial probability of being in that state with the first observation:
$$\text{viterbi}(1, x, c) = P_{\text{init}}(c|start) \cdot P_{\text{emiss}}(x_1|c).$$

2. **Recursion**: For each time step $i = 2, \ldots, N$, update the probability for each state based on the previous states:
$$\text{viterbi}(i, x, c) = P_{\text{emiss}}(x_i|c) \cdot \max_{\tilde{c} \in \Lambda} \left(P_{\text{trans}}(c|\tilde{c}) \cdot \text{viterbi}(i-1, x, \tilde{c}) \right).$$

3. **Termination**: Identify the most probable final state:
$$\text{viterbi}(N+1, x, \text{stop}) := \max_{y \in \Lambda^N} P(X = x, Y = y)$$

4. **Backtracking**: Trace back the states that lead to the maximum probability to reconstruct the most likely state sequence, using the following recurrence:
$$\text{backtrack}(N+1, x, \text{stop}) = \arg\max_{c_l \in \Lambda} \left(P_{\text{final}}(\text{stop}|c_l) \cdot \text{viterbi}(N, x, c_l) \right)$$

$$\text{backtrack}(i, x, c) = \arg\max_{\tilde{c} \in \Lambda} \left(P_{\text{trans}}(c|\tilde{c}) \cdot \text{viterbi}(i-1, x, \tilde{c}) \right)$$

To do this we need to keep track of the backtrack quantities when we compute the viterbi quantities

2.3 Structured Perceptron Algorithm

The structured perceptron algorithm adapts the classical perceptron to handle structured outputs, sequences in our case. The key steps in this algorithm are:

- **Objective:** The algorithm aims to learn a parameter vector W that maximizes the score of the correct output $Y^{(j)}$ relative to other possible outputs Y.
- **Prediction:** For each training example $(X^{(j)}, Y^{(j)})$, the algorithm computes the predicted output Y^* using the current parameter vector W:

  $$Y^* = \arg\max_Y W^\top \Phi(X^{(j)}, Y)$$

  This step finds the output Y^* that maximizes the score given by the dot product $W^\top \Phi(X^{(j)}, Y)$.
- **Update Rule:** If the predicted output Y^* does not match the true output $Y^{(j)}$, the algorithm updates the parameter vector W using the perceptron update rule:

  $$W \leftarrow W + r \left(\Phi(X^{(j)}, Y^{(j)}) - \Phi(X^{(j)}, Y^*) \right)$$

  Here,
  - $\Phi(X^{(j)}, Y^{(j)})$ is the feature vector corresponding to the true output $Y^{(j)}$.
  - $\Phi(X^{(j)}, Y^*)$ is the feature vector corresponding to the predicted output Y^*.
  - r is the learning rate that controls the step size of the update.

The structured perceptron is capable of learning complex dependencies in structured output spaces, making it suitable for tasks like NER where the output is a sequence of labels.

3 Implementation

The zip folder that we deliver to the professor follows a very similar tree structure as the defined in the guide:

```
JaimeSanchez_AlejandroVara_BrandonAlfaro_DavidIniguez
    main.pdf
    skseq
    nltk_data
    data
        train_data_ner.csv
        test_data_ner.csv
        tiny_test.csv
    train_models.ipynb
    reproduce_results.ipynb
    fitted_models
        ner_dl_distilbert
        train_seq_pkl
        ...
    utils
        utils.py
```

Divison of the work:

Jaime Sanchez:

- **main.pdf Document**:
  - Provided the explanations for the Deep Learning Model and analyzed and presented the results of this approach.
- **reproduce_results.ipynb Notebook**:
  - Realized the post processing of the Deep Learning Model developing the `reproduce_results.ipynb` notebook and its functions.
  - Addressed errors encountered during the implementation process.

Alejandro Vara:

- **main.pdf Document**:
  - Authored the `main.pdf` file, providing a comprehensive theoretical explanation of the structured perceptron.
  - Detailed the structured perceptron models implemented, discussing their methodologies and functionalities. Also, commenting their ability to handle unseen words during test.

  - Analyzed and presented the results of the structured perceptron models, offering insights and interpretations based on the experimental outcomes.

- **reproduce_results.ipynb Notebook**:
  - Collaboratively worked with Brandon on developing the `reproduce_results.ipynb` notebook and its functions.
  - Addressed errors encountered during the implementation process.

Brandon Alfaro:

- **train_models.ipynb Notebook**:
  - Developed the training for both of the implementations of Structured Perceptron (with Default and Extended Features).
  - Added a new `extended_feature.py` file to `skseq` folder, to add new features to the Structured Perceptron
  - Created new functions in `utils.py`, necessary to train both implementations of the Strcutured Perceptron.
- **reproduce_results.ipynb Notebook**:
  - Developed the testing for both of the implementations of Structured Perceptron (with Default and Extended Features).
  - Created new functions in `utils.py`, necessary to test both implementations of the Structured Perceptron.

David Iniguez:

- **main.pdf Document**:
  - Provided the explanations for the Deep Learning Model and redacted the general conclusions of the project.
- **train_models.ipynb Notebook**:
  - Developed the training for the Deep Learning Model and the creation of the correspondent functions in `utils.py`
- **reproduce_results.ipynb Notebook**:
  - Collaboratively worked with Jaume on developing the `reproduce_results.ipynb` notebook and its functions.
  - Addressed errors encountered during the implementation process.

4 Models

4.1 Structured Perceptron with default features

The structured perceptron model in this implementation utilizes a set of default features that replicate those used in a Hidden Markov Model (HMM). The main features include word-tag pairs and tag-tag transitions. This approach ensures that the model captures both the identity of the words and the relationships between consecutive tags in the sequence. The features are extracted from the dataset using two primary classes, `IDFeatures` and `UnicodeFeatures`, which build and manage the feature sets required for the structured perceptron.

Feature Extraction Process

The `IDFeatures` class is responsible for extracting features from the dataset. It maintains a dictionary of all features (`feature_dict`), which maps feature names to unique IDs. It also stores the feature list for each sentence in the corpus, which includes node and edge features. The class provides methods to build features for the entire dataset (`build_features`) and to get features for individual sequences (`get_sequence_features`).

For each sequence, the features are divided into four categories:

- **Initial features**: Captured at the beginning of the sequence.
- **Transition features**: Captured between consecutive tags.
- **Final features**: Captured at the end of the sequence.
- **Emission features**: Captured for each word-tag pair in the sequence.

During the feature extraction process, initial and final state features are extracted using `add_initial_features` and `add_final_features` methods respectively. Transition features between consecutive tags are added using `add_transition_features`, and emission features for each word-tag pair are extracted using `add_emission_features`. These methods generate feature names, obtain their IDs, and append them to the respective feature lists.

4.1.1 Handling Unseen Words

When the model encounters an unseen word during testing, any feature related to that word and different tags will be activated. If no additional features are used, the model lacks sufficient information to tag the word correctly. This limitation underscores the importance of incorporating meaningful extra features that can define the word without prior observation.

4.2 Structured Perceptron with additional features

The structured perceptron model can be significantly enhanced by incorporating additional features beyond the default word-tag pairs and tag-tag transitions. These additional features provide the model with more contextual information, thereby improving its ability to correctly tag sequences, especially when encountering unseen words. The `ExtendedFeatures` class extends the basic `IDFeatures` class to include a comprehensive set of extra features.

Additional Feature Extraction

The `ExtendedFeatures` class introduces several new types of features extracted from the dataset, including:

- **Capitalization Features**:
  - First letter uppercase (e.g., "Capitalized::Tag").
  - All letters uppercase (e.g., "ALL_CAPS::Tag").
  - All letters lowercase (e.g., "lower::Tag").
- **Character Type Features**:
  - Presence of digits (e.g., "contains_digit::Tag").
  - Alphanumeric characters (e.g., "alphanum::Tag").
- **Punctuation Features**:
  - Presence of punctuation marks (e.g., "punctuation::Tag").
  - Presence of hyphens (e.g., "hyphen::Tag").
- **Linguistic Features**:
  - Stopwords (e.g., "stopword::Tag").
  - Prepositions (e.g., "preposition::Tag").
- **Prefix and Suffix Checks**:
  - For each type of entity (e.g., Artifact, Event, Geographic, etc.), specific prefixes and suffixes are checked.
  - If a word starts with a predefined prefix, the feature "prefix:Prefix::Tag" is added.
  - If a word ends with a predefined suffix, the feature "suffix:Suffix::Tag" is added.

4.2.1 Handling Unseen Words

The additional features help the model handle unseen words more effectively. By incorporating character type features, affixes, and linguistic properties, the model gains more context about each word. This context allows the model to make better-informed predictions even when encountering words not seen during training. For instance, recognizing that a new word is capitalized, contains a common suffix, or matches a known prefix can provide crucial hints about its likely tag.

4.3 Deep Learning model

The other approach that was taken into consideration was the use of deep learning. The first family of models that come to mind are transformers from huggingface. These will be the models that will be tuned for this problem. Within all the different type of pretrained structures that one can find in hugginface, the most useful ones for named entity recognition are the BERTForTokenClassification models.

These models receive as inputs a sequence of tokens that are the result of applying a tokenizer to a sentence. The output of the model is a label for each of the tokens with extra information like the score of the prediction, the token that corresponds to that label, and where can one find that token in the original sentence.

As one could expect, here the preprocess is key to maximizing the obtained model performance, as well as being able to revert the different transformations in order to avoid lags between the actual sentence and the reconstructed one. Let us explain first how the preprocessing works.

Preprocessing is understood as the diverse transformations the data suffer, so that the model receives as inputs the more easily interpretable input and with the correct distribution of the labels. All the techniques that are applied in this project have to do with how does the tokenizer that is used here work when facing different scenarios. Tokenizers are known for their tendency to splitting words in sequences interpretable by the model, called tokens, which have embeddings associated to them that map the token to a higher dimensional space. When one word is splitted in two or more tokens, a '##' string is added at the beginning of each token (excepting the first one), to mark that this token is attached to the previous one. This must be also considered when building the label of the tokens, since splitting a word in several tokens induces a lag between the actual labels and inputs of the model. To do that, the true outcome of a word is replicated in all tokens. This is called alignment of tokens and labels.

The other factor that is problematic when it comes to building the model inputs are the punctuation signs, like dots and commas. Every time the tokenizer sees

those signs, splits the word. For example for 'U.S.A', the tokenizer gets 5 tokens: ['U', '.', 'S', '.' , 'A']. We do not want this to happen, since the word losses all its meaning, moreover we have an unintended misalignment of the labels and tokens. This was specially appreciated in acronyms and numbers. To avoid it, we simply removed those punctuation signs. Note that when we find a dot or a comma alone we do not remove it, since in this case they are relevant to understand the meaning of the sentence. We also found more conflictive signs, like hyphens or apostrophes, but they do not interfere so much in the training process, but they do when undoing the tokenization process.

With all this, we fit the model. We implemented an early stopping since the model tended to overfit very quickly. We used the validation loss as our early stopping criteria, because this subset does not take part in the weight adjusting process. Note that since this is a classification problem, we use the cross-entropy loss function. After that, we saved the model to be used later for testing purposes.

In the testing process we had to deal with the biggest challenge in this project, that is reverting the tokens and rebuild words when punctuation signs (hyphens or apostrophes) are involved. This tokens, when they are in the middle of words, force them to split when they should not do that. We had to detect these signs are in the middle of words and put them again together. It is worth mentioning that despite being part of the same word as the previous tokens, the ones that represent punctuation signs do not have a beginning that indicates that they are attached to the anterior token. This is the greatest difference with letters tokens. Note that this is not conflictive for predicting, since we took this into account also for training, but this is not negligible when it comes to computing different metrics such as accuracy, f1-score or plotting the confusion matrix.

So, the methodology we followed to reconvert the tokens into the original words consisted on two steps. First, we corrected the apostrophes and finally the hyphens.

In the case of apostrophes, what was happening was that for words like "he's" we were initially predicting three different tags. One for "he" and the other two for the apostrophe and the "s". In most cases they are typical contractions of verbs (['s','m','ll','re','ve']) or possessives so we decided to contract these tokens and assign the tag relative to the first token. Following the same criteria we have taken into account the contraction "n't" (does not, for instace) and an exceptional case that was repeated a lot which was "Shi'ite". For all these cases we had to take also into account the indices that we were eliminating (tokens indices) in order to be able to properly reconstruct the sentence.

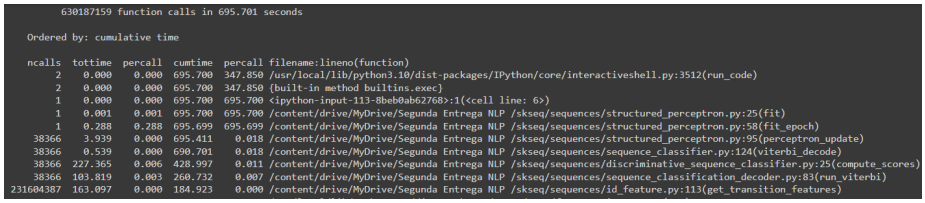
Regarding the hyphens, we followed these steps: first, we detected the indices where we have hyphens, and if the result of concatenating this index with the previous and following token is an original word, we put them together. Then,

we checked that if we have two consecutive signs and they do not belong to another word, to put them together. Finally, the most complex case was when we have words like '46-year-old'. We corrected examples like that by concatenating these strings. We take note of the indices that are joint and realign the labels together. Note that we associated the label of the first token of the word, since there were cases where tokens at the end of the word had a 'I-' label and the first one a 'B-', which is correct and the 'I-' is a secondary effect of splitting the word. There are weird cases that we had to dismiss, since we lost performance because we lost generality and less cases were correctly rebuilt.

Having lags between the actual and rebuilt sentence made us lack accuracy. This is because we compared the words located at the indices of the actual sentences that were not tagged as 'O' with whatever word is at the same index in the recovered sentence. So, if we have lags, we are not exactly comparing the same words. We think this is a but unfair, because the accuracy may be smaller than expected due to those lags, but the fact that the original sentence was not exactly conserved after the tokenizing and reconverting processes can be also considered as a mistake made us decide that this is a way to measure how good was the post-processing process.

4.4 Extra Exercise

Additional to the requested tasks, we displayed the profiling results for the training of both the Structured Perceptron with Default and Extended Features. This would enable us to check which tasks take the most time to execute during training. By looking at column *tottime*, one can see that the functions *compute_score* and *run_viterbi* took 222 seconds and 111 seconds, respectively. Summed up, these two executions account for 46% of the total execution time in training (1-epoch training), for both approaches of the Structured Perceptron. Knowing this, it would be benefitial to look up at alternatives or solutions on how to improve the execution time of the overall training process, specifically targeting at these functions or Python files (.py) where they are defined. Alternatives such as Cython are well-known for gaining significant performance improvements by compiling the code to C.



```
         630187159 function calls in 695.701 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.000    0.000  695.700  347.850 /usr/local/lib/python3.10/dist-packages/IPython/core/interactiveshell.py:3512(run_code)
        2    0.000    0.000  695.700  347.850 {built-in method builtins.exec}
        1    0.000    0.000  695.700  695.700 <ipython-input-113-8beb0ab62768>:1(<cell line: 6>)
        1    0.001    0.001  695.700  695.700 /content/drive/MyDrive/Segunda Entrega NLP /skseq/sequences/structured_perceptron.py:25(fit)
        1    0.288    0.288  695.699  695.699 /content/drive/MyDrive/Segunda Entrega NLP /skseq/sequences/structured_perceptron.py:58(fit_epoch)
    38366    3.939    0.000  695.411    0.018 /content/drive/MyDrive/Segunda Entrega NLP /skseq/sequences/structured_perceptron.py:95(perceptron_update)
    38366    0.539    0.000  690.701    0.018 /content/drive/MyDrive/Segunda Entrega NLP /skseq/sequences/sequence_classifier.py:124(viterbi_decode)
    38366  227.365    0.006  428.997    0.011 /content/drive/MyDrive/Segunda Entrega NLP /skseq/sequences/discriminative_sequence_classifier.py:25(compute_scores)
    38366  103.819    0.003  260.732    0.007 /content/drive/MyDrive/Segunda Entrega NLP /skseq/sequences/sequence_classification_decoder.py:83(run_viterbi)
231604387  163.097    0.000  184.923    0.000 /content/drive/MyDrive/Segunda Entrega NLP /skseq/sequences/id_feature.py:113(get_transition_features)
```

Figure 1: Profiling for Structured Perceptron Training

5 Results

5.1 Structured Perceptron with default features

The Table 3 presents the performance metrics, acurracy and weighted F-score, of the Structured Perceptron model with default features on all datasets.

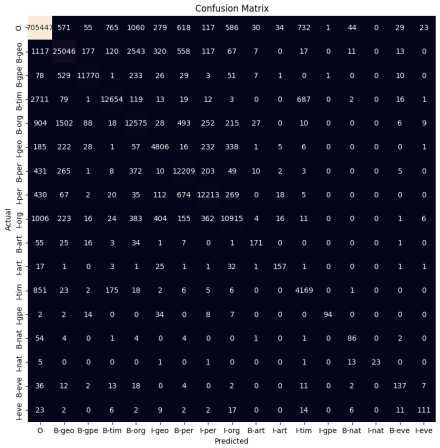
Dataset	**Accuracy**	**F-Score**
Train	0.9683	0.9682
Test	0.8808	0.8579
Tiny	0.9041	0.9036

Table 1: Structured Perceptron with default features Performance Metrics

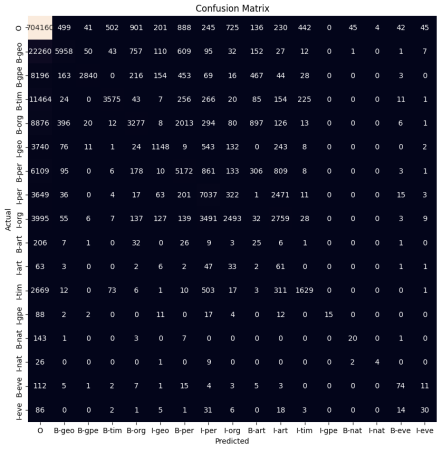
On the training dataset The model achieves an accuracy of 0.9683 and an F-score of 0.9682. This high level of performance indicates that the model has learned the training data very well. However, it is important to verify whether this performance generalizes to unseen data.

On the test dataset, the model's accuracy drops to 0.8808, and the F-score to 0.8579. This decrease in performance suggests that the model may not generalize as well as desired. This is something expected due to the model's handling of unseen words during training, as commented on the previous section.

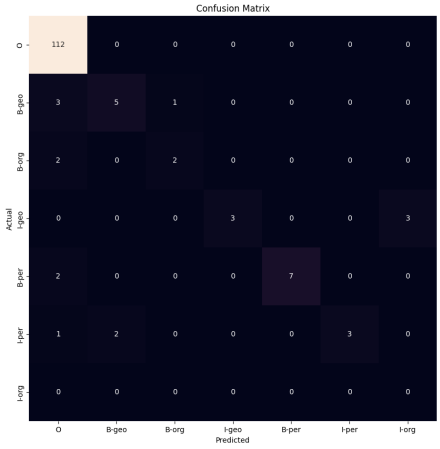
The tiny dataset, which contains a small subset of data for quick evaluations, shows an accuracy of 0.9041 and an F-score of 0.9036. These metrics are better than those on the test dataset but not as high as on the training dataset, which provides a balanced view of the model's generalization capability.



(a) Train



(b) Test



(c) Tiny

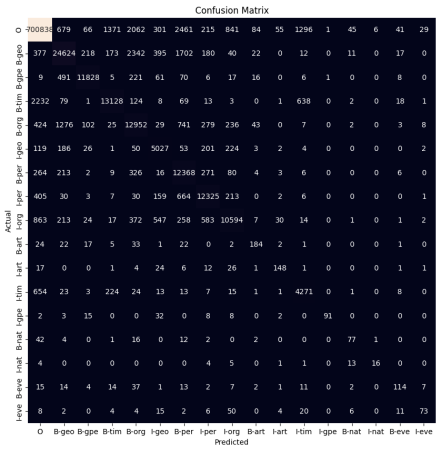
Figure 2: Confusion Matrices for all datasets of the Structured Perceptron with default features

5.2 Structured Perceptron with additional features

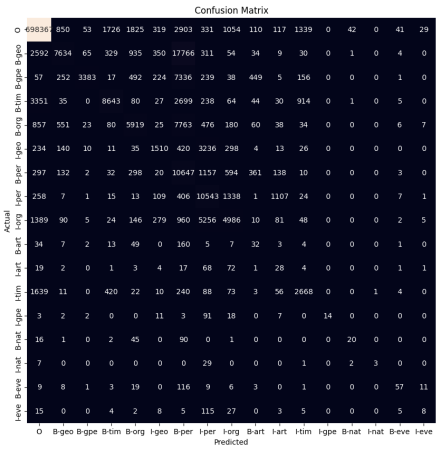
In Table 2, we observe that the performance on the training dataset is nearly identical to that of the Structured Perceptron with default features. However, the improved performance on the tiny dataset suggests better potential for generalization. This enhanced generalization is confirmed by the improved performance on the test dataset, supporting our hypothesis about the model's ability to handle unseen values during testing.

Dataset	Accuracy	F-Score
Train	0.9637	0.9644
Test	0.9010	0.9029
Tiny	0.9452	0.9370

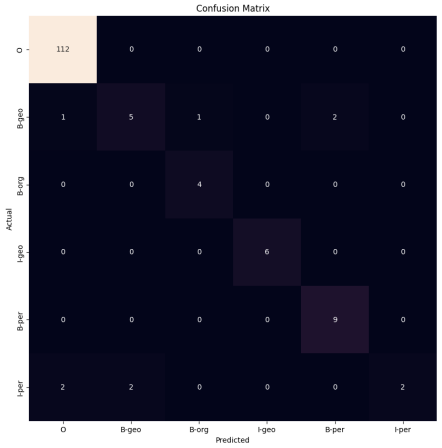
Table 2: Structured Perceptron with additional features Performance Metrics



(a) Train



(b) Test



(c) Tiny

Figure 3: Confusion Matrices for all datasets of the Structured Perceptron with additional features

About the confusion matrices, we can see how in both cases, with additional features and without all the misclassifications are located in the upper side of the matrix, mostly with B-org and I-org.

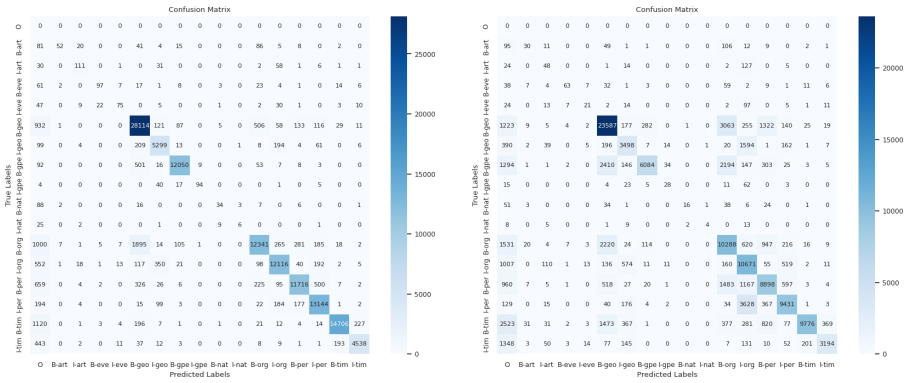
5.3 Deep Learning Model

The Table 3 presents the performance metrics, acurracy and weighted F-score, of the Deep Larning Model (Distilibert):

Dataset	**Accuracy**	**F-Score**
Train	0.89	0.91
Test	0.67	0.70
Tiny	0.99	0.99

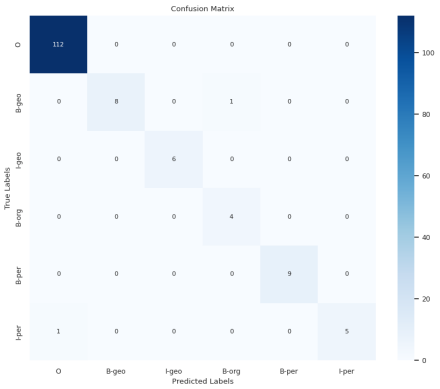
Table 3: Structured Perceptron with default features Performance Metrics

We can see how our performance on the training set is not as good as with the Perceptron approach (in both Accuracy and F-Score terms). Nevertheless it is still a good performance. As expected, in the test set we have a worse performance than the Perceptron. Finally, we can see how in the tiny test we obtain better results than in training and test, but also compared with the Perceptron. It is worth mentioning that it was a really small group of sentences and, even if we obtain a higher accuracy in this tiny set we would not say that the Distilbert performance (wihtout further modifications) is better than the Perceptron one.



(a) Train

(b) Test



(c) Test

Figure 4: Confusion Matrices for all datasets of the Deep Learning Model.

Having a look at the confusion matrices we can detect that in training and testing most of our mistakes are in B-org and B-geo. This matches the explanation we gave when explaining the problems we had in our Deep Learning Model. Note that the performance in the tiny test is really good, only detecting two misclassifications. Also is it worth mentioning that the first row of the first two matrices do not have any number because we excluded, in order to compute the accuracy as asked.

6 Conclusions

In this study, we implemented and compared two approaches for Named Entity Recognition (NER): a structured perceptron and a fine-tuned DistilBERT model. Our preprocessing and postprocessing techniques were meticulously designed and applied, particularly to enhance the performance of the DistilBERT model.

The structured perceptron outperformed the fine-tuned DistilBERT model on both the training and test datasets. This indicates the perceptron's robustness and effectiveness in capturing the underlying patterns necessary for accurate NER. However, the DistilBERT model exhibited remarkable results on the tiny test set, making only two errors. This suggests that, despite its slightly lower performance overall, DistilBERT possesses strong potential for precise entity recognition in specific, constrained scenarios.

These findings highlight the trade-offs between traditional machine learning models and modern transformer-based approaches. While structured perceptrons may offer superior performance on broader datasets, the fine-tuned transformer models like DistilBERT demonstrate exceptional precision in limited contexts. Future work could explore hybrid models that leverage the strengths of both methods to achieve even better results in NER tasks.