UNIVERSITAT DE BARCELONA
Facultat de Matemàtiques i Informàtica

Natural Language Processing

NAMED ENTITY RECOGNITION

Sara Bardají Serra
Sergi Bech Sala
Àlex Pujol Vidal
David Rosado Rodríguez
Jordi Segura Pons

Barcelona, June 18, 2023

How to compile

To successfully execute the "Reproduce Results" notebook, it is essential to follow the steps below. Firstly, navigate to the "Train Models" notebook and execute the necessary Cython import. This step is crucial as it generates the required files for Cython to function properly. Once the import cell has been executed, proceed to the "Reproduce Results" notebook and run the entire notebook to observe the desired outcomes.

Contents

1 Introduction

Named Entity Recognition (NER) is a natural language processing (NLP) task that involves identifying and classifying named entities within text. A named entity refers to a specific type of entity, such as names of persons, organizations, locations, cites, etc. In our case, the dataset contains the following entities:

- **geo**: Geographical entity.
- **org**: Organization entity.
- **per**: Person name entity.
- **gpe**: Geopolitical entity.
- **tim**: Time indicator entity.
- **art**: Artifact entity.
- **eve**: Event entity.
- **nat**: Natural Phenomenon entity.

The dataset provided utilizes the widely recognized IOB (Inside, Outside, Beginning) format for representing entities. In this structure, each token in a sentence is tagged with a label indicating whether it is inside an entity, at the beginning of an entity, or outside any entity. The format uses three types of tags:

- **B-<label>**: Indicates the beginning of an entity with the specified label.
- **I-<label>**: Indicates that the token is inside an entity with the specified label.
- **O**: Indicates that the token is outside any entity.

Allow us to provide an illustration by constructing a sentence that includes tagged entities for the purpose of elucidation:

Token	John	Smith	is	an	engineer	at	Microsoft
Label	B-PER	I-PER	O	O	O	O	B-ORG

We have been provided with three distinct datasets: the training set consists of 38,366 sentences, the test set contains 38,367 sentences, and lastly, there is a tiny test set composed of 13 sentences.

Now that we have been introduced to the problem and have a clear understanding of our data, let us delve into the structure of this report. Firstly, we will provide a detailed explanation of how the structured perceptron operates. Subsequently, we will elaborate on the implementations we have employed to address the problem. Finally, we will present an overview of the results obtained and draw conclusions based on them.

2 Theoretical model explanation

In this section, we will explore the concepts behind the structured perceptron algorithm. To ensure a clear understanding of this algorithm, we will begin by elaborating on the foundational principles that underlie Hidden Markov Models.

2.1 Hidden Markov Models

In order to address the issue at hand, we have first studied Hidden Markov Models (HMM). In an HMM, we assume that there is an underlying sequence of hidden states that we cannot directly observe. Instead, we only observe a sequence of outputs or observations, which are influenced by the hidden states. The goal of an HMM is to infer the most likely sequence of hidden states given the observed sequence of outputs. More precisely, a HMM is a probabilistic model that defines a probability distribution over input/output pairs as follows:

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

The model is based on three main 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).
- **Homogeneous transition**: the probability of making a transition from state c_l to state c_k is independent of the particular sequence position.
- **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.

Given an HMM, the Viterbi algorithm can be used to find the most likely sequence of hidden states that generated a given sequence of observations. The forward-backward algorithm is commonly used to compute the probability of an observation sequence given the model. Let us explain in detail the Viterbi algorithm, since it will be useful for the following section.

2.1.1 Viterbi Algortihm

The Viterbi algorithm is a dynamic programming[1] algorithm that is used to find the most likely sequence of hidden states in a HMM that generated a given sequence of observations. Mathematically,

$$y^* = \underset{y \in \Lambda^N}{\arg\max}\, P(Y_1 = y_1, \ldots, Y_N = y_N | X_1, \ldots, X_N = x_N).$$

[1] Dynamic programming refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.

Let us explain step-by-step the Viterbi Algorithm:

- **Initialization**. In the first step, for every state $c \in \Lambda$, we define
$$\text{viterbi}(1, x, c) = P_{init}(c|start) \cdot P_{emiss}(x_1|c).$$
- **Recursion**. For each subsequent time step, $i \in [2, N]$,
$$\text{viterbi}(i, x, c) = P_{emiss}(x_i|c) \cdot \max_{\tilde{c} \in \Lambda} \left(P_{trans}(c|\tilde{c}) \cdot \text{viterbi}(i-1, x, \tilde{c})\right).$$
- **Termination**. Once all time steps have been processed, we can define the Viterbi quantity at the stop position as:
$$\text{viterbi}(N+1, x, \text{stop}) = \max_{c \in \Lambda} \left(P_{final}(\text{stop}|c) \cdot \text{viterbi}(N, x, c)\right).$$
Observe that this corresponds to the overall most likely path, the maximum probability, i.e.,
$$\text{viterbi}(N+1, x, \text{stop}) = \max_{y \in \Lambda^N} P(X = x, Y = y).$$
- **Backtracking**. Once the Viterbi value at position N is computed, the algorithm can backtrack and determine the most likely sequence of hidden states that produced the observed sequence. The recurrence is as follows:
$$\text{backtrack}(N+1, x, \text{stop}) = \arg\max_{c_l \in \Lambda} \left(P_{final}(\text{stop}|c_l) \cdot \text{viterbi}(N, x, c_l)\right),$$
$$\text{backtrack}(i, x, c) = \arg\max_{\tilde{c} \in \Lambda} (P_{trans}(c|\tilde{c}) \cdot \text{viterbi}(i-1, x, \tilde{c})).$$
The final output is the most likely sequence of hidden states corresponding to the observed sequence.

The Viterbi algorithm is widely used in various applications, including speech recognition, part-of-speech tagging, and bioinformatics, where finding the most likely sequence of hidden states is crucial for accurate prediction and classification.

2.2 Structured Perceptron

It is not difficult to see that by utilizing logarithms of the probabilities discussed in section 2.1 as the weights for a linear classifier, and considering the associated counts as feature vectors, the HMM can be perceived as a linear model, i.e.,
$$P(X, Y) = W^T \Phi(X, Y).$$
The vector Φ can be defined as the sum of HMM-like features, $\phi(y, y', X, i)$, as described in Table 1.

Conditions	
y=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 1: Features explanation for every state.

The structured perceptron is a discriminative learning algorithm used for structured prediction tasks, where the output is a structured object such as a sequence or a graph. It is an extension of the perceptron algorithm that can handle structured outputs instead of just binary or scalar outputs. The structured perceptron algorithm can be seen as a linear model that assigns scores to different combinations of observations and labels. The weights associated with each feature determine the contribution of that feature to the overall score.

The perceptron algorithm is a linear classification algorithm that learns a weight vector to make predictions based on feature vectors. It updates the weight vector by iteratively comparing the predicted output with the true output and adjusting the weights based on any errors made. The structured perceptron algorithm maintains a weight vector for each possible output structure. During training, it makes predictions based on the current weights and compares the predicted structure with the true structure. If the predicted structure is incorrect, the algorithm updates the weights to favor the correct structure and penalize the incorrect one. The weight updates are typically based on the feature vectors associated with the predicted and true structures. To perform these weight updates, the structured perceptron algorithm utilizes gradient descent to update the model's parameters by finding the direction of steepest descent in the parameter space.

It is worth noting that we can incorporate additional features beyond those outlined in Table 1. We will provide a more comprehensive explanation of the structure and utilization of these new features by the model in Section 3.2.2.

After the structured perceptron algorithm has acquired the model's parameters (weights) through training data, we can employ the Viterbi algorithm (section 2.1.1) to carry out decoding. This enables us to identify the most probable output sequence by taking into account the learned weights.

2.3 Out-of-Vocabulary Words at Test Time

A question that naturally arises when assessing the model is as follows: What happens when the model is given a word during testing that it has never come across during training?

As the structured perceptron algorithm works as a linear model, no significant changes occur. The emission feature linked to that word may not have a corresponding weight in the weight vector, which means it does not directly contribute to the overall score for that word, but that's the extent of it. However, in cases where the basic features fail to accurately classify the word, additional features can assist the model in achieving a satisfactory classification for the unobserved word. These additional features can provide supplementary information that aids in accurately classifying unobserved words or handling specific cases where the basic features fail to capture the necessary patterns. For example, when encountering a proper noun in the test set that was not present in the training set, we expect that the inclusion of a supplementary feature that identifies capital letters (which is often a useful indicator for proper nouns) would enable us to accurately classify the word.

3 Approaches and Methodologies

In this section we will briefly review the code structure in which our project is presented and provide an in-depth exploration of the specific models we have trained.

3.1 Code Structure

The codebase of our project is organized in a well-structured manner, ensuring ease of navigation and efficient execution. The repository follows a clear structure that consists of the following key components:

- **train_models.ipynb**: This Jupyter notebook serves as the primary code file for training the models. It contains all the necessary code to train the models and store them in the "fitted model" directory. By executing the cells in this notebook, one can replicate the training process and generate updated versions of the models if desired.
- **reproduce_results.ipynb**: Another Jupyter notebook, provided specifically for reproducing the results obtained by the trained models. This notebook loads the required data and the pre-trained models from the disk, allowing for the evaluation of the models' performance. By executing the cells in this notebook, one can replicate the evaluation process and verify the reported results.
- **utils.py**: This Python module contains various helper functions that assist in both the training and evaluation processes. The functions defined in this module are utilized by the Jupyter notebooks mentioned above. They serve to streamline the code, improve reusability, and enhance overall code readability.
- **skseq**: This set of Python files defines classes that are specifically designed to work with sequential data. These classes provide essential functionalities for tasks such as data representation, sequence labeling, feature extraction, and model training.

3.2 Models

In this section, we discuss the models we trained to solve the NER labeling problem. Our approach involves two structured perceptron models, one with default features and the other including additional features. The structured perceptron model, its training and inference process and the definition of basic features is explained in Section 2.2. We also trained a deep learning model to compare its performance with the structured perceptron approach.

3.2.1 Structured Perceptron with default features

This model is based on the structure perceptron class using the default features defined by the feature mapper class. The feature mapper class allows us to extract the HMM features from our training dataset, meaning the initial, emission, transmission and final features.

3.2.2 Structured Perceptron with additional features

One key benefit of utilizing a structured perceptron model instead of an HMM is its ability to incorporate additional features beyond those offered by the HMM. This enables the model to integrate problem-specific knowledge, enhancing its performance and adaptability.

Then, to further improve the strctured perceptron default model, we have added features that we believe are usefule in the Named Entity Recognition Task. We have incorporated the following additional features into our model:

- First letter is uppercase: This feature checks if the first letter of a word is capitalized.
- Check if it is a digit: This feature determines whether a word consists solely of numerical digits.
- Check if it is uppercase: This feature identifies whether a word is entirely composed of uppercase letters.
- Check if it contains numbers: This feature examines whether a word contains numerical digits.
- Check if it contains non-ASCII/punctuation/hyphens: This feature detects the presence of non-ASCII characters, punctuation marks, or hyphens in a word.
- Check if it is lowercase: This feature determines if a word is entirely in lowercase.
- Check if it contains only alphanumeric characters: This feature verifies if a word consists solely of letters and numerical digits.
- Check if it is a certain word from a preposition list: This feature checks if a word matches a specific word from a pre-defined list of prepositions.
- Check if it is a stopword: This feature identifies whether a word belongs to a predetermined list of stopwords, which are commonly used words that are typically excluded from text analysis.
- Check for specific prefixes and suffixes useful in recognizing specific entities: This feature examines if a word contains specific prefixes or suffixes that are indicative of particular entities or patterns.

These additional features provide our model with a richer set of information, enabling it to capture more nuanced aspects of the problem at hand.

3.2.3 Deep Learning model

In order to compare the performance of the structured perceptron model, we also trained a deep learning model, more precisely, a Bi-directional Long Short-Term Memory (Bi-LSTM).

A Bi-LSTM is a type of recurrent neural network (RNN) architecture that combines information from both past and future contexts. Traditional LSTMs process sequential

data in a unidirectional manner, where the hidden states are updated based on the previous information in the sequence. In contrast, a bidirectional LSTM processes the sequence in two directions: from the beginning to the end and from the end to the beginning. This structure allows the networks to have both backward and forward information about the sequence at every time step.

One of the main benefits of utilizing a Bi-LSTM lies in its ability to incorporate insights from both preceding and upcoming data, enhancing prediction accuracy and sequence comprehension. Furthermore, this architectural design empowers us to effectively capture long-term dependencies within sequential data, a highly advantageous feature within our specific framework.

The model's architecture consists of several key components. Firstly, there's an embedding layer responsible for transforming each word within the input sequence into a dense vector representation. Additionally, we have a spatial dropout layer which effectively applies dropout to the input elements. At each timestep, 10% of the elements are randomly set to zero, enhancing the model's robustness. Lastly, the Bi-LSTM layer is employed to process the input sequence in both the forward and backward directions. By doing so, it adeptly captures long-term dependencies between the words. For a visual representation of the architecture, see Figure 1, where further details can be found.

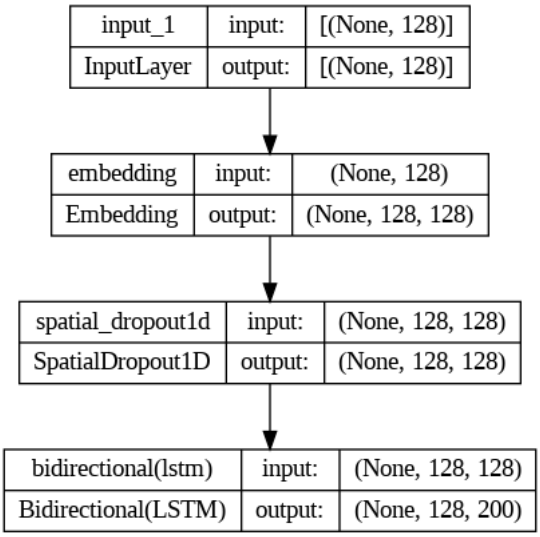


Figure 1: Bi-LSTM model architecture.

In terms of data preprocessing for this model, our initial step involved padding the sequences to a length of 128, ensuring uniformity across all sentences. Additionally, we crafted a vocabulary dictionary to represent the sentences, converting each sentence into a series of numbers that corresponded to the indices within the vocabulary. Lastly, we applied the same process to the tags, resulting in two numerical vectors per sentence, one representing the encoded sentence and the other representing the tags.

In Figure 2, we observe the progress of the accuracy and loss in both the training and validation sets during the model's training. It is worth noting that for validation purposes, we utilized the provided test set.

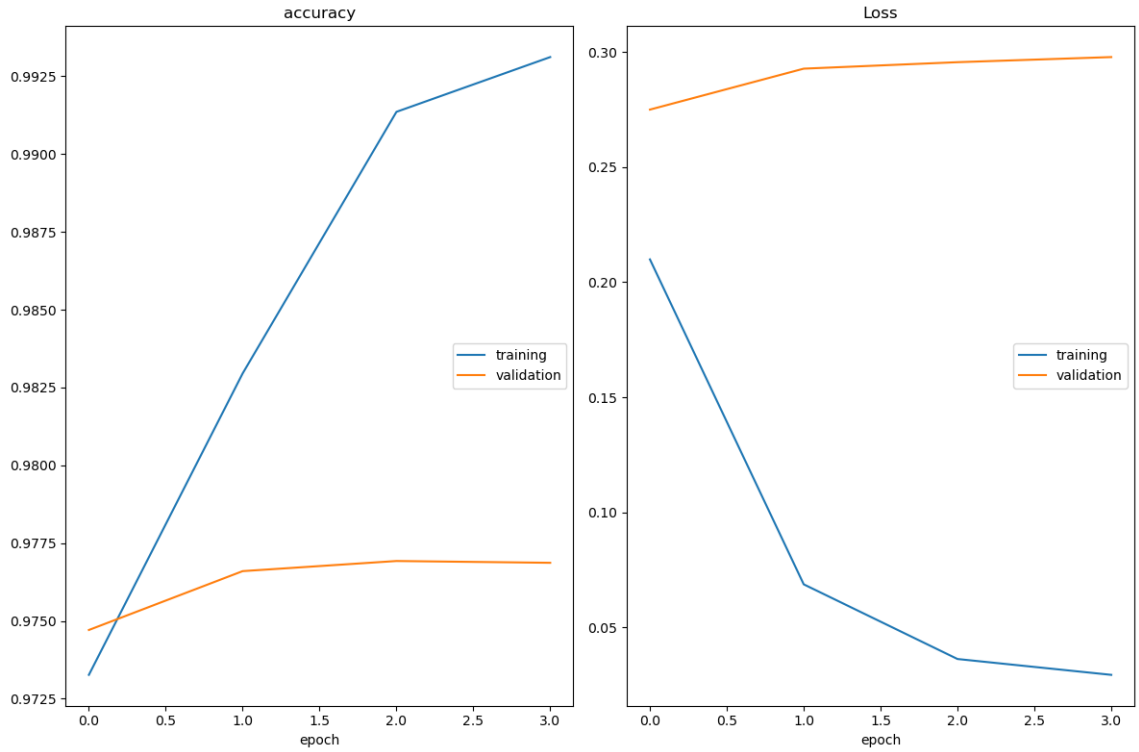


Figure 2: The accuracy and loss of the training and validation sets over epochs.

3.3 Cython enhancement

Due to the significant execution time observed in certain parts of the code, we have opted to leverage Cython to improve their performance. The areas where we encountered elevated execution times were during the creation of the sequence lists, which is needed to train the structured perceptron model and, the training method of the structured perceptron.

To leverage the speed benefits of Cython, we have made modifications to the code by introducing variable types. By explicitly specifying the data types of variables, Cython can generate more efficient C code, resulting in improved execution times.

By providing variable types, Cython can optimize memory access and reduce the overhead associated with dynamic typing, leading to faster execution. This optimization allows us to significantly enhance the performance of these code sections and improve the overall efficiency of the program.

In addition to incorporating variable types, we have further enhanced the code's performance by utilizing function decorators. Specifically, we have included the following decorators:

- @cython.boundcheck(False): This decorator disables bounds checking for array accesses within the decorated function. By eliminating bounds checking, we can improve the execution speed at the cost of potential safety risks. It assumes that array accesses are within the valid range.

- @cython.wraparound(False): This decorator disables negative indexing for arrays within the decorated function. By turning off negative indexing, we can gain a performance boost, assuming that negative indices are not used or necessary in the given context.

These function decorators help optimize the code by eliminating unnecessary checks and enabling more efficient array accesses. By carefully assessing the specific requirements and safety considerations of the code, we can strike a balance between performance improvements and potential risks.

Cython scripts have been incorporated into the skseq directory. We have developed a setup.py file to define the scripts that require building and distributing our Python extension modules based on Cython. Next, to enhance the efficiency of generating the sequence list, we have introduced a new file named sequence_list_c.pyx. In this file, we have implemented the Cython optimizations mentioned earlier. Lastly, to accelerate the training process of the structured perceptron, we have introduced a file named structured_perceptron_c.pyx. Once again, the aforementioned optimizations have been included in this file to expedite the method.

To conclude this section, let us present a table (Table 2) comparing the execution times with and without Cython optimization of the structured perceptron.

	Without Cython	With Cython
Default Features	94	59
New Features	125	60

Table 2: Comparison, in minutes, of the training execution times.

4 Results

In this section, we will provide an analysis of the results obtained using the different models mentioned earlier. The evaluation comprises various metrics that are utilized to assess the performance of the models:

- **Accuracy**: we are required to compute accuracy for both the train and test sets but taking into account only tags that are not "O".
- **Confusion matrix**: we are required to compute the confusion matrix for both the train and test sets. With this metric, we are able to identify patterns and understand how well the model is performing in terms of correctly and incorrectly classifying instances for each class.
- **F1-score**: we are required to compute the F1-score for both the train and test sets. The F1-score is a metric commonly used in machine learning to evaluate the performance of a classification model, particularly when dealing with imbalanced datasets. It combines precision and recall into a single value and provides a balanced measure of a model's accuracy.
- **Tiny_test predictions**: we are required to present the sentences of the tiny test with every predicted tag.

4.1 Structured Perceptron

Let us now examine the outcomes of training two distinct structured perceptron models: one employing the default features and the other incorporating the newly added features. It is worth noting (and expected) that the results obtained from the models, whether utilizing Cython or not, are identical.

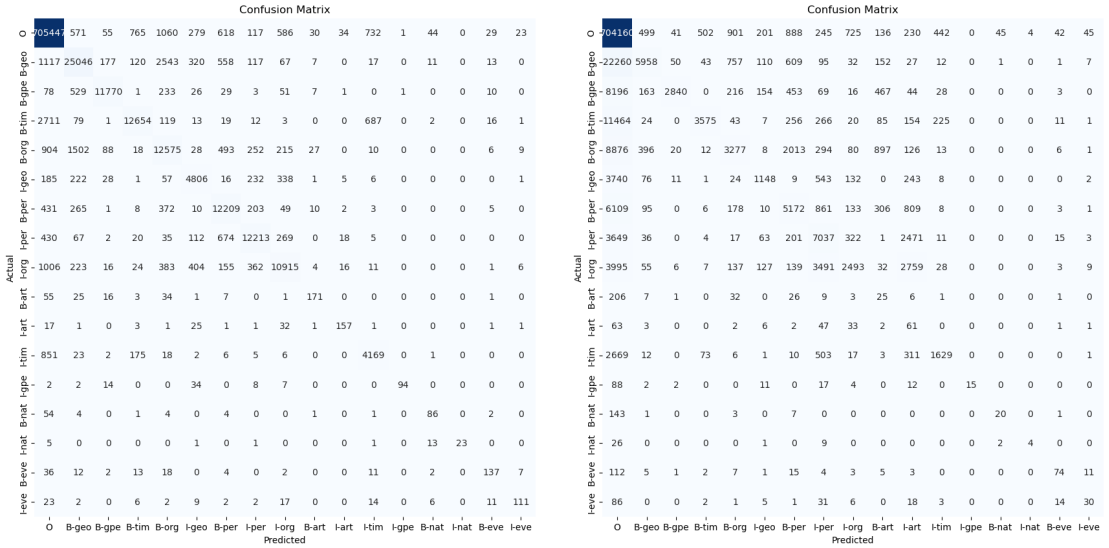
4.1.1 Structured Perceptron with default features

Table 3 and Figure 3 showcases the results obtained using the structured perceptron model with default features.

	Accuracy	F1-score
Train	0.9683	0.9682
Test	0.8808	0.8579
Tiny_test	0.9041	0.9036

Table 3: Results in the train, test and tiny_test sets for the structured perceptron with default features.

The model demonstrates a seemingly strong performance overall. However, by examining the tiny_test set, we can identify some errors made by the model. For instance, the misspelled word "Microsof" and the proper noun "Alice" are misclassified as "O". Let us determine if the structured perceptron model with the new features can rectify these mistakes.



(a) Train (b) Test

Figure 3: Confusion matrix of the train and test sets for the structured perceptron with default features.

4.1.2 Structured Perceptron with new features

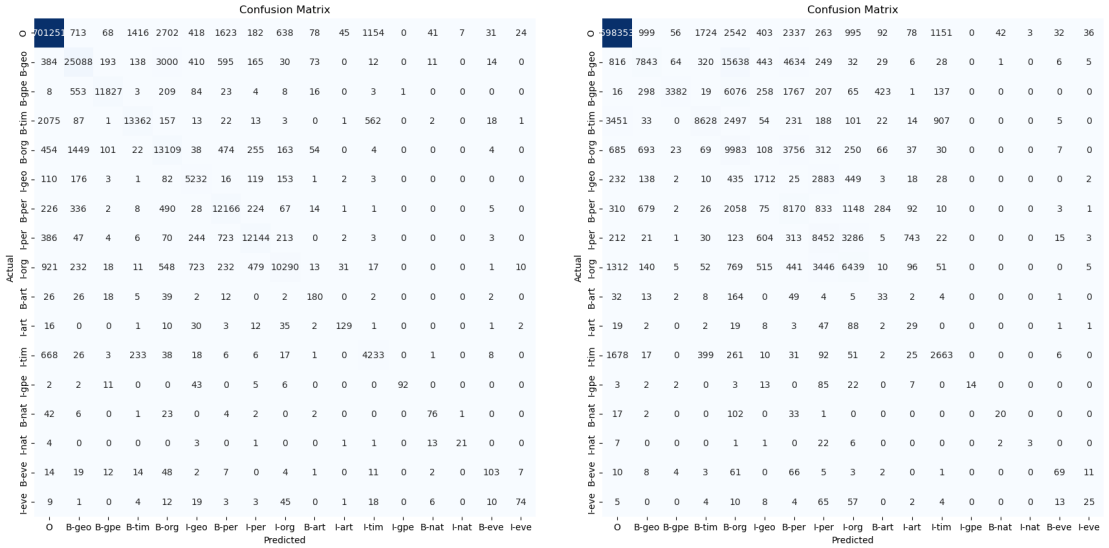
Table 4 and Figure 4 showcases the results obtained using the structured perceptron model with the new features.

	Accuracy	F1-score
Train	0.9645	0.9653
Test	0.9026	0.9043
Tiny_test	0.9452	0.9404

Table 4: Results in the train, test and tiny_test sets for the structured perceptron with new features.

It is worth mentioning that the accuracy and F1-score of the train set for the structured perceptron model with the new features are nearly identical to those of the model with default features. However, it is noteworthy that the F1-score has exhibited a growth of 5% and 4% in the test and tiny_test sets, respectively. In terms of accuracy, there has been an improvement of 2% and 4% in the test and tiny_test sets, respectively.

Furthermore, with the inclusion of the new features, we were able to rectify the aforementioned errors in the tiny_test set. The misspelled word “Microsof” has been correctly classified as B-org, and the proper noun “Alice” has also been accurately classified as B-per. Upon examining the confusion matrix (Figure 4) and accuracy results (Table 4), it can be concluded that the addition of the new features has been successful in correctly classifying certain words that were previously misclassified using the default features.



(a) Train (b) Test

Figure 4: Confusion matrix of the train and test sets for the structured perceptron with new features.

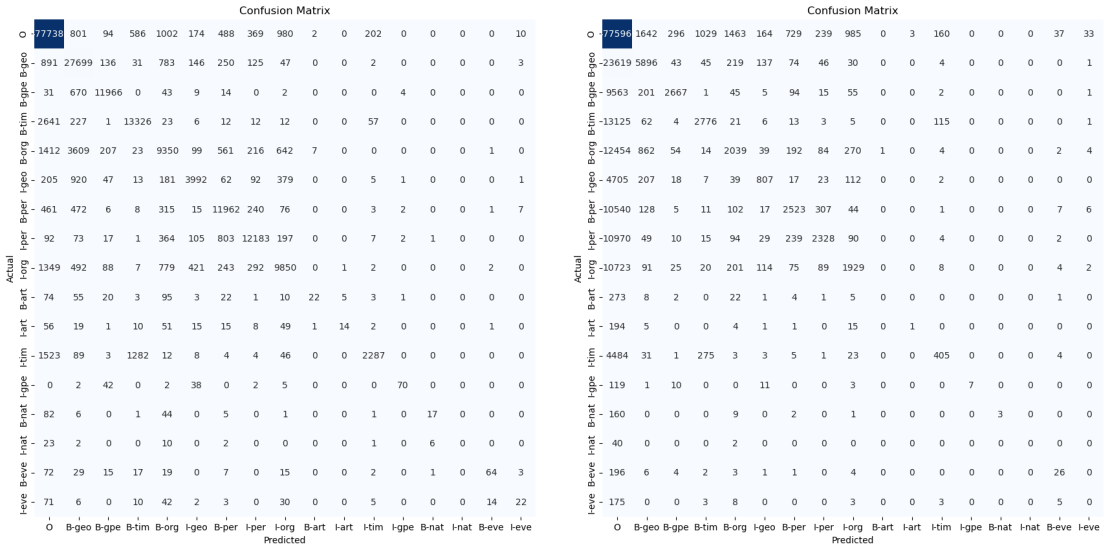
4.2 Deep Leaning model

Table 5 and Figure 5 showcases the results obtained using the Bi-LSTM.

	Accuracy	F1-score
Train	0.7986	0.9935
Test	0.1669	0.9699
Tiny_test	0.5294	0.9873

Table 5: Results in the train, test and tiny_test sets for the Bi-LSTM model.

We have observed that these are the most unfavorable outcomes achieved thus far. It appears that the model's learning capacity is insufficient to accurately determine the tags for out-of-vocabulary words or certain misspelled words. As an illustration, within the tiny_test dataset, the classification error occurs where Jack London is mistakenly classified as B-per B-geo, instead of the correct labeling of B-per I-per. Probably, during the training phase, the model has learned to associate the word "London" with the B-geo tag, which hinders its ability to accurately predict the correct tag



(a) Train (b) Test

Figure 5: Confusion matrix of the train and test sets for the structured perceptron with new features.

5 Conclusions

In conclusion, our work aimed to address the NER problem in NLP through various methods. We began by explaining the structured perceptron method from a theoretical standpoint. Subsequently, we explored two different approaches using the structured perceptron model, one with default features and another with added features, which yielded superior results.

Additionally, we delved into the realm of Deep Learning by implementing a bidirectional LSTM model. However, despite high expectations, the results did not meet our initial hopes. As a result, we decided to retain the structured perceptron model with added features as the best model among the three options we explored.

Furthermore, we recognized the need to optimize the computational efficiency of our chosen model. By identifying sections of the code that consumed significant time, we successfully implemented Cython, resulting in improved performance and reduced execution time.