UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

Fundamental Principles of Data Science

Natural Language Processing Final Project

NAMED ENTITY RECOGNITION

Dafni Tziakouri, Madison E. Chester

2023-2024

Contents

1 Introduction

For Delivery 2 of the Natural Language Processing course, our task was to fully understand the structured perceptron algorithm applied to Named Entity Recognition (NER). NER problems are very useful in many contexts, from information retrieval to question answering systems. The goal of this project is not to achieve the best results, but to fully understand all the details about a simple solution. To do this project, we will work with a training dataset with 38,366 sentences that contains 839,149 words. The test set has 38,367 sentences and 837,339 words.

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, cities, 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.

In this problem, the tags are following the 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: "I love Python. Python is great" has the following tags: ["0", "0", "B"].

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

2 Model Explanation

Before training the model and presenting the results obtained on the train and test sets, let's provide an overview of the structured perceptron algorithm. The Structured Perceptron (SP) is an algorithm that combines the perceptron algorithm for learning linear classifiers with an inference algorithm, which in our case will be the Viterbi algorithm. Once model probabilities are estimated, sequence labels can be obtained using Viterbi decoding. We will begin by introducing these concepts, discussing generative models and Hidden Markov Models.

2.1 Hidden Markov Models

Sequence labeling tasks can be approached using a generative approach. One of the most well-known generative models is Hidden Markov Models (HMM). These types of models assume that the system is a Markov process with unobservable (hidden) states. Thus, their primary objective is to maximize the probability of the hidden states, which depends on an initial state. We can define their probability as follows:

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

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 transitioning 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.

If we take the logarithms of the previous expression and introduce a set of indicator functions, certain terms can be interpreted as counts. Thus, the set of indicator functions are:

$$I[e] = \begin{cases} 1 & \text{if } e \text{ is true} \\ 0 & \text{if } e \text{ is false} \end{cases} \tag{2}$$

If we introduce the indicator functions in the log-probability expression and order terms, we get

the following:

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

Certain terms can be interpreted as counts:

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

This results in a matrix form as the next one:

$$\log P(X,Y) = \begin{bmatrix} \log P_{\text{init}}(\cdot) \\ \log \vec{P}_{\text{trans}}(\cdot) \\ \log \vec{P}_{\text{emiss}}(\cdot) \\ \log P_{\text{final}}(\cdot) \end{bmatrix}^T \begin{bmatrix} e_{y_1} \\ \vec{C_Y} \\ \vec{C_{X,Y}} \\ e_{y_N} \end{bmatrix} \tag{4}$$

From the previous matrix, the left vector contains the model parameters (does not depend on X or Y) and the right vector describes X and Y (knows nothing about the model). Thus, we have a linear model, since the left vector can be seen as the weight vector W and the right one can be seen as the feature vector.

With an HMM, the Viterbi algorithm is utilized to determine the most probable sequence of hidden states that generated a given observation sequence. Meanwhile, the forward-backward algorithm is frequently employed to compute the probability of an observation sequence given the model. We will delve into the Viterbi algorithm in detail as it serves as a crucial foundation for the subsequent section.

2.1.1 Viterbi algorithm

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,

$$y^* = \arg\max_{y \in \Lambda^N} P(Y_1 = y_1, \ldots, Y_N = y_N | X_1, \ldots, X_N = x_N) \tag{5}$$

Let's explain step-by-step the Viterbi Algorithm:

[1]Dynamic programming involves simplifying complex problems by decomposing them into simpler subproblems in a recursive manner.

- **Initialization:** In the first step, for every state $c \in \Lambda$, we define

$$\text{viterbi}(1, x, c) = P_{\text{init}}(c|\text{start}) \cdot P_{\text{emiss}}(x_1|c).$$

- **Recursion:** For each subsequent time step $i \in [2, N]$,

$$\text{viterbi}(i, x, c) = P_{\text{emiss}}(x_i|c) \cdot \max_{c' \in \Lambda} \left(P_{\text{trans}}(c|c') \cdot \text{viterbi}(i-1, x, c') \right).$$

- **Termination:** Once all time steps have been processed, we define the Viterbi quantity at the stop position as:

$$\text{viterbi}(N+1, x, \text{stop}) = \max_{c \in \Lambda} \left(P_{\text{final}}(\text{stop}|c) \cdot \text{viterbi}(N, x, c) \right).$$

  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:

$$\text{backtrack}(N+1, x, \text{stop}) = \arg\max_{c_l \in \Lambda} \left(P_{\text{final}}(\text{stop}|c_l) \cdot \text{viterbi}(N, x, c_l) \right),$$

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

  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

In our report, it becomes evident that by employing logarithms of the probabilities outlined in Section 2.1 as weights for a linear classifier, and regarding the corresponding counts as feature vectors, the HMM can be viewed as a linear model, represented as:

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

Here, the vector Φ can be defined as the sum of HMM-like features, $\phi(y, y', X, i)$, as elaborated in Table 1.

The structured perceptron is a discriminative learning algorithm utilized for tasks involving structured prediction, where the output is a structured object like a sequence or a graph. It extends the perceptron algorithm to handle structured outputs, moving beyond binary or scalar outputs. In this algorithm, scores are assigned to various combinations of observations and

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 1: *Features explanation for every state.*

labels, making it akin to a linear model. The weights associated with each feature dictate its contribution to the overall score.

The perceptron algorithm, a linear classification method, learns a weight vector to make predictions based on feature vectors. It iteratively updates the weight vector by comparing predicted outputs with true outputs and adjusting weights based on errors. Unlike the standard perceptron, the structured perceptron maintains weight vectors for each possible output structure. During training, predictions are made based on these weights, and if the predicted structure is incorrect, weights are updated to favor the correct structure and penalize the incorrect one. Weight updates typically rely on feature vectors associated with predicted and true structures. Gradient descent is commonly used to update the model's parameters by finding the direction of steepest descent in the parameter space.

It's notable that additional features beyond those outlined in Table 1 can be incorporated. A more detailed explanation of these new features and their utilization in the model will be provided in Section 3.1.2.

Once the structured perceptron algorithm has learned the model's parameters (weights) from training data, the Viterbi algorithm (Section 2.1.1) can be employed for decoding. This allows the identification of the most probable output sequence, taking into account the learned weights.

3 Implementation

3.1 Models

In this section, we present the models we developed to address the NER labeling problem. Our approach encompasses two structured perceptron models: one with default features and another incorporating additional features. The structured perceptron model, along with its training and inference process and the definition of basic features, is detailed in Section 2.2. Additionally, we trained a deep learning model to compare its performance with the structured perceptron approach.

3.1.1 Structured Perceptron with default features

This model relies on the Structured Perceptron class utilizing default features defined by the Feature Mapper class. The Feature Mapper class enables us to extract HMM features from our training dataset, including initial, emission, transmission, and final features.

3.1.2 Structured Perceptron with additional features

Incorporating a structured perceptron model instead of an HMM offers a significant advantage: it allows us to include additional features beyond those offered by the HMM. This capability enables the model to leverage problem-specific knowledge, thereby enhancing its performance and adaptability.

To further enhance the default structured perceptron model, we have incorporated features tailored for the Named Entity Recognition Task. These additional features include:

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

These additional features enrich our model with a broader range of information, enabling it to capture more nuanced aspects of the task at hand.

3.1.3 Deep Learning Model

To compare the performance of the structured perceptron model, we also trained a deep learning model known as a Bi-directional Long Short-Term Memory (Bi-LSTM).

A Bi-LSTM belongs to the family of recurrent neural networks (RNNs) and is distinguished by its ability to assimilate information from both past and future contexts. While traditional LSTMs process sequential data in a unidirectional manner, updating hidden states based solely on previous information, a bidirectional LSTM operates bidirectionally, processing sequences from both ends simultaneously. This bidirectional structure equips the model with comprehensive information about the sequence at each time step.

The primary advantage of employing a Bi-LSTM lies in its capacity to incorporate insights from both preceding and upcoming data, thereby improving prediction accuracy and sequence comprehension. Moreover, this architecture excels in capturing long-term dependencies within sequential data, a crucial feature within our framework.

The model's architecture comprises several crucial components. Firstly, an embedding layer transforms each word in the input sequence into a dense vector representation. Additionally, a spatial dropout layer is applied to the input elements, with 10% of elements randomly set to zero at each timestep, enhancing the model's robustness. Finally, the Bi-LSTM layer processes the input sequence in both forward and backward directions, adeptly capturing long-term dependencies between words.

Regarding data preprocessing for this model, our first step was to pad the sequences to a length of 128, ensuring uniformity across all sentences. Additionally, we constructed a vocabulary dictionary to represent the sentences, converting each sentence into a series of numbers corresponding to the indices within the vocabulary. Lastly, we followed the same process for the tags, resulting in two numerical vectors per sentence: one representing the encoded sentence and the other representing the tags.

Figure 1 illustrates the progress of accuracy and loss in both the training and validation sets during the model's training. It's important to note that for validation purposes, we used the provided test set.

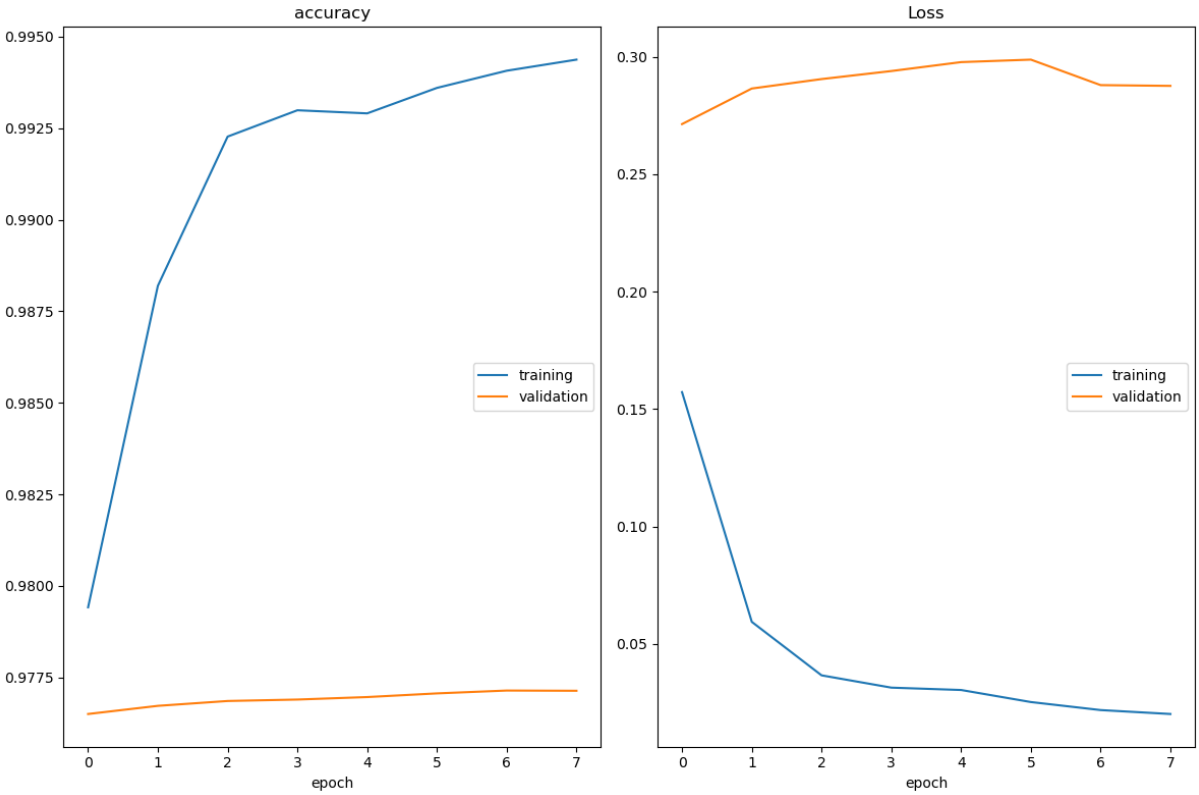


Figure 1: *The accuracy and loss of the training and validation sets across epochs.*

3.2 Cython enhancement

To address the long execution times observed in certain parts of our code, we have decided to utilize Cython to enhance their performance. The sections where we noticed prolonged execution times include the creation of sequence lists required for training the structured perceptron model and the training method itself.

To take advantage of Cython's speed benefits, we modified the code by introducing explicit variable types. By defining data types for variables, Cython can produce more efficient C code, thereby improving execution times.

Explicitly specifying variable types allows Cython to optimize memory access and minimize the overhead associated with dynamic typing, resulting in faster execution. This optimization significantly boosts the performance of these code sections and enhances the overall efficiency of the program.

In addition to incorporating variable types, we have further boosted the code's performance by using function decorators. Specifically, we have implemented the following decorators:

- **@cython.boundcheck(False):** This decorator turns off bounds checking for array accesses within the decorated function. Disabling bounds checking can enhance execution speed, though it comes with potential safety risks, as it assumes array accesses are within valid ranges.
- **@cython.wraparound(False):** This decorator disables negative indexing for arrays within the decorated function. By disabling negative indexing, we achieve a performance improvement, assuming that negative indices are neither used nor needed in the given context.

These function decorators optimize the code by removing unnecessary checks and enabling more efficient array accesses. By carefully evaluating the specific needs and safety considerations of the code, we can balance performance improvements with potential risks.

Cython scripts have been added to the skseq directory. We developed a setup.py file to specify the scripts needed for building and distributing our Python extension modules using Cython. To improve the efficiency of generating the sequence list, we introduced a new file named 'sequence-list-c.pyx', where we implemented the previously mentioned Cython optimizations. Additionally, to speed up the training process of the structured perceptron, we created a file named 'structured-perceptron-c.pyx', incorporating the same optimizations to accelerate the method.

We present a 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 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

Now, let's analyze the results of training two distinct structured perceptron models: one utilizing the default features and the other incorporating the newly added features.

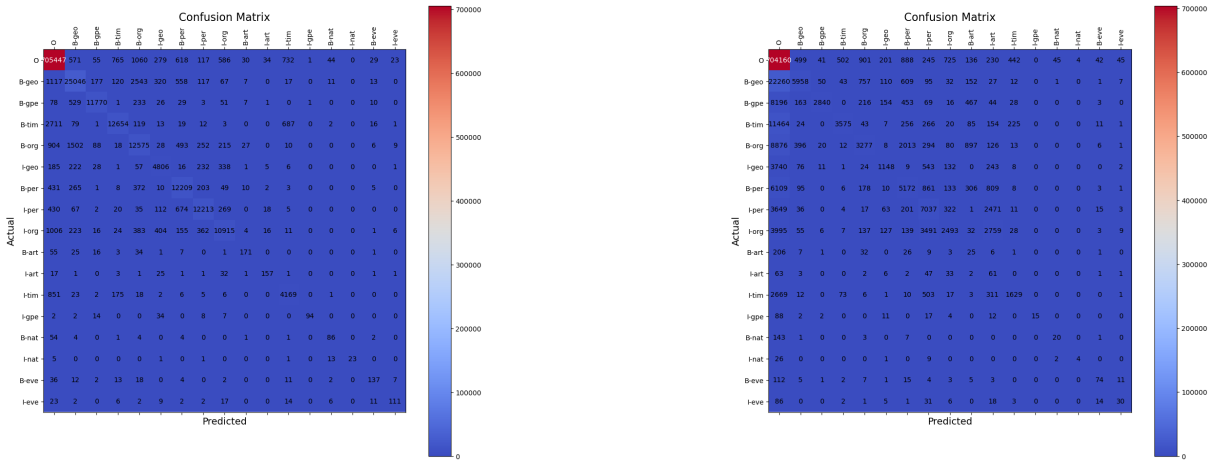
4.1.1 Structured Perceptron with default features

The following Table 3 and Figure 2 illustrates the results obtained using the structured perceptron model with default features.

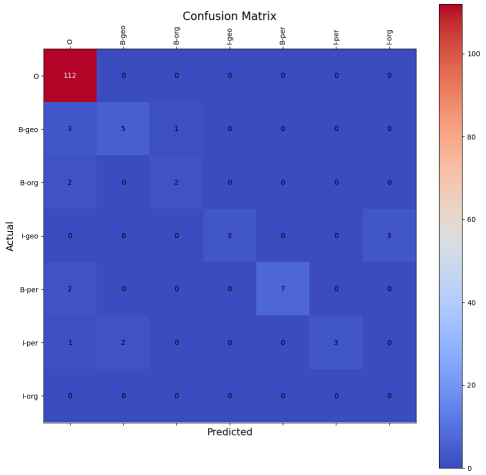
	Accuracy	F1-score
Train	0.9683	0.9682
Test	0.8808	0.8579
Tiny test	0.9041	0.9036

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

The model shows generally strong performance. However, examining the tiny test set reveals some errors. For example, the misspelled word "Microsof" and the proper noun "Alice" are incorrectly classified as "O". Let's see if the structured perceptron model with the new features can correct these mistakes.



(a) *Train* (b) *Test*



(c) *Tiny Test*

Figure 2: *Confusion matrix of the train, test and Tiny test sets for the structured perceptron with default features.*

4.1.2 Structured Perceptron with new features

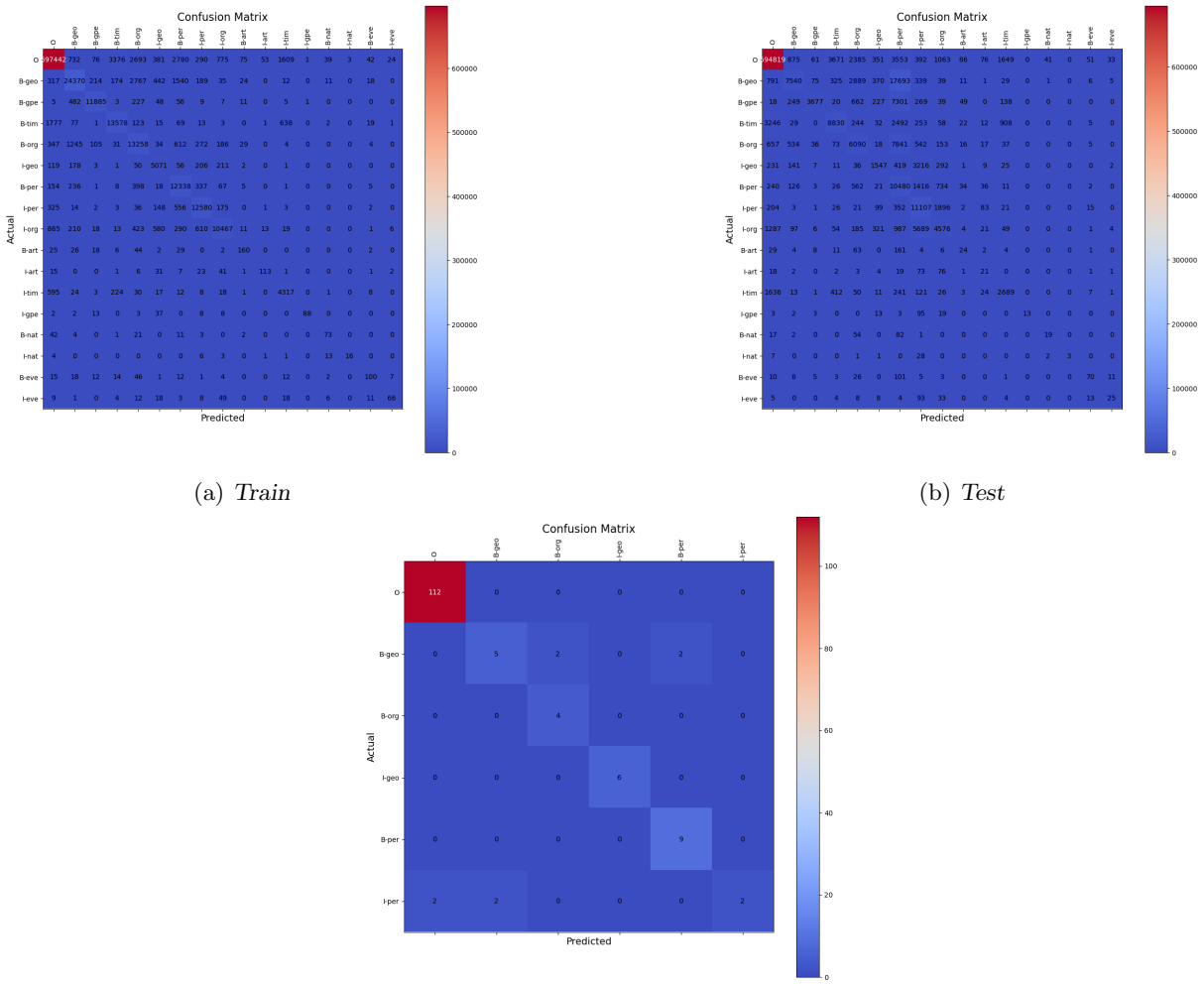
The following Table 4 and Figure 3 illustrates the results obtained using the structured perceptron model with the new features.

	Accuracy	F1-score
Train	0.9604	0.9617
Test	0.8975	0.9004
Tiny test	0.9452	0.9379

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

It is important to note that the accuracy and F1-score of the training set for the structured perceptron model with the new features are almost the same as those of the model with default features. However, the F1-score has increased by 5% and 4% in the test and tiny test sets, respectively. Additionally, accuracy has improved by 1% in the test set and 4% in the tiny test set.

Additionally, by incorporating the new features, we successfully addressed the earlier iden-



(a) *Train*

(b) *Test*

(c) *Tiny Test*

Figure 3: *Confusion matrix of the train, test and Tiny test sets for the structured perceptron with new features.*

tified errors in the small test dataset. Specifically, the incorrectly spelled word "Microsof" is now correctly categorized as B-org, and the name "Alice" has been accurately identified as B-per. Analysis of the confusion matrix (Figure 3) and accuracy results (Table 4) indicates that the introduction of these new features effectively improved the classification accuracy of previously misclassified words compared to the default feature set.

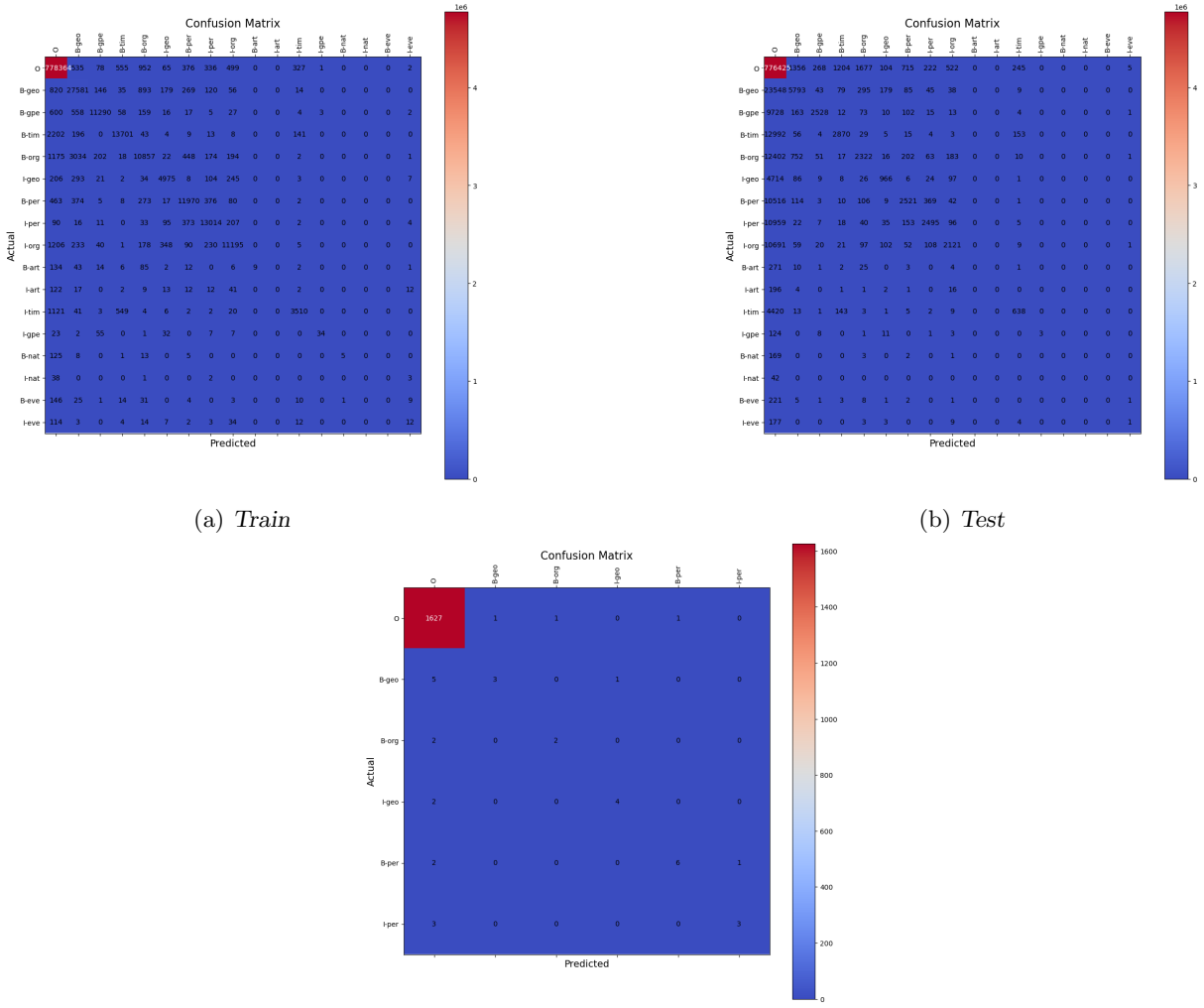
4.2 Deep Leaning model

The following Table 5 and Figure 4 illustrates the results obtained using Bi-LSTM.

	Accuracy	F1-score
Train	0.8400	0.9948
Test	0.1736	0.9702
Tiny test	0.5294	0.9872

Table 5: *Results in the train, test and tiny test sets for the structured perceptron with new features.*

These represent the most undesirable results observed to date. It seems that the model's



(a) *Train*

(b) *Test*

(c) *Tiny Test*

Figure 4: *Confusion matrix of the train, test and Tiny test sets for Bi-LSTM model.*

learning capability is inadequate in accurately assigning tags to out-of-vocabulary words or certain misspelled words. For instance, in the small test dataset, an error in classification occurs when "Jack London" is incorrectly labeled as B-per B-geo instead of the correct sequence B-per I-per. This discrepancy likely stems from the model associating the word "London" with the B-geo tag during its training phase, thereby hindering its ability to predict the correct tag accurately.

5 Conclusions

In summary, our study aimed to tackle the NER challenge in NLP using various methodologies. Initially, we provided a theoretical overview of the structured perceptron approach. We then investigated two implementations of the structured perceptron model: one using default features and another enhanced with additional features, which produced superior outcomes.

Furthermore, we ventured into Deep Learning by deploying a bidirectional LSTM model. Despite optimistic expectations, the achieved results fell short of our initial aspirations. Consequently, after evaluating all options, we concluded that the structured perceptron model with

added features stood out as the most effective choice.

Moreover, we acknowledged the necessity to optimize the computational efficiency of our selected model. Through the identification and optimization of time-intensive code segments, we successfully integrated Cython, resulting in enhanced performance and reduced execution times.