
UNIVERSITAT DE BARCELONA

# Delivery 2: Named Entity Recognition

*Natural Language Processing*

Manuel Andrés Hernández Alonso
Alexandru Ioan Oarga Hategan
Carmen Casas Herce

# Introduction

This work focuses on implementing and improving existing algorithms to solve a *Named Entity Recognition (NER)* task. The main objective of this task is to identify specific entities from a dataset (corpus). The categories into which words can be classified vary from a range of themes, such as names, geographical locations, companies, dates, etc; or, even more, other particular entities [1]. This way, it is possible to convert plain, unlabelled text to structured data, pointing out and classifying the most relevant tokens from some data.

However, this problem has some drawbacks too, as usually the amount of data available for training is very limited and, even if it is available, generalizing from it can be challenging [2]. In the context of supervised learning, the methods with best results use Hidden Markov Models (HMM) (used in the Structured Perceptron, developed in the following section), SVM or Conditional Random Fields (CRF). In order to overcome the limitations stated, semi-supervised methods were developed, using unstructured text [3]; and unsupervised learning was used to create new features that could be combined with some existing methods.

In this work we will focus on developing the *Structured Perceptron (SP)*, providing a theoretical framework of the algorithm along with some existing limitations. Using the skseq library provided during the course as a baseline, we have implemented the project as a baseline. Furthermore, we have added some additional features of the data as input and we have studied how the SP behaves. Results are developed in the following section.

On the other hand, when a large amount of data corpora is given, deep learning models can be leveraged to automatically learn intrinsic features that can later be tailored for classification tasks such as NER. Motivated by the success of *Large Language Models (LLM)* in multiple Natural Language Processing (NLP) tasks [7, 8, 9], in this work we suggest the use of LLMs, in particular, an instance of Bidirectional Encoder Representations Transformer (BERT) [7] model fine-tuned for our NER classification task. This means that, instead of training a deep learning model from the beginning, a pre-trained model trained with general-purpose data is then re-trained for a particular downstream task. Given that the base model is already trained, a reduced amount of data is then required for fine-tuning.

Since the fine-tuning of LLMs is computationally expensive, alternative methods such as *Low Rank Adaptation (LoRA)* [10], have also been proposed as efficient alternatives for this effort. In this work, two different approaches are suggested: (1) fully fine-tuning a BERT model, and (2) fine-tuning a BERT model by training only a LoRA module. At the end of the document, a comparison of the performance obtained in each case is presented.

# The Structured Perceptron

## Theoretical model

The Structured Perceptron is a variant of the Perceptron algorithm first developed in 2002. [4] It is based on the perceptron algorithm, a linear classifier. However, it is combined with generative models in order to be able to make it applicable to a wide range of tasks as machine translation, parsing or positional tagging or named entity recognition with HMMs (the one we are implementing in this work).

Hidden Markov Models are statistical models that states that the observable output is dependent on a probabilistic distribution that is a function of the input and the previous outputs. If we have a sequence as an input, the HMM is defined as:

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

The observation states **X** are not observable directly (they are considered hidden states), and the probability of the final output sequence can be decomposed in probabilities that depend on the i-th pair of observation-state of the sequence $P(x_i|y_i)$ and the transition probabilities of the outputs $P(y_{i+1}|y_i)$.

Hidden Markov Models rely on 3 main assumptions:

- As a first-order chain, the probability of a state $y_i$ only depends on the previous state $y_{i-1}$:

  $$P\left(y_i|y_1, ..., y_{i-1}\right) = P\left(y_i|y_{i-1}\right)$$

  This doesn't mean that there is no dependence with previous states, but the impact is already taken into account implicitly in the last state.

- The probability of the observation $x_i$ only depends on the state $y_i$:

  $$P(x_i|y_1...y_i, ..., y_N; x_1, ..., x_i, ..., x_N) = P(x_i|y_i)$$

- Furthermore, the probability of a transition from one state to another is independent of the position of those states in the sequence:

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

When training a HMM one approach is to establish the hidden states as the learning parameters, given observations and a matrix of probabilities. The process of training can also update this matrix in order to get a better prediction of the result.

One algorithm applied when training HMM is the Viterbi Algorithm (VA). In the Viterbi algorithm, given a set of hidden states and the observations, the objective is to maximize:

$$y^* = argmax_{y \epsilon \Lambda^N} P\left(Y = y_1, ..., y_N | X = x_1, ..., x_N\right)$$

The way the algorithm computes this optimization is by transforming the probabilities into learnable weights and a feature vector as follow:

$$y^* = argmax_{y \epsilon \Lambda^N} \cdot \omega \cdot f(x, y)$$

And if we take into account eq. (1), we can rewrite the optimization problem into:

$$\arg\max_{y \in \Lambda^N} \sum_{i=1}^{N} \text{score}_{\text{emiss}}(i, x, y_i) + \text{score}_{\text{init}}(x, y_1) + \sum_{i=1}^{N-1} \text{score}_{\text{trans}}(i, x, y_i, y_{i+1}) + \text{score}_{\text{final}}(x, y_N)$$

In the Viterbi Algorithm a variant Forward-Backward Algorithm is computed. In the first place, the Forward step calculates the best path to a state by computing the feature weights for all possible states and finding the most optimal one (in the Forward-Backward Algorithm what is computed are the log probabilities, searching for the most negative one). In the end, the Viterbi Algorithm calculates the best possible predicted sequence given a set of hidden states and original observations.

Now, the structured perceptron algorithm takes this predictions $y^*$ as an start point and compares them to the original observation. If there is a difference between both, the weight matrix is updated by trying to increase the score of the positive examples and decrease the score of the negative ones as follow:

$$\omega \leftarrow \omega + \varphi(X, Y) - \varphi(X, Y^*)$$

Where **φ** are the feature vectors, given as a parameter of the model. The main difference between the perceptron and the structured perceptron is that the first one is a linear classifier, so the output will be a binary value based on a probability for a certain output. However, for the second one we are using structured objects, as sequences, to get a likewise prediction, rather than just a value.

Summing up, the Structured Perceptron algorithm works as follow:

Given a pair of observations and states $(x, y)$, and a feature vector:

Initialize a matrix of weights ω

For 1,...,I iterations:

For each pair $(x_i, y_i)$:

$y_i^*$ = VA(ω,X) (optimization function)

**φ**$^*$ =FV $(X_i, y_i^*)$

$\boldsymbol{\varphi}$ =FV ($X_i, y_i$ )

if $y_i^* \neq y_i$:

$$\omega \leftarrow \omega + r(\varphi - \varphi*)$$

Where r is the learning rate. The Feature Vector (FV) function depends on the transition and emission probabilities described in the equations above.

## Words never seen before

One characteristic of the Structured Perceptron algorithm is that it is able to learn based on the inputs given, i.e., a set of labelled words and sequences (in our exercise). This is why it is reasonable to think how the model classifies a new word that was not available in the training dataset.

As the weight matrix would not take into account this new token, it wouldn't be considered when computing the best prediction $y_i^*$. However, as we are adding a feature vector as part of the algorithm, and this vector is not dependent on the weights but on the initial state, and it is based on the characteristics of the token (capital letters, length, etc) rather than the true that needs to be predicted, it will get a score in this part of the algorithm, in a way that these words will have a weight assigned in following iterations and, therefore, will have a chance to be properly classified.

## Additional features

From the HMM features we can extend the features used for prediction. We can append these features in the emission features of the Structured Perceptron. Such features can be defined manually with a distinctive tag to the list. We added them programmatically to the module. The features used are as follows:

- The word is title-cased, that is, the word starts with uppercased then followed by lowercase.
- The word is all uppercase.
- The word is all lowercase.
- The word is a digit, that is, all of the characters are numbers.
- The word contains a number.
- The word is composed only of alpha-numeric characters.
- The word is composed only of alphabetic characters.
- The word ends by different suffixes:
    - "day", mainly for weekdays.
    - "ber", mainly for months.
    - "ing", some verbs can be used in -ing form as a noun.
    - Person suffixes, "man", "woman", "person".
    - Geographical.
- The word is a stopword, as given from a list of stopwords.
- The word is a preposition, as from the list of English prepositions.

- The word contains a hyphen; dot; or quote; as different features.
- The word has different prefixes:
    - Chemical prefixes.
    - International Standard system units.
    - Honorifics.
- The word has a corporate abbreviation; Corp., Inc., etc..

These new features should help capture the different labels better than the baseline model, as well as give a better base for prediction.

## Results

In this section we will look into the results of the different models over the three data splits (Train, Test, Tiny Test). Particularly, we will see the following evaluation metrics and assessments:

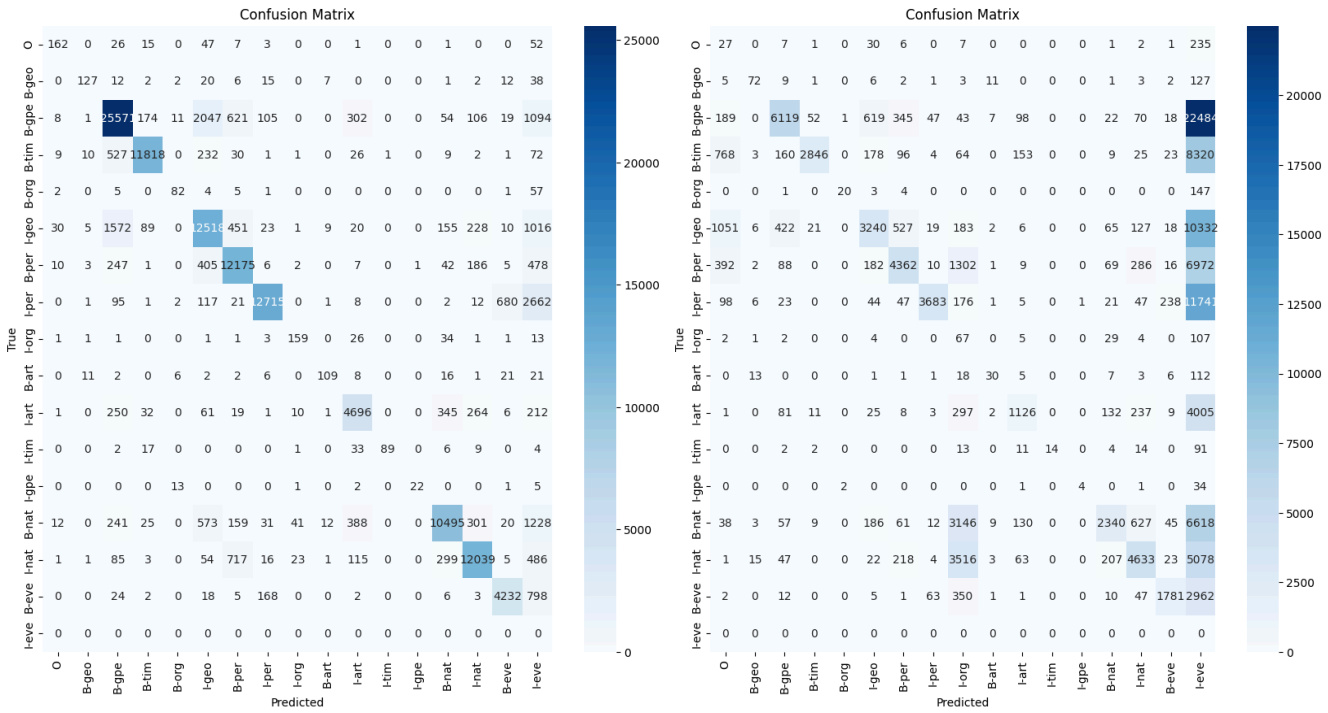
- **Accuracy:** Without taking the “O” label into account.
- **Confusion Matrix:** Of all labels, without the “O” label.
- **Weighted F1-score:** Computed with all labels.
- **Tiny_test predictions:** The tag predictions for the tiny test

**Vanilla Structured Perceptron**

Firstly, let us look at the accuracies and F1-scores for the vanilla structured perceptron on the different data splits:

| | Accuracy | F1-Score |
|---|---|---|
| Train | 83.11% | 96.82% |
| Test | 23.68% | 85.32% |
| Tiny Test | 55.88% | 88.49% |

Here we can see that the accuracy on the labels for the test set is 23%, with low performance in these tests. We can then see the confusion matrix of this model.




The first confusion matrix corresponds to the train set, and the second one the test set. we can see that it has a lot of confusion with the label I-eve on the test set, with it heavily predicting I-eve.

Finally, let us see the Tiny Test prediction tags:

```
The/O programmers/O from/O Barcelona/B-org might/O write/O a/O sentence/O without/O a/O spell/O checker/O ./O

The/O programmers/O from/O Barchelona/O cannot/O write/O a/O sentence/O without/O a/O spell/O checker/O ./O

Jack/B-per London/B-geo went/O to/O Parris/O ./O

Jack/B-per London/B-geo went/O to/O Paris/B-geo ./O

Bill/B-per gates/O and/O Steve/B-per jobs/O never/O thought/O Microsoft/B-org would/O become/O such/O a/O big/O company/O ./O

Bill/B-per Gates/I-per and/O Steve/B-per Jobs/I-per never/O thought/O Microsoft/B-org would/O become/O such/O a/O big/O company/O ./O

The/O president/O of/O U.S.A/O thought/O they/O could/O win/O the/O war/O ./O

The/O president/O of/O the/O United/B-geo States/I-geo of/I-geo America/I-geo thought/O they/O could/O win/O the/O war/O ./O

The/O king/O of/O Saudi/B-per Arabia/I-per wanted/O total/O control/O ./O

Robin/O does/O not/O want/O to/O go/O to/O Saudi/B-per Arabia/I-per ./O

Apple/O is/O a/O great/O company/O ./O

I/O really/O love/O apples/O and/O oranges/O ./O

Alice/O and/O Henry/B-per went/O to/O the/O Microsoft/B-org store/O to/O buy/O a/O new/O computer/O during/O their/O trip/O to/O New/B-geo York/I-geo ./O
```

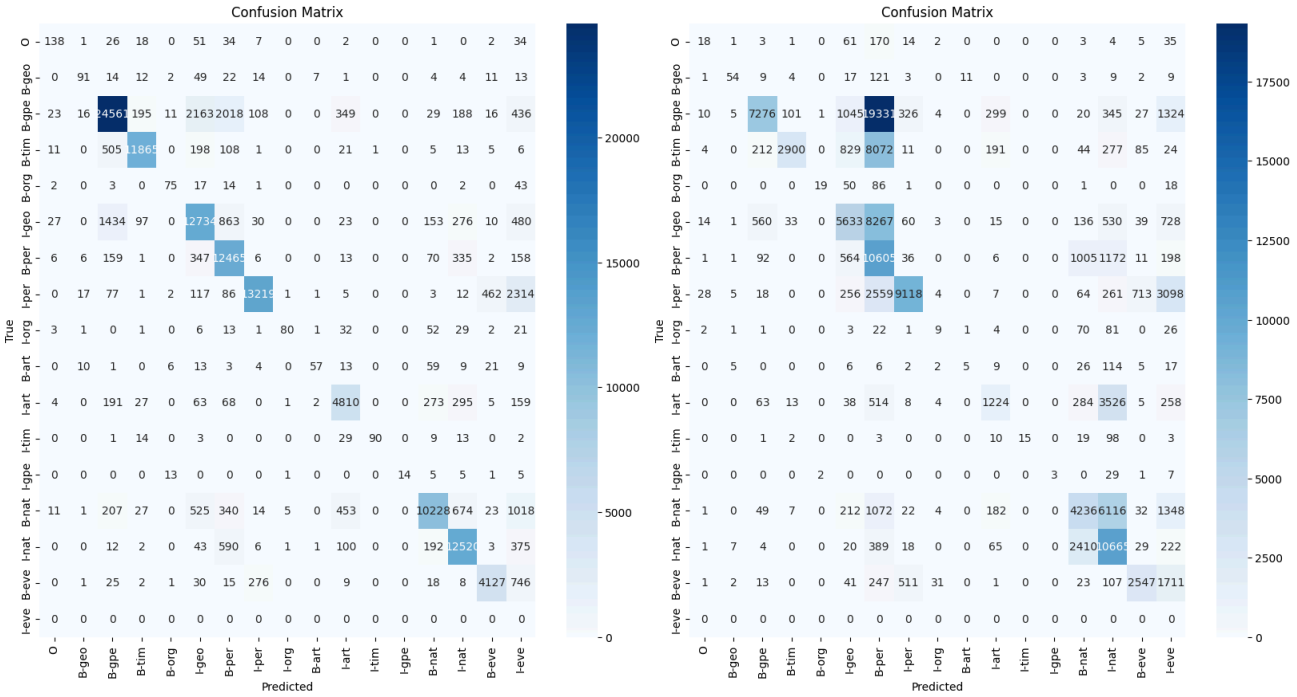
So, here we can see that it doesn't capture the misspellings, and even some simple things like U.S.A. This shows that there's still room for improvement.

**Structured Perceptron with Additional Features**

Similarly to the previous case, we computed the accuracy and got the results as shown in the following table:

| | **Accuracy** | **F1-Score** |
|---|---|---|
| Train | 83.16% | 96.39% |
| Test | 42.37% | 90.09% |
| Tiny Test | 82.35% | 88.49% |

The accuracy for this model is a significant improvement to the previous one, it captures almost double of labels with an accuracy of 42% on the test set. Subsequently, we can see the confusion matrices for this model.




Now, as we can see in the matrix for the test set (Matrix 2), the confusion seems to lie more in the B-per tag, rather than in the I-eve tag prediction. The B-per tag is heavily favoured in this case, but overall the predictions are better than the previous model.

Lastly, let's check the performance on the tiny test dataset.

---

```
The/O programmers/O from/O Barcelona/B-geo might/O write/O a/O sentence/O without/O a/O spell/O checker/O ./O

The/O programmers/O from/O Barchelona/B-per cannot/O write/O a/O sentence/O without/O a/O spell/O checker/O ./O

Jack/B-per London/B-geo went/O to/O Parris/B-per ./O

Jack/B-per London/B-geo went/O to/O Paris/B-geo ./O

Bill/B-per gates/O and/O Steve/B-per jobs/O never/O thought/O Microsoft/B-org would/O become/O such/O a/O big/O company/O ./O

Bill/B-per Gates/I-per and/O Steve/B-per Jobs/I-per never/O thought/O Microsoft/B-org would/O become/O such/O a/O big/O company/O ./O

The/O president/O of/O U.S.A/B-geo thought/O they/O could/O win/O the/O war/O ./O

The/O president/O of/O the/O United/B-geo States/I-geo of/I-geo America/I-geo thought/O they/O could/O win/O the/O war/O ./O
```

```
The/O king/O of/O Saudi/B-geo Arabia/I-geo wanted/O total/O control/O ./O

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

Apple/B-org is/O a/O great/O company/O ./O

I/O really/O love/O apples/O and/O oranges/O ./O

Alice/B-per and/O Henry/B-per went/O to/O the/O Microsoft/B-org store/O to/O buy/O
a/O new/O computer/O during/O their/O trip/O to/O New/B-geo York/I-geo ./O
```

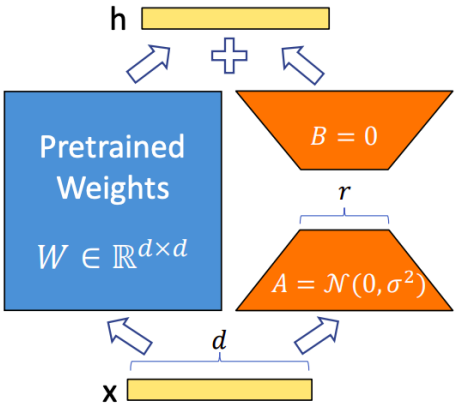
---

The model is still not able to capture the misspellings properly, but it is labeling them as some named entity. Additionally, simple things that weren't properly labeled before such as Apple, Robin, U.S.A., and others are now properly labeled.

# Deep learning model

As has been mentioned in the Introduction, in this work, two different approaches are followed for fine-tuning an LLM, in this case, a BERT model, this is, fine-tuning the full model, and training a LoRA module for fine-tuning. The BERT model that was used in this work is the bert-base-uncased [10], this is, a BERT model where the vocabulary is not case-sensitive. This model is one of the first LLMs introduced in the original BERT work, and it is still widely used in many NLP tasks. The model has 110M parameters and a total of 12 layers.

Figure 1: LoRA module [10].




Before proceeding, let us first recall the definition of the LoRA. The LoRA method assumes that many tasks do not require large weight matrices to be learned, but instead have weights that tend to have a low-rank structure. This is, the weights of the model can be approximated by a low-rank matrix. Based on this assumption, the LoRA module is introduced. The LoRA module is composed of encoder A and decoder B modules (see Figure 1). The goal of this structure is to reduce the dimensionality of an input feature map, to a lower-rank representation, and then to reconstruct the original feature map. Encoder A is initialized sampling from a normal distribution, while decoder B is initialized as zero so that B * A is zero at the beginning of the training. The output of the LoRA module is then added to the original output of the weight matrix. When the LoRA module is trained, the weights of the weight matrix are frozen, and the LoRA module is trained, this is, the LoRA goal is to learn a transformation of the output of the weight matrix such that the output is fine-tuned for a specific task.

In this work, we used a common approach to fine-tuning BERT with LoRA, which is to use a rank = 8, for the low-rank approximation. In the case of transformer models such as the one we are using, a LoRA model is included for each weight matrix that is applied to the Key (K), Query (Q) and Value (V) before computing the attention coefficients. Note that LoRA modules are not used in the Fully Connected (FC) layers that proceed the attention coefficients, only to the Q, K, V transformations.

## Results

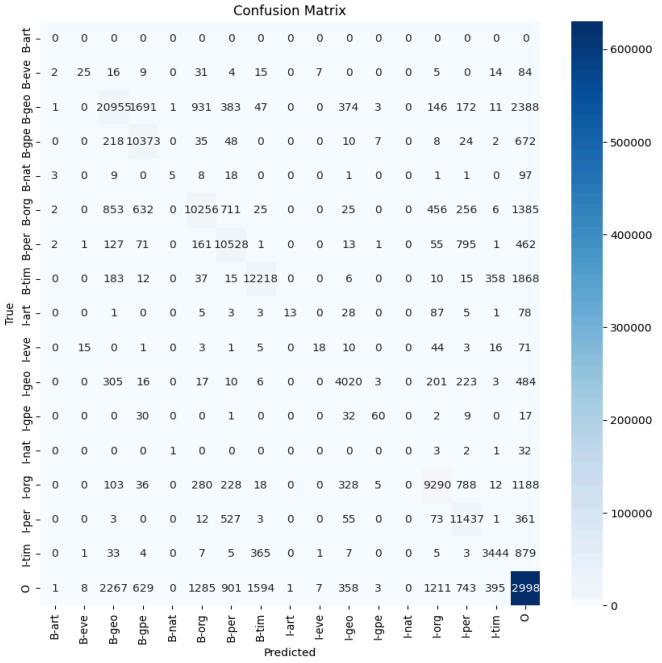
The performance comparison of both approaches can be found in the table below. In this table first we can see that the single BERT model outperforms the BERT + LoRA model in both accuracy and F1-score in the test dataset. Similarly in the train dataset, we can see that the LoRA-based solution follows closely the only-BERT solution. This can be explained by the fact that, since in the only-BERT solution we are training the whole model, it is easier to get an overall lower error on the model performance as in general it is more powerful. However, it is worth mentioning that, for the same reason, the only-BERT solution easily reaches an overfitting regime while the LoRA-based solution still seems to be able to

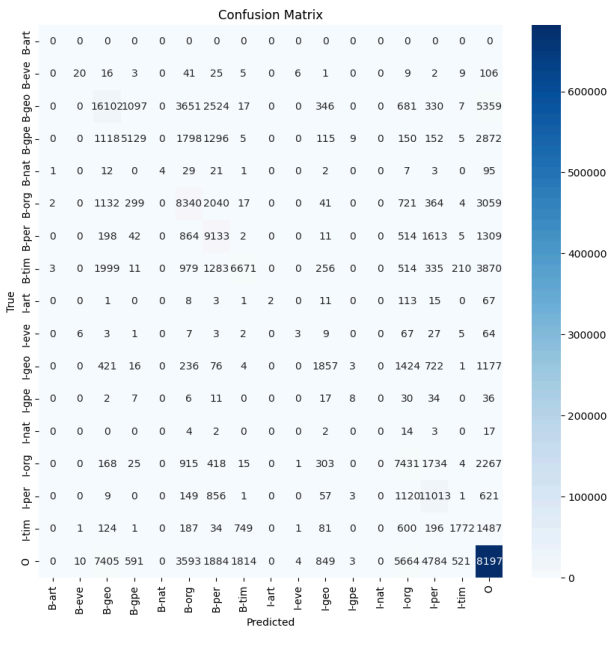
improve the performance. As a result, we believe that the LoRA-based solution should be able to obtain similar results to the BERT-only solution albeit with a longer computation time.

Generally, we can see that the best result that we obtain is a F1-score of 89.55. Note that this is a weighted F1-score which suggests that the model shows an overall good performance in the test dataset. If we now compare these results to the ones obtained with the structured perceptron, it seems that the bert-case-uncased model performs better than the structured perceptron in terms of accuracy, but underperforms if we look at the F1-score. Since accuracy is more sensitive to the majority class, this means that the structured perceptron does a better job at predicting classes other than the “O” label, while the BERT model has a better performance in detecting such class.

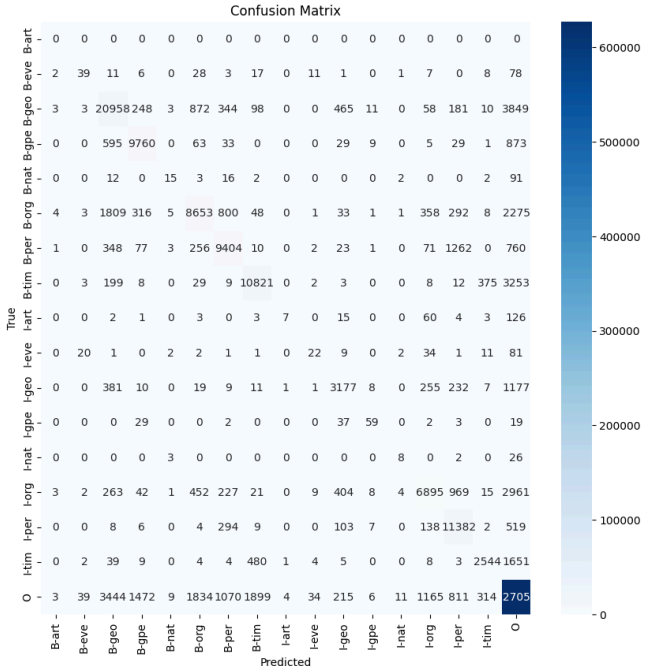
| **Model** | F1-score training | F1-core test | F1-core tiny | Accuracy training | Accuracy test | Accuracy tiny |
|---|---|---|---|---|---|---|
| bert-base-uncased | 95.58% | 89.55% | 100.00% | 79.82% | 52.63% | 100.00% |
| bert-base-uncased + LoRA | 93.90% | 88.63% | 98.51% | 72.15% | 45.59% | 94.11% |

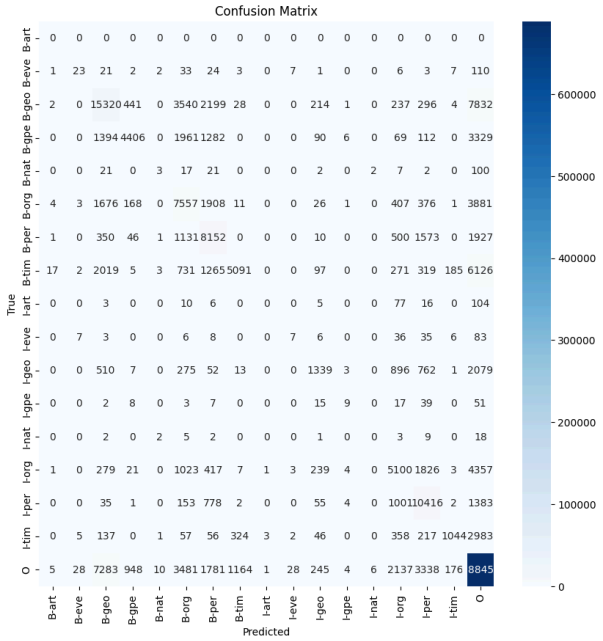
In the figures below we can see the confusion matrix from the train and test set respectively from the full-training BERT model. Compared with the structured perceptron, we can confirm that the model does a better job of predicting the majority class. Furthermore, we can see that most of the errors are explained by words that are incorrectly predicted with the label “O”. The classes that seem to be more problematic for the model are the “B-geo”, ·B-gpe”, “I-org” and “I-per”.



If we look now at the confusion matrix of the bert-base-uncased + LoRA model (see figures below), we will see a similar result as with the previous model, this is, the model shows a good performance in detecting the majority class, with slightly more errors than with the previous model.




Regarding the tiny, test, we can see that, unlike the structured perceptron, this task seems easy for both models. The bert-base-uncased model is able to get a perfect prediction on the dataset, while the bert-base-uncased + LoRA has only one error. The output of the last is shown below, where the error is highlighted.

---

```
The/O programmers/O from/O Barcelona/B-geo might/O write/O a/O sentence/O without/O a/O spell/O checker/O ./O

The/O programmers/O from/O Barchelona/B-geo cannot/O write/O a/O sentence/O without/O a/O spell/O checker/O ./O

Jack/B-per London/I-per went/O to/O Parris/B-geo ./O

Jack/B-per London/I-per went/O to/O Paris/B-geo ./O

Bill/B-per gates/I-per and/O Steve/B-per jobs/I-per never/O thought/O Microsoft/B-org would/O become/O such/O a/O big/O company/O ./O

Bill/B-per Gates/I-per and/O Steve/B-per Jobs/I-per never/O thought/O Microsoft/B-org would/O become/O such/O a/O big/O company/O ./O

The/O president/O of/O U.S.A/B-geo thought/O they/O could/O win/O the/O war/O ./O
```

```
The/O president/O of/O the/O United/B-geo States/I-geo of/I-geo America/I-geo thought/O they/O could/O win/O the/O war/O ./O

The/O king/O of/O Saudi/B-geo Arabia/I-geo wanted/O total/O control/O ./O

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

Apple/B-org is/O a/O great/O company/O ./O I/O really/O love/O apples/O and/O oranges/O ./O

Alice/B-per and/O Henry/B-per went/O to/O the/O Microsoft/B-org store/O to/O buy/O a/O new/O computer/O during/O their/O trip/O to/O New/B-geo York/I-geo ./O
```

---

## Hyperparameters

This section includes the hyperparameters used for training each of the deep learning models. The hyperparameters can be found in the next table. In each case, the parameters were optimized given the computational budget allowed by the Google colab environment. The best model selection was done using the lower validation loss. To generate a validation dataset, the training dataset was split into 90% training and 10% validation. The loss function used was a multi-class cross entropy.

| **Hyperparameter** | **bert-base-uncased** | **bert-base-uncased + LoRA** |
|---|---|---|
| Optimizer | AdamW | AdamW |
| Learning rate | 3e-5 | 5e-5 |
| Epochs | 20 | 40 |
| Batch size | 64 | 64 |
| LoRA scaling factor | X | 32 |
| LoRA dropout | X | 0.1 |

**Note:** recall that transformer models make use of tokenizers to split text into individual tokens. It might thus happen that a word is split into different tokens. In such cases, to get the predicted label for that word we used the label predicted for the first token of the word.

## Experimental setup

As mentioned earlier, Google colab was used for the deep learning training section. In the case of reproducing these results on a personal computer, the torch version used was torch==2.2.1 with cuda 1.2.1. For the structured perceptron, it was trained using the module provided during the course.

# References


[1] Naseer, Salman & Ghafoor, Muhammad & Sohaib, & Khalid Alvi, Sohaib & Kiran, Anam & Rehman, Shafique Ur & Murtaza, Ghulam & Campus, Jehlum & Jehlum, Pakistan. (2022). *Named Entity Recognition (NER) in NLP Techniques, Tools Accuracy and Performance.*

[2] Guillaume Lample and Miguel Ballesteros and Sandeep Subramanian and Kazuya Kawakami and Chris Dyer. (2016). *Neural Architectures for Named Entity Recognition*

[3] Arya Roy. (2021). *Recent Trends in Named Entity Recognition (NER)*

[4] Collins, Michael. (2002). *Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms.*

[5] Huang. (2008) *Forest Reranking: Discriminative Parsing with Non-Local Features*

[6] Liang et al. (2006). *An End-to-End Discriminative Approach to Machine Translation*

[7] Devlin, Jacob, et al. (2018). *"Bert: Pre-training of deep bidirectional transformers for language understanding."*

[8] Radford, Alec, et al. (2019) *"Language models are unsupervised multitask learners."*

[9] Wei, Jason, et al. (2022). *"Emergent abilities of large language models."*

[10] Hu, Edward J., et al. (2021) *"Lora: Low-rank adaptation of large language models.".*