**Natural Language Processing**

Master on Fundamentals of Data science
Universitat de Barcelona

---

# Delivery 2
# Named Entity Recognition

---

Montse Comas, Blai Ras and Fritz Pere Nobbe

June 20, 2021

# 1 Introduction

Named Entity Recognition (from now on, NER) is an information extraction method which goal is to identify a certain group of entities or "tags" from a corpus. These tags can be names of cities, people, geographic location, time, currency, etc. In this problem, our set of tags has been defined as follows [1]:

- **geo**: geographical tag
- **gpe**: assumed to be a geopolitical entity
- **tim**: time measure
- **org**: organization tag
- **per**: person name
- **art**: assumed to be a tag for "artifact"
- **nat**: natural phenomenon
- **eve**: event tag
- **O**: a 'wild card' token (no chunk)

In addition, tags can be formatted using the IOB (inside, outside, beginning) format, where we add a prefix to each tag indicating tag order. For example, the sentence `European Union` would be tagged as `European/B-org, Union/I-org`.

Our corpus is divided in the given train and test sets. The dimension of our training dataset is 38366 sentences, which include 839149 words. Our test set includes 38367 sentences with 837339 words. Therefore, an example of our problem can be:

```
X = ['Henry', 'went', 'yesterday', 'to', 'the', 'United', 'States', '.']

Y = ['B-per', 'O',    'B-tim',     'O',  'O',   'B-geo',  'I-geo',  'O']
```

# 2 Structured Perceptron

## 2.1 Hidden Markov Models

To deal with Sequential Tagging Problems, we have previously studied HMM's. These models assume two random processes (taking into account input and output sequences) and the goal is to maximize a given probability that depends on an initial state, the

[1] Source: Kaggle

hidden ones (corresponding to the length of the sequence, and a final one for the stopping probability.

A HMM is a generative sequence model that defines the probability distribution over input-output pairs as follows:

$$P(X_1 = x_1, \ldots, X_N = x_N, Y_1 = y_1, \ldots, T_N = y_N) = \\ = P_{init}(y_1|\text{start}) \cdot \Big( \prod_{i=1}^{N-1} P_{trans}(y_{i+1}|y_i) \Big) \times P_{final}(\text{stop}|y_N) \cdot \prod_{i=1}^{N} P_{emiss}(x_i|y_i) \tag{2.1}$$

We are not explaining the details of how these probabilities are computed, but the following sections are strongly related to this model. Therefore, let us define with a little more detail what these probabilities refer to, as well as some notation that can be reused.

Let $\Sigma := w_1, ..., w_J$ bet a set of words, $\Lambda := c_1, ..., c_K$ bet the set of labels. Then,

- $P_{init}(c|\text{start})$: we need $|\Sigma|$ floats.
- $P_{trans}(c|\hat{c})$: we need $|\Sigma| \cdot |\Sigma|$ floats.
- $P_{final}(\text{stop}|c)$: we need $|\Sigma|$ floats.
- $P_{emiss}(w|c)$: we need $|\Sigma| \cdot |\Lambda|$ floats.

Moreover, notice that this model works under the first order HMM independence assumptions over the joint distribution $P(X = x, Y = y)$ :

- **Independence of previous states**: the probability of being in a given state only depends on the state of the previous position.
- **Homogeneous transition**: the probability of making a transition from two states is independent of the particular position in the sequence.
- **Observation independence**: the probability of observing a given word at a given position is fully determined by the state at that position.

We can take the logarithms of the previous probabilities as the weights of a linear classifier and the corresponding counts as feature vectors. In this case, we can see HMM as a linear classifier. Therefore, we can use the algorithm to train non-normalized scores (not working with probabilities anymore) and use additional features to create a discriminative model. In section 2.2, we focus on this idea.

### 2.1.1 Decoding

There are two important problems that one might want to solve with HMM,

- Posterior Decoding: $y_i^* = \arg\max_{y_i \in \Lambda} P(Y_i = y_i | X_1 = x_1, \ldots, X_N = x_N)$
- Viterbi Decoding: $y_i^* = \arg\max_{y \in \Lambda^N} P(Y_1 = y_1, \ldots, Y_N = y_N | X_1 = x_1, \ldots, X_N = x_N)$

Although both have their advantages and disadvantages, we will only focus on Viterbi decoding since it is the algorithm we want for our Structured Perceptron.

We need to compute $P(Y_1 = y_1, \ldots, Y_N = y_N | X_1 = x_1, \ldots, X_N = x_N)$ for $\forall y \in \Lambda^N$ in order to compute the maximum value and find the correct sequence of labels.

- For the first position and for a given state $y_1 \in \Lambda$ let us define:

$$\text{Viterbi}(1, x, y_1) = P_{init}(y_1 | \text{start}) \times P_{emiss}(x_1 | y_1)$$

- For every position $i \in [2, N]$:

$$\text{Viterbi}(i, x, y_i) = P_{emiss}(x_i | y_i) \times \max_{y_k \in \Lambda} \Big( P_{trans}(y_i | y_k) \times \text{viertbi}(i-1, x, y_k) \Big)$$

- We can define the Viterbi quantity at the stop position as:

$$\text{Viterbi}(N+1, x, \text{stop}) = \max_{y \in \Lambda} \Big( P_{final}(\text{stop} | y) \times \text{Viterbi}(N, x, y) \Big)$$

Using the recurrence rule, we find that the Viterbi algorithm can be described as follows.

$$y^* = \text{Viterbi}(N+1, x, \text{stop}) = \arg\max_{y_1 \ldots y_N} P(Y_1 = y_1, \ldots, Y_N = y_N, X_1, = x_1, \ldots, X_N = x_n)$$

Once the Viterbi value at position $N$ is computed, the algorithm can backtrack using the following recurrence:

- $\text{backtrack}(N+1, x, \text{stop}) = \arg\max_{y \in \Lambda} \Big( P_{final}(\text{stop} | y) \times \text{Viterbi}(N, x, y) \Big)$
- $\text{backtrack}(i, x, y_i) = \arg\max_{y_{i-1} \in \Lambda} \Big( P_{trans}(y_i | y_{i-1}) \times \text{Viterbi}(i-1, x, y_{i-1}) \Big)$

**Note:** We need to keep track of the backtrack quantities when we compute the Viterbi quantities in order to be able to compute everything.

## 2.2 Structured Perceptron

Discriminative sequence models aim to solve the following:

$$\arg\max_{y \in \Lambda^N} P(Y = y \,|\, X = x) = \arg\max_{y \in \Lambda^N} \boldsymbol{w} \cdot \boldsymbol{f}(x, y)$$

where $\boldsymbol{w}$ is the model's weight vector, and $\boldsymbol{f}(x, y)$ is a feature vector. Notice that now both $y$ and $x$ are $N$-dimensional vectors.

In this case, the scoring of the sequences is computed as the product of the weights with the feature vector. As we saw in 2.1, we can separate the feature vector in the four parts of the sequence (*init, trans, final, emiss*). The feature vector depends on the input and:

- The first output label for $f_{init} \Rightarrow f_{init}(x, y_1)$
- A pair of output labels for $f_{trans} \Rightarrow f_{trans}(i, x, y_i, y_{i+1})$
- The last output label for $f_{final} \Rightarrow f_{init}(x, y_N)$
- The corresponding output label for $f_{emiss} \Rightarrow f_{init}(i, x, y_i)$

Then, our lineal classifier is defined as follows:

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

We can rewrite the previous expression as:

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

### 2.2.1 Definition of new features

Previously, we saw the main features used for the HMM related to the four types of events, which depend on a certain position and the output state. However, by focusing on the discriminative approach, instead of modeling $P(X, Y)$ we model $P(X|Y)$. Those will also be used for the Structured Perceptron, but now we can add new features, implicit in the HMM. These basic features are:

| **Condition** | **Name** |
|---|---|
| $y_i = c_k$, $i = 0$ | Initial feature |
| $y_i = c_k$, $y_{i-1} = c_j$ | Transition feature |
| $y_i = c_k$, $i = N$ | Final feature |
| $x_i = w_k$, $y_i = c_k$ | Basic Emission feature |

Then, we can use more than one feature at the same time for a given word and position. To this end, we create new features, specified at the end of section 3.2. We also explain how these features are structured and used by the model, with a bit more detail in section 3.1.

The decoding process, finding the most likely label $y_i$ for the word $x_i$, stays the same as

in a standard HMM. Meaning, we can use the previously mentioned Viterbi Algorithm to compute $y^* \in \Lambda^N$ that maximizes $P(Y = y^*|X = x)$.

# 3 Implementation

The main goal of this project is to understand how the Structured Perceptron works. Therefore, we compare the results of this model with the default features (same as the ones in HMM) against the newly added features.

To this end, we trained two different Structured Perceptrons with different input features and compared several indicators that you can check in section 4 and that show if the added ones really help for the predictive task.

## 3.1 Feature mapper

Thanks to the fact that this approach provides extra features, there is no need to generate all the possible combinations of basic transmission and emission features, but the ones observed in the training set.

The implementation of the feature mapper is based on the concatenation of specific prefix strings with the corresponding label indicator. This enables the model to identify to which part of the formula each feature corresponds.

For the sake of clarifying, let us show this with an example of the `train_seq`. Consider the following Sequence:

`Doctors/O say/O they/O expect/O Mr./B-per Sharon/I-per will/O make/O a/O full/O recovery/O ./O '`.

The main features, related to the four types we mentioned, are triggered as follows:

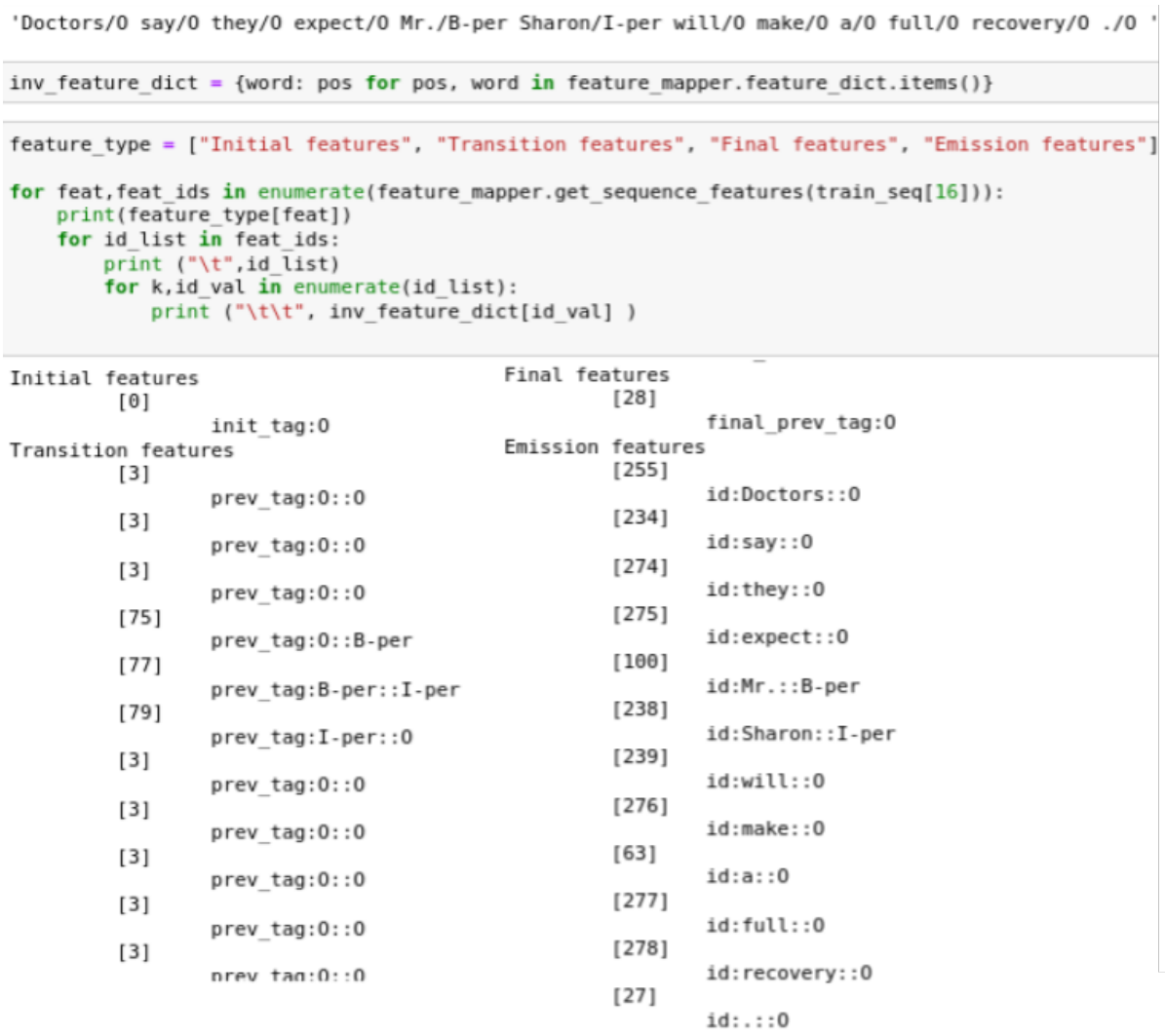


Figure 3.1: activated features for a given sentence

As you can see, the string before the first ':' enables the program to identify to what they refer:

- init_tag: refers to *Initial* features. In 3.1 we see that the feature related to 'The initial tag being `O`' is activated.
- prev_tag: refers to the *Transition* features. In the image, one can observe many activations of the feature related to 'The tag being `O` when the previous tag is `O`'. There are other features activated corresponding to the `B-per` and `I-per` tags, their interaction with the tags `O` before and after those and their own interaction.
- id: Correspond to the *Emiss* features. In our example, we see the activation of the features corresponding to 'The tag being $y$ when the word is $x$' for each word and label of our sequence.
- final_prev_tag: Lastly, this string corresponds to the final features, and we ob-

serve that The feature related to 'The last tag being `O`' activates.

For the second model with extended features, several features can be activated at the same time. If one takes a look at 3.2, one cans see that for each word, more than one feature can activate.

```
inv_feature_dict = {word: pos for pos, word in added_feature_mapper.feature_dict.items()}
for feat,feat_ids in enumerate(added_feature_mapper.get_sequence_features(train_seq[16])):
    print(feature_type[feat])
    for id_list in feat_ids:
        print ("\t",id_list)
        for k,id_val in enumerate(id_list):
            print ("\t\t", inv_feature_dict[id_val] )
```

```
Initial features
        [0]
                init_tag:0
Transition features
        [5]
                prev_tag:0::0
        [5]
                prev_tag:0::0
        [5]
                prev_tag:0::0
        [91]
                prev_tag:0::B-per
        [94]
                prev_tag:B-per::I-per
        [96]
                prev_tag:I-per::0
        [5]
                prev_tag:0::0
        [5]
                prev_tag:0::0
        [5]
                prev_tag:0::0
        [5]
                prev_tag:0::0
        [5]
                prev_tag:0::0
Final features
        [37]
                final_prev_tag:0
Emission features
        [280, 2]
                id:Doctors::0
                capi_ini::0
        [259]
                id:say::0
        [299]
                id:they::0
        [300]
                id:expect::0
        [118, 90, 119]
                id:Mr.::B-per
                capi_ini::B-per
                inside_point::B-per
        [263, 93]
                id:Sharon::I-per
                capi_ini::I-per
        [264]
                id:will::0
        [301]
                id:make::0
        [75]
                id:a::0
        [302]
                id:full::0
        [303]
                id:recovery::0
        [35, 36]
                id:.::0
                inside_point::0
```

Figure 3.2: activated features for a given sentence with the added features

For example, we see that `capi_ini::B-per` feature, which activates when the first letter of a word is a capital letter, is activated for both words `Mr.` and `Sharon` (as well as in the first word) at the same time that their *Emission* features. The word `Mr.` also activates the feature `inside_point::B-per`, which corresponds to the word having the character `"."`. These extra features can help this model being more discriminative in order to distinguish better the words.

### 3.1.1 Non observed words

In case there appear new words that were not observed in the training set, we expect the model to be able to predict labels using other activated features, such as if the word is capitalized. Let us take the following sentence as an example.

`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'.`

The word `Alice` is not observed in the training set. That only means that the feature `id:Alice::O` can't be activated and won't sum to the score, but that is all, if we initialize the sequence (as in the example) the algorithm works perfectly. Since this will be the same for every label, it can still classify right. In fact, firstly Alice is wrongly classified with tag `O`, but after adding the new features it correctly assigns the label `B-per`

## 3.2 Added features

As said before, in order to really understand the behavior of the described perceptron we decided to build two different ones: one with the default, given features and another one with personalized features made by us.

When building this personalized features, we took into account the `TINY_TEST` dataset. `TINY_TEST` is nothing more than a document containing 13 different sentences made specially for testing interesting decisions made by our models. For example, it is interesting to look at the behavior when predicting a tag for `Bill Gates` and `Bill gates` or `Parris` versus `Paris`.

To start with, we tested this set with the default features model. We examined various miss-classified interesting words such as `U.S.A./O` or `Apple/O` and built, accordingly, personalized features that would (in theory) help to identify correctly this tricky words. All these features are manually introduced in the `ExtendedFeatures` class inside the file `extended_features.py`.

Below there is a detailed implementation of how we solved these issues:

1. **Solving 'U.S.A'**: we built a feature that is triggered when the character "." is found inside a word:

Listing 1: added feature that detects a point in a word

```
if str.find(word, ".") != -1:
    # Generate feature name.
    feat_name = "inside_point::%s" % y_name
    # Get feature ID from name.
    feat_id = self.add_feature(feat_name)
    # Append feature.
    if feat_id != -1:
        features.append(feat_id)
```

2. **Solving geographical names**: names can be easily tagged building a feature that checks if the first word is capitalized. Nevertheless, after its addition, some names are still miss-classified. In order to solve this issue, we created more helper features, like one that is triggered when the word is "from" (solving "from <name>"), or another one that is triggered when the word is "to" (solving "to <name>"):

Listing 2: added feature that detects the word "the"

```
if word == 'the':
    # Generate feature name.
    feat_name = "article_the::%s" % y_name
    # Get feature ID from name.
    feat_id = self.add_feature(feat_name)
    # Append feature.
    if feat_id != -1:
        features.append(feat_id)
```

Apart, we built features that help the overall accuracy of train and test sets, like detecting a hyphen in a word ("-") or detecting explicit suffixes like "-ly", "-ing" or "-ed". The emission features defined in the Python script specified above are the following ones:

| **Condition** | **Name** |
|---|---|
| $x_0 = w_j,\ w_j[0]$ is capitalized | Capitalized Initial feature |
| $x_i = w_j,$ contains capitalized letter | Capitalized Any feature |
| $x_i = w_j,$ is digit | Digit feature |
| $x_i = w_j,$ contains digit character | In Digit feature |
| $x_i = w_j,$ contains dot character | Inside point feature |
| $x_i = w_j,$ ends with -ing | -ing ending feature |
| $x_i = w_j,$ ends with -ed | -ed ending feature |
| $x_i = w_j,$ ends with -ness | -ness ending feature |
| $x_i = w_j,$ ends with -ship | -ship ending feature |
| $x_i = w_j,$ ends with -ity | -ity ending feature |
| $x_i = w_j,$ ends with -ty | -ty ending feature |
| $x_i = w_j,$ ends with -ly | -ly ending feature |
| $x_i = w_j,$ contains "-" | Hyphen feature |
| $x_i = w_j,$ is 'of' | Preposition-of feature |
| $x_i = w_j,$ is 'from' | Preposition-from feature |
| $x_i = w_j,$ is 'the' | Article-the feature |

# 4 Results

In this section we are going to comment the overall performance of each model we created. We are asked to look at the accuracy, F1 Score and confusion matrix, and we

decided to also count the overall number of sentences that are fully right predicted, i.e., every predicted tag is the same as the ground truth of every dataset.

- **Accuracy**: we are demanded to not take in consideration tags that are 'O'. Accuracy, by definition, is the number of correctly predicted tags divided by the total number of tags. Therefore, we decided to compute each of this sums following the below scheme:

| **Ground Truth** | **Predicted** | **Counts as correct prediction?** | **Tag counts for total number of tags?** |
|---|---|---|---|
| O | O | No | No |
| O | <not O> | No | Yes |
| <not O> | O | No | Yes |
| <same tag, not O> | <same tag, not O> | Yes | Yes |

- **F1 Score**: we are interested in looking at the weighted version of the F1 Score, meaning that we set a weight of each F1 Score of each class by looking the number of samples from that class.
- **Confusion matrix**: what we have to do here is to find TP, TN, FP and FN for each type of tag. Instead of just returning the NumPy matrix, we plot the matrix using the Seaborn heatmap plot.
- **Sentences fully right predicted**: we consider that a sentence is fully right predicted when every predicted tag is the same as the ground truth of every dataset, including tags that are 'O'.

#### 4.0.1 Default features

Within the `ExtendedFeature` class we are given only the addition of one transition feature: *Suffixes*. That piece of code is able to create new features from each word taking from it the last N characters. By default, N is set to 3, which means the following:

| **Word** | **Suffix 1** | **Suffix 2** | **Suffix 3** |
|---|---|---|---|
| Making | g | ng | ing |
| Make | e | ke | ake |
| Mac | c | ac | - |
| Ma | a | - | - |

In other words, when building our perceptron with default features, we are at least creating 1 new feature per word, if and only if this word has more than one character. With this philosophy, these are our results from test, train and `TINY_TEST` sets:

| **Dataset** | **Correct Sentences (%)** | **Accuracy (%)** | **Weighted F1-Score** |
|---|---|---|---|
| Test | 25,78 | 25,05 | 85,79 |
| Train | 63,92 | 80,13 | 96,82 |
| TINY_TEST | 23,08 | 55,88 | 89,34 |

So, overall, a 25% of the tags that are not "O" in our test set are predicted correctly. Taking into account that no features were added, this results is totally reasonable. What is also reasonable is the fact that when predicting with our train set, accuracy and number of correct sentences increases.

Let's look at figure 4.1. We can see that the 99% of tags "O" that were actually "O" were identified correctly. No other tag has a similar recall, "I-per" is the next higher one with 50%.

When looking at "O" precision we see that we correctly predicted 90% of "O" tags out of all predicted "O" tags. Interestingly, the precision for "I-gpe" is 1 (100%) and for "B-gpe" we have 95%, whereas for tags like "I-per" we only have 52% of precision.

| | precision | recall | f1-score |
|---|---|---|---|
| **O** | 0.907607 | 0.993025 | 0.948397 |
| **B-geo** | 0.812602 | 0.197848 | 0.318218 |
| **B-gpe** | 0.955585 | 0.224524 | 0.363613 |
| **B-tim** | 0.845753 | 0.221623 | 0.351213 |
| **B-org** | 0.585074 | 0.204570 | 0.303145 |
| **I-geo** | 0.619536 | 0.193364 | 0.294737 |
| **B-per** | 0.527701 | 0.377766 | 0.440320 |
| **I-per** | 0.520450 | 0.508821 | 0.514570 |
| **I-org** | 0.620304 | 0.187712 | 0.288208 |
| **B-art** | 0.011843 | 0.078864 | 0.020593 |
| **I-art** | 0.008386 | 0.276018 | 0.016278 |
| **I-tim** | 0.676495 | 0.311175 | 0.426272 |
| **I-gpe** | 1.000000 | 0.099338 | 0.180723 |
| **B-nat** | 0.294118 | 0.114286 | 0.164609 |
| **I-nat** | 0.500000 | 0.095238 | 0.160000 |
| **B-eve** | 0.422857 | 0.304527 | 0.354067 |
| **I-eve** | 0.267857 | 0.152284 | 0.194175 |
| **accuracy** | 0.880788 | 0.880788 | 0.880788 |
| **macro avg** | 0.563304 | 0.267117 | 0.314067 |
| **weighted avg** | 0.875903 | 0.880788 | 0.857919 |

Figure 4.1: Classification report of the test evaluation by tag

When looking at our `TINY_TEST` [2], we can see the following issues:

- **Misspelled names are predicted as "O"**: This means that `Parris` is not identified as "B-geo" whilst `Paris` is. Our approach is the following: even though a name or word is misspelled, it should be predicted as its tag when its well-written. In the case of `Parris`, then, it should be tagged as "B-geo".
- **Composed geographical names are predicted correctly**: `Saudi Arabia`, `New York` but not `United States of America`, which is tagged as "org".
- **Other**: surprisingly, `Bill Gates` is tagged correctly whilst `Robin` or `Alice`, who also are at the beginning of the sentence but are not compose, are not tagged correctly. `U.S.A` is tagged as "O", and `Jack London` is tagged partially good: `Jack/B-per London/B-geo`. It should be `Jack/B-per London/I-per`.

### 4.0.2 Added features

As said in section 3.2, we trained another perceptron with our own addition of features. When doing so, we focused on the problems seen in our `TINY_TEST` . Remark that we removed the given feature "Suffixes" used in the default perceptron. Here is the recap of metrics for this new model:

| **Dataset** | **Correct Sentences (%)** | **Accuracy (%)** | **Weighted F1-Score** |
|---|---|---|---|
| Test | 25,95 | 35,13 | 89,328 |
| Train | 57,68 | 77,42 | 96,36 |
| TINY_TEST | 30,77 | 58,82 | 89,14 |

In our training sets, each metric is improved. In the test set, we get 35.13% accuracy from 26%. In our `TINY_TEST` , we got 31% of the sentences fully well predicted, while in the previous model we got 23%.

Weirdly, we did not improve our train set metrics. In figure 4.2 we can see the confusion matrix plot for each tag. Of course, the tag that appears the most and is more accurate is "O". We got solid results with tags "B-tim", "B-org" and "B-geo". Overall, though, we got a good Weighted Average F1-Score: 96%.

[2]You can see the full result for each sentence in the `reproduce_results` notebook

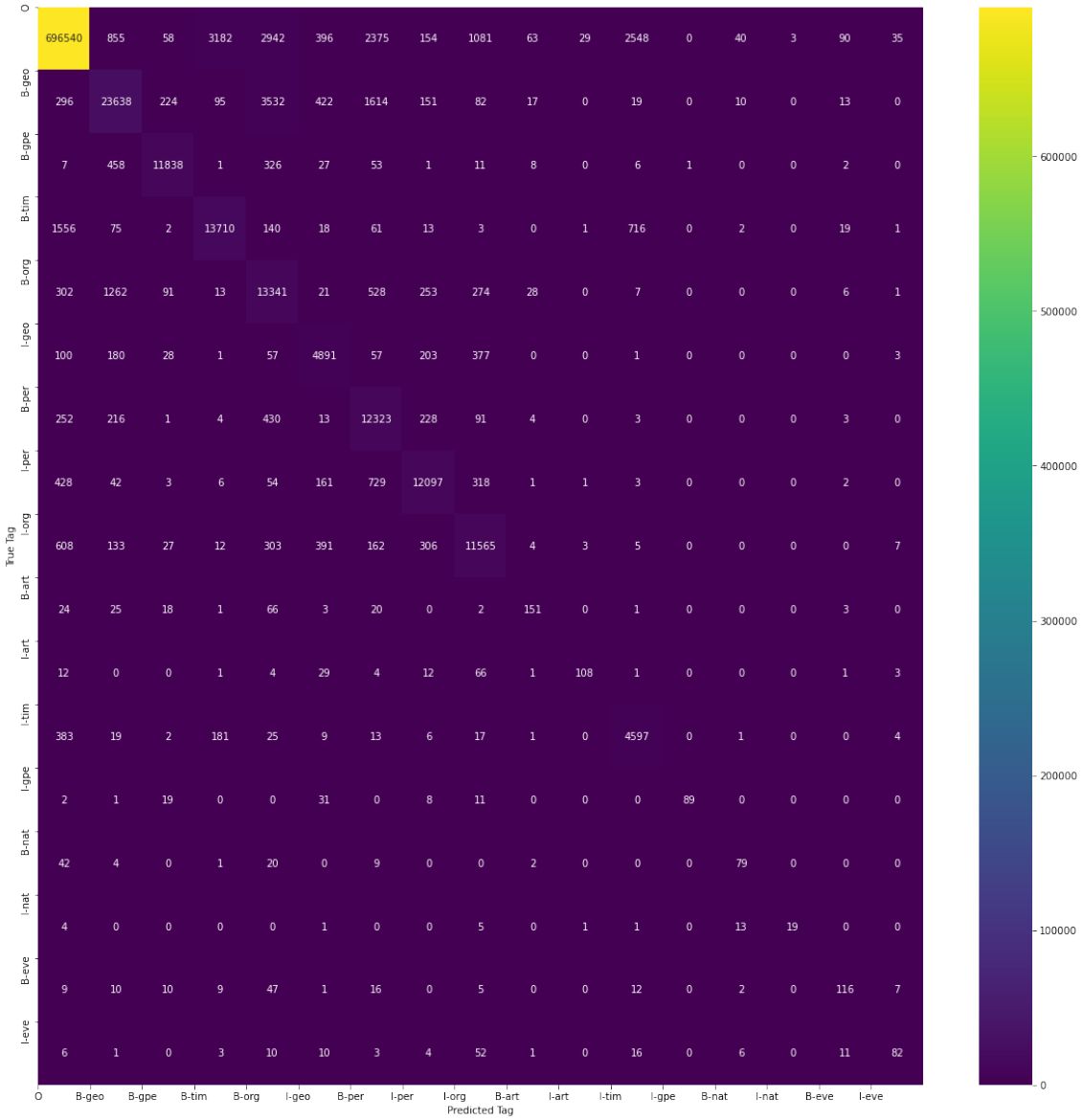


Figure 4.2: Confusion matrix of our train test

The real deal here is the `TINY_TEST` comparison between the two models:

- **Misspelled names are predicted as "B-per"**: instead of "O", misspelled names like `Parris` are now "B-per". A possible explanation is the added feature that is triggered when we detect the word "the", "from" or "to". In our train set, most of the words that come after this trigger words are "B-per", so probabilities don't help us in this `TINY_TEST` .
- **Composed geographical names**: `New York` is still correctly identified. Curiously, `Saudi Arabia` is now tagged as "per" instead of "geo" but we fixed `United States of America`.

- **Other**: `Alice`, `Apple`, `Robin` are now correctly identified. We could not fix `U.S.A` and the `London` part of `Jack London`. The word `apples` is still correctly identified as "O".

Overall, we upgraded the accuracy on this tricky dataset, but more improvement is needed, specially when distinguishing between persons and geographical names.

# 5 Conclusions

To conclude, we would like to mention a few remarks focusing on how good the predictions can be using this approach, and also which improvements it provides compared with the default HMM.

On one hand, we must take into account that this model is not state of the art. The performance is limited and can be trickily related to our training data together with the defined features. Actually, we observed that, although we added specific features based on the initial results on the tiny test, the overall accuracy values may decrease instead. Having to tag a word that has not been seen in the training set makes it harder to make a good prediction, since some of the main features are missing.

However, this project allowed us to understand the improvements that the Structured Perceptron provides versus the initial HMM; we can add customized features, adding several of them in one single timestep. Overall, this model gives us more flexibility to adapt the model to our needs while taking advantadge of the strengths of the HMM.

# 6 Extra exercise

## 6.1 Where it spends more time

Definitely, where the code spends more time is on training. Specifically, the `fit_epoch` on average lasts 5 minutes per epoch. The `perceptron_update` function is the principal reason of this 5 minutes, since we are updating all the features sentence by sentence.

The inference or evaluation part is also very heavy. Take in consideration that our evaluation function is predicting sentence by sentence, and, for every predicted sentence, is checking tag by tag with the ground truth. Remember the dimensions of our data: 38366 sentences with 839149 words in both datasets.

In minutes, here it is a recap of all the timings for the default model:

| **Dataset** | **Training** | **Evaluation** |
|---|---|---|
| Train | 79 | 6 |
| Test | - | 6,5 |

And for the added features model (also in minutes):

| Dataset | Training | Evaluation |
|---|---|---|
| Train | 83 | 5,9 |
| Test | - | 6 |

Remark, also, that the sequence list creation also takes its time. In this case, the `add_sequence` is the heaviest function, since we are appending sequence by sequence. In average, this sequence list creation lasts 10 minutes.

## 6.2 Improve the code

Due to time constraints we did not apply any kind of improvement to the original given code. Nevertheless, we tried to speed up the `add_sequence` function of the file `sequence_list`. It is not the worse function in terms of computing time (5 minutes for dataset) but we though it was easy to optimize using Cython:

Listing 3: improved add_seq function

```
def add_seq_cython(self, x, y, x_dict, y_dict):
    num_seqs = len(self.seq_list)
    dict_size = len(x_dict)
    cdef np.array[double, dim=1] x_ids = np.zeros((dict_size,))
    cdef np.array[double, dim=1] y_ids = np.zeros((dict_size,))

    for name in x:
        x_ids[i] = x_dict.get_label_id(name)
        y_ids[i] = x_dict.get_label_id(name)

    self.seq_list.append(seq.Sequence(x_ids, y_ids))
```

This improvement is possible due that we know the length of both `x_ids` and `y_ids` arrays. As a consequence, instead of doing `dict_size` x 2 appends we are just assigning each position of the array one by one.

The same logic can be applied to the global `seq_list` array, because we know its length: `num_seqs`. Nevertheless, in order to do that we would have to modify the whole class.