Named Entity Recognition

(NLP Assignment 2)

Ignacio Garcia & Gabriela Zemenčíková

May 2024

1 Introduction

Named Entity Recognition (NER) stands as a cornerstone in the realm of Natural Language Processing (NLP), tasked with the identification and classification of named entities within textual data. These entities span a diverse spectrum, encompassing individuals' names, organizational affiliations, geographical locations, geopolitical entities, temporal references, artifacts, events, and natural phenomena. Central to our endeavor is a dataset meticulously curated to adhere to the IOB (Inside, Outside, Beginning) format, where each token within a sentence bears a tag signifying its placement within an entity, initiation of an entity, or exclusion from any entity.

Illustratively, consider the sentence "John Smith is an engineer at Microsoft", annotated as "B-PER I-PER O O O O B-ORG", elucidating the commencement of a person's name, continuation thereof, and delineation of an organizational entity.

Our dataset constitutes three discernible subsets: a comprehensive training set comprising 38,366 sentences, a rigorous test set comprising 38,367 sentences, and a diminutive test set, featuring a modest ensemble of 13 sentences.

With a firm grasp of our problem domain and dataset intricacies, we now embark on an expedition into the inner workings of the structured perceptron. Subsequently, we traverse the landscape of implemented methodologies aimed at addressing the challenge at hand. Finally, we cast our gaze upon the panorama of results obtained, culminating in a discerning synthesis of insights garnered.

2 Hidden Markov Models

Hidden Markov Models (HMMs) have been a focal point in our exploration of Sequential Tagging Problems. These models operate under the assumption of two random processes, one for input sequences and one for output sequences, aiming to maximize a probability dependent on initial and final states. In essence, an HMM serves as a generative sequence model, defining the probability distribution over input-output pairs.

The probability distribution is composed of several components:

- $P_{init}(c_j|start)$: Initial state probabilities, requiring $|V|$ floats, where V is the set of words.
- $P_{trans}(c_j|\hat{c})$: Transition probabilities between states, requiring $|V| \times |V|$ floats.
- $P_{final}(stop|c_j)$: Final state probabilities, also needing $|V|$ floats.
- $P_{emiss}(w_i|c_j)$: Emission probabilities, requiring $|V| \times |L|$ floats, where L is the set of labels.

These models abide by fundamental assumptions:

- Independence of previous states: The probability of a state depends solely on the preceding state.
- Homogeneous transition: Transition probabilities remain constant regardless of sequence position.
- Observation independence: Observations are fully determined by the state at each position.

HMMs can be viewed as linear classifiers, where probabilities are transformed into weights for a classifier. The logarithms of probabilities can be used as weights, enabling the training of non-normalized scores and facilitating the creation of discriminative models.

2.1 Decoding in Hidden Markov Models

Two significant decoding problems arise in the context of Hidden Markov Models (HMMs):

- **Posterior Decoding**:

$$y_i = \arg\max_{y_i \in \mathcal{Y}} P(Y_i = y_i | X_1 = x_1, \ldots, X_N = x_N)$$

- **Viterbi Decoding**:

$$y_i = \arg\max_{y \in \mathcal{Y}^N} P(Y_1 = y_1, \ldots, Y_N = y_N | X_1 = x_1, \ldots, X_N = x_N)$$

We focus solely on Viterbi decoding as it aligns with our Structured Perceptron algorithm's requirements.

To compute $P(Y_1 = y_1, \ldots, Y_N = y_N | X_1 = x_1, \ldots, X_N = x_N)$ for all $y \in \mathcal{Y}^N$, we utilize the Viterbi algorithm. This involves computing the maximum value and determining the correct sequence of labels as follows: For the first position and a given state $\mathrm{y}_1 \in \mathcal{Y}, define$:
$Viterbi(1, x, y_1) = P_{init}(y_1|start) \times P_{emiss}(x_1|y_1)$
$For positions i \in [2, N]$:
$Viterbi(i, x, y_i) = P_{emiss}(x_i|y_i) \times \max_{y_k \in \mathcal{Y}} (P_{trans}(y_i|y_k) \times Viterbi(i-1, x, y_k))$
$Define the Viterbi quantity at the stop position as$:
$Viterbi(N+1, x, stop) = \max_{y \in \mathcal{Y}} (P_{final}(stop|y) \times Viterbi(N, x, y))$ Using the recurrence rule, the Viterbi algorithm computes $y^* = Viterbi(N+1, x, stop)$, yielding the sequence of labels that maximizes the joint probability.

2.2 Viterbi Algorithm

The Viterbi algorithm, a dynamic programming approach, is instrumental in determining the most likely sequence of hidden states in an HMM given a sequence of observations:

$$y^* = \arg\max_{y \in \mathcal{Y}^N} P(Y_1 = y_1, \ldots, Y_N = y_N | X_1 = x_1, \ldots, X_N = x_N)$$

The algorithm unfolds as follows:

- **Initialization**: Compute $viterbi(1, x, c) = P_{init}(c|start) \times P_{emiss}(x_1|c)$ for every state $c \in \mathcal{Y}$.
- **Recursion**: At each subsequent time step $i \in [2, N]$, compute $viterbi(i, x, c)$ recursively using the formula provided.
- **Termination**: Compute $viterbi(N+1, x, stop)$ using the termination formula.
- **Backtracking**: Use the backtrack recurrence to determine the most likely sequence of hidden states.

The Viterbi algorithm, with its ability to efficiently find the optimal hidden state sequence, finds applications in diverse fields such as speech recognition, part-of-speech tagging, and bioinformatics.

2.3 Structured Perceptron

Discriminative sequence models aim to solve the following optimization problem:

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

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

In this scenario, sequence scoring is computed as the product of the weights with the feature vector. As discussed in Section 2.1, the feature vector can be separated into four parts of the sequence (init, trans, final, emiss). The feature vector depends on the input and:

- The first output label for $finit$ denoted as $finit(x, y_1)$
- A pair of output labels for $ftrans$ denoted as $ftrans(i, x, y_i, y_{i+1})$
- The last output label for $ffinal$ denoted as $ffinal(x, y_N)$
- The corresponding output label for $femiss$ denoted as $femiss(i, x, y_i)$

Thus, our linear classifier is defined as follows:

$$\arg\max_{y \in \mathcal{Y}^N} \sum_{i=1}^{N} scoreemiss(i, x, y_i) + scoreinit(x, y_1) + \sum_{i=1}^{N-1} scoretrans(i, x, y_i, y_{i+1}) + scorefinal(x, y_N)$$

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 extends the perceptron algorithm to handle structured outputs instead of just binary or scalar outputs. The structured perceptron algorithm can be viewed 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 learns a weight vector for linear classification by iteratively updating it based on feature vectors and observed outputs. Similarly, the structured perceptron algorithm maintains a weight vector for each possible output structure. During training, it compares predicted and true output structures, updating the weights to favor correct structures and penalize incorrect ones.

Additional features beyond those outlined in Table 1 can be incorporated into the model. A more comprehensive explanation of these new features and their utilization by the model will be provided in Section 3.2.2.

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 by considering the learned weights.

2.4 Handling Out-of-Vocabulary Words and New Feature Definitions

When evaluating a model, a common concern is its behavior when encountering words during testing that were not seen during training. In the context of the structured perceptron algorithm, which operates as a linear model, encountering such out-of-vocabulary (OOV) words does not prompt significant changes. These OOV words may lack corresponding weights in the weight vector, thus not directly contributing to the overall score. However, to address cases where basic features are inadequate for accurate classification, additional features can assist the model in achieving satisfactory classification for OOV words.

Supplementary features provide extra information that aids in accurately classifying unobserved words or handling specific cases where basic features fail to capture necessary patterns. For instance, encountering a proper noun in the test set not present in the training set might prompt the inclusion of a supplementary feature identifying capital letters (often indicative of proper nouns), facilitating accurate word classification.

The structured perceptron leverages various features, including initial, transition, final, and basic emission features, to model the relationship between input and output sequences. These features, derived from the Hidden Markov Model (HMM) framework, define conditions based on specific positions and output states. For instance, an initial feature corresponds to the state of the first output label, while a transition feature captures the transition between consecutive output states. Additionally, basic emission features consider the emission of specific words associated with output states.

In the context of structured prediction tasks, such as sequence labeling, it's essential to define new features that enhance the model's capabilities. New features can be introduced to address specific challenges or capture additional patterns in the data. These features are typically structured based on the observed data and the underlying task requirements.

By incorporating a combination of basic and supplementary features, the model can effectively handle OOV words and adapt to diverse patterns in the data. Furthermore, a detailed understanding of feature definitions and their relevance to the task is crucial for optimizing model performance.

3 Implementation

The primary objective of this project is to gain a comprehensive understanding of how the Structured Perceptron functions. To achieve this, we compare the performance of the model using the default features (identical to those used in HMM) with the performance using newly added features.

We trained two variations of the Structured Perceptron, each with different input features. The results of these models are compared using several metrics, which are detailed in Section 4. These comparisons help determine if the additional features improve the model's predictive capability.

The implementation involves several key steps: data transformation, corpus creation, sequence list creation, feature mapping, model training, and evaluation. Please note that we used functions from lecture notes provided in the class and followed the logic for our code implementation.

3.1 Data Transformation

The data transformation process involves converting the raw data into a format suitable for training the Structured Perceptron model. Note that all these defined functions are in the `utils.py` file. The code below demonstrates how sentences and their corresponding tags are extracted from the data:

```
def transform_data_sentence_tag(df):
    """
    Extracts sentences and tags from the provided df.

    Parameters:
        df: A df object containing sentence_id, words, tags columns
            .

    Returns:
        A tuple (X, y) containing sentences and corresponding tags.
    """
    X, y = [], []
    id_s = df.sentence_id.unique()
    progress = tqdm(id_s, desc="Creating", unit= "sentence")
    for sentence in progress:
        X.append(list(df[df["sentence_id"] == sentence]["words"].
            values))
        y.append(list(df[df["sentence_id"] == sentence]["tags"].
            values))
    return X, y
```

3.2 Corpus Creation

Next, we create a corpus from the list of sentences and tags, mapping each word and tag to a unique integer index. As a result we produced two outputs corresponding to word and tag dictionaries respectively.

```python
def create_corpus(sentences, tags):
    """
    Create a corpus from a list of sentences and corresponding tags.

    Parameters:
        sentences (list): A list of sentences.
        tags (list): A list of corresponding tags for each sentence.

    Returns:
        word_dict (dict): A dictionary mapping words to integer indices.
        tag_dict (dict): A dictionary mapping tags to integer indices.
    """
    word_dict = {}
    tag_dict = {}
    for sentence, tag_sequence in zip(sentences, tags):
        word_indices = []
        tag_indices = []
        for word, tag in zip(sentence, tag_sequence):
            if word not in word_dict:
                word_dict[word] = len(word_dict)
            if tag not in tag_dict:
                tag_dict[tag] = len(tag_dict)
            word_indices.append(word_dict[word])
            tag_indices.append(tag_dict[tag])
    return word_dict, tag_dict
```

3.3 Sequence List Creation

We then create a sequence list object from the training data, which will be used to train the model:

```python
def create_sequence_list(X, y, word_dict, tag_dict):
    """
    Create a SequenceList object from training data.

    Parameters:
        X (list): A list of sentences.
        y (list): A list of corresponding tags for each sentence.
        word_dict (dict): A dictionary mapping words to integer indices.
        tag_dict (dict): A dictionary mapping tags to integer indices.

    Returns:
        SequenceList: A SequenceList object containing the sequences.
    """
    sequence_list = SequenceList(LabelDictionary(word_dict), LabelDictionary(tag_dict))
    for sent_x, sent_y in tqdm(zip(X, y), desc="Creating Sequence List", total=len(X)):
```

```
        sequence_list.add_sequence(sent_x, sent_y, LabelDictionary(
            word_dict), LabelDictionary(tag_dict))
    return sequence_list
```

3.4 Loading and Transforming Data

To combine all previously defined functions together and optimise the code, we load and transform the data, creating the necessary dictionaries and sequence lists for training:

```
def load_and_transform_data(file_path):
    df = pd.read_csv(file_path)
    df['words'] = df['words'].astype(str)
    X, y = transform_data_sentence_tag(df)
    return X, y

data_files = {
    "train": "data/train_data_ner.csv",
    "test": "data/test_data_ner.csv",
    "tiny": "data/tiny_test.csv"
}

save_models=True
if save_models:
    data_dict = {}
    for key, file_path in data_files.items():
        X, y = load_and_transform_data(file_path)
        data_dict[f"X_{key}.pkl"] = X
        data_dict[f"y_{key}.pkl"] = y
    for filename, data in data_dict.items():
        save_p(filename, data)

    word_dict, tag_dict = create_corpus(data_dict["X_train.pkl"],
        data_dict["y_train.pkl"])
    sequence_list = create_sequence_list(data_dict["X_train.pkl"],
        data_dict["y_train.pkl"], word_dict, tag_dict)
    save_p("sequence_list.pkl", sequence_list)
else:
    X_train = load_p("X_train.pkl")
    y_train = load_p("y_train.pkl")
    word_dict, tag_dict = create_corpus(X_train, y_train)
    sequence_list = create_sequence_list(X_train, y_train,
        word_dict, tag_dict)
    save_p("sequence_list.pkl", sequence_list)
    sequence_list = load_p("sequence_list.pkl")
```

3.5 Model Training with Default Features

The training of the Structured Perceptron model in this project is based on a set of default features that can be compared to those used in HMMs. They are crucial for capturing the relationships between words and tags, as well as the transitions between tags. As a result the model can effectively recognize named entities. In our folder `skseq/sequences` we have a file `id_feature.py` where

the default builder is defined. It is a generic function to build features for a given dataset. It iterates through all sentences in the dataset and extracts its features, saving the node/edge features in feature list. Let's take a closer look at its key features. The 4 different key features used in the model are:

- **Word-Tag Pair Features:** These features capture the direct link between individual words and their corresponding tags. They are also called emission features. By mapping words to their potential tags, the model learns which tags are likely to be associated with specific words based on the training data.
- **Tag-Tag Transition Features:** These features capture the probability of transitioning from one tag to another. They are crucial for maintaining the logical flow of tags in a sentence. These transition probabilities help the model predict sequences of tags in the context.
- **Initial Tag Features:** These features indicate the likelihood of a tag being the first tag in a sequence. They help the model understand which tags are commonly found at the beginning of sentences.
- **Final Tag Features:** Opposite to initial tags, these features represent the probability of a tag being the last one in a sequence. This helps the model learn typical ending tags for sentences.

These features are used in the Default settings, now let's define model with several additional features and compare the results to see if we can obtain better results.

3.6 Model Training with Additional Features

3.7 3.2 Added Features

To gain a deeper understanding of the perceptron's behavior, we created two different models: one with the default features and another with custom features designed by us. We used the TINY_TEST dataset, which consists of 13 sentences, specifically created to test various decision-making scenarios in our models. This dataset helps us examine behaviors like how the model tags "Bill Gates" versus "Bill gates" or distinguishes between "Parris" and "Paris."

First, we evaluated this dataset with the default features model. We identified several misclassified words, such as "Apple/O" and "Microsoft/O," and developed custom features to improve the model's accuracy in these tricky cases. These custom features are manually added to the `ExtendedFeatures` class in the `extended_features.py` file. Below is an explanation of how we addressed these issues:

1. **Handling Geographical Names:** While capitalized initials usually help in tagging names correctly, some names were still misclassified. To address this, we created additional features that activate for words like "from" (e.g., "from ¡name¿") and "to" (e.g., "to ¡name¿"):

```
if word == 'the':
    feat_name = f"article_the::{y_name}"
    feat_id = self.add_feature(feat_name)
    if feat_id != -1:
        features.append(feat_id)
```

2. **Overall Accuracy Improvement:** We introduced features that detect hyphens in words ("-") or specific suffixes like "-ly," "-ing," or "-ed" to enhance the overall accuracy of the model:

```
suffixes = ['ing', 'ed', 'ness', 'ship', 'ity', 'ty', 'ly']
for suffix in suffixes:
    if word.endswith(suffix):
        feat_name = f"ending_{suffix}::{y_name}"
        feat_id = self.add_feature(feat_name)
        if feat_id != -1:
            features.append(feat_id)
```

The emission features defined in the Python script are as follows:

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

3.8 Results Analysis

3.8.1 Model without New Features

3.9 3.2 Added Features

To gain a deeper understanding of the perceptron's behavior, we created two different models: one with the default features and another with custom fea-

tures designed by us. We used the TINY_TEST dataset, which consists of 13 sentences, specifically created to test various decision-making scenarios in our models. This dataset helps us examine behaviors like how the model tags "Bill Gates" versus "Bill gates" or distinguishes between "Parris" and "Paris."

First, we evaluated this dataset with the default features model. We identified several misclassified words, such as "Apple/O", but Microsoft was correctly and '/B-org', for these mistakes we developed custom features to improve the model's accuracy in these tricky cases. These custom features are manually added to the `ExtendedFeatures` class in the `extended_features.py` file. Below is an explanation of how we addressed these issues:

1. **Handling Geographical Names:** While capitalized initials usually help in tagging names correctly, some names were still misclassified. To address this, we created additional features that activate for words like "from" (e.g., "from ¡name¿") and "to" (e.g., "to ¡name¿"):

```
if word == 'the':
    feat_name = f"article_the::{y_name}"
    feat_id = self.add_feature(feat_name)
    if feat_id != -1:
        features.append(feat_id)
```

2. **Handling Organizational Names:** To help the model recognize organization names like "Apple" correctly, we added features that detect certain suffixes and patterns within the word:

```
if word[0].isupper():
    # Generate feature name.
    feat_name = "capi_ini::%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)
```

3. **Additional Specific Features:** We introduced features to detect the word "the," which often indicates the presence of an article:

```
if word == 'the':
    feat_name = f"article_the::{y_name}"
    feat_id = self.add_feature(feat_name)
    if feat_id != -1:
        features.append(feat_id)
```

4. **Suffix and Prefix Features:** Features were added to detect hyphens within words and specific suffixes such as "-ly," "-ing," or "-ed" to enhance the model's ability to accurately classify different types of entities:

```
suffixes = ['ing', 'ed', 'ness', 'ship', 'ity', 'ty', 'ly']
for suffix in suffixes:
    if word.endswith(suffix):
        feat_name = f"ending_{suffix}::{y_name}"
        feat_id = self.add_feature(feat_name)
        if feat_id != -1:
            features.append(feat_id)
```

The emission features defined in the Python script are as follows:

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

3.10 Evaluation

Finally, we evaluate the model's performance using various metrics. However, due to the length of the function, we provide only the definition of the listings for more details please see the `utils.py` file. We defined a function that calculates the % of correct sentences, % of tags correct, F1 Score and accuracy.

```
1 def evaluate(data_, model, data_tag_pos, corpus_tag_dict, y_true,
      name, load=False):
2     """
3     Evaluate the model's performance on the given dataset and
          calculate various metrics.
4
5     Parameters:
```

```
        data_ (list): A list of sentences.
        model (object): The trained model object.
        data_tag_pos (dict): A dictionary containing the true tags
            for each sentence.
        corpus_tag_dict (dict): A dictionary mapping tags to
            indices.
        y_true (list): A list of true tags.
        name (str): The name to use for saving results.
        load (bool): Whether to load previously saved predictions
            and metrics.

    Returns:
        metrics = pd.DataFrame({
            "% of correct sentences",
            "% of tags correct",
            "F1 Score",
            "Accuracy"
        })
    """
```

4 Results

Here, we delve into the holistic evaluation of each model we created. Our focus encompasses accuracy, F1 Score, and the confusion matrix. Additionally, we assess the total number of sentences where every predicted tag precisely matches the ground truth across all datasets

- **Accuracy**: We exclude 'O' tags from consideration. Accuracy is defined as the number of correctly predicted tags divided by the total number of tags.
  - **Ground Truth: "O" / Predicted: "O"**
    - **Counts as correct prediction?:** No
    - **Tag counts for total number of tags?:** No
  - **Ground Truth: "O" / Predicted: not "O"**
    - **Counts as correct prediction?:** No
    - **Tag counts for total number of tags?:** Yes
  - **Ground Truth: not "O" / Predicted: "O"**
    - **Counts as correct prediction?:** No
    - **Tag counts for total number of tags?:** Yes
  - **Ground Truth: not "O" / Predicted: not "O"**
    - **Counts as correct prediction?:** Yes
    - **Tag counts for total number of tags?:** Yes

- **F1 Score**: We focus on the weighted version of the F1 Score, where the score for each class is weighted by the number of samples from that class.
- **Confusion Matrix**: For each type of tag, we determine true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Instead of returning a NumPy matrix, we visualize the matrix using a Seaborn heatmap.
- **Fully Correctly Predicted Sentences**: A sentence is considered fully correctly predicted when every predicted tag matches the ground truth, including 'O' tags.

4.1 Default Features

Thus, when building our perceptron with default features, we generate at least one new feature per word.The results are as follows:

Model	% of correct sentences	% of tags correct	F1 Score	Accuracy
train-default	0.41	0.56	0.92	0.93
test-default	0.21	0.14	0.82	0.87
tiny-default	0.08	0.26	0.79	0.83

Table 1: Performance metrics for different models

In general, the model achieved high accuracy, with 87% on the standard test set and 83% on the smaller test set (tiny test). This performance is primarily driven by the tag "O", which achieved an accuracy of 0.869, compared to an average of 0.691 for other tags. The tag with the highest accuracy was B-gpe, achieving 0.969.

Label	Precision	Recall	F1-Score
O	0.868	0.999	0.929
B-geo	0.829	0.157	0.264
B-gpe	0.969	0.150	0.260
B-tim	0.923	0.163	0.277
B-org	0.762	0.137	0.232
I-geo	0.877	0.054	0.102
B-per	0.851	0.133	0.230
I-per	0.855	0.155	0.262
I-org	0.751	0.121	0.209
B-art	0.000	0.000	0.000
I-art	0.000	0.000	0.000
I-tim	0.912	0.070	0.129
I-gpe	0.786	0.073	0.133
B-nat	1.000	0.046	0.087
I-nat	1.000	0.048	0.091
B-eve	0.461	0.243	0.318
I-eve	0.222	0.041	0.069
accuracy	0.868	0.868	0.868
macro avg	0.710	0.152	0.211
weighted avg	0.865	0.868	0.823

Table 2: Classification report

The main difference lies in recall: the tag "O" achieved a recall of 0.999, as shown in the following table, while other tags averaged 0.139. The tag with the highest recall was B-eve, reaching 0.242.

These results underscore the model's strong performance in correctly identifying instances of the "O" tag, while also highlighting opportunities to improve recall for other tags to ensure comprehensive and balanced entity classification.

When examining our tiny dataset, several observations stand out. For instance, in sentences containing "Bill Gates", "Gates" is recognized as a person when it starts with a capital letter, but not when it starts with a lowercase letter.

We also notice that "Parris" is not considered a location due to misspelling, whereas "Paris" is recognized correctly. Another curious thing is that "Apple" is not shown as Organization, meanwhile Microsoft is. Finally, neither "Alice" nor "Henry" is recognized as a person but rather as "O".

4.2 Added Features

These are the results from the other perceptron using additional features. Similar metrics were employed as in the previous case:

Model	% of correct sentences	% of tags correct	F1 Score	Accuracy
train-extra	0.50	0.74	0.96	0.95
test-extra	0.23	0.35	0.89	0.89
tiny-extra	0.38	0.65	0.91	0.92

Table 3: Performance metrics for different models

Overall, all metrics improved when comparing the results of added_features with respect to default. The improvements are evident across all indicators, as shown in the following table.

Model	% of correct sentences	% of tags correct	F1 Score	Accuracy
train	+21.9%	+32.1%	+4.4%	+2.2%
test	+9.5%	+150%	+8.5%	+2.3%
tiny	+375%	+150%	+15.2%	+10.8%

Table 4: Improvement in performance metrics (Extra vs Default)

We observe significant improvements across several metrics when comparing "added_features" with "default". Specifically, accuracy increased by approximately 2.2-2.3% in both train and test sets. Moreover, there was a remarkable 150% improvement in % of tags corrected in both the test and tiny sets. Additionally, noteworthy enhancements include the increase in % of correct sentences in the train set and the F1 score in the test set.

If we look at the metrics on precision, recall and F1 score of our tags we will that both precision and recall improve, the first from 0.86.5 to 0.922 and the second from 0.868 to 0.885

Label	Precision	Recall	F1-Score
O	0.989052	0.974246	0.981593
B-geo	0.765496	0.185362	0.298455
B-gpe	0.951917	0.223812	0.362414
B-tim	0.499205	0.369723	0.424817
B-org	0.469604	0.359261	0.407088
I-geo	0.488000	0.205491	0.289202
B-per	0.180908	0.784968	0.294048
I-per	0.443243	0.743962	0.555517
I-org	0.452756	0.389654	0.418842
B-art	0.086505	0.078864	0.082508
I-art	0.084881	0.144796	0.107023
I-tim	0.425605	0.544222	0.477659
I-gpe	1.000000	0.092715	0.169697
B-nat	0.296875	0.108571	0.158996
I-nat	0.500000	0.071429	0.125000
B-eve	0.291339	0.304527	0.297787
I-eve	0.218045	0.147208	0.175758
accuracy	0.885486	0.885486	0.885486
macro avg	0.479025	0.336989	0.330965
weighted avg	0.922136	0.885486	0.889370

Table 5: Classification report

Finally, upon reviewing our predictions on the tiny dataset, we observe that "Apple" is correctly classified as an organization. Similarly, both "Alice" and "Henry" are recognized as persons. However, "Parris" continues to be incorrectly classified as a person.

5 Conclusions

In evaluating the perceptron model enhanced with additional features against its default counterpart, several key observations emerge. The introduction of custom features aimed to refine prediction accuracy and enhance performance metrics across various datasets. Notably, while the initial results on the tiny test set showed promising improvements, the overall accuracy metrics were subject to variability due to the nature of unseen data in the training set.

The perceptron model, while not state-of-the-art, provided a significant improvement over the default Hidden Markov Model (HMM). By incorporating specific features tailored to linguistic nuances and contextual patterns observed in training data, the model exhibited enhanced predictive capabilities. However, challenges arose when predicting tags for previously unseen words, underscoring the importance of robust feature engineering and comprehensive training data coverage.

Ultimately, this project underscored the versatility of the Structured Perceptron model in adapting to diverse linguistic tasks, leveraging strengths from traditional HMM frameworks while accommodating customized feature sets. This approach not only offered flexibility in model adaptation but also highlighted opportunities for further refinement in handling unseen data scenarios effectively.