Contents

Entropy Optimization Paradigms: Analysis of Probabilistic Modelling via Maximum Entropy Frameworks

WONG KAI YUAN, DC026157
CHAN KA WAI, DC226165
Department of Computer and Information Science
Project Supervisor: Derek, Fai Wong

April 18, 2024

Abstract

This project embarks on the journey of constructing a Maximum Entropy Model [6] (MEM) with the preliminary objective of identifying personal names within newswire texts. The labels used for identification are divided into 'PERSON' and 'O'. The main goal of this project is to create a Named Entity Recognition [4] (NER) model that can correctly find person names in text data.

1 Introduction

1.1 Project Overview

In Natural Language Processing (NLP), Named Entity Recognition is a very important job that lets you get certain kinds of information (named entities) from unstructured text. Our model is designed to find person names by using the Maximum Entropy Model (MEM) to sort text parts into groups based on how likely they are to contain a person's name. This project came about because of the need to quickly sort through and understand large amounts of textual data, like news stories, where recognizing individual names can give you important clues about what the text is about.

1.2 Importance of NER

Named Entity Recognition is a basic technology used in many natural language processing (NLP) tasks, from simple ones like finding information, sorting material into categories, and analysing sentiment to more complicated ones like finding relationships and building knowledge graphs. NER systems make it easier to understand and analyse big datasets by finding and grouping things like person names, locations, and organizations. In particular, being able to recognize person names in texts is very helpful for jobs like summarizing news stories, checking social media to see how people feel, automating customer service, and even security for keeping an eye on important communications. Being able to correctly identify person names makes it possible for more in-depth semantic analysis, which helps systems figure out who is being talked about, how they relate to each other, and how they relate to the material.

1.3 Objectives

1. Develop a NER Model: Create and use a Maximum Entropy Model that can accurately and completely recognize and group person names in newswire text data.
2. Feature Engineering and Optimization: Look into and add a variety of linguistic and orthographic traits to help the model do a better job of telling person names apart from other text elements.
3. Evaluation and Analysis: Use standard NER metrics like precision, recall, and F1 score to assess the model's performance. Pay special attention to how well it can correctly identify person names.

4. Application and Demonstrative: Making a web-based app that lets users enter text and see the model's named object predictions will show how well the model works in the real world.

By reaching these goals, the project hopes to make a strong named entity recognition tool that can be added to larger NLP systems to help them understand and process human language better.

1.4 Terms and Formula

1.4.1 The Maximum Entropy (MaxEnt) Model

The maximum entropy model[6] is a conditional probability model $p(y|x)$ that allows us to predict class labels given a set of features for a given data point. It does the inference by taking trained weights and performs linear combinations to find the tag with the highest probability by finding the highest score for each tag.

Similar to logistic regression, the maximum entropy (MaxEnt) model is also a type of log-linear model. The MaxEnt model is more general than logistic regression. It handles multinomial distribution where logistic regression is for binary classification.

The maximum entropy principle is defined as modeling a given set of data by finding the highest entropy to satisfy the constraints of our prior knowledge. The feature function of MaxEnt model would be multi-classes. For example, given (x,y), the feature function returns 0,1, or 2

$$f(x,y) = \begin{cases} 1 & \text{if } y = M, x = \text{'e'} \\ 2 & \text{if } y = B, x = \text{'t'} \\ 0 & \text{otherwise} \end{cases}$$

To find the probability for each tag/class, MaxEnt defined as:

$$p(y|x) = \frac{1}{Z(x)} e^{\sum_{i=1}^{m} w_i f_i(x,y)}$$

We defind f_i as a feature function and w_i as the weight vector. The summation of i = 1 to m is summing of all feature functions where m is the number of unique states. The denominator Z(x) helped normalize the probability as:

$$Z(x) = \sum_{y'} e^{\sum_{i=1}^{m} w_i f_i(x,y)}$$

The MaxEnt model makes uses of the log-linear model approach with the feature function but does not take into account the sequential data.

1.4.2 Maximum Entropy Markov Model (MEMM)

From the Maximum Entropy model, we can extend into the Maximum Entropy Markov Model (MEMM). This approach allows us to use Hidden Markov Model(HMM) that takes into account the sequence of data and to combine it with the Maximum Entropy model for features and normalization. The Maximum Entropy Markov Model (MEMM) has dependencies between each state and the full observation sequence explicitly. This is more expressive than HMMs.

In the HMM model, we saw that it uses two probabilities matrices (state transition and emission probability). We need to predict a tag given an observation, but HMM predicts the probability of a tag producing a certain observation. This is due to its generative approach. Instead of the transition and observation matrices in HMM, MEMM has only one transition probability matrix. This matrix encapsulates all combinations of previous states y_{i-1} and current observation x_i pairs in the training data to the current state y_i.

Note: HMM is a **generative** model because words as modelled as observations generated from hidden states. Even the formulation of probabilities uses likelihood $P(W|T)$ and prior P(T). On the other hand, MEMM is a **discriminative** model. This is because it directly uses posterior probability

$P(T|W)$; that is, probability of a tag sequence given a word sequence. Thus, it discriminates among the possible tag sequences. Figure 1 demonstrates about the differences for Part of Speech(POS) tagging.[2]

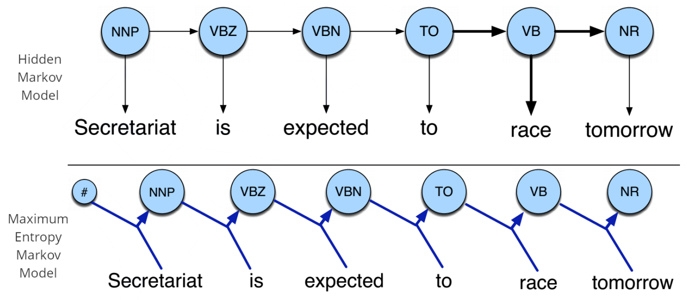


Figure 1: HMM vs. MEMM for POS tagging. Source: Adapted from Jurafsky and Martin 2009, fig. 6.20[2]

Our goal is to find the p($y1$, $y2$, …, yn — $x1$, $x2$, … xn). Since HMM only depend on the previous state, we can limit the condition of y_n given y_{n-1} This is the Markov independence assumption:

$$P(y_1 \dots y_n) = \prod_{i=1}^{n} (P(y_n | y_{i-1}, x_1 \dots x_n))$$

So the Maximum Entropy Markov Models (MEMM) defines using Log- linear likelihood model as:

$$P(y_i|y_{i-1}, x) = \frac{1}{Z(y_{i-1}, x)} e^{\sum_{i=1}^{n} w_i f_i(y_i, y_{i-1}, x)}$$

where x is a full sequence of inputs of x_1 to x_n. Let y be corresponding labels or sequence of tags (0 and 1 in our case). The variable i is the position to be tagged and n is the length of the sentence. The denominator Z($y_i - 1$, x) is the normalizer that defines as:

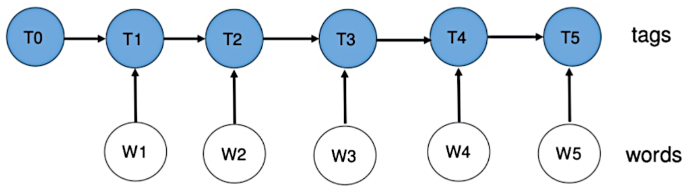
$$Z_w(y_{i-1}, x) = \sum_{y \in Y} e^{\sum_{i=1}^{n} w_i f_i(y_i, y_{i-1}, x)}$$

MEMM can incorporate more features from its feature function as input while HMM required the likelihood of each of the features to be computed since it is a likelihood-based. The feature function of MEMM also has dependencies on previous tag yi-1. As an example:

$$P(y_i, y_{i-1}, x) = \begin{cases} 1 & \text{if } y_i = M, y_{i-1} = B, x = \text{'e'} \\ 0 & \text{otherwise} \end{cases}$$

Example function for letter ‘e’ in ‘test’ where the current tag is M and the previous tag is B. The MEMM has a richer set of observation features that can describe observations in terms of many overlapping features.

Figure 2 shows a example of POS tagging with MEMM.



Discriminative model, model conditional probability Pr(T | W) directly.

$$\Pr(\boldsymbol{T}|\boldsymbol{W}) = \prod_{i=1}^{L} \Pr(t_i|t_{i-1}, w_i) = \prod_{i=1}^{L} \frac{\exp(\sum_j \beta_j f_j(t_{i-1}, w_i))}{Z(t_{i-1}, w_i)}$$

t_0 is a dummy start state.

Figure 2: Example of POS tagging with MEMM. Source: Gui et al. 2019, slide 46.[2]

2 Methods and Implementation

This is the flowchart of MME:

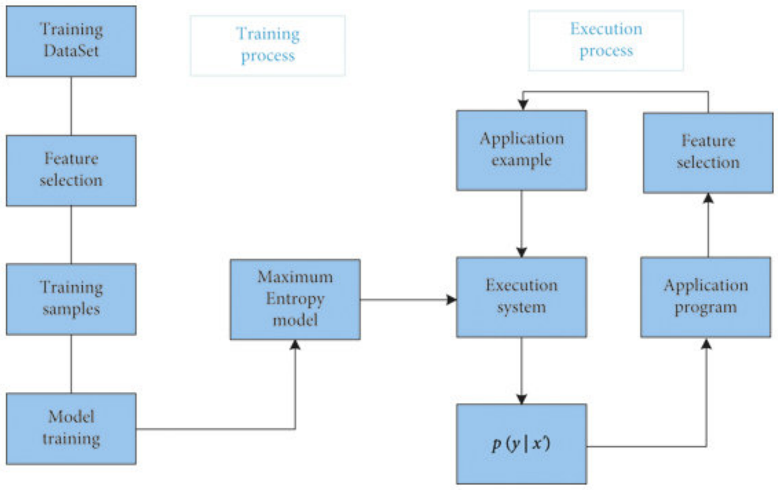


Figure 3: This is the flowchart of the Maximum Entropy Model(MEM) framework. [5]

For the features part, we add 5 new features besides the three simple starter features provided in the starter code, for example, the 'Titlecase' feature. They are 'ALLCAP', 'lowercase', 'after_symbol', 'number' and 'pretitle'.

2.1 Features Addition

2.1.1 'ALLCAP' Feature

If the current word is composed of uppercase letters, we add this feature. It is because we will not use all capital letters to represent the person names in most cases. This feature can let the model knows that there is high probability that the word does not belong to the 'PERSON' label when the word consists of all uppercase letters.

2.1.2 'lowercase' Feature

If the current word is composed of lowercase letters, we add this feature. It is because we found that most of the person names are represented by starting with a capital letter and following with some lowercase letters. This feature can let the model knows that there is high probability that the word does not belong to the 'PERSON' label when the word consists of all lowercase letters.

2.1.3 'after_symbol' Feature

If there is any symbol, such as ' . ', ' , ', ' " ' , ' ' ', ' (' , ') ', before the current word, we add this feature. It is because we found that there is high probability for the word to be a person name if it appears after such symbols when we look up the provided training set.

2.1.4 'number' Feature

If the current word consists of any numbers, we add this feature. It is because most of the person names only consist of letters. This feature can let the model knows that there is high probability that the word does not belong to the 'PERSON' label when the word consists of any numbers.

2.1.5 'pretitle' Feature

If there is any word, such as *'mr.'*, *'ms.'*, *'mrs.'*, *'mister'*, *'mistress'*, *'miss'*, *'president'*, *'minister'*, *'professor'*,*'prof.'*, *'dr.'*, *'sir'*, *'lady'*, *'lord'*, before the current word, we add this feature. It is because we found that there is high probability for the word to be a person name if it appears after such English honorifics.

Besides, we also add a predict_sentence() function in class MEMM() to output predictions and integrate with our web application which is available for the sentence input. In this function, we use word_tokenize() from NLTK to split the input sentence. Then we predict each word in the sentence by using the trained model and save the result in a list. We found that the first word of the input sentence sometimes may not be predicted as person, therefore, adding a '.' in front of the first word of the sentence is to avoid this situation.

Each detail of feature and prediction of sentences, you may refer to the First Appendix A.1

2.2 Training and Testing Classifier

2.2.1 Training Classifier

1. Load the training data from the *train_path* using the load_data method. The words are stored in a list called words, and the labels are stored in a list called labels.
2. Create a new list called *previous_labels* and add the label "O" to the beginning of the labels list so as to provide context for the feature extraction process
3. We need to transform the data into suitable format for the classifier, ensure it to learn patterns and make predictions based on the provided context.Therefore, we extract features from the combination of words, *previous_labels*, and word indices using the features method. The extracted features are stored in a list called features.
4. Zipping the features and labels lists together are also necessary, because we have to associate each feature with its corresponding label. Moreover, the classifier can learn from the training data to make accurate predictions on unseen data.
5. In the code snippet, the classifier is trained using the Maximum Entropy algorithm, which is implemented by the *'MaxentClassifier'* in NLTK. Training the classifier by calling the *'MaxentClassifier.train'* function with *train_samples* as the training data and the *max_iter* attribute to specify the maximum number of iterations.

Note: Once the Maximum Entropy model is trained, it can be used to calculate the posterior probabilities of different class labels given new input features. These posterior probabilities can then be used for making predictions or decisions. (Details has been shown in 1.4)

2.2.2 Testing Classifier

1. Load the test data from the dev_path using the load_data method. The words are stored in a list called words, and the labels are stored in a list called labels.

2. Create a new list called previous_labels and add the label ”O” to the beginning of the labels list. This is done to provide context for the feature extraction process.

3. Extract features from the combination of words, previous_labels, and word indices using the features method. The extracted features are stored in a list called features.

4. We need to organize and store the predictions made by the classifier for each individual feature. So, we classify each feature in the features list using the trained classifier. The classification results are stored in a list called results.

5. Evaluate the performance of the classifier using evaluation metrics which consist of F1 score, precision, recall, and accuracy. These metrics help assess the classifier's effectiveness in correctly classifying instances and managing false positives and false negatives, which is crucial for many real-world classification tasks.

We can summarize that the provided MEMM class implements a model for named entity recognition using a **Maximum Entropy Markov Model**. The model utilizes various features to predict named entity labels for words in a sentence. It can be trained on labeled data, tested on separate development data, and used to predict labels for new sentences. The model's performance can be evaluated using metrics such as F1 score, accuracy, recall, and precision.

2.3 Web Application

2.3.1 In 'app.py'

For the web application, we create a new python file called 'app.py' to finish it. Here we need to install the flask package and using the Flask class, which is passed the name of the module or package of the application. Once it is created it will act as a central registry for the view functions, the URL rules, template configuration and much more.

2.3.2 In 'NLP-group-project.html'

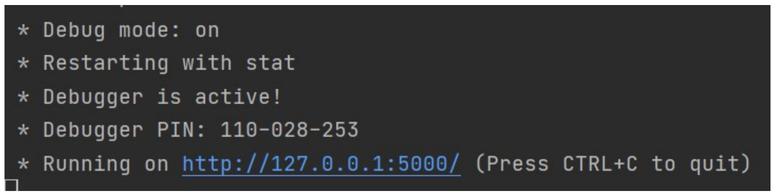
Moreover, we also use render_template() function inside the package to render a template, which named 'NLP-group-project.html', from the templates folder with the given context. The CSS code is written in the html files together as well. The CSS code has been shown in Appendix pageA.4.

3 Evaluation and Discussion

3.1 Running the program

You can use three different command flags ('-t', '-d', '-s') to train, test, and show, respectively in 'run.py'. These flags can be used independently or jointly.

For running the web application, you need to run the 'main.py' file and you will get an url and you need to copy and paste it on a browser. (Figure 2)



```
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 110-028-253
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Figure 4: This is the results after running the code

3.2 Results

3.2.1 Train the Data

Iteration	Log Likelihood	Final Accuracy
5	-0.04436	0.981
20	-0.01954	0.998
25	-0.01670	0.999
30	-0.01464	0.999

Table 1: Results of running the run.py to train the data in the training set and print out the training process

This is a reference to the Second Appendix A.2

From Table 1, we can see that the more the iteration, the higher the final accuracy and log likelihood.

3.2.2 Test the Trained Model

Iteration	F1 Score	Accuracy	Recall	Precision
5	0.9219	0.9739	0.7924	0.9780
20	0.9657	0.9871	0.9002	0.9862
25	0.9657	0.9871	0.9008	0.9858
30	0.9660	0.9872	0.9019	0.9859

Table 2: Results of running the run.py to test the trained model and print out your score on the dev set

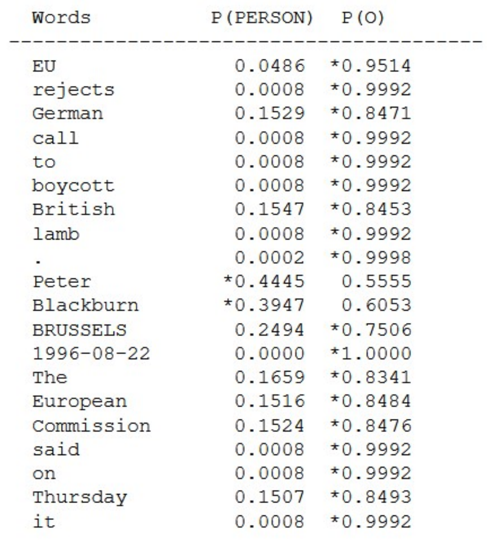
This is a reference to the Third Appendix A.3

From Table 2, we can see that the more the iteration, the higher F1 score, hence, the better the performance of the model to identify the person names.

3.2.3 Prediction of Word's Probability

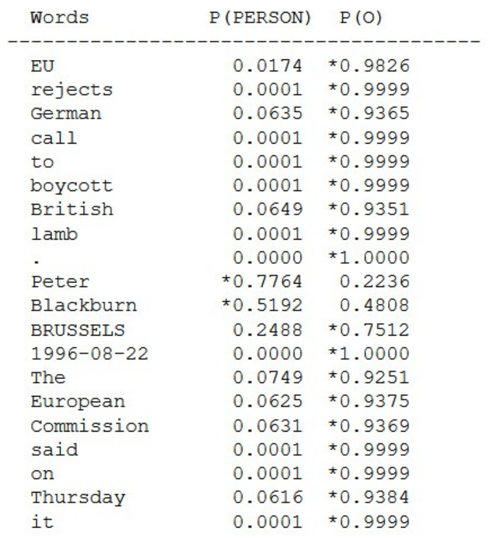
Results of running the run.py to show verbose sample results. The first column is the word, the last two columns are your program's prediction of the word's probability to be PERSON or O. The star '*' indicates the gold result.

We will show you the Figure 3-6 for 5, 20, 25, 30 iterations respectively below.



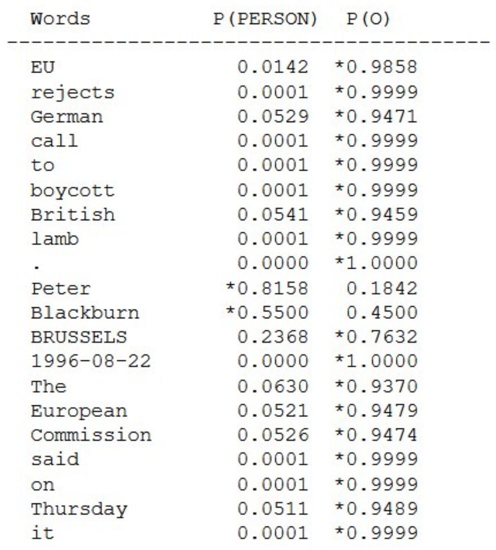
```
 Words           P(PERSON)  P(O)
---------------------------------------
 EU                0.0486  *0.9514
 rejects           0.0008  *0.9992
 German            0.1529  *0.8471
 call              0.0008  *0.9992
 to                0.0008  *0.9992
 boycott           0.0008  *0.9992
 British           0.1547  *0.8453
 lamb              0.0008  *0.9992
 .                 0.0002  *0.9998
 Peter            *0.4445   0.5555
 Blackburn        *0.3947   0.6053
 BRUSSELS          0.2494  *0.7506
 1996-08-22        0.0000  *1.0000
 The               0.1659  *0.8341
 European          0.1516  *0.8484
 Commission        0.1524  *0.8476
 said              0.0008  *0.9992
 on                0.0008  *0.9992
 Thursday          0.1507  *0.8493
 it                0.0008  *0.9992
```

Figure 5: Screenshot for 5 iterations by using '-s' flag in run.py



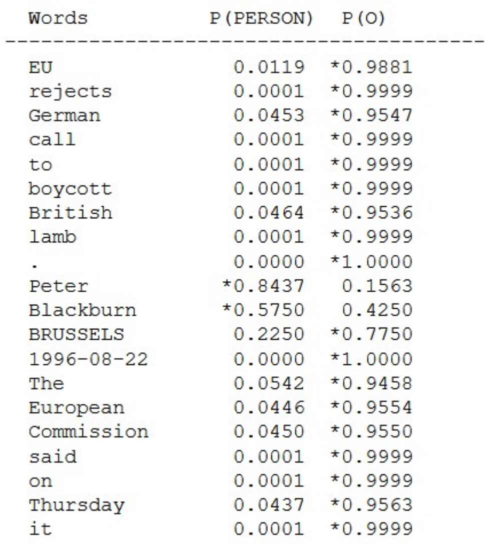
```
 Words           P(PERSON)  P(O)
---------------------------------------
 EU                0.0174  *0.9826
 rejects           0.0001  *0.9999
 German            0.0635  *0.9365
 call              0.0001  *0.9999
 to                0.0001  *0.9999
 boycott           0.0001  *0.9999
 British           0.0649  *0.9351
 lamb              0.0001  *0.9999
 .                 0.0000  *1.0000
 Peter            *0.7764   0.2236
 Blackburn        *0.5192   0.4808
 BRUSSELS          0.2488  *0.7512
 1996-08-22        0.0000  *1.0000
 The               0.0749  *0.9251
 European          0.0625  *0.9375
 Commission        0.0631  *0.9369
 said              0.0001  *0.9999
 on                0.0001  *0.9999
 Thursday          0.0616  *0.9384
 it                0.0001  *0.9999
```

Figure 6: Screenshot for 20 iterations by using '-s' flag in run.py



```
 Words           P(PERSON)  P(O)
---------------------------------------
 EU                0.0142  *0.9858
 rejects           0.0001  *0.9999
 German            0.0529  *0.9471
 call              0.0001  *0.9999
 to                0.0001  *0.9999
 boycott           0.0001  *0.9999
 British           0.0541  *0.9459
 lamb              0.0001  *0.9999
 .                 0.0000  *1.0000
 Peter            *0.8158   0.1842
 Blackburn        *0.5500   0.4500
 BRUSSELS          0.2368  *0.7632
 1996-08-22        0.0000  *1.0000
 The               0.0630  *0.9370
 European          0.0521  *0.9479
 Commission        0.0526  *0.9474
 said              0.0001  *0.9999
 on                0.0001  *0.9999
 Thursday          0.0511  *0.9489
 it                0.0001  *0.9999
```

Figure 7: Screenshot for 25 iterations by using '-s' flag in run.py



```
 Words           P(PERSON)  P(O)
---------------------------------------
 EU                0.0119  *0.9881
 rejects           0.0001  *0.9999
 German            0.0453  *0.9547
 call              0.0001  *0.9999
 to                0.0001  *0.9999
 boycott           0.0001  *0.9999
 British           0.0464  *0.9536
 lamb              0.0001  *0.9999
 .                 0.0000  *1.0000
 Peter            *0.8437   0.1563
 Blackburn        *0.5750   0.4250
 BRUSSELS          0.2250  *0.7750
 1996-08-22        0.0000  *1.0000
 The               0.0542  *0.9458
 European          0.0446  *0.9554
 Commission        0.0450  *0.9550
 said              0.0001  *0.9999
 on                0.0001  *0.9999
 Thursday          0.0437  *0.9563
 it                0.0001  *0.9999
```

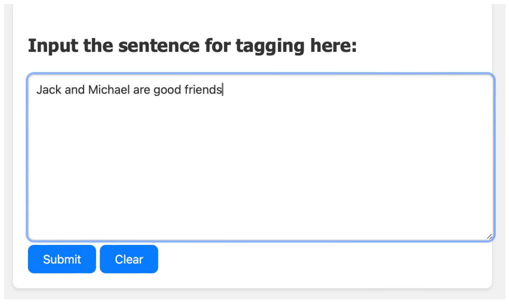
Figure 8: Screenshot for 30 iterations by using '-s' flag in run.py

3.2.4 Finding from the Results

From the above results, we can see that the gold results for the program's prediction of the word's probability to be PERSON are 'Peter' and 'Blackburn' among the first 20 words in the training data, which means that these two words belong to person name according to our trained model. And the best F1 score from our experiments is 0.9660, which trains the data set for 30 iterations. An F1 score of 0.9660 indicates a high level of accuracy and effectiveness in identifying named entities.However, it's important to consider that these findings are specific to the training data and the specific parameters used in the experiments. The model's performance may vary on different datasets or with different parameter settings.

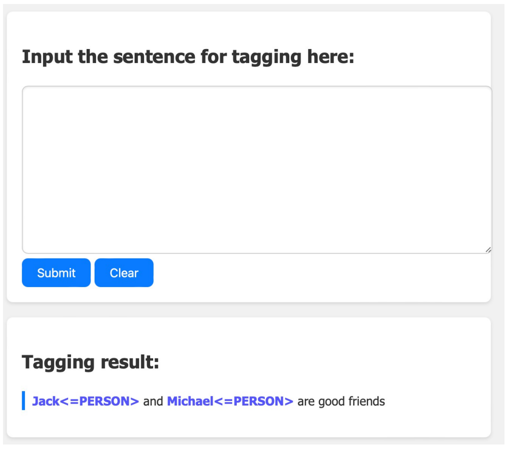
3.2.5 Results of Running the Web Application

We can input a sentence in the input area(Figure 7), and the result (Figure 8) Here are some examples:



Jack and Michael are good friends

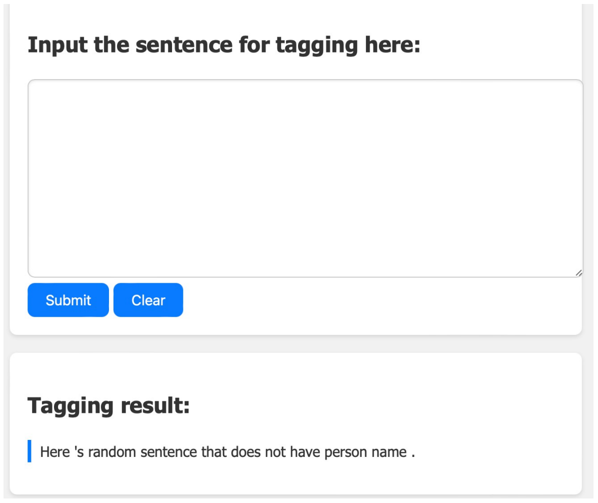
Figure 9: This is the input sentence in web application



Jack<=PERSON> and Michael<=PERSON> are good friends

Figure 10: This is the output in web application

We might have one case that does not have person name. Thus, the result(Figure 9):



Here 's random sentence that does not have person name .

Figure 11: This is the input sentence without person name in web application

3.3 Challenges and its Solutions in this Project

3.3.1 Challenges

The challenges in this project we met:

1. It is hard to find the right feature to optimize the model.
2. We found that if the person name is the first word in the sentence, it won't be predicted as the "PERSON" label in the web app when forecast sentences.

3.3.2 Solutions

The solutions with corresponding difficulties from above:

1. Increasing the number of rounds, on the other hand, is a much easier way to raise the F1 score and make the model work better. Initially, a set of basic features can be chosen, such as word identity, capitalization patterns, and presence of symbols. Then, additional features can be gradually added or modified based on their impact on the model's performance. The performance can be evaluated using evaluation metrics like F1 score, and feature engineering techniques can be employed to enhance the model's effectiveness.

2. We put a "." before the first word in the word list in the predict_sentence() function's word tokenization part. By adding a period before the first word in the sentence, the model receives context information that aids in predicting the correct named entity label. This approach ensures that the first word is not treated differently from subsequent words in terms of labeling. It allows the model to consider the same set of features for all words, leading to more accurate predictions.

4 Conclusion and Future Work

4.1 Conclusion

In conclusion, working on this project likely involved a mix of challenges, excitement, and satisfaction. The process of finding the right features to optimize the model's performance might have posed difficulties, requiring careful analysis and experimentation. Overcoming challenges, such as addressing the issue with predicting the first word in a sentence, likely brought a sense of accomplishment and problem-solving. It's always a good practice to evaluate the model's performance on different evaluation metrics and validate it on separate test data to ensure the reliability of the results. To improve the F1 score and make the model work better, we can look for and add more useful features.

4.2 Future Work

For future work, we can mix different models, like **Support Vector Machines(SVM)** [3] and **Neural Networks(NN)** [1], to make the model work better. Also, we can make the web app's interface easier to use and add more features to it.

Moreover,Conditional random fields (CRFs) and multi-layer neural network approaches have been proposed as alternative methods to address the label bias issue and improve sequence labeling tasks in NLP.

CRFs, introduced by Lafferty et al. (2001), offer a solution to the label bias problem in sequence models. In traditional sequence models, features at a given state compete locally, meaning they do not compete with features at other states in the sequence. However, CRFs directly model the probability distribution over the entire sequence, allowing for global competition of features. This enables better modeling of dependencies and interactions between different states in the sequence. The parameter estimation for CRFs is related to the Generalized Iterative Scaling algorithm used in maximum entropy models. [6]

There are several areas that can be explored to further enhance the model's performance:

1. Feature Engineering: Continuously refining and exploring additional features can improve the model's ability to capture patterns and characteristics of named entities. Experimentation with different feature combinations and advanced techniques, such as word embeddings or contextualized representations, may yield better results.

2. Model Optimization: Besides increasing the number of training iterations, other optimization techniques can be employed, such as regularization methods (e.g., L1 or L2 regularization) to prevent overfitting, parameter tuning, or exploring different algorithms like Conditional Random Fields (CRF) for named entity recognition.

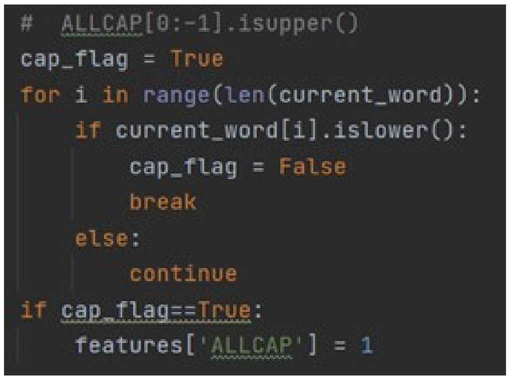
3. Data Augmentation: Expanding the training data by augmenting it with synthetic or labeled external data can help improve the model's generalization and performance. Techniques like data augmentation through synonym replacement, back-translation, or using pre-trained language models for data generation can be explored.

4. Error Analysis and Fine-tuning: Conducting a thorough error analysis can provide insights into the model's weaknesses and areas for improvement. Fine-tuning the model based on the specific errors and misclassifications can lead to better performance on challenging cases.

5. Scaling and Deployment: If the model is intended for production use, considerations should be given to scalability and deployment infrastructure. Optimizing the model's efficiency and integrating it into a robust and scalable system can ensure its effective utilization.

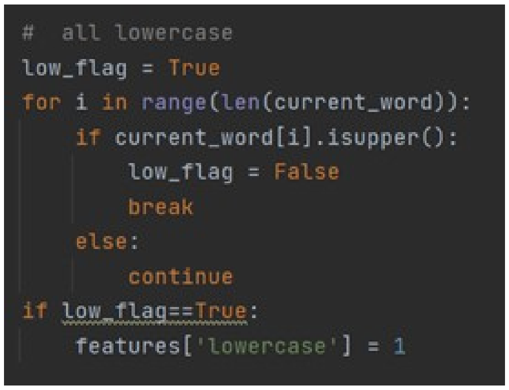
A Appendix

A.1 First Appendix: Features Addition and Predict_sentence



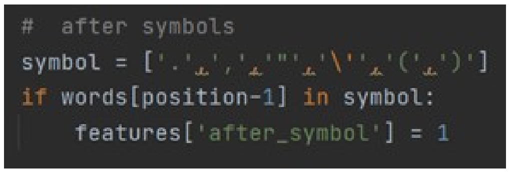
```python
#  ALLCAP[0:-1].isupper()
cap_flag = True
for i in range(len(current_word)):
    if current_word[i].islower():
        cap_flag = False
        break
    else:
        continue
if cap_flag==True:
    features['ALLCAP'] = 1
```

Figure 12: 'ALLCAP' Feature



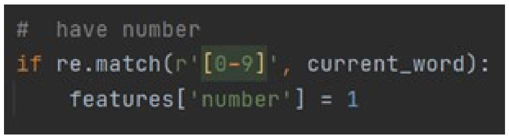
```python
#  all lowercase
low_flag = True
for i in range(len(current_word)):
    if current_word[i].isupper():
        low_flag = False
        break
    else:
        continue
if low_flag==True:
    features['lowercase'] = 1
```

Figure 13: 'lowercase' Feature



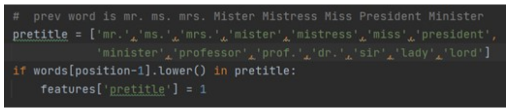
```python
#  after symbols
symbol = ['.',',','"','\'','(',')']
if words[position-1] in symbol:
    features['after_symbol'] = 1
```

Figure 14: 'after_symbol' Feature



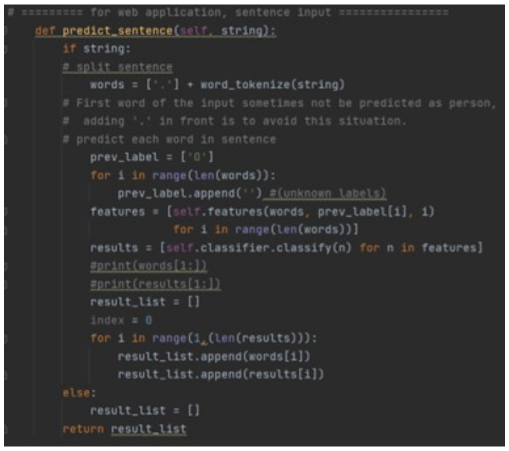
```python
#  have number
if re.match(r'[0-9]', current_word):
    features['number'] = 1
```

Figure 15: 'number' Feature



```python
#  prev word is mr. ms. mrs. Mister Mistress Miss President Minister
pretitle = ['mr.','ms.','mrs.','mister','mistress','miss','president',
            'minister','professor','prof.','dr.','sir','lady','lord']
if words[position-1].lower() in pretitle:
    features['pretitle'] = 1
```

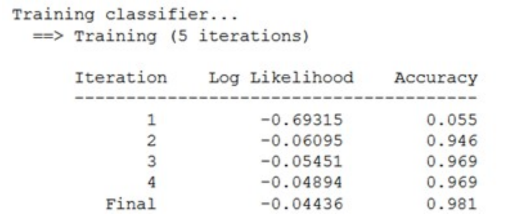
Figure 16: 'pretitle' Feature



```python
# ========= for web application, sentence input ================
    def predict_sentence(self, string):
        if string:
        # split sentence
            words = ['.'] + word_tokenize(string)
        # First word of the input sometimes not be predicted as person,
        #  adding '.' in front is to avoid this situation.
        # predict each word in sentence
            prev_label = ['O']
            for i in range(len(words)):
                prev_label.append('') #(unknown labels)
            features = [self.features(words, prev_label[i], i)
                        for i in range(len(words))]
            results = [self.classifier.classify(n) for n in features]
            #print(words[1:])
            #print(results[1:])
            result_list = []
            index = 0
            for i in range(1,(len(results))):
                result_list.append(words[i])
                result_list.append(results[i])
        else:
            result_list = []
        return result_list
```

Figure 17: Predict_sentence

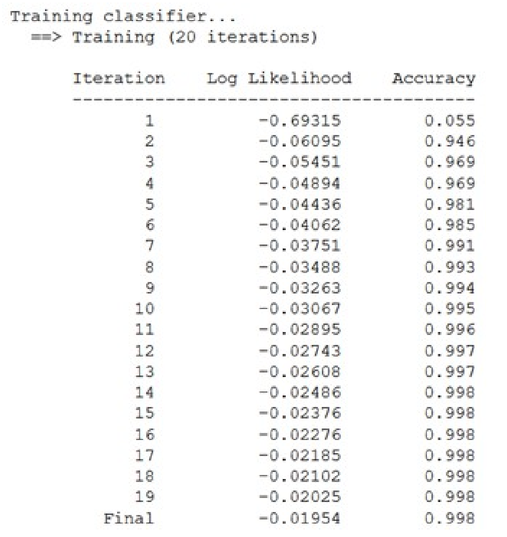
A.2 Second Appendix: Iterations By Using '-t' Flag in run.py



```
Training classifier...
  ==> Training (5 iterations)

      Iteration    Log Likelihood    Accuracy
      ---------------------------------------
             1          -0.69315        0.055
             2          -0.06095        0.946
             3          -0.05451        0.969
             4          -0.04894        0.969
         Final          -0.04436        0.981
```

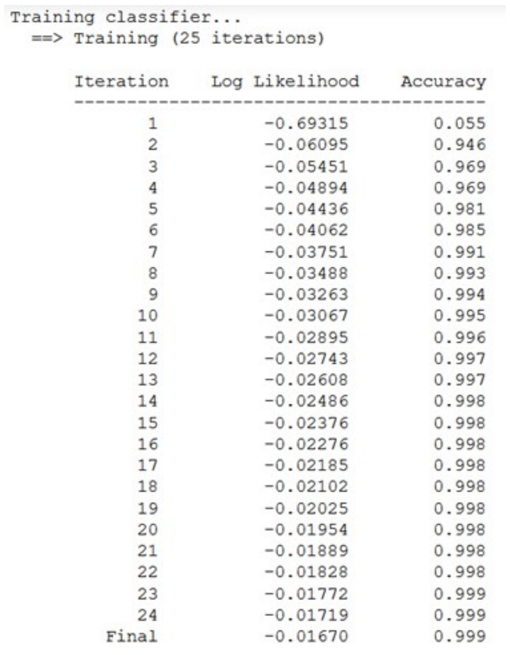
Figure 18: Screenshot for 5 iterations by using '-t' flag in run.py



```
Training classifier...
  ==> Training (20 iterations)

      Iteration    Log Likelihood    Accuracy
      ---------------------------------------
             1          -0.69315        0.055
             2          -0.06095        0.946
             3          -0.05451        0.969
             4          -0.04894        0.969
             5          -0.04436        0.981
             6          -0.04062        0.985
             7          -0.03751        0.991
             8          -0.03488        0.993
             9          -0.03263        0.994
            10          -0.03067        0.995
            11          -0.02895        0.996
            12          -0.02743        0.997
            13          -0.02608        0.997
            14          -0.02486        0.998
            15          -0.02376        0.998
            16          -0.02276        0.998
            17          -0.02185        0.998
            18          -0.02102        0.998
            19          -0.02025        0.998
         Final          -0.01954        0.998
```

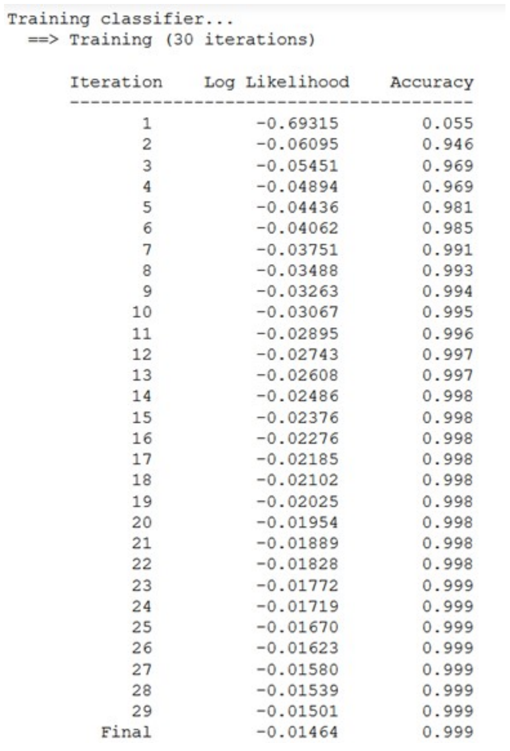
Figure 19: Screenshot for 20 iterations by using '-t' flag in run.py



```
Training classifier...
  ==> Training (25 iterations)

      Iteration    Log Likelihood    Accuracy
      ---------------------------------------
             1          -0.69315        0.055
             2          -0.06095        0.946
             3          -0.05451        0.969
             4          -0.04894        0.969
             5          -0.04436        0.981
             6          -0.04062        0.985
             7          -0.03751        0.991
             8          -0.03488        0.993
             9          -0.03263        0.994
            10          -0.03067        0.995
            11          -0.02895        0.996
            12          -0.02743        0.997
            13          -0.02608        0.997
            14          -0.02486        0.998
            15          -0.02376        0.998
            16          -0.02276        0.998
            17          -0.02185        0.998
            18          -0.02102        0.998
            19          -0.02025        0.998
            20          -0.01954        0.998
            21          -0.01889        0.998
            22          -0.01828        0.998
            23          -0.01772        0.999
            24          -0.01719        0.999
         Final          -0.01670        0.999
```

Figure 20: Screenshot for 25 iterations by using '-t' flag in run.py

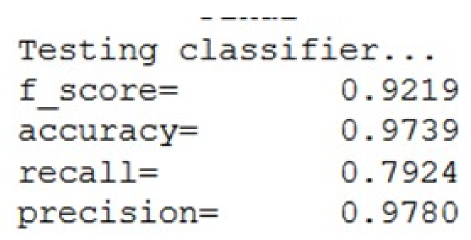


```
Training classifier...
  ==> Training (30 iterations)

      Iteration    Log Likelihood    Accuracy
      ---------------------------------------
             1          -0.69315        0.055
             2          -0.06095        0.946
             3          -0.05451        0.969
             4          -0.04894        0.969
             5          -0.04436        0.981
             6          -0.04062        0.985
             7          -0.03751        0.991
             8          -0.03488        0.993
             9          -0.03263        0.994
            10          -0.03067        0.995
            11          -0.02895        0.996
            12          -0.02743        0.997
            13          -0.02608        0.997
            14          -0.02486        0.998
            15          -0.02376        0.998
            16          -0.02276        0.998
            17          -0.02185        0.998
            18          -0.02102        0.998
            19          -0.02025        0.998
            20          -0.01954        0.998
            21          -0.01889        0.998
            22          -0.01828        0.998
            23          -0.01772        0.999
            24          -0.01719        0.999
            25          -0.01670        0.999
            26          -0.01623        0.999
            27          -0.01580        0.999
            28          -0.01539        0.999
            29          -0.01501        0.999
         Final          -0.01464        0.999
```

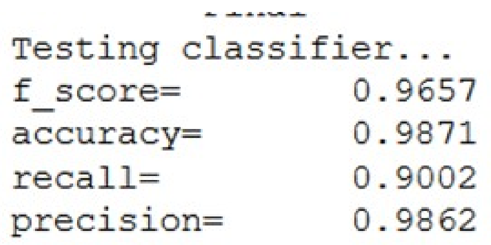
Figure 21: Screenshot for 30 iterations by using '-t' flag in run.py

A.3 Third appendix: Iterations By Using '-d' Flag in run.py



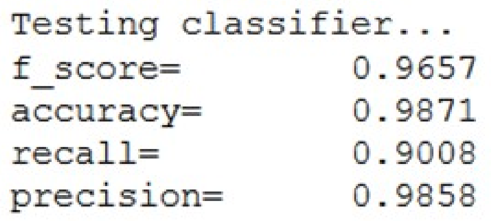
```
Testing classifier...
f_score=          0.9219
accuracy=         0.9739
recall=           0.7924
precision=        0.9780
```

Figure 22: Screenshot for 5 iterations by using '-d' flag in run.py



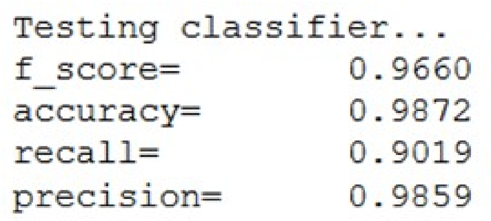
```
Testing classifier...
f_score=          0.9657
accuracy=         0.9871
recall=           0.9002
precision=        0.9862
```

Figure 23: Screenshot for 20 iterations by using '-d' flag in run.py



```
Testing classifier...
f_score=          0.9657
accuracy=         0.9871
recall=           0.9008
precision=        0.9858
```

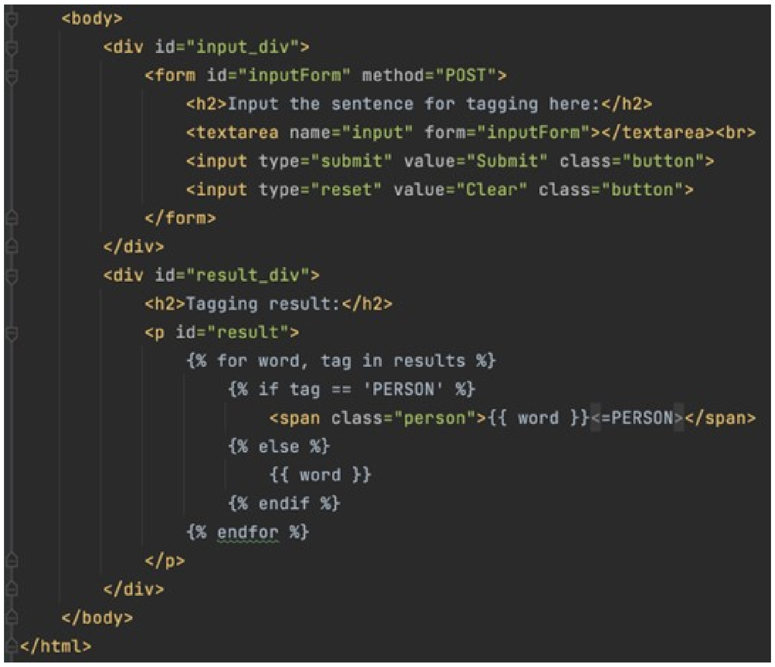
Figure 24: Screenshot for 25 iterations by using '-d' flag in run.py



```
Testing classifier...
f_score=          0.9660
accuracy=         0.9872
recall=           0.9019
precision=        0.9859
```

Figure 25: Screenshot for 30 iterations by using '-d' flag in run.py

A.4 Fourth appendix: CSS code



```
    <body>
        <div id="input_div">
            <form id="inputForm" method="POST">
                <h2>Input the sentence for tagging here:</h2>
                <textarea name="input" form="inputForm"></textarea><br>
                <input type="submit" value="Submit" class="button">
                <input type="reset" value="Clear" class="button">
            </form>
        </div>
        <div id="result_div">
            <h2>Tagging result:</h2>
            <p id="result">
                {% for word, tag in results %}
                    {% if tag == 'PERSON' %}
                        <span class="person">{{ word }}<=PERSON></span>
                    {% else %}
                        {{ word }}
                    {% endif %}
                {% endfor %}
            </p>
        </div>
    </body>
</html>
```

Figure 26: This is CSS code is written in the html files together as well

References

[1] Arvindpdmn, "Neural networks for NLP," 11 2019. [Online]. Available: https://devopedia.org/neural-networks-for-nlp

[2] A. d. Arvindpdmn, "Maximum-Entropy Markov model," 2 2022. [Online]. Available: https://devopedia.org/maximum-entropy-markov-model

[3] Elton, "The Support Vector Machines (SVM) algorithm for NLP," 12 2021. [Online]. Available: https://pythonwife.com/the-support-vector-machines-svm-algorithm-for-nlp/

[4] J. Li, A. Sun, J. Han, and C. Li, "A survey on deep learning for named entity recognition," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, pp. 50–70, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9039685

[5] T. Li and C. Zhang, "Research on the application of multimedia entropy method in data mining of retail business," *Scientific Programming*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247645031

[6] A. Ratnaparkhi, *Maximum Entropy Models for Natural Language Processing*. Boston, MA: Springer US, 2017, pp. 800–805. [Online]. Available: https://doi.org/10.1007/978-1-4899-7687-1_525