Evaluation and Optimization of a MEMM-based Model for Named Entity Recognition

Maijiajun, Guopengze

April 28, 2025

Abstract

This report describes the implementation and optimization of a Maximum Entropy Markov Model (MEMM) for Named Entity Recognition (NER) to identify people's names in text. The model utilizes feature engineering and a maximum entropy classifier, incorporating gazetteers for both Latin script and Chinese Pinyin names. Additionally, challenges in model performance, training efficiency, and multilingual coverage are addressed, along with solutions such as ternary search for optimization.

1 Introduction

Named Entity Recognition (NER) is a critical task in natural language processing (NLP) that aims to detect entities like people, places, and organizations in text. This project implements a MEMM for the task of recognizing people's names using various features, including morphological properties, gazetteer-based name lists, and non-English name recognition. The model is evaluated based on metrics like accuracy, precision, recall, and F1-score. An interface is built using Gradio, enabling interactive training and sentence prediction.

2 Methodology

2.1 Model Overview

The Maximum Entropy Markov Model (MEMM) uses sequential labeling and prediction, where the model is trained to predict labels based on both the current word and the label of the previous word. The primary components of the methodology include:

- **Feature Engineering:** Features such as word case, punctuation, and whether a word is part of a gazetteer are used. Additionally, the model checks for non-English names using regular expressions.
- **Classifier:** The model employs a Maximum Entropy classifier from NLTK, which is trained using the features extracted from the training data.

- **Evaluation:** The model is evaluated using precision, recall, accuracy, and F1-score. Beta-F1 score with beta of 1.5 is also utilized to better optimize the model with low recall observed in some test cases.

2.2 Gradio User Interface

A Gradio interface is implemented to provide a simple web-based interface for interacting with the model. The interface is divided into two tabs:

- **Training Tab:** Allows users to visualize the training process and monitor metrics like accuracy and loss.
- **Prediction Tab:** Allows users to input sentences and see the predicted labels for each word in the sentence.

2.3 Challenges and Solutions

2.3.1 Feature Extraction

This section explains the design and rationale behind the feature extraction process.

1.Basic Token Properties

```
current_word = words[position]
current_word_lower = current_word.lower()
features['has_(%s)' % current_word] = 1
features['cur_word_len'] = len(current_word) // 2
if current_word_lower in self.nltk_stopwords:
    features['cur_word_is_stopword'] = 1
```

The basic features describe the word's surface form, its relative length, and stopword status. They help distinguish frequent functional words from informative content words or entities.

2.Name Gazetteer Matching

```
features['cur_word_in_first_name_gazetteer'] = (current_word_lower
    in self.first_name_gazetteer)
features['cur_word_in_last_name_gazetteer'] = (current_word_lower
    in self.last_name_gazetteer)
features['cur_word_in_cn_first_name_gazetteer'] =
    (current_word_lower in self.cn_first_name_gazetteer)
features['cur_word_in_cn_last_name_gazetteer'] =
    (current_word_lower in self.cn_last_name_gazetteer)
```

Matching against English and Chinese name gazetteers improves recall in identifying person names, especially when contextual cues are limited.

Two gazetteers are both received online and were pre-processed by us that the model can quickly load them when being used.

3.Rough Features Of Previous Word

```
prev_word = words[position - 1] if position > 0 else "."
features['prev_word_label'] = previous_label
features['prev_word_is_capitalized'] = prev_word[0].isupper()
features['prev_word_ends_with_punctuation'] = prev_word[-1] in
    string.punctuation
features['prev_word_is_title'] =
    any((prev_word.lower().startswith(t) and
    len(prev_word.lower()) <= len(t) + 1) for t in self.titles)
features['prev_word_len'] = len(prev_word) // 2
features['prev_word_is_digit'] = prev_word.isdigit()
```

Capturing the previous word's characteristics (such as punctuation or title words) models local dependencies crucial for sequence prediction, and hints whether a named entity starts.

4.Letter Case Features

```
if current_word[0].isupper():
    features['Titlecase'] = 1
if current_word.isupper():
    features["Allcapital"] = 1
if self.camel_regex.fullmatch(current_word):
    features["Camelcase"] = 1
```

Letter case patterns are strong cues for entity recognition. Many named entities appear in title case, while acronyms and abbreviations are often all capitals.

The `Camelcase` is focus on the words like 'iPhone' and 'McDonald'.

5.Two Punctuation Features

```
if "'" in current_word:
    features["Apostrophe"] = 1
if "-" in current_word:
    features["Hyphen"] = 1
```

Punctuation symbols provide syntactic signals. Apostrophes often relate to contractions or possessives, while hyphens commonly occur in compound proper nouns.

6.Prefix and Suffix Hashing

```
max_record_len = 4
_prefix = current_word[:max_record_len]
_suffix = current_word[-max_record_len:]
features["prefix_hash"] =
    int(hashlib.md5(_prefix.encode()).hexdigest(), 16) % 100000
features["suffix_hash"] =
    int(hashlib.md5(_suffix.encode()).hexdigest(), 16) % 100000
```

Hashing prefixes and suffixes encodes morphological information compactly, enabling the model to capture common word-form patterns without introducing high-dimensional sparse features.

7.Non-English and Special Character Features

```
if self.pinyin_regex.fullmatch(current_word_lower):
    features["Pinyin"] = 1
```

```
if current_word_lower.endswith("lyu") or
    current_word_lower.startswith("lyu"):
    features["Pinyin_lyu"] = 1
if current_word_lower in self.pinyin_confusion:
    features["Pinyin_confusion"] = 1

if any(ord(c) > 127 and self.is_latin_char(c) for c in
    current_word):
    features["Contain_non_ascii_latin"] = 1
if any(char.isdigit() for char in current_word):
    features["Contain_any_number"] = 1
if not current_word.isalpha():
    features["Contain_no_alpha"] = 1
```

These features aim to handle non-English input, spelling variations, transliterations, and noisy text. Identifying patterns like Pinyin, digits, or non-alphabetic characters ensures broader linguistic coverage.

The Pinyin regularization is as follow. The `pinyin_confusion` is prepared for the words have correct Pinyin spelling and also English meanings.

```
self.pinyin_regex = re.compile(
    r"^("
    r"(a[io]?|ou?|e[inr]?|ang?|ng|[bmp](a[io]"
    r"?|[aei]ng?|ei|ie?|ia[no]|o|u)|"
    r"pou|me|m[io]u|[fw](a|[ae]ng?|ei|o|u)|fou"
    r"|wai|[dt](a[io]?|an|e|[aeio]ng|"
    r"ie?|ia[no]|ou|u[ino]?|uan)|dei|diu|[nl](a[io]?"
    r"|ei?|[eio]ng|i[eu]?|i?ang?|"
    r"iao|in|ou|u[eo]?|ve?|uan)|nen|lia|lun|[ghk](a[io]"
    r"?|[ae]ng?|e|ong|ou|u[aino]?|uai|uang?)|[gh]ei|[jqx](i"
    r"(ao?|ang?|e|ng?|ong|u)?|u[en]?|uan)"
    r"|([csz]h?|r)([ae]ng?|ao|e|i|ou|u[ino]?"
    r"|uan)|[csz](ai?|ong)|[csz]h(ai?|uai|"
    r"uang)|zei|[sz]hua|([cz]h|r)ong|y(ao?"
    r"|[ai]ng?|e|i|ong|ou|u[en]?|uan))"
    r"){1,4}$"
)
self.pinyin_confusion = {"me", "ma", "bin", "fan", "long", "sun",
    "panda", "china"}
```

We use `unicodedata` to identify whether a char is Latin letter or not.

```
@staticmethod
    def is_latin_char(check_char):
        try:
            return unicodedata.name(check_char).startswith('LATIN')
        except ValueError:
            return False
```

2.3.2 Low Recall

One significant issue observed was low recall during initial testing. Despite high F1 scores, most names were classified as "O" (non-entity). To address this, we emphasized optimizing for recall by using the Beta-F1 score instead of focusing solely on F1 score.

2.3.3 Training Efficiency

Training the model was slow due to the constraints of the MaxentClassifier, which requires retraining from scratch each time. To optimize this, a ternary search algorithm was implemented to find the optimal number of iterations, reducing the time complexity from $O(Kn^2)$ to $O(Kn \log n)$.

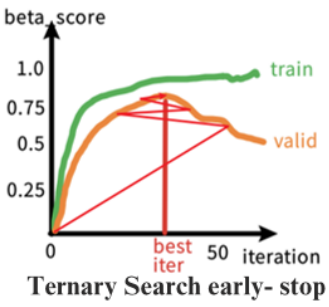

Figure 1: Ternary Search Early Stop

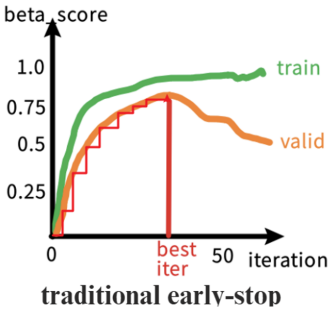

Figure 2: Traditional Early Stop

2.3.4 Gazetteer Expansion

To handle multilingual named entity recognition, two gazetteers were constructed:

- **Latin Script Gazetteer:** Contains approximately 160k first names and 100k last names, covering 18 languages using Latin script.
- **Chinese Pinyin Gazetteer:** Includes 35k+ common names and 308 family names to help the model recognize Chinese Pinyin names effectively.

3 Results and Discussion

The model was evaluated on a development set, and the performance metrics were as follows:

- **F1-Score:** 0.9693
- **Accuracy:** 0.9689
- **Precision:** 0.9693
- **Recall:** 0.7538

Although the F1 score was high, recall remained low initially, primarily due to insufficient coverage of diverse names in the training set. Expanding the gazetteers and adjusting for recall using Beta-F1 score improved the results significantly.

4 Conclusion

The MEMM-based NER system demonstrates strong performance in identifying person names, especially when using a combination of feature engineering and specialized gazetteers. Challenges such as low recall, slow training, and limited multilingual coverage were addressed through careful optimization, including ternary search for training efficiency and expanding the gazetteer for diverse names. Future work could explore integrating deep learning-based models to further improve performance, especially in cases with complex linguistic variations.