Hidden Markov Model for Named Entity Recognition

Submitted by
Mitesh Kumar (M23MAC004)

Colab: assignment_2.ipynb

1. Introduction

Named Entity Recognition (NER) is a fundamental task in Natural Language Processing that aims to locate and classify named entities mentioned in unstructured text into predefined categories. This report details the implementation of an HMM-based approach for NER on Twitter data, where we identify and classify ten different types of named entities: person, product, company, geolocation, movie, music artist, TV show, facility, sports team, and others.

2. Dataset Analysis

The dataset used for this task consists of three files:

- **Train.txt**: Used to train the model (2,394 sentences, 46,469 tokens)
- **Dev.txt**: Used for validation (1,000 sentences, 16,261 tokens)
- **Test.txt**: Used for final evaluation (3,850 sentences, 61,908 tokens)

Each file contains Twitter messages with tokens annotated in the BIO (Beginning, Inside, Outside) tagging scheme, where:

- **B-tag**: Marks the beginning of a named entity
- **I-tag**: Indicates a token inside a named entity
- **O**: Denotes tokens that are not part of any named entity

2.1 Dataset Characteristics

A detailed analysis of the dataset reveals several important characteristics:

2.1.1 Tag Distribution

The dataset is highly imbalanced, with the 'O' tag (non-entity) dominating across all splits:

- Train set: 94.70% 'O' tags
- Dev set: 93.06% 'O' tags
- Test set: 90.38% 'O' tags

This imbalance presents a significant challenge, as the model may be biased toward predicting the majority class.

2.1.2 Entity Type Distribution

Among the entity types, the most frequent in the training data are:

- Person: 1.43% (B-person + I-person)

- Other: 1.17% (B-other + I-other)
- Location: 0.70% (B-loc + I-loc)
- Facility: 0.45% (B-facility + I-facility)
- Company: 0.45% (B-company + I-company)

The least frequent entities are:

- Sports team: 0.16% (B-sportsteam + I-sportsteam)
- TV show: 0.14% (B-tvshow + I-tvshow)
- Movie: 0.17% (B-movie + I-movie)

2.1.3 Sentence Length

The average sentence lengths are:

- Train set: 19.41 tokens
- Dev set: 16.26 tokens
- Test set: 16.08 tokens

2.1.4 Vocabulary Size

The dataset has a considerable vocabulary size:

- Train set: 10,586 unique words
- Dev set: 6,255 unique words
- Test set: 18,320 unique words

The large number of unique words in the test set (greater than in the training set) indicates potential challenges with out-of-vocabulary words during testing.

3. Methodology

3.1 Hidden Markov Model Overview

Hidden Markov Models (HMMs) are probabilistic sequence models that assume the sequence of observations (words) is generated by a sequence of hidden states (entity tags). Three components define the model:

1. **Initial State Probabilities (π)**: The probability of a sentence starting with a particular tag
2. **Transition Probabilities (A)**: The probability of transitioning from one tag to another
3. **Emission Probabilities (B)**: The probability of a word being generated by a particular tag

3.2 Model Variants

We implemented four variants of the HMM model with different configurations:

1. **Bigram model without context**: Uses bigram transitions between tags and calculates emission probabilities without considering the previous tag
2. **Bigram model with context**: Uses bigram transitions and incorporates the previous tag when calculating emission probabilities
3. **Trigram model without context**: Uses trigram transitions (two previous tags) and calculates emission probabilities without context
4. **Trigram model with context**: Uses trigram transitions and incorporates the previous tag in emission probabilities

3.3 HMM Parameter Estimation

3.3.1 Finding States (Tags)

The first step is to identify all possible tags in the training data. In our dataset, we found 21 unique tags, including the 'O' tag and the 'B-' and 'I-' prefixes for the ten entity types.

3.3.2 Calculating Start Probabilities (π)

For each tag, we calculate the probability of a sentence starting with that tag:

```
Unset
π(tag) = (count of sentences starting with tag + smoothing) /
(total sentences + smoothing * |tags|)
```

Smoothing is applied to handle tags that may not appear at the beginning of sentences in the training data.

3.3.3 Calculating Transition Probabilities (A)

Bigram Model: For each pair of tags (prev_tag, curr_tag), we calculate:

```
Unset
A(prev_tag -> curr_tag) = (count of transitions from prev_tag to
curr_tag + smoothing) /
                        (total transitions from prev_tag +
smoothing * |tags|)
```

Trigram Model: For each triplet of tags (prev_prev_tag, prev_tag, curr_tag), we calculate:

```
Unset
A((prev_prev_tag, prev_tag) -> curr_tag) = (count of transitions
from (prev_prev_tag, prev_tag) to curr_tag + smoothing) /
                                         (total transitions from
(prev_prev_tag, prev_tag) + smoothing * |tags|)
```

3.3.4 Calculating Emission Probabilities (B)

Without Context: For each (tag, word) pair, we calculate:

```
Unset
B(word | tag) = (count of word with tag + smoothing) /
              (total words with tag + smoothing * |vocabulary|)
```

With Context: For each (prev_tag, curr_tag, word) triplet, we calculate:

```
Unset
B(word | prev_tag, curr_tag) = (count of word with (prev_tag,
curr_tag) + smoothing) /
                             (total words with (prev_tag,
curr_tag) + smoothing * |vocabulary|)
```

3.4 Decoding with the Viterbi Algorithm

To predict the most likely sequence of tags for a given sentence, we use the Viterbi algorithm:

1. **Initialization**: For each possible tag, compute the probability of starting with that tag and emitting the first word
2. **Recursion**: For each subsequent word and each possible tag, find the most likely previous tag that maximizes the probability of the sequence
3. **Termination**: Select the most likely final tag and trace back to find the entire sequence

The algorithm uses dynamic programming to find the optimal tag sequence.

3.5 Implementation Details

The implementation includes several important features:

- **Smoothing**: Laplace smoothing (add-one) is applied to handle unseen words and transitions
- **Log Probabilities**: To prevent numerical underflow, we work with log probabilities during the Viterbi algorithm
- **Default Values**: For unseen words, we use a default emission probability based on the tag's distribution
- **Special Handling for Trigrams**: The trigram model requires special handling for the first and second words in a sentence

4. Experimental Results

4.1 Model Performance

The performance of the four model variants on the test set is summarized below:

Model	Accuracy	Precision	Recall	F1 Score
Bigram without context	0.6378	0.8949	0.6378	0.7424
Bigram with context	0.6748	0.8746	0.6748	0.7597
Trigram without context	0.9029	0.8168	0.9029	0.8577
Trigram with context	0.9033	0.8168	0.9033	0.8579

4.2 Performance Analysis

4.2.1 Bigram Models

The bigram models show moderate performance with F1 scores of 0.74-0.76. Adding context to the emission probabilities improves performance across all metrics:

- **Accuracy**: Increases from 0.6378 to 0.6748 (+3.7%)
- **F1 Score**: Improves from 0.7424 to 0.7597 (+1.7%)

The precision is slightly lower in the context model, but this is outweighed by the substantial improvement in recall.

Looking at entity-specific performance, the bigram models struggle with rare entity types like "movie," "tvshow," and "sportsteam," with F1 scores near zero. They perform best on more common entities like "facility," "loc," and "person."

4.2.2 Trigram Models

The trigram models show significantly higher accuracy and F1 scores compared to the bigram models. However, a closer examination reveals a critical issue: they consistently predict the "O" tag for all tokens.

This is evidenced by:

- Perfect recall (1.0) for the "O" class
- Zero precision, recall, and F1 score for all entity classes
- High overall accuracy (0.90) due to the dominance of the "O" class

This suggests that the trigram models have learned to always predict the majority class, effectively ignoring the entity classes altogether. This is a classic example of the class imbalance problem.

4.3 Comparative Analysis

1. **Bigram vs. Trigram**: While the trigram models appear to outperform the bigram models in terms of overall metrics, they fail to identify any named entities, making the bigram models more useful in practice.

2. **Context vs. No Context**: Adding context to emission probabilities consistently improves performance in both the bigram and trigram settings, demonstrating the value of considering the previous tag when calculating word probabilities.

3. **Entity Type Performance**: For the bigram models, performance varies significantly across entity types:

    - Location (B-loc): F1 = 0.39 (without context), 0.22 (with context)
    - Facility (I-facility): F1 = 0.31 (without context), 0.15 (with context)
    - Person (I-person): F1 = 0.22 (without context), 0.18 (with context)

5. Error Analysis

5.1 Trigram Model Issues

The trigram models' bias toward the "O" tag can be attributed to:

1. **Data Sparsity**: With more complex transitions (trigrams vs. bigrams), the model encounters more unseen patterns.
2. **Severe Class Imbalance**: With over 90% of tokens labeled as "O," the model learns that predicting "O" for everything minimizes error on the training data.
3. **Overfitting**: The trigram model may overfit to the patterns in the training data, particularly the dominance of the "O" tag.

5.2 Bigram Model Challenges

The bigram models show more balanced performance but still face challenges:

1. **Rare Entity Types**: Poor performance on infrequent entity types due to limited training examples
2. **Precision-Recall Tradeoff**: The models with context show lower precision but higher recall, indicating a tendency to predict more entities but with lower confidence
3. **Context Limitations**: While adding context helps, a single previous tag may not provide enough context for accurate NER in the complex Twitter domain

6. Conclusions

1. HMM-based approaches can achieve reasonable performance for NER on Twitter data, with the best model achieving an F1 score of 0.76.
2. Adding context to emission probabilities consistently improves performance, highlighting the importance of considering surrounding tokens.
3. The extreme class imbalance in the dataset poses significant challenges, particularly for more complex models like the trigram HMM.
4. Simpler models (bigram) outperform more complex ones (trigram) in this task due to their better handling of data sparsity and class imbalance.

Overall, while the HMM approach provides a strong baseline for NER on Twitter data, the task remains challenging due to the nature of social media text and the imbalanced distribution of entity types. Future work should focus on addressing these specific challenges to improve performance further.

Appendix: Implementation Details

The implementation was done in Python, utilizing libraries such as NumPy for numerical operations, scikit-learn for evaluation metrics, and matplotlib/seaborn for visualization. The core components include:

1. **HMMTagger class**: Implements the HMM model with methods for:
    - Data preprocessing
    - Parameter estimation (start, transition, emission probabilities)
    - Viterbi decoding
    - Model evaluation
2. **Data Analysis Functions**:
    - Dataset statistics calculation
    - Visualization of tag distributions

    - Performance comparison across models
3. **Smoothing Technique**:

    - Laplace smoothing with a small value (1e-10) to handle unseen patterns
4. **Serialization**:

    - Model parameters are saved using pickle for later reuse
    - Predictions are saved to text files for submission