Machine Learning Design Project Report

Aishwarya Iyer 1007141, Khoo Zi Qi 1006984

April 2025

Overview

This project implements core components of a Hidden Markov Model (HMM) for Named Entity Recognition (NER). It provides functions to compute emission probabilities, apply smoothing for rare words, generate predictions, and evaluate model performance using precision, recall, and F1 score. The code requires a training dataset where each line contains a word and its corresponding tag, separated by whitespace. Sentences are separated by blank lines. For Part 4, we applied the Averaged Perceptron algorithm to further improve tagging performance.

In terms of evaluation metrics, the progression across methods was observed:

$$\text{Baseline} < \text{Viterbi 4th Best} < \text{Viterbi} < \text{Averaged Perceptron}$$

Part 1

Computing Emission Parameters (Unsmoothed)

> **Input:** Path to training data file.
>
> **Output:** A nested dictionary `emission_parameters[tag][word] = probability`.

The function `compute_emission_parameters(train_file_path)` estimates the unsmoothed emission parameters from the training set using MLE (Maximum Likelihood Estimation). Firstly, the function initializes a counter `emission_counts` and a counter `tag_counts` using the data structure `defaultdict`. The data structure of `defaultdict` was chosen as it automatically handles missing keys. The `emission_counts` keeps track of the number of times a word appears with a tag and is initialized to 0. The `tag_counts` keeps track of the number of occurrences per tag and is also initialized to 0.

The function then opens the file from the `train_file_path`, and counts the word-tag co-occurrences, and the tag frequencies, and updates it to `emission_counts` and `tag_counts` respectively. After the `emission_counts` and `tag_counts` are updated, the emission probabilities are calculated and updated into another `defaultdict` labelled `emission_parameters`. The emission probabilites are calculated through the following formula:

$$e(x|y) = \frac{\text{Count}(y \to x)}{\text{Count}(y)} \tag{1}$$

The function returns the nested dictionary `emission_parameters` that contains the emission parameters.

Computing Emission Parameters (Smoothed)

Input: Training file path, threshold k.

Output: Smoothed emission parameters dictionary.

The function `compute_emission_parameters_smoothing(train_file_path, k)` adds smoothing to handle rare words. The function initializes a counter `emission_counts` and a counter `tag_counts`. The `emission_counts` keeps track of the number of times a word appears with a tag and is initialized to 0. The `tag_counts` keeps track of the number of occurrences per tag and is also initialized to 0. Similar to the previous function, this function then opens the file from the `train_file_path`, and counts the word-tag co-occurrences, and the tag frequencies, and updates it to `emission_counts` and `tag_counts` respectively. However, there is an added step to ensure that words with frequency $< k$ are replaced with a special token #UNK#. This handles Out-of-Vocabulary words to improve generalization and reduces sparsity. The emission probabilities (calculated by (1)) are then computed using this modified training data and updated into `emission_parameters`.

Predict Tags

Input: Development file path, emission parameters, unknown tag.

Output: List of (word, predicted_tag) tuples.

The function `predict_tags(dev_in_file_path, emission_parameters, unknown_tag='O')` predicts tags for a sentence using emission probabilities.

$$y* = \arg\max_{y} e(x|y) \tag{2}$$

The function iterates through all of the possible `tags` and checks if the current `word` exists in the `emission_probabilities`. It then finds the `best_tag` by implementing (2).

Write Predictions

Input: A list of predicted pairs and an output file path.

The function `write_predictions(predicted_list, output_file_path)` writes predicted (`word`, `tag`) pairs to a file, one pair per line.

For part 1, the output is found in 'EN/dev.p1.out'

Precision, recall and F scores of baseline system

```
#Entity in gold data: 13179
#Entity in prediction: 16653

#Correct Entity : 8
Entity  precision: 0.0005
Entity  recall: 0.0006
Entity  F: 0.0005

#Correct Sentiment : 8
Sentiment  precision: 0.0005
Sentiment  recall: 0.0006
Sentiment  F: 0.0005
```

The baseline system achieves extremely low performance, with an entity precision, recall, and F-score of only 0.0005, and similar results for sentiment classification. This shows that predicting tags based solely on emission probabilities is highly unreliable, as it ignores the structure and sequence of the tags.

Part 2

Computing Transition Parameters with Laplace Smoothing

Input: Training file path, smoothing factor (default 0.1).

Output: Smoothed transition parameters dictionary.

The function `compute_transition_parameters(train_file_path, smoothing=0.1)` calculates the transition parameters from the training set using Maximum Likelihood Estimation (MLE) with Laplace smoothing. It initializes counters to track tag-to-tag transitions and tag occurrences.

The transition probability is estimated as:

$$q(y_i \mid y_{i-1}) = \frac{\text{Count}(y_{i-1}, y_i)}{\text{Count}(y_{i-1})} \tag{3}$$

Special transitions are also handled:

$$q(\text{STOP} \mid y_n) \quad \text{and} \quad q(y_1 \mid \text{START}) \tag{4}$$

This smoothing technique ensures that even unseen transitions are assigned a small non-zero probability, making the model more robust.

Implementing the Viterbi Algorithm

Input: Sentence (list of words), transition parameters, emission parameters, set of all tags.

Output: Predicted tag sequence.

The function `viterbi(sentence, transition_params, emission_params, all_tags)` implements the Viterbi algorithm to find the most probable sequence of tags for a given sentence. It initializes a `viterbi_matrix` to store the best log-probabilities for each tag at each position, and a `backpointer` to keep track of the optimal path.

The function first computes the scores for the first word by combining the transition probability from `START` to each tag and the emission probability of the word given the tag. Small ϵ values are used to handle unseen transitions or emissions and to avoid computing `log(0)`.

Then, for each word in the sentence, it recursively updates the best score for each tag by considering all possible previous tags and selecting the one that maximizes the total log-probability.

At the final word, the function transitions to the `STOP` state, selecting the best final tag. It then reconstructs the most probable tag sequence by backtracking through the `backpointer` from the end to the start.

This approach ensures both robustness to missing probabilities and numerical stability through the use of log-space computations.

Running Viterbi on Development Set

Input: Development input file path, output file path, transition parameters, emission parameters, set of all tags.

Output: Predicted word-tag pairs written to the output file.

The function `run_viterbi_on_dev_set(dev_in_path, output_path, transition_params, emission_params all_tags)` reads the development input file line by line, groups words into sentences, and applies the Viterbi algorithm to each sentence to predict the most probable tag sequence.

For each completed sentence, it writes the (word, tag) pairs to the output file, preserving sentence boundaries with blank lines.

Precision, recall and F scores with Viterbi

```
#Entity in gold data: 13179
#Entity in prediction: 13280

#Correct Entity : 11094
Entity  precision: 0.8354
Entity  recall: 0.8418
Entity  F: 0.8386

#Correct Sentiment : 10592
Sentiment  precision: 0.7976
Sentiment  recall: 0.8037
Sentiment  F: 0.8006
```

Comparison between Viterbi and Baseline System

The Viterbi algorithm performs significantly better than the baseline system, which only uses emission probabilities without considering transition probabilities. By incorporating both emission and transition probabilities, Viterbi is able to model the sequential nature of the tagging problem, leading to more coherent and accurate predictions. The results clearly demonstrate that modelling tag dependencies is crucial for achieving good performance.

Part 3

Finding the k-th Best Sequence with Viterbi

Input:
- `sentence`: List of words in the sentence.
- `transition_params`: Transition probability dictionary.
- `emission_params`: Emission probability dictionary.
- `all_tags`: Set of all possible tags.
- `k`: Rank of the sequence to return (e.g., $k = 4$ returns the 4th best path).

Output: List of tags representing the k-th best sequence for the given sentence.

The function `kth_best_viterbi(sentence, transition_params, emission_params, all_tags, k=4)` modifies the standard Viterbi algorithm to compute not only the best tag sequence but the k-th best.

It initializes a dynamic programming table `viterbi_matrix` to store the top k probabilities for each tag at each time step, and a `backpointer` structure to trace back the corresponding paths. At each step, it evaluates all possible previous paths and retains only the top k highest scoring ones.

After processing the sentence, it selects the k-th highest scoring path ending with the STOP transition. The sequence is then reconstructed by backtracking through the stored pointers. If there are fewer than k valid paths, the last available path is returned instead.

Applying the k-th Best Viterbi to a Development Set

Input:
- `dev_in_path`: Path to the input file (one word per line, blank lines between sentences).
- `output_path`: Path to write the output (word-tag pairs).
- `transition_params`: Transition probability dictionary.
- `emission_params`: Emission probability dictionary.
- `all_tags`: Set of all possible tags.
- `k`: The rank of the path to predict (e.g., $k = 4$ for the 4th best path).

Output: Writes predicted $(word, tag)$ pairs to `output_path`, preserving sentence boundaries.

The function `run_kth_best_viterbi(dev_in_path, output_path, transition_params, emission_params, all_tags, k=4)` reads a development set sentence by sentence, applies the k-th best Viterbi decoding via `kth_best_viterbi`, and writes the predicted word-tag pairs to the output file.

Precision, recall and F scores of 4th best sequence with Viterbi

```
#Entity in gold data: 13179
#Entity in prediction: 13408

#Correct Entity : 10625
Entity  precision: 0.7924
Entity  recall: 0.8062
Entity  F: 0.7993

#Correct Sentiment : 10095
Sentiment  precision: 0.7529
Sentiment  recall: 0.7660
Sentiment  F: 0.7594
```

The performance of the 4th best sequence with Viterbi is slightly worse than that in Part 2, which is expected as the 4th best sequence here naturally performs below the best sequence used in Part 2.

Part 4

Model and Method: Averaged Perceptron for Sequence Labeling

We implemented a **first-order sequence labeling model** using the **Averaged Perceptron** algorithm. This model is used to assign chunk-level tags (e.g., `B-NP`, `I-VP`, `O`) to each token in a sentence, learning from features that capture both the word and its context.

Model Overview

The **Averaged Perceptron** is an online, linear classifier. It updates its weights based on prediction errors and averages weights across all iterations to improve generalization and reduce overfitting. At each step, it:

1. Predicts a label using current feature weights.
2. Compares the prediction with the gold label.
3. If incorrect, updates the weights for the correct and incorrect labels.
4. Averages weights over time.

The update rule for label weights is:

$$w_y = w_y + \phi(x), \quad w_{\hat{y}} = w_{\hat{y}} - \phi(x) \tag{5}$$

Averaged weights are calculated as:

$$\bar{w} = \frac{1}{T} \sum_{t=1}^{T} w^{(t)} \tag{6}$$

where T is the total number of updates.

Implementation Details

AveragedPerceptron Class

This class implements the core algorithm:

```
class AveragedPerceptron:
    def __init__(self):
        self.weights = defaultdict(lambda: defaultdict(float))
        self.totals = defaultdict(lambda: defaultdict(float))
        self.timestamps = defaultdict(lambda: defaultdict(int))
        self.i = 0 # Global update counter

    def predict(self, features):
        scores = defaultdict(float)
        for feat, value in features.items():
            for label, weight in self.weights[feat].items():
                scores[label] += value * weight
        return max(scores, key=scores.get) if scores else 'O'

    def update(self, truth, guess, features):
        self.i += 1
        if truth == guess:
            return
        for feat, value in features.items():
            self._update_feat(feat, truth, value)
            self._update_feat(feat, guess, -value)

    def _update_feat(self, feat, label, value):
        weight = self.weights[feat][label]
        self.totals[feat][label] += (self.i - self.timestamps[feat][label]) * weight
        self.timestamps[feat][label] = self.i
        self.weights[feat][label] += value

    def average_weights(self):
        for feat, weights in self.weights.items():
            for label in weights:
                total = self.totals[feat][label]
                total += (self.i - self.timestamps[feat][label]) * weights[label]
                averaged = total / self.i
                if averaged:
                    self.weights[feat][label] = averaged
                else:
                    del self.weights[feat][label]
```

Feature Extraction

extract_features(sentence, i) Generates a dictionary of features for the word at position i. These features include:

- Current word
- Previous and next word
- Prefix and suffix
- Capitalization, digit, title-case status
- Word shape (e.g., capitalization pattern)
- Presence of hyphens
- Presence of digits
- Presence of punctuation
- Special tokens for beginning and end of sentence (BOS, EOS)

```
def extract_features(sentence, index):
    word = sentence[index]
    features = {
        'bias': 1.0,
        'word.lower=' + word.lower(): 1.0,
        'word[-3:]=' + word[-3:]: 1.0,
        'word[-2:]=' + word[-2:]: 1.0,
        'word.isupper=' + str(word.isupper()): 1.0,
        'word.istitle=' + str(word.istitle()): 1.0,
        'word.isdigit=' + str(word.isdigit()): 1.0,
        'word.shape=' + word_shape(word): 1.0,
        'word.has_hyphen=' + str('-' in word): 1.0,
        'word.has_digit=' + str(any(char.isdigit() for char in word)): 1.0,
        'word.has_punct=' + str(any(char in string.punctuation for char in word)): 1.0,
    }

    if index > 0:
        prev = sentence[index - 1]
        features.update({
            '-1:word.lower=' + prev.lower(): 1.0,
            '-1:word.istitle=' + str(prev.istitle()): 1.0,
            '-1:word.shape=' + word_shape(prev): 1.0,
        })
    else:
        features['BOS'] = 1.0

    if index < len(sentence) - 1:
        next = sentence[index + 1]
        features.update({
            '+1:word.lower=' + next.lower(): 1.0,
            '+1:word.istitle=' + str(next.istitle()): 1.0,
            '+1:word.shape=' + word_shape(next): 1.0,
        })
    else:
        features['EOS'] = 1.0

    return features
```

These features provide context-aware information for sequence tagging.

Training Procedure

train_perceptron(dataset, dev_data, gold_dev_tags, epochs=20) Trains the model over multiple epochs and optionally evaluates on development data each time.

```
def train_perceptron(dataset, dev_data, gold_dev_tags, epochs=20):
    model = AveragedPerceptron()
    for epoch in range(epochs):
        for sentence, tags in dataset:
            for i in range(len(sentence)):
                features = extract_features(sentence, i)
                pred = model.predict(features)
                model.update(tags[i], pred, features)
        model.average_weights()
    return model
```

Precision, recall and F scores of Averaged Perceptron for Dev Set

```
#Entity in gold data: 13179
#Entity in prediction: 13646

#Correct Entity : 12155
Entity  precision: 0.8907
Entity  recall: 0.9223
Entity  F: 0.9062

#Correct Sentiment : 11884
Sentiment  precision: 0.8709
Sentiment  recall: 0.9017
Sentiment  F: 0.8860
```

The Averaged Perceptron performs better than Viterbi in all metrics for entity classification. The higher precision, recall, and F-score suggest that the Averaged Perceptron is more accurate at both identifying the correct entities and recovering all relevant entities from the data. Specifically, the Averaged Perceptron shows a higher recall (0.9223 vs. 0.8418), which means it is better at identifying all instances of entities, even when they are harder to detect. Additionally, the precision is also higher, indicating that it makes fewer false positive errors compared to the Viterbi algorithm.

Similarly, for sentiment classification, the Averaged Perceptron outperforms Viterbi. The higher F-score of the Averaged Perceptron (0.8860 vs. 0.8006) indicates that it achieves a better balance between precision and recall, making it a more reliable model overall for sentiment analysis. Its higher recall value (0.9017 vs. 0.8037) shows that it is more effective at capturing all relevant sentiment instances, while the precision (0.8709 vs. 0.7976) demonstrates it is also more accurate in its predictions.

Precision, Recall, and F Scores of Averaged Perceptron for Test Set

Since we do not have the gold standard for the test set, we are unable to evaluate the Precision, Recall, and F-scores. However, we can examine the output and assess if it makes sense. Upon review, the output appears to be largely consistent with expectations.

The output for the test set is as follows:

```
Kary B-NP
Moss I-NP
, O
an B-NP
attorney I-NP
with B-PP
the B-NP
ACLU I-NP
's B-NP
Women I-NP
's B-NP
Rights I-NP
Project I-NP
, O
said B-VP
, O
`` O
They B-NP
wanted B-VP
a B-NP
svelte-looking I-NP
woman I-NP
, O
and O
a B-NP
pregnant I-NP
woman I-NP
is B-VP
not O
svelte I-NP
. O
```

In the output, the predicted labels appear to be reasonable, with entities like "Kary", "Moss", "attorney", and "ACLU" being correctly identified and classified under the appropriate entity types (e.g., 'B-NP' for noun phrases). The use of 'O' indicates non-entity words, which is also consistent with expectations. While we cannot provide precise metrics without a gold standard, the logical consistency of the labels suggests that the Averaged Perceptron model is performing well on this task.

Evaluation

Evaluation is based on **chunk-level F1 score** using `EvalScript/evalResult.py`. A predicted chunk is correct if its label and span match the gold standard. Metrics computed:

- **Precision** = Correct chunks / Predicted chunks
- **Recall** = Correct chunks / Gold chunks
- **F Score** = 2 / (1/Precision + 1/Recall)