CS 5740 - Project 3 Report

Philip Su (ps845)
Kelvin Jin (kkj9)
11/13/2015

Overview

In this project, we implemented a model to tag entities in sentences with the categories *person*, *location*, *organisation* or *miscellaneous*. In order to do this, we first developed a simple baseline model to use as a basis of comparison. Then, we created a Hidden Markov Model and implemented the Viterbi algorithm with transition probabilities that are the probability of a tag given previous *n-1* tags and emission probabilities that are the probability of a tag given a word to determine the most likely sequence of tags given a test sentence.

Implementation Details

We chose to implement Hidden Markov Models in our approach of sequence tagging. No existing packages or external tools were used in this project - everything in our project was implemented from scratch.

Baseline Model

```
class BaselineNER:
    """
    Simple naive Baseline Named Entity Recognition system, memorizes training data
    and classifies test data based on training data
    """
    # Mapping of first words to list of entities and tags
    named_entities = {}
    all_types = ["PER", "LOC", "ORG", "MISC"]
```

Figure 1. Information stored in the BaselineNER class, contains a map of all seen words during train

The baseline model implemented was a simple memorizer. When the baseline model was trained, it memorized the first word of all named entities and their associated tags, and stored these values in a map called *named_entities* as seen in Figure 1. In testing, the model selected the most often seen tag associated with a word. Because named entities may span multiple tokens, the baseline model greedily matches every word in a test sentence with the first word of named entities seen in training data, as seen in Figure 2.

```
# Check if first word has been seen
if entity_word in self.named_entities:
    for data in self.named_entities[entity_word]:
        entities, type = data
        # See if any before seen named-entities match this
        if self.matches(entities.split(), tokens[entity_index:]):
            if indices:
                interval = Interval(int(indices[entity_index]),
                                    int(indices[entity_index]) + len(entities.split()) - 1)
            else:
                interval = Interval(entity_index, entity_index + len(entities.split()) - 1)
            result[type].append(interval)
```

Figure 2. Baseline NER utilizes a greedy approach to classify sentences in test data

Hidden Markov Model

Our Hidden Markov Model (HMM) utilized the Viterbi algorithm to determine the most likely sequence of tags for a sentence. The HMM class used 2 data structures - a map **words_labels_counts** of words to labels and counts and an **Ngram** object of tags. The map **words_labels_counts** allows us to generate the emission probabilities *P(word | tag)* as shown in Figure 3 and the **Ngram** object allows us to use different orders of n-grams for transition probabilities *P(tag | n-1 previous tags)* as shown in Figure 4.

```
class HMM:
    """
    A Hidden Markov Model that utilizes the Viterbi algorithm to determine the most likely
    sequence of tags for a Named Entity Recognition task. Tunable hyperparameters include:
        - Unknown token threshold
        - Smoothing amount
        - n value for N-gram model
    """
    # Mapping of each labels to a map of word : count
    words_labels_counts = {}
```

Figure 3.HMM stores a map of words to labels and counts.

```
class Ngram:
    """
    A class representing Ngrams of labels. This class has a recursive implementation to allow for a
    varying n-value of ngrams. Each Ngram objects has mappings to other Ngram objects, n-1 number of
    times until n = 1 in which case the object contains a mapping to counts. The Ngram probability
    can be calculated by taking the nth Ngram object count dividing by the n-1th Ngram object.
    """
    def __init__(self, label):
        self.label = label
        # Mapping of label to Ngram object
        self.next_grams = {}
        self.count = 0
```

Figure 4.Ngram object is used to allow various orders of Ngrams.

After training, the HMM class uses the Viterbi algorithm to determine the most likely sequence of tags given a word sequence. The Viterbi algorithm consists of 3 steps: **Initialization**, **Iteration** and finally, **Identification** as shown in Figure 5. The **Initialization** step calculates initial probabilities for all states using only the lexical generation probabilities *P(word | tag)*. The probability values are stored in **scores**, a list of iteration scores, where each score element contains a map of the probability of a label to its score at that iteration. The **Iteration** step iterates through every word in the word sequence and uses the max score from the previous iteration, the lexical generation probabilities and the Ngram label probabilities *P(tag | previous n-1 tags)* to determine which label gives the highest score at the current iteration. The most likely label is added to the path for each iteration. Finally, the **Identification** step chooses the highest score for the last iteration and identifies the sequence of labels, the path, that resulted in this score. The implementation details of the **Initialization**, **Iteration** and **Identification** steps in the Viterbi algorithm are shown in Figures 6, 7, 8, respectively.

```python
def viterbi(self, word_seq):
    """
    Runs the Viterbi algorithm for a given word sequence to determine the most probable
    sequence of tags.

    :param word_seq: to determine tag for
    :return: list of most likely sequence of tags
    """
    # Initialize scores
    scores = [{}]
    path = {}
    # Populate scores
    for i in range(0, len(word_seq)):
        for label in self.label_type_map:
            scores[i][label] = 0
        scores.append({})
    self.initialize(scores, word_seq, path)
    path = self.iterate(scores, word_seq, path)
    return self.identify(scores, word_seq, path)
```

Figure 5. High level view of the implemented Viterbi algorithm - Initialization, Iteration and Identification

```python
def initialize(self, scores, word_seq, path):
    """
    Initialize the probabilities of tags for a word sequence.

    :param scores: most likely probabilities for a word sequence at iteration 0
    :param word_seq: sequence of words to determine tags for
    :param path: the current paths
    """
    for label in self.label_type_map:
        label_prob = self.get_ngram_prob([label])
        lexical_generation_prob = self.get_lexical_generation_prob(word_seq[0], label)
        scores[0][label] = label_prob * lexical_generation_prob
        path[label] = [label]
```

Figure 6. Initialization of Viterbi algorithm - assigns scores for iteration 0

```python
def iterate(self, scores, word_seq, path):
    """
    Iterate through all tags to determine most likely sequence of tags using the Viterbi
    algorithm

    :param scores: most likely probabilities for a word sequence at each iteration
    :param word_seq: sequence of words to determine tags for
    :param path: the current paths
    :return: the most likely paths for each tag
    """
    for i in range(1, len(word_seq)):
        new_path = {}
        for label in self.label_type_map:
            prev_labels = path[label][:]
            # Conditioned on N previous terms
            if len(prev_labels) > self.N_VALUE:
                prev_labels = prev_labels[-self.N_VALUE+1:]
            # Emission probability: P(w | t)
            lexical_generation_prob = self.get_lexical_generation_prob(word_seq[i], label)
            prob, state = float('-inf'), None
            # Iterate and find next state that is most probable
            for label0 in self.label_type_map:
                # Transition probability P(t | t-1, t-2...t-n)
                curr_label = prev_labels[:]
                curr_label.append(label0)
                label_prob = self.get_ngram_prob(curr_label)
                curr_prob, curr_state = scores[i-1][label0] * label_prob * lexical_generation_prob, label0
                # Update if a better state is found
                if curr_prob > prob:
                    prob, state = curr_prob, curr_state
            scores[i][label] = prob
            new_path[label] = path[state] + [label]
        path = new_path
    return path
```

Figure 7. Iteration of Viterbi algorithm - iterates through every word and determines most likely path for each iteration

```
def identify(self, scores, word_seq, path):
    """
    Identify the most likely path of tags of a given word sequence after running the
    Viterbi algorithm.

    :param scores: the final scores of the probability of the paths
    :param word_seq: sequence of words to determine tags for
    :param path: the final paths of most likely sequences of tags
    :return: the most likely sequence of tags.
    """
    n = len(word_seq) - 1
    (prob, state) = max((scores[n][label], label) for label in self.label_type_map)
    return path[state]
```

Figure 8. Identification of Viterbi algorithm - identifies the most likely path for given word sequence

Preprocessing

The data was preprocessed in 2 steps. Firstly, we ran a script, *extract_train_validation.py,* to partition the training data into *train_partial.txt* and *validation_partial.txt* in a 1:4 ratio. Additionally, we noticed that the data contained named entities that had different cases (e.g. “TORONTO” and “Toronto” were both named entities with same labels), so we made our model case insensitive by converting all letters to lowercase to reduce the sparsity of the data.

Experiments

We trained our model on *train_partial.txt* and calculated the F-Score on *validation_partial.txt* with different parameters. Our goal was to find the best parameters to use in our HMM model to achieve the highest accuracy for the named entity recognition task. The 3 parameters we changed were the **order** of Ngrams for label sequences, the **Smoothing Constant** for additive smoothing, and the **Unknown Threshold** of word count that would be categorized as “unknown” tokens. We hypothesized that bigrams (N=2), a little bit of smoothing (around 0.1) and a medium unknown token count threshold (roughly around 3) would give us the most accurate model. To test our hypothesis, we compared the F-Scores of various values for **N**, **Smoothing Constant** and **Unknown Threshold**, changing only 1 parameter with each successive run. The results are summarized in Table 1 below.

Results

Parameters	**Accuracy**
N = 1; Smoothing Constant = 1, Unknown Threshold = 0	Precision: 56.3% Recall: 76.7% F-score: 65.0%
N = 12; Smoothing Constant = 1, Unknown Threshold = 0	Precision: 52.2% Recall: 31.9% F-score: 39.6%

N = 3; Smoothing Constant = 1, Unknown Threshold = 0	Precision: 49.4% Recall: 27.1% F-score: 35.0%
N = 3; Smoothing Constant =1, Unknown Threshold = 3	Precision: 1.7% Recall: 7.9% F-score: 2.8%
N = 3; Smoothing Constant =1, Unknown Threshold = 5	Precision: 1.8% Recall: 7.7% F-score: 2.9%
N = 1; Smoothing Constant = 0.1, Unknown Threshold = 0	**Precision: 56.2%** **Recall: 91.7%** **F-score: 69.7%**
N = 1; Smoothing Constant = 0.01, Unknown Threshold = 0	Precision: 50.3% Recall: 93.5% F-score: 65.4%
N = 1; Smoothing Constant = 5, Unknown Threshold = 0	Precision: 52.4% Recall: 68.7% F-score: 59.5%
N = 3, Smoothing Constant = 0.1, Unknown Threshold = 0, Generalized tags	Precision: 63.3% Recall: 72.8% F-score: 67.7%
N = 5, Smoothing Constant = 0.1, Unknown Threshold = 0, Generalized tags	Precision: 65.1% Recall: 72.2% F-score: 68.4%

Table 1. F-Score on validation set with different model parameters

The results are not entirely what we hypothesized. The parameters that gave us the most accurate model were N = 1, Smoothing Constant = 0.1, and an Unknown Threshold = 0, with an accuracy of 69.7% on the validation set.

We saw that as we increased the order of n-gram features, the accuracy went down. We attempted to generalize the tags by mapping all “B-[TAG]” and “I-[TAG]” tags to the same “[TAG]” tag, where [TAG] is a label category (e.g. PER, LOC or MISC). The last 2 runs in Table 1 show that this resulted in more accurate results when the order of n-grams is increased.

A **Smoothing Constant** of 0.1 gave us the most accurate model, which we expected in our hypothesis..

Finally, having an Unknown threshold of 0 gave us the most accurate model. In fact, we saw that as we increased the **Unknown Threshold**, the accuracy dramatically decreased We

believe this is the case because many of the words in the training corpus appeared only once with a specific tag, and by aggregating all these counts, the <UNK> unknown token would have a large count (~200 counts for a given tag). During classification, the <UNK> unknown probability would significantly larger than the probability of other words and greatly affect the categorized label.

Competition Score

Our team name was *pskj* and the best score we received was an F-Score of 0.71031, as shown in Figure 9.

Figure 9. Our best score on Kaggle achieved an F-Score of 0.71031

Extensions

We chose to implement different orders of n-gram based features for sequence tagging. This was implemented by using the recursive **Ngram** data structure shown in Figure 4. Each **Ngram** object contains a map of a label to another **Ngram** object, as well as a count. At the nth **Ngram** object, only a count of the number of times the label sequence is seen is stored. Increasing the order of n-gram based features for sequence tagging led to a higher F-Score through higher precision, but also resulted in lower recall. As seen in the last 2 rows for Table 1, running our model with N = 3 led to a F-Score of 67.7%, while running our model with N = 5 led to a F-Score of 68.4%.

We only used additive smoothing, but we ran tests with different smoothing constants to add to find the most optimal value, ranging from 0.01 to 1. We determined that a smoothing value of 0.1 gave us the most accurate model.

Individual Member Contribution

Everyone on the team contributed an equal amount. We were both involved in determining what kind of data structures to use and implemented the algorithms together. We pair programmed on several occasions.