Connected to miniconda3 (Python 3.11.5)

POS Tagger Prediction Explanation

By Luopeiwen (Tina) Yi

Table of Content

- Main Code and Output
- General Observation
- Investigation of Wrong Prediction
- Summary Report
  - General Observation
  - Code Overview
  - The Correct POS Prediction Explanation
  - The Incorrect POS Prediction Explanation

Main Code and Output

```
In [ ]: """Viterbi Algorithm for inferring the most likely sequence of states from an HMM.
        Patrick Wang, 2021
        """

        from typing import Sequence, Tuple, TypeVar
        import numpy as np
        import nltk

        # nltk.download("brown")
        # nltk.download("universal_tagset")


        def getWordTag(corpus):
            words = set()
            tags = set()
            for sentence in corpus:
                for word, tag in sentence:
                    words.add(word)
                    tags.add(tag)
            # Add OOV token to word list
            words.add("OOV")
            # Convert sets to lists for index-based operations
            word_map = list(words)
            tag_map = list(tags)
            return word_map, tag_map


        def genMatrix(corpus, word_map, tag_map):
            num_tags = len(tag_map)
            num_words = len(word_map)
            # Create dictionaries for efficient index lookup
            word_to_idx = {word: idx for idx, word in enumerate(word_map)}
            tag_to_idx = {tag: idx for idx, tag in enumerate(tag_map)}
            # 0. Initial State Matrix (initState)
            initState = np.ones(num_tags)  # Laplace Smoothing
            for sentence in corpus:
                _, tag = sentence[0]  # consider only the first word of each sentence
                initState[tag_to_idx[tag]] += 1
            initState /= initState.sum()  # Normalize
            # 2. Transition Matrix
            matState = np.ones((num_tags, num_tags))  # Laplace Smoothing
            for sentence in corpus:
                for i in range(len(sentence) - 1):
                    curr_tag_idx = tag_to_idx[sentence[i][1]]
                    next_tag_idx = tag_to_idx[sentence[i + 1][1]]
                    matState[curr_tag_idx][next_tag_idx] += 1
            # Normalize each row
            matState /= matState.sum(axis=1, keepdims=True)
            # 3. Observation Matrix
            matObs = np.ones((num_tags, num_words))  # Laplace Smoothing
            for sentence in corpus:
                for word, tag in sentence:
                    matObs[tag_to_idx[tag]][word_to_idx[word]] += 1
            # Normalize each row
            matObs /= matObs.sum(axis=1, keepdims=True)
            return matState, matObs, initState


        # Example Usage:

        training = nltk.corpus.brown.tagged_sents(tagset="universal")[:10000]
        word_map, tag_map = getWordTag(training)
        matState, matObs, initState = genMatrix(training, word_map, tag_map)
        # print("Initial State (initState):", initState)
        # print("Transition Matrix (matState):", matState)
        # print("Observation Matrix (matObs):", matObs)
        # Q represents the states
        # V represents the observations

        Q = TypeVar("Q")
        V = TypeVar("V")
        # obs: A sequence of observed values (usually word indices).
        # pi: Initial state probabilities.
        # A: State transition probabilities.
        # B: Emission probabilities.
        # It returns a tuple containing the most probable sequence of states (qs) and its probability.


        def viterbi(
            obs: Sequence[int],
            pi: np.ndarray[Tuple[Q], np.dtype[np.float_]],
            A: np.ndarray[Tuple[Q, Q], np.dtype[np.float_]],
            B: np.ndarray[Tuple[Q, V], np.dtype[np.float_]],
        ) -> Tuple[Sequence[Q], np.float_]:
            """Infer most likely state sequence using the Viterbi algorithm.
            Args:
                obs: An iterable of ints representing observations.
                pi: A 1D numpy array of floats representing initial state probabilities.
                A: A 2D numpy array of floats representing state transition probabilities.
                B: A 2D numpy array of floats representing emission probabilities.
            Returns:
                A tuple of:
                * A 1D numpy array of ints representing the most likely state sequence.
                * A float representing the probability of the most likely state sequence.
            """
            # N: The number of observations.
            # Q: The number of states.
            # V: The number of unique observations.
            N = len(obs)
            Q, V = B.shape  # num_states, num_observations
            # d_t[i] = max prob of being in state i at step t
            #  AKA viterbi
            # ipsi_t[i] = most likely state preceeding state i at step t
            #  AKA backpointer
            # initialization
            log_d = [np.log(pi) + np.log(B[:, obs[0]])]
            log_psi = [np.zeros(Q,)]
            # recursion
            for t in obs[1:]:
                log_da = np.expand_dims(log_d[-1], axis=1) + np.log(A)
                log_d.append(np.max(log_da, axis=0) + np.log(B[:, t]))
                log_psi.append(np.argmax(log_da, axis=0))
            # termination
            log_p = np.max(log_d[-1])
            qs = [-1] * N
            qs[-1] = int(np.argmax(log_d[-1]))
            for i in range(N - 2, -1, -1):
                qs[i] = log_psi[i + 1][qs[i + 1]]
            return qs, np.exp(log_p)


        def words_to_indices(words, word_map):
            """Convert a list of words to their respective indices."""
            return [
                word_map.index(word) if word in word_map else word_map.index("OOV")
                for word in words
            ]


        # Extracting test data

        test_data = nltk.corpus.brown.tagged_sents(tagset="universal")[10150:10153]
        # Function to compute accuracy


        def accuracy(predicted, actual):
            return sum(p == a for p, a in zip(predicted, actual)) / len(predicted)


        total_correct_tags = 0
        total_tags = 0
        # Looping over each sentence in the test data

        for test_sentence in test_data:
            words, _ = zip(
                *word for word, _ in test_sentence
            )  # Extracting words from the (word, tag) tuples
            obs_indices = words_to_indices(words, word_map) # Convert words to indices
            state_sequence, _ = viterbi(obs_indices, initState, matState, matObs)
            predicted_tags = [tag_map[idx] for idx in state_sequence]
            actual_tags = [tag for _, tag in test_sentence]
            total_correct_tags += sum(p == a for p, a in zip(predicted_tags, actual_tags))
            total_tags += len(actual_tags)
            acc = accuracy(predicted_tags, actual_tags)
            print("Test sentences:", words)
            print("Predicted tags:", predicted_tags)
            print("Actual tags:", actual_tags)
            print("Accuracy:", acc)
            print("Accuracy in Percentage:", round(acc * 100, 2), "%")
            print("\n" + "=" * 50 + "\n")

        overall_accuracy = total_correct_tags / total_tags
        print("Overall Accuracy for the three sentences:{overall_accuracy}")
        print(
            f"Overall Accuracy for the three sentences in percentage: {overall_accuracy * 100:.2f}%"
        )

Test sentence: ['The', 'coming', 'from', 'other', 'denominations', 'will', 'welcome', 'the', 'opportunity', 'to', 'become', 'informed', ',']
Predicted tags: ['DET', 'NOUN', 'ADP', 'ADJ', 'NOUN', 'VERB', 'VERB', 'DET', 'NOUN', 'PRT', 'VERB', 'VERB', ',']
Actual tags: ['DET', 'VERB', 'ADP', 'ADJ', 'NOUN', 'VERB', 'VERB', 'DET', 'NOUN', 'PRT', 'VERB', 'VERB', ',']
Accuracy: 0.9230769230769231
Accuracy in Percentage: 92.31 %

==================================================

Test sentence: ['The', 'preparatory', 'class', 'is', 'an', 'introductory', 'face-to-face', 'group', 'in', 'which', 'new', 'members', 'become', 'acquainted', 'with', 'on
e', 'another', '.']
Predicted tags: ['DET', 'ADJ', 'NOUN', 'VERB', 'DET', 'ADJ', 'NOUN', 'NOUN', 'ADP', 'DET', 'ADJ', 'NOUN', 'VERB', 'VERB', 'ADP', 'NUM', 'NOUN', '.']
Actual tags: ['DET', 'ADJ', 'NOUN', 'VERB', 'DET', 'ADJ', 'ADJ', 'NOUN', 'ADP', 'DET', 'ADJ', 'NOUN', 'VERB', 'VERB', 'ADP', 'NUM', 'DET', '.']
Accuracy: 0.8888888888888888
Accuracy in Percentage: 88.89 %

==================================================

Test sentence: ['It', 'provides', 'a', 'natural', 'transition', 'into', 'the', 'life', 'of', 'the', 'local', 'church', 'and', 'its', 'organizations', '.']
Predicted tags: ['PRON', 'VERB', 'DET', 'ADJ', 'NOUN', 'ADP', 'DET', 'NOUN', 'ADP', 'DET', 'ADJ', 'NOUN', 'CONJ', 'DET', 'NOUN', '.']
Actual tags: ['PRON', 'VERB', 'DET', 'ADJ', 'NOUN', 'ADP', 'DET', 'NOUN', 'ADP', 'DET', 'ADJ', 'NOUN', 'CONJ', 'DET', 'NOUN', '.']
Accuracy: 1.0
Accuracy in Percentage: 100.0 %

==================================================

Overall Accuracy for the three sentences: 0.9361702127659575
Overall Accuracy for the three sentences in percentage: 93.62%
```

General Observation

- Overall, my model's POS tagger prediction achieve a 93.62% accuracy.
- The accuracy of the first sentence POS tagger prediction is around 92.31 %. In the first sentence, the word "coming" was predicted as a 'NOUN' but the actual tag is 'VERB'.
- The accuracy of the second sentence POS tagger prediction is around 88.89 %. In the second sentence, the word "face-to-face" was predicted as a 'NOUN', but the actual tag is 'ADJ'. Additionally, "another" was predicted as 'NOUN', but the actual tag is 'DET'.
- The accuracy of the first sentence POS tagger prediction is 100%. In the third sentence, the tags predicted by the model match perfectly with the actual tags.

Investigation of Wrong Prediction

```
In [ ]: # Check whether the words "coming", "face-to-face", and "another" are present in the training data and check their word tags
        # Words to check
        words_to_check = ["coming", "face-to-face", "another"]

        # Create a dictionary to store tags for each word
        word_tags = {word: set() for word in words_to_check}

        # Loop through the training data to gather tags
        for sentence in training:
            for word, tag in sentence:
                if word in words_to_check:
                    word_tags[word].add(tag)

        # Display results
        for word, tags in word_tags.items():
            if tags:
                print(
                    f"The word '{word}' appears in the training data with the following tags: {', '.join(tags)}."
                )
            else:
                print(
                    f"The word '{word}' does not appear in the training data. It is an OOV token."
                )

The word 'coming' appears in the training data with the following tags: VERB, NOUN.
The word 'face-to-face' does not appear in the training data. It is an OOV token.
The word 'another' appears in the training data with the following tags: DET.
```

Summary Report

General Observation:

- The overall accuracy of the POS tagger is 93.62%. This indicates that in the majority of cases, the model correctly predicted the POS tags.
- This is a very good result considering perfect accuracy in POS tagging is challenging due to the intricacies of human language and the limitations of the training set.
- A more detailed explanation of why my POS tagger does or does not produce the correct tags is provided below.

Code Overview:

1. Word and Tag Mapping:

Function: `getWordTag`

This function extracts all distinct words and tags from the corpus. To accommodate words that didn't appear during training, I included an "OOV" (Out-Of-Vocabulary) token.

2. Generating Matrices for HMM:

Function: `genMatrix`

Three matrices play pivotal roles in the Hidden Markov Model:

- `initState`: Represents the initial probabilities for each tag at the beginning of a sentence.
- `matState`: Acts as the transition matrix, indicating the probabilities of transitioning from one tag to the next.
- `matObs`: As the observation matrix, it signifies the likelihood of observing a specific word given a particular tag.

(1). Initial State Distribution Matrix:

- **Objective**: Denote the probabilities for each tag's appearance at a sentence's start.
- **Procedure**:
  1. Generate a 1xN matrix (N being the total unique tags).
  2. Apply Laplace Smoothing by setting all matrix values to 1.
  3. Loop through the corpus. For each first word in a sentence, increment the count of its associated tag.
  4. Normalize the counts by dividing them by the total sum, ensuring the sum of probabilities equals 1.

(2). Transition Matrix:

- **Objective**: Denote the probabilities of transitioning from one tag to another.
- **Procedure**:
  1. Construct an NxN matrix based on the unique tag count (N being the total unique tags).
  2. Apply Laplace Smoothing by setting all matrix values to 1.
  3. Loop through the corpus. For every pair of consecutive tags, increment the count in the corresponding cell in the matrix.
  4. Normalize rows by dividing values by their row's sum, ensuring the sum of probabilities at 1 for each row.

(3). Observation Matrix:

- **Objective**: Denote the probabilities of a word being emitted under a given tag.
- **Procedure**:
  1. Frame an NxM matrix (N being the total unique tag count, M being the total unique word count).
  2. Apply Laplace Smoothing by setting all matrix values to 1.
  3. Loop through the corpus. For every word-tag pair, increment the corresponding cell in the matrix.
  4. Normalize rows by dividing values by their row's sum, ensuring the sum of probabilities at 1 for each row.

3. Viterbi Algorithm:

Function: `viterbi`

For a provided word sequence (observations), this algorithm determines the most probable sequence of POS tags (states) leveraging the aforementioned matrices. It employs dynamic programming to ascertain the likelihood of each word being in a specific state, factoring in the previous state. To deduce the optimal state sequence (tags) for a word sequence, backtracking is utilized.

4. Testing and Evaluating Accuracy:

The code subjects each test sentence to the following procedure:

1. Words undergo a conversion to their respective indices as per `word_map`. Unfamiliar words get substituted with the "OOV" index.
2. Utilizing the Viterbi algorithm on the indexed sentence yields the most probable tag sequence.
3. For accuracy computation, the system contrasts the predicted tags against the actual ones.

The Correct POS Prediction Explanation

My code used a Hidden Markov Model (HMM) for POS tagging. Utilizing probability matrices derived from training data and the Viterbi algorithm, it determines the likeliest tags for word sequences. The model is considered to be successful because of the diversity of the training data, and the correct implementation of the matrix-generation methods and Viterbi algorithm.

The Incorrect POS Prediction Explanation

1. **First Sentence:**
   - Accuracy is approximately 92.31%.
   - The word "coming" was predicted as a 'NOUN' but was actually a 'VERB'.
   - **Analysis:** Upon checking the training data, it was found that the word "coming" has been tagged both as 'VERB' and 'NOUN'. In some contexts, "coming" might be used as a noun (e.g., "The coming of winter"), while in others, it's a verb (e.g., "She is coming to the party"). The model seems to have been influenced by the dual nature of this word in the training data, leading to a potential misclassification in this specific context.
2. **Second Sentence:**
   - Accuracy is around 88.89%.
   - The word "face-to-face" was predicted as 'NOUN', but the actual tag is 'ADJ'.
   - The word "another" was predicted as 'NOUN', but the actual tag is 'DET'.
   - **Analysis:** The word "face-to-face" was not found in the training data. This implies it's an OOV (Out-Of-Vocabulary) word for the model. Hence, any prediction for this word would be based on the model's generalizations and might not be accurate. For "another," the training data showed it has been tagged as 'DET'; but the model incorrectly predicted it as a 'NOUN'. This could be attributed to the model's bias towards more frequent patterns or contexts in the training data where similar words appear as nouns.
3. **Third Sentence:**
   - Accuracy is 100%.
   - **Analysis:** This suggests that the words in this sentence and their respective contexts were well-represented or resembled patterns seen in the training data. The model made accurate predictions in this instance.

Conclusion:

The model's errors can be attributed to three main factors:

1. Ambiguity of certain words in the training data (words can serve different roles based on context), as seen with the word "coming".
2. Model's bias towards more frequent patterns or contexts in the training data where similar words appear as a different tag, as seen with the word "another".
3. Absence of certain words from the training data, leading to possibly inaccurate generalizations, as observed with "face-to-face".