hw2-csci544

September 26, 2023

1 CSCI 544 Natural Language Processing

1.1 Krushang Satani

1.1.1 Homework 2

Due Date	Assignment Type	Submission Format
September 26, 2023	Hands-on Assignment	Jupyter Notebook (.ipynb)

This assignment focuses on using Hidden Markov Models (HMMs) for part-of-speech tagging. The task involves building an HMM model using the Wall Street Journal section of the Penn Treebank. In the 'data' folder, you will find three files: 'train', 'dev', and 'test'. 'Train' and 'dev' contain sentences with human-annotated part-of-speech tags, while 'test' provides raw sentences for part-of-speech tagging prediction.

Assignment Overview: - Task 1: Vocabulary Creation - Task 2: Hidden Markov Model (HMM) Implementation - Task 3/4: Part-of-Speech Tagging Prediction on 'test' data

1.2 Task 1: Vocabulary Creation (20 points)

[1]:
```
import json
```

[2]:
```
with open('data/train.json') as f:
    train_data = json.load(f)
```

1.3 Converting All Words to Lowercase for Training

In the provided code, all words in the input sentences are converted to lowercase for processing. This preprocessing step serves several important purposes:

1. **Normalization**: Converting all words to lowercase ensures uniformity in the text data. It helps treat words like "Word" and "word" as the same, preventing the model from treating them differently due to case differences.

1.4 Data Preprocessing: Word and Tag Frequency Analysis

In this section, we perform an analysis of word and tag frequencies in the training data to gain insights into the dataset. We use the `Counter` class from the `collections` module to calculate word and tag frequencies.

1.4.1 Word Frequency Analysis

We calculate the total number of words in the training data and determine the total number of unique words. This analysis helps us understand the vocabulary size.

1.4.2 Tag Frequency Analysis

We also analyze the frequency of tags in the training data. This is essential for understanding the distribution of tags in the dataset.

Let's start by analyzing the data.

[3]:
```
from collections import Counter

word_freq = Counter()
tag_freq = Counter()
count = 0
number_of_sentences = len(train_data)

for record in train_data:

    sentence = record['sentence']
    label = record['labels']

    for word in sentence:
        count += 1
        word_freq[word.lower()] += 1

    for tag in label:
        tag_freq[tag] += 1

total_unique_words = len(word_freq)
tag_labels = [i for i in tag_freq]

print(f'total words in the train data: {count}')
print(f'total unique words in the train data: {total_unique_words}')
print(f'Total number of sentences are: {number_of_sentences}')
```

```
total words in the train data: 912095
total unique words in the train data: 38558
Total number of sentences are: 38218
```

1.5 assigning thresold value = 3

1.6 Handling Low-Frequency Words with <unk>

In this section, we address the issue of low-frequency words in the training data by merging them into a special <unk> token. Words that appear with a frequency less than or equal to a threshold (in this case, 3) are considered low-frequency.

1.6.1 Word Frequency Merging

We iterate through the word frequencies and identify words with counts below the threshold. These words are then merged into a single `<unk>` token, while retaining the counts for statistical purposes.

This step is essential for improving the robustness of the NLP model and reducing the vocabulary size by handling infrequent words.

Let's proceed with the merging process.

[4]:
```
threshold = 2
merged_word_freq = {}
unk_count = 0

for word, count in word_freq.items():
    if count <= threshold:
        unk_count += count
    else:
        merged_word_freq[word] = count

merged_word_freq['< unk >'] = unk_count
print(f'Total unique words after merging <unk>: {len(merged_word_freq)}')
```

```
Total unique words after merging <unk>: 15568
```

[5]:
```
sorted_merged_word_freq = sorted(merged_word_freq.items(), key=lambda x: x[1],␣
↪reverse=True)
print(f'count of unknown words are: {unk_count}')
```

```
count of unknown words are: 28581
```

1.7 Saving Vocabulary to File

In this section, we save the vocabulary to a text file named `vocab.txt`. The vocabulary includes both the `< unk >` token and the sorted merged word frequencies.

1.7.1 Vocabulary Structure

- `< unk >`: A special token representing low-frequency words.
- Other words: Sorted by frequency, with unique identifiers (indices) and their respective counts.

Saving the vocabulary to a file is crucial for reference and future use in NLP tasks.

Let's proceed with saving the vocabulary.

[6]:
```
with open('vocab.txt', 'w') as vocab_file:
    vocab_file.write(f'< unk >\t0\t{unk_count}\n')

    for index, (word, freq) in enumerate(sorted_merged_word_freq):
        if word == '< unk >':
```

```
            continue
        vocab_file.write(f'{word}\t{index + 1}\t{freq}\n')
```

[7]:
```
print(f'Threshold value for identifying unknown words: {threshold}')
print(f'Overall size of the vocabulary: {len(sorted_merged_word_freq)}')
print(f'Occurrences of "<unk>" following the replacement process: {unk_count}')
```

```
Threshold value for identifying unknown words: 2
Overall size of the vocabulary: 15568
Occurrences of "<unk>" following the replacement process: 28581
```

1.8 Task 2: Model Learning (20 points)

[8]:
```
def dictadd(dict,item):
    if(item not in dict):
        dict[item] = 1
    else:
        dict[item] += 1
    return dict
```

1.9 HMM Probability Calculation

In this section, we calculate Hidden Markov Model (HMM) probabilities:

- Transition probabilities: likelihood of tag transitions.
- Emission probabilities: likelihood of word emissions.
- Initial probabilities: distribution of initial tags in sentences.

These probabilities are crucial for part-of-speech tagging.

Let's calculate these probabilities.

[9]:
```
transitions = {}
emissions = {}
initials = {}

for record in train_data:
    sentence = record['sentence']
    labels = record['labels']

    for i in range(len(sentence)-1):
        word1, word2 = sentence[i].lower(), sentence[i+1].lower()
        label1, label2 = labels[i],labels[i+1]

        if(word1 not in merged_word_freq): word1 = '< unk >'
        if(word2 not in merged_word_freq): word2 = '< unk >'

        if(i==0):
            initials = dictadd(initials,label1)
```

```
        transitions = dictadd(transitions, (label1,label2))
        emissions = dictadd(emissions,(label1,word1))

    emissions = dictadd(emissions,(label2,word2))

# print(transitions)
# print(emissions)
# print(initials)
```

1.10 Calculating Transition and Emission Probabilities

In this section, we compute the transition and emission probabilities as well as the initial probabilities for the Hidden Markov Model (HMM). These probabilities are essential for training and using the HMM in part-of-speech tagging.

1.10.1 Transition Probabilities

- Transition probabilities represent the likelihood of transitioning from one part-of-speech tag to another.
- We calculate these probabilities based on tag sequences observed in the training data.

1.10.2 Emission Probabilities

- Emission probabilities capture the probability of observing a specific word given a particular tag.
- We calculate these probabilities by counting the occurrences of words associated with their tags.

1.10.3 Initial Probabilities

- Initial probabilities define the probability distribution of the first tag in a sentence.
- We determine these probabilities based on the frequency of the initial tags in the training data.

Let's proceed with computing these probabilities.

[10]:
```
transition_probability = {}
emission_probability = {}
initial_probability = {}

for pair in transitions:

    s1,s2 = pair
    if((s1,s2) not in transition_probability):
        transition_probability[(s1,s2)] = transitions[(s1,s2)] / tag_freq[s1]

for pair in emissions:
```

```
    s1,x1 = pair
    if((s1,x1) not in emission_probability):
        emission_probability[(s1,x1)] = emissions[(s1,x1)] / tag_freq[s1]

for ini in initials:

    if(ini not in initial_probability):
        initial_probability[ini] = initials[ini] / number_of_sentences


# print(transition_probability)
# print(emission_probability)
# print(initial_probability)
```

1.11 Saving HMM Model to JSON

In this section, we convert the Hidden Markov Model (HMM) parameters, including transition and emission probabilities, into a JSON format. The model is saved to a file named `hmm.json`.

1.11.1 Model Parameters

- Transition probabilities represent tag transition likelihoods.
- Emission probabilities capture word-tag likelihoods.

These parameters are essential for HMM-based part-of-speech tagging.

Let's save the HMM model and count the number of parameters.

[11]:
```
transition_probability_str = {str(k): v for k, v in transition_probability.
 ↪items()}
emission_probability_str = {str(k): v for k, v in emission_probability.items()}

hmm_model = {
    "transition": transition_probability_str,
    "emission": emission_probability_str
}

with open('hmm.json', 'w') as json_file:
    json.dump(hmm_model, json_file, indent=4)

num_transition_parameters = len(transition_probability)
num_emission_parameters = len(emission_probability)

print(f'Number of transition parameters: {num_transition_parameters}')
print(f'Number of emission parameters: {num_emission_parameters}')
```

```
Number of transition parameters: 1351
Number of emission parameters: 23449
```

1.12 Creating Double Hashtable

In this section, we define a Python function `create_double_hashtable` that creates a double hashtable from input data.

1.12.1 Function Signature

"'python def create_double_hashtable(data, flag=0):

[12]:
```python
def create_double_hashtable(data, flag=0):
    double_hashtable = {}

    for key, value in data.items():
        first_key, second_key = key

        # Swap keys if flag is set to 1
        if flag == 1:
            first_key, second_key = second_key, first_key

        if first_key not in double_hashtable:
            double_hashtable[first_key] = {}
        double_hashtable[first_key][second_key] = value

    return double_hashtable
```

[13]:
```python
double_emission_probability = create_double_hashtable(emission_probability,1)
double_transition_probability = create_double_hashtable(transition_probability)
# print(double_emission_probability)
# print(double_transition_probability)
# print(initial_probability)
```

1.13 Task 3: Greedy Decoding withHMM(30 points)

[14]:
```python
def calculate_accuracy(correct_predictions, total_predictions):
    accuracy = correct_predictions / total_predictions
    return accuracy
```

1.14 Accessing Values from Nested Dictionary

In this section, we define a Python function `getfromdict` for accessing values from a dictionary, including nested dictionaries.

1.14.1 Function Signature

"'python def getfromdict(dict, key, flag=1):

[15]:
```python
def getfromdict(dict,key,flag=1):
    if(flag==1):
```

```
        if(key not in dict):
            return 0
        else:
            return dict[key]

    if(flag == 2):
        if(key[0] not in dict or key[1] not in dict[key[0]]):
            return 0
        else:
            return dict[key[0]][key[1]]
```

1.15 Predicting Part-of-Speech Tags with `Greedy Decoding`

In this section, we use the greedy decoding algorithm to predict part-of-speech tags for the development data. The algorithm selects the tag with the `highest local probability` for each word in a sentence.

1.15.1 Algorithm Description

- For each sentence in the development data:
  - We iterate through each word and calculate the local probability for each possible tag.
  - The local probability is based on emission probabilities, transition probabilities, and initial probabilities.
  - We select the tag with the highest local probability as the predicted tag for each word.
  - Predicted tags are stored for each sentence in the `dev_data_predicted_tags` dictionary.

This algorithm is a simple but effective way to perform part-of-speech tagging.

Let's proceed with predicting the tags.

[16]:
```
def greedy_decoding(sentence, tag_labels, initial_probability,␣
↪double_emission_probability, double_transition_probability,␣
↪merged_word_freq):
    predicted_tags = []
    last_predicted_tag = None

    for i, word in enumerate(sentence):
        word = word.lower()
        if word not in merged_word_freq:
            word = '< unk >'

        semi_local_prob = -1
        semi_pred_tag = None

        if i == 0:
            for tag in tag_labels:
```

```
                local_prob = getfromdict(initial_probability, tag) *␣
↪getfromdict(double_emission_probability, (word, tag), 2)
                if local_prob > semi_local_prob:
                    semi_local_prob = local_prob
                    semi_pred_tag = tag
        else:
            for tag in tag_labels:
                local_prob = getfromdict(double_emission_probability, (word,␣
↪tag), 2) * getfromdict(double_transition_probability, (last_predicted_tag,␣
↪tag), 2)

                if local_prob > semi_local_prob:
                    semi_local_prob = local_prob
                    semi_pred_tag = tag

        last_predicted_tag = semi_pred_tag
        predicted_tags.append(last_predicted_tag)

    return predicted_tags
```

1.16 Evaluating Greedy Decoding Accuracy

In this section, we evaluate the accuracy of the greedy decoding algorithm on the development data. We compare the predicted part-of-speech tags with the ground truth labels.

1.16.1 Evaluation Process

- For each sentence in the development data:
  - We compare the predicted part-of-speech tags with the ground truth labels.
  - Correct predictions are counted to calculate accuracy.

1.16.2 Accuracy Calculation

- We calculate the accuracy as the ratio of correct predictions to the total number of predictions.

Let's calculate and report the accuracy for the greedy decoding algorithm on the development data.

[17]:
```
with open('data/dev.json') as f:
    dev_data = json.load(f)

dev_data_predicted_tags = {}

for record in dev_data:
    index_value = record['index']
    predicted_tags = greedy_decoding(
        record['sentence'], tag_labels, initial_probability,␣
↪double_emission_probability, double_transition_probability, merged_word_freq)
    dev_data_predicted_tags[index_value] = predicted_tags
```

[18]:
```
correct_predictions = 0
total_predictions = 0

for record in dev_data:

    words  = record['sentence']
    labels = record['labels']

    pred_labels = dev_data_predicted_tags[record['index']]

    for i in range(len(words)):
        total_predictions += 1
        if(labels[i] == pred_labels[i]):
            correct_predictions += 1

Greedy_accuracy = calculate_accuracy(correct_predictions,total_predictions)
print(f'Accuracy for Greedy decoding on Dev data is: {Greedy_accuracy}')
```

```
Accuracy for Greedy decoding on Dev data is: 0.9153436342662862
```

1.17 Predict the part-of-speech tags of the sentences in the test data and output the predictions in a file named greedy.json, in the same format of training data.

1.17.1 Motivation and Plan

Motivation: This code aims to predict part-of-speech tags for a set of sentences using a greedy decoding approach.

Plan: 1. Load test data from 'test.json'. 2. Initialize data structures for storing predictions. 3. Iterate through each sentence in the test data. 4. Predict part-of-speech tags using 'greedy_decoding'. 5. Store predictions in a JSON file named 'greedy.json'.

[19]:
```
with open('data/test.json') as f:
    test_data = json.load(f)

test_data_predicted_tags = {}
greedy_dict_array = []

for record in test_data:
    index_value = record['index']
    predicted_tags = greedy_decoding(
        record['sentence'], tag_labels, initial_probability,␣
 ↪double_emission_probability, double_transition_probability, merged_word_freq)

    test_data_predicted_tags[index_value] = predicted_tags
    greedy_dict_array.append({"index": index_value,
                              "sentence": record['sentence'],
                              "labels": predicted_tags})
```

```
with open('greedy.json', 'w') as json_file:
    json.dump(greedy_dict_array, json_file, indent=4)
```

1.18 Task 4: Viterbi Decoding withHMM(30 Points)

The fourth task is to implement the viterbi decoding algorithm with HMM. Modified Viterbi Decoding Function Explanation

- **Objective:** This function computes Viterbi decoding for a given sentence using Hidden Markov Model (HMM) parameters.
- **Data Structures:** The function utilizes two dictionaries, `Pi` and `Backtrack`, for storing probabilities and backtrack information.
- **Word-by-Word Processing:**
  - The function iterates through each word in the input sentence.
  - It computes probabilities for each possible tag associated with the word.
- **Handling the Initial Word:**
  - For the initial word (when `i == 0`), it initializes probabilities based on initial state and emission probabilities.
  - It also initializes the `Backtrack` dictionary for this word.
- **Handling Subsequent Words:**
  - For subsequent words (when `i > 0`), it calculates probabilities considering transition, emission, and previous probabilities.
  - Updates both `Pi` and `Backtrack` dictionaries accordingly.
- **Tracking the Most Probable Sequence:**
  - Throughout the process, the function keeps track of the most probable tag sequence using the `Backtrack` dictionary.
- **Final Tag for the Sentence:**
  - The function identifies the most probable tag for the last word in the sentence.
  - It then appends this tag to the tag sequence before returning it.
- **Note:** The modification made to the original function ensures that the most probable final tag is included in the returned tag sequence. This can be valuable when you want to include the most likely tag for the last word in your output.

This function is crucial for tasks like part-of-speech tagging and other sequence labeling tasks in the field of natural language processing.

[20]:
```
def viterbi_decoding(sentence, tag_labels, initial_probability,␣
↪double_emission_probability, double_transition_probability,␣
↪merged_word_freq):

    Pi = {}
```

```
    Backtrack = {}

    for i,word in enumerate(sentence):

        word = word.lower()
        if word not in merged_word_freq:
            word = '< unk >'

        Pi[i] = {}
        Backtrack[i] = {}

        if(i == 0):
            for tag in tag_labels:
                Pi[i][tag] = getfromdict(initial_probability, tag) *␣
↪getfromdict(double_emission_probability, (word, tag), 2)
                Backtrack[i][tag] = []
        else:
            for tag in tag_labels:

                Pi[i][tag] = -1
                for prev_tag in tag_labels:

                    local_prob = Pi[i-1][prev_tag] *␣
↪getfromdict(double_transition_probability, (prev_tag, tag), 2) *␣
↪getfromdict(double_emission_probability, (word, tag), 2)

                    if(local_prob > Pi[i][tag]):
                        Pi[i][tag] = local_prob
                        prev_final_tag = prev_tag

                Backtrack[i][tag] = [prev_final_tag] +␣
↪Backtrack[i-1][prev_final_tag]

    lastEntryCount = len(sentence)-1

    max_tag_prob = -1
    max_tag = None

    for i in Pi[lastEntryCount]:
        tag_prob = Pi[lastEntryCount][i]
        if(tag_prob>max_tag_prob):
            max_tag_prob = tag_prob
            max_tag = i

    return Backtrack[lastEntryCount][max_tag][::-1] + [max_tag]
```

Loading and Predicting with Viterbi Decoding on Dev Data

- **Data Loading:** The code starts by loading data from the 'dev.json' file using a `with open` block. The loaded data is stored in the `dev_data` dictionary.
- **Prediction Loop:** Next, the code initializes an empty dictionary called `viterbi_dev_data_predicted_tags` to store predicted tag sequences for each sentence in the dev data.
- **Prediction Process:** It then iterates through each record in the `dev_data` dictionary. For each record:
  - It extracts the 'index' value, which appears to be an identifier.
  - It calls the `viterbi_decoding` function to predict the tag sequence for the 'sentence' in the record. The function is provided with various HMM parameters and data structures.
  - The predicted tag sequence is stored in the `viterbi_dev_data_predicted_tags` dictionary with the 'index' as the key.

This code performs Viterbi decoding on the dev data, generating predicted tag sequences for each sentence. These predicted tags can be further evaluated and compared against ground truth labels to assess the model's performance on the development dataset.

[21]:
```
with open('data/dev.json') as f:
    dev_data = json.load(f)

viterbi_dev_data_predicted_tags = {}

for record in dev_data:
    index_value = record['index']
    predicted_tags = viterbi_decoding(
        record['sentence'], tag_labels, initial_probability,␣
 ↪double_emission_probability, double_transition_probability, merged_word_freq)
    viterbi_dev_data_predicted_tags[index_value] = predicted_tags
```

Calculating Viterbi Decoding Accuracy on Dev Data

- **Objective:** This code computes the accuracy of Viterbi decoding results on the dev data.
- **Accuracy Calculation:** The code iterates through dev data records, comparing predicted labels to ground truth labels. It counts correct predictions and tracks total predictions.
- **Accuracy Display:** The final accuracy for Viterbi decoding on the dev data is calculated and displayed.

This code assesses the model's performance on the dev data by measuring how accurately it predicts labels.

[22]:
```
correct_predictions = 0
total_predictions = 0

for record in dev_data:

    words  = record['sentence']
    labels = record['labels']
```

```
    pred_labels = viterbi_dev_data_predicted_tags[record['index']]

    for i in range(len(words)):
        total_predictions += 1
        if(labels[i] == pred_labels[i]):
            correct_predictions += 1

viterbi_accuracy = calculate_accuracy(correct_predictions,total_predictions)
print(f'Accuracy for Viterbi decoding on Dev data is: {viterbi_accuracy}')
```

```
Accuracy for Viterbi decoding on Dev data is: 0.9320851800133568
```

Generating Viterbi Decoding Results for Test Data

- **Objective:** This code generates Viterbi decoding results for the test data and saves them to a JSON file.
- **Processing Test Data:** The code loads the test data from a JSON file and iterates through each record.
- **Viterbi Decoding:** For each record, Viterbi decoding is performed using the specified Hidden Markov Model (HMM) parameters.
- **Results Storage:** The predicted tags for each record are stored in a dictionary and saved to a JSON file.

This code is responsible for applying Viterbi decoding to the test data and saving the results for further analysis or evaluation.

[23]:
```
with open('data/test.json') as f:
    test_data = json.load(f)

viterbi_test_data_predicted_tags = {}
viterbi_dict_array = []

for record in test_data:
    index_value = record['index']
    predicted_tags = viterbi_decoding(
        record['sentence'], tag_labels, initial_probability,␣
 ↪double_emission_probability, double_transition_probability, merged_word_freq)

    viterbi_test_data_predicted_tags[index_value] = predicted_tags
    viterbi_dict_array.append({"index": index_value,
                               "sentence": record['sentence'],
                               "labels": predicted_tags})

with open('viterbi.json', 'w') as json_file:
    json.dump(viterbi_dict_array, json_file, indent=4)
```