CS187 Lab 2-4: Sequence labeling with hidden Markov models

September 9, 2023

[ ]:
```
# Please do not change this cell because some hidden tests might depend on it.
import os

# Otter grader does not handle ! commands well, so we define and use our
# own function to execute shell commands.
def shell(commands, warn=True):
    """Executes the string `commands` as a sequence of shell commands.

       Prints the result to stdout and returns the exit status.
       Provides a printed warning on non-zero exit status unless `warn`
       flag is unset.
    """
    file = os.popen(commands)
    print (file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status != None:
        print(f"Completed with errors. Exit status: {exit_status}\n")
    return exit_status

shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
 rm -rf .tmp
 git clone https://github.com/cs187-2021/lab2-4.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

[ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

[ ]:
```
import math
from collections import defaultdict
```

1 Introduction

Hidden Markov models (HMM) are a fundamental generative method for sequence labeling NLP tasks such as part-of-speech tagging (as in the present lab) and information extraction (as in the second project segment). In this lab, you'll train, apply, and evaluate some simple sequence labeling algorithms culminating in an HMM.

To keep things manageable, the dataset you'll use will involve very few word types, only six (plus a special beginning of sentence token), but these word types are quite ambiguous with regard to part of speech. We'll use the following labels for parts of speech:

[ ]:
```
parts_of_speech = [
    "<bos>", # beginning of sentence marker
    "N",     # noun
    "V",     # main verb
    "M",     # modal verb
    "P",     # preposition
    "A",     # adjective
    "R"      # adverb
]
```

The vocabulary of word types, along with their possible parts of speech, is given by the following dictionary:

[ ]:
```
vocabulary = {
    "<bos>":   ["<bos>"],
    "can":     ["N", "V", "M"],
    "canned":  ["A", "V"],
    "canners": ["N"],
    "fish":    ["N", "V"],
    "for":     ["P"],
    "not":     ["R"]
}
```

Here are a few sentences constructed with these words:

[ ]:
```
text = """
    <bos> canners canned fish
    <bos> can canners can fish
    <bos> fish can not fish
    <bos> can fish can fish can
    <bos> canners fish fish for can
    <bos> canners can fish for fish
    <bos> canners fish for fish
    <bos> fish can canned fish
    <bos> canners can not can canned fish
    <bos> fish can can fish for canners
"""
```

and the corresponding POS sequences, for the first few sentences. You complete the rest.

[ ]:
```
#TODO -- Provide part-of-speech sequences in this format for the rest of the␣
 ↪sample sentences.
text_pos = """
    <bos> N V N
    <bos> M N V N
    <bos> N M R V
    <bos> M N V N N
    <bos> N V N P N
    <bos> N M V P N
"""
```

[ ]:
```
grader.check("text_pos")
```

> Just to make sure we're all on the same page – and because you'll need the right tagging to calculate some probabilities below – you can find our intended solution to this question at https://go.cs187.info/lab2-4-q1. You should check your solution before moving on.

We tokenize the sentences and label the tokens with their POS.

[ ]:
```
def tokenize(text):
  result = []
  for line in text.strip().split("\n"):
    result.append([item for item in line.strip().split()])
  return result

tagged_text = [list(zip(sentence, poses))
                 for sentence, poses
                   in zip(tokenize(text), tokenize(text_pos))]
```

Here are a couple of examples to indicate what the tagged sentences look like.

[ ]:
```
print(tagged_text[0])
print(tagged_text[1])
```

For reference, here is a table showing the frequency distribution for each word type and each part of speech it can be used as.

[ ]:
```
counts = defaultdict(lambda: defaultdict(int))
for sentence in tagged_text:
  for type, pos in sentence:
    counts[type][pos] += 1

print(f'{"TYPE":8} {"POS":6} {"COUNT"}')
for type, type_counts in counts.items():
  for pos, count in type_counts.items():
    print(f'{type:8} {pos:6} {count:2}')
```

2 Majority label

The first sequence labeling method we'll use is simply to choose for each word the POS label it most frequently occurs as in the training data. The table above provides the required information directly.

Choosing the majority label for a word sequence $\mathbf{w} = \langle w_1, w_2, \ldots, w_m \rangle$ is tantamount to maximizing the probability of the label sequence assuming independence of the label conditioned on the word, that is, selecting the tag sequence $\mathbf{t} = \langle t_1, t_2, \ldots, t_m \rangle$ given by

$$\operatorname*{argmax}_{\mathbf{t}} \prod_{i=1}^{m} \Pr(t_i \mid w_i)$$

How would the majority label method label the following test sentence (which we've marked with the words' correct ("gold") parts of speech)?

<bos>[<bos>] canners[N] can[V] canned[A] fish[N]

Give your answer in the next cell in the form of a list of strings for the POS labels.

[ ]:
```
#TODO
example_majority_labeling = ...
```

[ ]:
```
grader.check("example_majority_labeling")
```

By inspection, what is the accuracy of the majority labeling, given as a proportion of the words in the sentence (including the beginning of sentence token)?

[ ]:
```
#TODO
example_maj_label_accuracy = ...
```

[ ]:
```
grader.check("example_maj_label_accuracy")
```

3 Majority bigram labeling

It may occur to you that what part of speech a word has *depends on its context*. Suppose we relax the assumption that tag probabilities depend only on the word being tagged, and condition them on the previous word as well. (For the first word in the sentence, we'll condition on that fact, by conditioning it on the special start token.) In summary, we'll condition on the bigram that ends at the word being tagged:

$$\operatorname*{argmax}_{\mathbf{t}} \prod_{i=1}^{m} \Pr(t_i \mid w_{i-1} w_i)$$

What is the majority bigram labeling of the test sentence? Again, give your answer in the form of a list of strings for the POS labels.

```
[ ]: #TODO
example_majority_bigram_labeling = ...
```

```
[ ]: grader.check("example_majority_bigram_labeling")
```

By inspection, what is the accuracy of the majority bigram labeling, given as a proportion of the words?

```
[ ]: #TODO
example_maj_bigram_label_accuracy = ...
```

```
[ ]: grader.check("example_maj_bigram_label_accuracy")
```

4 Hidden Markov models

Now we get to the real point, using an HMM model. Recall that in an HMM model, we assume that the joint tag/word sequence is generated by

1. Selecting a tag sequence according to a Markov model whose states correspond to tags and whose transitions from state t_i to t_j are governed by a *transition probability* $a_{ij} = \Pr(t_i \to t_j)$, and then
2. Selecting a word sequence from the tag sequence where for tag t_i we observe word x_i of type w_j governed by an *emission probability* $b_i(w_j) = \Pr(t_i \to w_j)$.

   Here, we're using a notation $\Pr(t_i \to t_j)$ to indicate the probability that a word tagged t_i is followed by a word tagged t_j and $\Pr(t_i \to w_j)$ to indicate the probability that a word tagged t_i is the word w_j.

4.1 Estimating the transition and emission probabilities

We estimate these transition and emission probabilities by looking at the empirical probabilities in the training data, counting and perhaps smoothing as usual. That is, for the (unsmoothed) transition probabilities, we estimate

$$a_{ij} \approx \frac{\sharp(t_i \to t_j)}{\sum_k \sharp(t_i \to t_k)}$$

and for the emission probabilities

$$b_i(w_j) \approx \frac{\sharp(t_i \to w_j)}{\sharp(t_i)}$$

For instance, we note that there are 4 times in the training data where the tag N is followed by the tag M, out of the 21 occurrences of the tag N. Thus, we estimate the corresponding transition probability $a_{NM} \approx 4/21$.

Similarly, the emission probability $b_M(can)$ for tag M generating the word *can* is $6/6 = 1$, since every occurrence of the tag M corresponds to the word *can* in the training data.

For your convenience, we've computed and provided full tables for the transition and emission probabilities below.

[ ]:
```
# Generate counts
bigram_tag_counts = defaultdict(lambda: defaultdict(int))
unigram_tag_counts = defaultdict(int)
tag_word_counts = defaultdict(lambda: defaultdict(int))
tag_counts = defaultdict(int)

for sentence in tagged_text:
    for (w1, t1), (w2, t2) in list(zip(sentence[:-1], sentence[1:])):
        bigram_tag_counts[t1][t2] += 1
        unigram_tag_counts[t1] += 1
    for w, t in sentence:
        tag_word_counts[t][w] += 1
        tag_counts[t] += 1

# Generate transition and emission probabilities
a = defaultdict(lambda: defaultdict(int))
b = defaultdict(lambda: defaultdict(int))

for t1 in parts_of_speech:
    for t2 in parts_of_speech:
        a[t1][t2] = bigram_tag_counts[t1][t2] / unigram_tag_counts[t1]
    for w1 in vocabulary.keys():
        b[t1][w1] = tag_word_counts[t1][w1] / tag_counts[t1]

# Print tables of probabilities

print("Transition probabilities: a_ij")
print(f"{' ':6}", end="")
for t in parts_of_speech:
    print(f"{t:>6}", end="")
print()
for t1 in parts_of_speech:
    print(f"{t1:<6}", end="")
    for t2 in parts_of_speech:
        print(f"{a[t1][t2]:>6.2f}", end="")
    print("")

print("\nEmission probabilities: b_i(w_j)")
print(f"{' ':6}", end="")
for w in vocabulary.keys():
    print(f"{w:>8}", end="")
print()
for t in parts_of_speech:
    print(f"{t:<6}", end="")
```

```
    for w in vocabulary.keys():
        print(f"{b[t][w]:>8.2f}", end="")
    print()
```

4.2 An example HMM trellis

Now consider the HMM generating the example sentence "canners can canned fish". The figure at right contains the *trellis* for the sentence. The horizontal axis corresponds to the words in the sentence, one at a time. The vertical axis corresponds to the states of the HMM (that is, the parts of speech). The gray arrows that connect a tag on the left to a tag on the right correspond to the transition probabilities. The red arrows that connect a tag to a word directly below correspond to the emission probabilities.

For convenient reference, we've labeled the rows with letters and the columns with numbers (in teal!) so that particular nodes can be referred to as, for example, B1.

(The red lines are intended to go from a state node to the word node directly below it; for instance, the red line immediately below B1 goes to the word node F1 at the bottom of the figure. Some of those lines are depicted with a crossbar to indicate that they are running "underneath" other graphic objects in the figure.)

We've highlighted two paths through the trellis from the beginning to the end of the sentence, corresponding to different taggings of the sentence:

1. A0-B1-C2-E3-B4
2. A0-B1-D2-C3-B4

Answer the following questions about this trellis.

What is the path through the trellis corresponding to the *majority bigram labeling* that you determined above? Give your answer as a string, like the path examples in the previous cell.

[ ]:
```
majority_path = ...
```

[ ]:
```
grader.check("majority_path")
```

The probability of a path is just the product of the probabilities of all the transitions along the path and all the emissions from nodes in the path to observed words. Use the tables above to calculate the probability of the first highlighted path (A0-B1-C2-E3-B4) by multiplying together the appropriate probabilities. Don't forget the emission probabilities, corresponding to the edges A0-F0, B1-F1, C2-F2, E3-F3, and B4-F4.

[ ]:
```
#TODO
highlight1_path_probability = ...
highlight1_path_probability
```

[ ]:
```
grader.check("highlight1_path_probability")
```

Do the same for the second highlighted path (A0-B1-D2-C3-B4).

[ ]:
```
#TODO
highlight2_path_probability = ...
highlight2_path_probability
```

[ ]:
```
grader.check("highlight2_path_probability")
```

These two paths turn out to be the two paths through the trellis with the highest probabilities. (You'll have to trust us.) Based on that fact, which tagging has the highest probability according to this HMM? Give your solution in the same format as you did for `example_majority_labeling` above.

[ ]:
```
#TODO
example_highest_labeling = ...
```

[ ]:
```
grader.check("example_highest_labeling")
```

By inspection, what is the accuracy of the highest probability HMM labeling, given as a proportion of the words?

[ ]:
```
#TODO
example_highest_label_accuracy = ...
```

[ ]:
```
grader.check("example_highest_label_accuracy")
```

Now, recall the majority bigram labeling, and the path for it that you developed above. What is the probability of that path according to the HMM?

[ ]:
```
majority_path_probability = ...
```

[ ]:
```
grader.check("majority_path_probability")
```

4.3 Calculating the highest probability tagging - The Viterbi algorithm

Above, we merely asserted that the two highlighted paths are the two most probable, so that it was a simple matter to find the highest probability tagging by just comparing the probabilities of those two. But in general there can be a huge number of paths through a trellis such as this.

Question: If there are N tags and a sentence of length M, how many paths through the HMM trellis will there be (using big-O notation)?

Type your answer here, replacing this text.

The Viterbi algorithm, named after famed electrical engineer Andrew Viterbi, is an efficient dynamic programming algorithm for performing this (otherwise impractical) computation. We'll do the first few steps of the Viterbi algorithm for the example here.

Given a string of words $\mathbf{x} = \langle w_0, w_1, \ldots, w_M \rangle$ and a set of states (tags) $\mathbf{q} = \{q_0, q_1, \ldots, q_N\}$, the algorithm works by calculating a series of values $v_i(j)$ where i ranges over the words in the sentence

from 1 to M and j ranges over the tags from 1 to N. For simplicity, we'll assume an extra word and tag at the beginning of the sentence, as above, so $w_0 =$ `<bos>` and $q_0 =$ `<bos>`.

The values $v_i(j)$ correspond to the probability of the best (highest probability) path starting in state 0, emitting $w_0, \ldots, w_i$, and ending in state q_j. The definition for v then is:

$$
\begin{aligned}
v_0(0) &= 1 \\
v_0(j) &= 0 && \text{for } j > 0 \\
v_i(j) &= \max_{j'=0}^{N} v_{i-1}(j') \cdot a_{j'j} \cdot b_j(w_i) && \text{for } i > 0
\end{aligned}
$$

In addition, we'll want to keep track of which states the best paths go through, so in addition to tracking the best probability with $v_i(j)$, we'll track the immediately preceding state that led to that best path – the "back pointer" – with $bp_i(j)$:

$$
bp_i(j) = \operatorname*{argmax}_{j'=0}^{N} v_{i-1}(j') \cdot a_{j'j} \cdot b_j(w_i) \qquad \text{for } i > 0
$$

> Notice how similar the back pointer definition is to the Viterbi best path definition. It merely records which state was used in computing the corresponding best path.

Question: For the sample sentence above ("canners can canned fish"), calculate the five "layers" of the Viterbi algorithm, that is, v_0, v_1, v_2, v_3, and v_4, and the corresponding back pointers by filling in the tables below. (We've filled in the v_0 column for you already, as per the first two lines in the definition of the Viterbi calculation. No bp_0 backpointer is needed (or defined).)

> For the back pointer table, you need only provide entries corresponding to cases where $bp_i(j)$ is non-zero.

	tag	v_0 <bos>	v_1 canners	v_2 can	v_3 canned	v_4 fish
0	<bos>	1				
1	N	0				
2	V	0				
3	M	0				
4	P	0				
5	A	0				
6	R	0				

	tag	bp_1 canners	bp_2 can	bp_3 canned	bp_4 fish
0	<bos>				
1	N				
2	V				
3	M				
4	P				
5	A				

	tag	bp_1 canners	bp_2 can	bp_3 canned	bp_4 fish
6	R				

Doing this calculation by hand is painful, but it should make clear what's going on. At each entry $v_i(j)$ in the table, we've calculated the probability of the best path through the trellis from the beginning of the sentence to the current word x_i, starting in the start state and ending in the current state q_j. To get the maximum probability of all paths in the trellis for the full sentence ending in any state, we merely look up the maximum value of $v_M(j)$ (recalling that M is the length of the sentence).

Question: Based on the tables you filled out above, what is the probability of the best path through the trellis for the sample sentence?

[ ]:
```
best_path_probability = ...
```

[ ]:
```
grader.check("best_path_probability")
```

Question: Based on the tables you filled out above, what is the best label sequence for the sentence? Provide it as a list of strings for the variable `best_label_sequence`, e.g., `['<bos>','V','A','M','A']`. (No, that's not the actual answer.)

Hint: You'll get this by tracing back the backpointers.

[ ]:
```
best_label_sequence = ...
```

[ ]:
```
grader.check("best_label_sequence")
```

Question: What is the complexity of filling in all of the entries in the Viterbi table? How does that compare with the complexity of the total number of paths through the trellis that you calculated above?

Type your answer here, replacing this text.

5 Lab debrief – for consensus submission only

Question: We're interested in any thoughts you have about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on include the following, but you're not restricted to these:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there additions or changes you think would make the lab better?

Type your answer here, replacing this text.

End of lab 2-4

To double-check your work, the cell below will rerun all of the autograder tests.

[ ]:
```
grader.check_all()
```