toy_example

February 1, 2023

1 Toy Example

This notebook walks through a toy example demonstrating the correct outputs for several functions from within `hmm.py`.

[1]:
```
%load_ext autoreload
%autoreload 2

import pickle
import pandas as pd
from hmm_answer import *
```

1.1 1. Load Data

This toy example is based on the textbook notes for hidden Markov models, which can be accessed using this link. Please note: the probability values (initial, transition, and emission) are NOT the same as in the textbook.

In the example they provide, our task is to create a hidden Markov model that relates the number of ice creams eaten by a man named Jason to the weather on a given day. The weather can be either "cold" or "hot"–these will be our hidden states. Jason eats either 1, 2, or 3 ice creams every day–these will be our observations.

Some example data has been loaded into `toy_example.txt`. The following code loads this example data:

[2]:
```
data = load_data('toy_example.txt')
print(f"Number of sentences: {len(data)}")
print(f"Number of tokens: {sum([len(i) for i in data])}")
print(f"Train data id 0: {data[0]}")
```

```
Number of sentences: 3
Number of tokens: 19
Train data id 0: [['1', 'cold'], ['2', 'cold'], ['2', 'hot'], ['3', 'hot'],
['1', 'cold'], ['</s>', '<end>']]
```

1.2 2. Frequency Count

Now we need to program `get_counts`. This function should take as its input `data` and compute the properties `initial_count`, `transition_count`, and `emission_count`.

Of course, we are not performing NER tagging for this toy example, but we can use `HMMNER` anyways.

[3]:
```
hmm = HMMNER()
hmm.get_counts(data)

print(f"Initial count: {hmm.initial_count}")
print(f"Transition count: {hmm.transition_count}")
print(f"Emission count: {hmm.emission_count}")
```

```
Initial count: {'cold': 2, 'hot': 1}
Transition count: {('cold', 'cold'): 2, ('cold', 'hot'): 4, ('hot', 'hot'): 4,
('hot', 'cold'): 3, ('cold', '<end>'): 1, ('hot', '<end>'): 2}
Emission count: {('cold', '1'): 4, ('cold', '2'): 3, ('hot', '2'): 4, ('hot',
'3'): 5, ('<end>', '</s>'): 3}
```

1.3 3. Get Tags and Vocabulary

[4]:
```
hmm.get_lists()
print(f"NER tags: {hmm.ner_tags}")
print(f"Index of cold: {hmm.tag_to_index['cold']}")
print(f"All observations: {hmm.observations}")
```

```
NER tags: ['<end>', 'cold', 'hot']
Index of cold: 1
All observations: ['1', '2', '3', '</s>']
```

1.4 4. Get Probabilities

In this cell we need to convert the raw frequency counts into probability values. We can use a relatively simple procedure for add-k smoothing:

- Calculate the probability values without smoothing
- Add k to each probability value
- Normalize the entire distribution (make sure it sums to 1)

This process should be done separately for the `initial_prob`, `transition_prob`, and `emission_prob` arrays.

One question that may arise is: across which axis/dimension should the normalization occur? Of course, since `initial_prob` is one-dimensional, normalization should take place along that one dimension. For `transition_prob` and `emission_prob`, both of which are two-dimensional, normalization should occur along each row (`axis=1`). The reason is that the formulae we use (for both the Viterbi/Beam Search and Forward algorithms) utilize conditional probability distributions described by each row. For instance, the transition probabilities are modeled as $\Pr(t_i|t_{i-1})$, where t_i is the i'th tag. The distribution over values of t_i is a single row in `transition_prob`. Thus, it should sum to 1.

[5]:
```
initial_k, transition_k, emission_k = 0.1, 0.1, 0.1

hmm.get_probabilities(initial_k, transition_k, emission_k)
```

```
print(f"Initial prob:\n{hmm.initial_prob}")
print(hmm.initial_prob.sum())
print(f"Initial prob of 'cold': {hmm.initial_prob[hmm.tag_to_index['cold']]}")

print("")

print(f"Transition prob:\n{hmm.transition_prob}")
print(hmm.transition_prob.sum(axis=1))
print(f"Transition prob from 'cold' to 'hot': {hmm.transition_prob[hmm.
  ↪tag_to_index['cold'], hmm.tag_to_index['hot']]}")

print("")

print(f"Emission prob:\n{hmm.emission_prob}")
print(hmm.emission_prob.sum(axis=1))
print(f"Emission prob from 'cold' to '1': {hmm.emission_prob[hmm.
  ↪tag_to_index['cold'], hmm.observation_to_index['1']]}")
```

```
Initial prob:
[0.03030303 0.63636364 0.33333333]
1.0
Initial prob of 'cold': 0.6363636363636364

Transition prob:
[[0.33333333 0.33333333 0.33333333]
 [0.15068493 0.28767123 0.56164384]
 [0.22580645 0.33333333 0.44086022]]
[1. 1. 1.]
Transition prob from 'cold' to 'hot': 0.5616438356164383

Emission prob:
[[0.02941176 0.02941176 0.02941176 0.91176471]
 [0.55405405 0.41891892 0.01351351 0.01351351]
 [0.0106383  0.43617021 0.54255319 0.0106383 ]]
[1. 1. 1.]
Emission prob from 'cold' to '1': 0.5540540540540541
```

1.5 5. Beam Search

Now we need to implement beam search. The output displayed below provides a concrete example of the `tags` and `backtrace` matrices.

[6]:
```
beam_width = 2

sample_data = ['1', '3', '3', '2', '1', '</s>']
weather_tags = hmm.beam_search(sample_data, beam_width, should_print=True)
print(weather_tags)
```

```
Tag Index Matrix:
 [[2 0 0 1 0 2]
 [1 2 2 2 1 0]]
Backtrace Matrix:
 [[0 1 1 1 1 1]
 [0 1 1 1 1 1]]
['cold', 'hot', 'hot', 'hot', 'cold', '<end>']
```

For analysis purposes, I have copied the `tags` and `backtrace` matrices below.

In this instance `tags[0,0] = 2`. To find out which tag this refers to, we can use `hmm.ner_tags`.

Now suppose we want to know which tag provided the maximum probability value for `tags[0,2]` (which tag preceded `tags[0,2]`). To do this, we can use the same indices to index into `backtrace`. Refer to the code below to see the concrete implementation.

[7]:
```
tags = np.array([[2, 0, 0, 1, 0], [1, 2, 2, 2, 1]])
backtrace = np.array([[0, 1, 1, 1, 1], [0, 1, 1, 1, 1]])

# Which tag does tags[0,0] refer to?
this_tag = hmm.ner_tags[tags[0,0]]
print(f"tags[0,0] = 2 refers to '{this_tag}'")

# Which tag provided the maximum probability value for tags[2,0]?
index_of_prev_tag = backtrace[0,2]
prev_tag_idx = tags[index_of_prev_tag, 2-1]
prev_tag = hmm.ner_tags[prev_tag_idx]
print(f"The tag in tags[0,2] was preceded by '{prev_tag}'")
```

```
tags[0,0] = 2 refers to 'hot'
The tag in tags[0,2] was preceded by 'hot'
```

1.6 6. Forward Algorithm

Now we need to implement the forward algorithm.

[8]:
```
sample_data = ['1', '3', '3', '2', '1', '</s>']
prob = hmm.forward_algorithm(sample_data)
print(f"Log Probability: {prob}")
```

```
Log Probability Matrix:
 [[ -7  -6  -7  -8  -9  -8]
 [ -1  -6  -7  -5  -6 -11]
 [ -5  -2  -3  -5  -9 -11]]
Log Probability: -8.218983217556692
```

The matrix/code structure for this algorithm is much simpler to understand than the code structure for beam search. For demonstration purposes, I have re-pasted the matrix below.

The log-probability matrix is of shape `(num_tags, num_tokens_in_sentence)`. Each row refers to a single tag, and each column refers to a single token. Thus, `log_prob_mat[i,j]` refers to the

log-probability that token `j` has tag `hmm.ner_tags[i]`.

[9]:
```
log_prob_mat = np.array([[-7, -6, -7, -8, -9], [-1, -6, -7, -5, -6], [-5, -2,␣
 ↪-3, -5, -9]])

sample_idx = [1,3]
token = sample_data[sample_idx[1]]
tag = hmm.ner_tags[sample_idx[0]]
log_prob = log_prob_mat[sample_idx[0], sample_idx[1]]
print(f"There is a log-probability of {log_prob} that the token '{token}' has␣
 ↪the tag '{tag}'")
```

```
There is a log-probability of -5 that the token '2' has the tag 'cold'
```