CS440/ECE448 Spring 2024

MP03: Hidden Markov Model

The first thing you need to do is to download this file: mp03.zip. It has the following content:

- `submitted.py`: Your homework. Edit, and then submit to Gradescope.
- `mp03_notebook.ipynb`: This is a Jupyter notebook to help you debug. You can completely ignore it if you want, although you might find that it gives you useful instructions.
- `grade.py`: Once your homework seems to be working, you can test it by typing `python grade.py`, which will run the tests in `tests/tests_visible.py`.
- `tests/test_visible.py`: This file contains about half of the unit tests that Gradescope will run in order to grade your homework. If you can get a perfect score on these tests, then you should also get a perfect score on the additional hidden tests that Gradescope uses.
- `data`: This directory contains the data.
- `util.py`: This is an auxiliary program that you can use to read the data, evaluate the accuracy, etc.

This file (`mp08_notebook.ipynb`) will walk you through the whole MP, giving you instructions and debugging tips as you go.

Table of Contents

For this MP, you will implement part of speech (POS) tagging using an HMM model. Make sure you understand the algorithm before you start writing code, e.g. look at the lectures on Hidden Markov Models and

Chapter 8 of Jurafsky and Martin.

Reading the data

The dataset consists of thousands of sentences with ground-truth POS tags.

The provided load_dataset function will read in the data as a nested list with the outer dimension representing each sentence and inner dimensin representing each tagged word. The following cells will help you go through the representation of the data.

The provided code converts all words to lowercase. It also adds a START and END tag for each sentence when it loads the sentence. These tags are just for standardization. They will not be considered in accuracy computation.

In [1]:
```
import utils
train_set = utils.load_dataset('data/brown-training.txt')
dev_set = utils.load_dataset('data/brown-test.txt')
```

In [2]:
```
print('training set has {} sentences'.format(len(train_set)))
print('dev set has {} sentences'.format(len(dev_set)))
print('The first sentence of training set has {} words'.format(len(train_
print('The 10th word of the first sentence in the training set is "{}" wi
```

```
training set has 35655 sentences
dev set has 9912 sentences
The first sentence of training set has 27 words
The 10th word of the first sentence in the training set is "investigation"
with ground-truth tag "NOUN"
```

In [3]:
```
print('Here is an sample sentence from the training set:\n', train_set[0]
```

```
Here is an sample sentence from the training set:
 [('START', 'START'), ('the', 'DET'), ('fulton', 'NOUN'), ('county', 'NOU
N'), ('grand', 'ADJ'), ('jury', 'NOUN'), ('said', 'VERB'), ('friday', 'NOU
N'), ('an', 'DET'), ('investigation', 'NOUN'), ('of', 'IN'), ("atlanta's",
'NOUN'), ('recent', 'ADJ'), ('primary', 'NOUN'), ('election', 'NOUN'), ('p
roduced', 'VERB'), ('``', 'PUNCT'), ('no', 'DET'), ('evidence', 'NOUN'),
("''", 'PUNCT'), ('that', 'CONJ'), ('any', 'DET'), ('irregularities', 'NOU
N'), ('took', 'VERB'), ('place', 'NOUN'), ('.', 'PERIOD'), ('END', 'END')]
```

Tagset

The following is an example set of 16 part of speech tags. This is the tagset used in the provided Brown corpus. **But remember you should not hardcode anything regarding this tagset because we will test your code on two other datasets with a different tagset**.

- ADJ adjective
- ADV adverb
- IN preposition
- PART particle (e.g. after verb, looks like a preposition)
- PRON pronoun
- NUM number
- CONJ conjunction
- UH filler, exclamation
- TO infinitive
- VERB verb

- MODAL modal verb
- DET determiner
- NOUN noun
- PERIOD end of sentence punctuation
- PUNCT other punctuation
- X miscellaneous hard-to-classify items

Taggers

You will need to write two main types of tagging functions:

- Baseline tagger
- Viterbi: HMM tagger

For implementation of this MP, You may use numpy (though it's not needed). **You may not use other non-standard modules (including nltk)**.

You should use the provided training data to train the parameters of your model and the test sets to test its accuracy.

In addition, your code will be tested on two hidden datasets that are not available to you, which has different number of tags and words from the ones provided to you. So do NOT hardcode any of your important computations, such as initial probabilities, transition probabilities, emission probabilities, number or name of tags, and etc. We will inspect code for hardcoding computations/values and will penalize such implementations.

Baseline Tagger

The Baseline tagger considers each word independently, ignoring previous words and tags. For each word w, it counts how many times w occurs with each tag in the training data. When processing the test data, it consistently gives w the tag that was seen most often. For unseen words, it should guess the tag that's seen the most often in training dataset.

For all seen word w:

$$Tag_w = \underset{t \in T}{\operatorname{argmax}}(\#\text{ times tag t is matched to word w})$$

For all unseen word w':

$$Tag_{w'} = \underset{t \in T}{\operatorname{argmax}}(\#\text{ times tag t appears in the training set})$$

A correctly working baseline tagger should get about 93.9% accuracy on the Brown corpus development set, with over 90% accuracy on multitag words and over 69% on unseen words.

In [4]:
```
import submitted
import importlib
importlib.reload(submitted)
print(submitted.__doc__)
```

```
This is the module you'll submit to the autograder.

There are several function definitions, here, that raise RuntimeErrors.  Y
ou should replace
each "raise RuntimeError" line with a line that performs the function spec
ified in the
function's docstring.

For implementation of this MP, You may use numpy (though it's not needed).
You may not
use other non-standard modules (including nltk). Some modules that might b
e helpful are
already imported for you.
```

In [5]:
```
help(submitted.baseline)
```

```
Help on function baseline in module submitted:

baseline(train, test)
    Implementation for the baseline tagger.
    input:  training data (list of sentences, with tags on the words)
            test data (list of sentences, no tags on the words, use utils.
strip_tags to remove tags from data)
    output: list of sentences, each sentence is a list of (word,tag) pair
s.
            E.g., [[(word1, tag1), (word2, tag2)], [(word3, tag3), (word4,
tag4)]]
```

In [6]:
```
import time
importlib.reload(submitted)
train_set = utils.load_dataset('data/brown-training.txt')
dev_set = utils.load_dataset('data/brown-test.txt')
start_time = time.time()
predicted = submitted.baseline(utils.strip_tags(dev_set), train_set)
time_spend = time.time() - start_time
accuracy, _, _ = utils.evaluate_accuracies(predicted, dev_set)
multi_tag_accuracy, unseen_words_accuracy, = utils.specialword_accuracies

print("time spent: {0:.4f} sec".format(time_spend))
print("accuracy: {0:.4f}".format(accuracy))
print("multi-tag accuracy: {0:.4f}".format(multi_tag_accuracy))
print("unseen word accuracy: {0:.4f}".format(unseen_words_accuracy))
```

```
time spent: 1.2834 sec
accuracy: 0.9388
multi-tag accuracy: 0.9021
unseen word accuracy: 0.6782
```

Viterbi: HMM Tagger

The Viterbi tagger should implement the HMM trellis (Viterbi) decoding algoirthm as seen in lecture or Jurafsky and Martin. That is, the probability of each tag depends only on the previous tag, and the probability of each word depends only on the corresponding tag. This model will need to estimate three sets of probabilities:

- Initial probabilities (How often does each tag occur at the start of a sentence?)
- Transition probabilities (How often does tag t_b follow tag t_a?)
- Emission probabilities (How often does tag t yield word w?)

You can assume that all sentences will begin with a START token, whose tag is START. **So your initial probabilities will have a very restricted form, whether you choose to handcode appropriate numbers or learn them from the data.** The initial probabilities shown in the textbook/texture examples will be handled by transition probabilities from the START token to the first real word.

It's helpful to think of your processing in five steps:

- Count occurrences of tags, tag pairs, tag/word pairs.
- Compute smoothed probabilities
- Take the log of each probability
- Construct the trellis. Notice that for each tag/time pair, you must store not only the probability of the best path but also a pointer to the previous tag/time pair in that path.
- Return the best path through the trellis.

You'll need to use smoothing to get good performance. Make sure that your code for computing transition and emission probabilities never returns zero. Laplace smoothing is the method we use to smooth zero probability cases for calculating initial probabilities, transition probabilities, and emission probabilities.

For example, to smooth the emission probabilities, consider each tag individually. For a fixed tag T, you need to ensure that $P_e(W|T)$ produces a non-zero number no matter what word W you give it. You can use Laplace smoothing (as in MP 2) to fill in a probability for "UNKNOWN" which will be the return value for all words W that were not seen in the training data. For this initial implementation of Viterbi, use the same Laplace smoothing constant α for all tags.

This simple Viterbi will perform slightly worse than the baseline code for the Brown development dataset (somewhat over 93% accuracy). However you should notice that it's doing better on the multiple-tag words (e.g. over 93.5%). **Please make sure to follow the description to implement your algorithm and do not try to do improvement in this part, as it might make your code fail some of our test cases. You will be asked to improve the algorithm in the next part.**

In [7]:
```
help(submitted.viterbi)
```

```
Help on function viterbi in module submitted:

viterbi(train, test)
    Implementation for the viterbi tagger.
    input:  training data (list of sentences, with tags on the words)
            test data (list of sentences, no tags on the words)
    output: list of sentences with tags on the words
            E.g., [[(word1, tag1), (word2, tag2)], [(word3, tag3), (word4,
tag4)]]
```

In [8]:
```
import time
importlib.reload(submitted)
train_set = utils.load_dataset('data/brown-training.txt')
dev_set = utils.load_dataset('data/brown-test.txt')
start_time = time.time()
predicted = submitted.viterbi(utils.strip_tags(dev_set), train_set)
time_spend = time.time() - start_time
accuracy, _, _ = utils.evaluate_accuracies(predicted, dev_set)
multi_tag_accuracy, unseen_words_accuracy, = utils.specialword_accuracies

print("time spent: {0:.4f} sec".format(time_spend))
print("accuracy: {0:.4f}".format(accuracy))
print("multi-tag accuracy: {0:.4f}".format(multi_tag_accuracy))
print("unseen word accuracy: {0:.4f}".format(unseen_words_accuracy))
```

```
time spent: 28.7358 sec
accuracy: 0.9372
multi-tag accuracy: 0.9373
unseen word accuracy: 0.2393
```

Viterbi_ec: Improving HMM Tagger (Optional, for Extra Credit only)

The previous Vitebi tagger fails to beat the baseline because it does very poorly on unseen words. It's assuming that all tags have similar probability for these words, but we know that a new word is much more likely to have the tag NOUN than (say) CONJ. For this part, you'll improve your emission smoothing to match the real probabilities for unseen words.

Words that appear zero times in the training data (out-of-vocabulary or OOV words) and words that appear once in the training data (hapax words) tend to have similar parts of speech (POS). For this reason, instead of assuming that OOV words are uniformly distributed across all POS, we can get a much better estimate of their distribution by measuring the distribution of hapax words. Extract these words from the training data and calculate the probability of each tag on them. When you do your Laplace smoothing of the emission probabilities for tag T, scale the Laplace smoothing constant by P(T|hapax), i.e., the probability that tag T occurs given that the word was hapax. Remember that Laplace smoothing acts by reducing probability mass for high-frequency words, and re-assigning some of that probability mass to low-frequency words. A large smoothing constant can end up skewing probability masses a lot, so experiment with small orders of magnitude for this hyperparameter.

This optimized version of the Viterbi code should have a significantly better unseen word accuracy on the Brown development dataset, e.g. over 66.5%. It also beat the baseline on overall accuracy (e.g. 95.5%). You should write your optimized version of Viterbi under the viterbi_ec function in submitted.py.

The hapax word tag probabilities may be different from one dataset to another, so your `viterbi_ec` method should compute them dynamically from its training data each time it runs.

Hints

- You should start with the original implementation of your viterbi algorithm in the previous part. As long as you understand what you need to do, the change in implementation should not be substantial.
- Tag 'X' rarely occurs in the dataset. Setting a high value for the Laplace smoothing constant may overly smooth the emission probabilities and break your statistical computations. A small value for the Laplace smoothing constant, e.g. 1e-5, may help.
- It's not advisable to use global variables in your implementation since gradescope runs a number of different tests within the same python environment. Global values set during one test will carry over to subsequent tests.

In [9]:
```
help(submitted.viterbi_ec)
```

```
Help on function viterbi_ec in module submitted:

viterbi_ec(train, test)
    Implementation for the improved viterbi tagger.
    input:  training data (list of sentences, with tags on the words). E.
g.,  [[(word1, tag1), (word2, tag2)], [(word3, tag3), (word4, tag4)]]
            test data (list of sentences, no tags on the words). E.g.,
[[word1, word2], [word3, word4]]
    output: list of sentences, each sentence is a list of (word,tag) pair
s.
            E.g., [[(word1, tag1), (word2, tag2)], [(word3, tag3), (word4,
tag4)]]
```

In [10]:
```
import time
importlib.reload(submitted)
train_set = utils.load_dataset('data/brown-training.txt')
dev_set = utils.load_dataset('data/brown-test.txt')
start_time = time.time()
predicted = submitted.viterbi_ec(utils.strip_tags(dev_set), train_set)
time_spend = time.time() - start_time
accuracy, _, _ = utils.evaluate_accuracies(predicted, dev_set)
multi_tag_accuracy, unseen_words_accuracy, = utils.specialword_accuracies

print("time spent: {0:.4f} sec".format(time_spend))
print("accuracy: {0:.4f}".format(accuracy))
print("multi-tag accuracy: {0:.4f}".format(multi_tag_accuracy))
print("unseen word accuracy: {0:.4f}".format(unseen_words_accuracy))
```

```
time spent: 28.8134 sec
accuracy: 0.9565
multi-tag accuracy: 0.9418
unseen word accuracy: 0.6550
```

Grade your homework

If you've reached this point, and all of the above sections work, then you're ready to try grading your homework! Before you submit it to Gradescope, try grading it on your own machine. This will run some visible test cases.

The exclamation point (!) tells python to run the following as a shell command. Obviously you don't need to run the code this way -- this usage is here just to remind you that you can also, if you wish, run this command in a terminal window.

For the visible test, if you get full points for the main parts (baseline and viterbi), you should get 50 points in total for completing baseline and viterbi functions. If you get full points for improving the viterbi, you should get 5 points in total for completing viterbi_ec function. The remaining 55 points are from the hidden tests on gradescope (50 for main parts + 5 for extra credit).

In [12]:
```
!python grade.py
```

```
{
    "tests": [
        {
            "name": "test baseline on Brown",
            "score": 20,
            "max_score": 20,
            "output": "time spent: 1.1647 sec; accuracy: 0.9388; multi-tag
accuracy: 0.9021; unseen word accuracy: 0.6782; ",
            "visibility": "visible"
        },
        {
            "name": "test viterbi on brown",
            "score": 30.0,
            "max_score": 30,
            "output": "start synthetic test; Synthetic test: passed!; time
spent: 28.9366 sec; accuracy: 0.9372; multi-tag accuracy: 0.9373; unseen w
ord accuracy: 0.2393; +10 points for accuracy, multi_tag_accuracy, unseen_
words_accuracy above [0.87, 0.87, 0.17] respectively.; +10 points for accu
racy, multi_tag_accuracy, unseen_words_accuracy above [0.9, 0.9, 0.19] res
pectively.; +10 points for accuracy, multi_tag_accuracy, unseen_words_accu
racy above [0.925, 0.925, 0.2] respectively.; ",
            "visibility": "visible"
        },
        {
            "name": "test viterbi_ec on Brown",
            "score": 5.0,
            "max_score": 5,
            "output": "time spent: 29.1041 sec; accuracy: 0.9565; multi-ta
g accuracy: 0.9418; unseen word accuracy: 0.6550; +1.5 points for accurac
y, multi_tag_accuracy, unseen_words_accuracy above [0.94, 0.92, 0.5] respe
ctively.; +1.5 points for accuracy, multi_tag_accuracy, unseen_words_accur
acy above [0.94, 0.92, 0.6] respectively.; +2 points for accuracy, multi_t
ag_accuracy, unseen_words_accuracy above [0.95, 0.932, 0.65] respectivel
y.; ",
            "visibility": "visible"
        }
    ],
    "leaderboard": [],
    "visibility": "visible",
    "execution_time": "63.74",
    "score": 55.0
}
```


Now you should try uploading `submitted.py` to Gradescope.

Gradescope will run the same visible tests that you just ran on your own machine, plus some additional hidden tests. It's possible that your code passes all the visible tests, but fails the hidden tests. If that happens, then it probably means that you hard-coded a number into your function definition, instead of using the input parameter that you were supposed to use. Debug by running your function with a variety of different input parameters, and see if you can get it to respond correctly in all cases.

Once your code works perfectly on Gradescope, with no errors, then you are done with the MP. Congratulations!