# Assignment 4

Due: Saturday 11/22/2024 @ 11:59pm EST

## Disclaimer

I encourage you to work together, I am a firm believer that we are at our best (and learn better) when we communicate with our peers. Perspective is incredibly important when it comes to solving problems, and sometimes it takes talking to other humans (or rubber ducks in the case of programmers) to gain a perspective we normally would not be able to achieve on our own. The only thing I ask is that you report who you work with: this is **not** to punish anyone, but instead will help me figure out what topics I need to spend extra time on/who to help.

## Setup: The Data

In this assignment you will build and improve a statistical BIO-tagging model for Named Entity Recognition (NER) on old tweets. This is a very difficult task: humans perform in the upper 60%s and machine performance ranges between 20-50%. Accompanying this pdf is a directory called `data` which contains all the data files for this assignment. The data files are:

- `train`: This file contains the training data for this task.
- `dev`: This file contains raw sentences (not trees) that will be our development data inputs. Your model will convert these sentences into trees.
- `test`: Ground-truth trees for each sentence in the development data.

Data has the following format:

```
Greek   B-other
Festival        I-other
at      O
St      B-facility
Johns   I-facility
before  O
ASPEN   B-geo-loc

Ew      O
wait    O
...
```

Blank lines mark sentence boundaries. Each nonblank line contains a word and its corresponding BIO-tag:

- `B-type` for the beginning of a named entity of type `type`
- `I-type` for a word inside a named entity of type `type`
- `O` for a word outside a named entity.

## Section: Starter Code

In addition to the `data` directory, there are quite a few code files included with this pdf. Some of these files are organized into sub-directories. The files and directories are:

- `eval/`: This directory contains all code files used to evaluate a model's performance. In this assignment we will use one flavor of performance: **f1** score:
  - `eval/conlleval.pl`: This file will be used to evaluate the f1 score of your model, both overall and as a function of each type of named entity. This script is written in *perl*, so please make sure you have perl installed on your machine.
  - `eval/eval_glue_script`: This executable file is used to help us call `conlleval.pl` from python.
  - `eval/do_eval.py`: This file provides an api that will run `conlleval.pl` as a backend.
- `from_file.py`: This file contains code for loading in the training/dev/test data.
- `layered_graph.py`: This file contains a special kind of graph that we will use in our viterbi algorithms: a graph whose topology is organized into layers. Each layer of this graph will be used in viterbi to handle the generation of a specific word, and each vertex in a layer will represent generating the word for the layer from a specific tag.
- `tables.py`: This file contains data types necessary for storing emission and bigram models, here represented as dense numpy arrays. This is to organize the tables inside a HMM to look more like they do on the lecture slides.
- `[part1/part2].py`: These files correspond to part1 and part2 of this assignment. I have implemented some of it for you so you can get the feel of how using the starter code should go. I also implemented these parts to provide more structure to the assignment. Some of the functions in these files I have left up to you. Even though there is a part 3 of this assignment, I have not provided you starter code for it. See the instructions below for more details.
- `models/`: This directory contains the meat of this assignment. It contains three files, all of which you will need to complete
  - `models/base.py`. Since a structured perceptron and a HMM have the same organization, those types will extend from a common class called `Base`. This class will contain some useful methods that will be shared by both structured perceptrons and HMMs: the most notable of which is `viterbi_traverse`.
  - `models/hmm.py`: This file contains a partial implementation of a HMM, where much of the boiler-plate code has been inherited through the `Base` class.
  - `models/sp.py`: This file contains a partial implementation of a structured perceptron. Most of the boiler-plate code has been inherited through the `Base` class, however there are some notable unique methods in this class: such as the `sp_training_algorithm` method.

## Part 1: Hidden Markov Model (45 points)

First, we will learn a HMM to solve this task. It will likely not perform very well.

a I have written code for you in `part1.parta` that will read in the training data, and calculates some summary statistics about the data: such as how many tokens, word types, and tag types there are. Include this in your report.

b I have written code for you in `part1.partb` which will try to train a HMM on the training data, however this code will not work until you complete the following methods:

- `Base.lm_count_bigram`. This method takes the tag corpus as an argument, and should generate the bigram counts (don't convert them into probabilities yet) by iterating through each sequence of the corpus and incrementing the correct entries into the pre-allocated bigram table.
- `HMM._train_lm`. This method takes the tag corpus as an argument, and after generating the bigram counts, should convert the counts into conditional probabilities.
- `HMM._train_tm`. This method takes the word corpus and the tag corpus as arguments, and instead of doing expectation maximization, should learn $Pr[w|t]$ by relative frequency estimation (i.e. counting and then normalizing). Be careful to deal with unknown words. We will also need to smooth $Pr[w|t]$. I found the model to be very sensitive to the smoothing, and I found add-0.1 smoothing to be the best.

Once you have those methods working, `part1.partb` should successfully train a HMM. It will then print out a bunch of probability values. Specifically the entries for:

```
p(B-person | O) =
p(B-person | B-person) =
p(I-person | B-person) =
p(B-person | I-person) =
p(I-person | I-person) =
p(O | I-person) =
```

and

```
p(God | B-person) =
p(God | O) =
p(Justin | B-person) =
p(Justin | O) =
p(Lindsay | B-person) =
p(Lindsay | O) =
```

Include these in your report.

c I have written code for you in `part1.partc` which will try to decode the dev data, but it will not work until you complete the following methods:

- `Base.viterbi_traverse`. This method will act like `cky_traverse` did from the previous assignment: it will only iterate over the data structures of a particular viterbi flavor. This method takes three arguments, the list of words, a function pointer that will produce an initialized `LayeredGraph` object (i.e. the data structure to populate), and another function pointer that will update a vertex within a layer of the `LayeredGraph` object. Your method is responsible for allocating new layers to the graph, and for iterating over each vertex within a layer.

- `HMM.viterbi`. This method will implement full viterbi decoding. Given a list of words as input, it should create two function pointers, one to initialize a `LayeredGraph` object, and another to update a layer within the object. You should call `viterbi_traverse` with these function pointers, and then assemble the most probable path through the graph. Your method should return the most probable path first, and the logprob of the most probable path second.

Once these methods are complete, `part1.partc` will generate a file called `generated/dev.out` which contains the following format:

```
Thinking O O
about O O
Bowlounge B-facility B-person
on O O
Saturday O O
night O O
. O O

sur O O
mon O O
...
```

The first column contains the words, the second column contains the *correct* tags, and the third column contains the *predicted* tags. In your report, include what your output looks like for the first five sentences of the dev data. This function will also run the evaluation script on your predictions, and will print out something like this:

```
processed 16261 tokens with 661 phrases; found: 931 phrases; correct: 77.
accuracy:  71.99%; precision:   8.27%; recall:  11.65%; FB1:   9.67
          company: precision:  85.71%; recall:  15.38%; FB1:  26.09  7
         facility: precision:  17.65%; recall:   7.89%; FB1:  10.91  17
          geo-loc: precision:  60.00%; recall:  15.52%; FB1:  24.66  30
            movie: precision:   0.00%; recall:   0.00%; FB1:   0.00  5
      musicartist: precision:   0.00%; recall:   0.00%; FB1:   0.00  67
            other: precision:   2.63%; recall:   8.33%; FB1:   3.99  419
           person: precision:  10.05%; recall:  21.64%; FB1:  13.73  368
          product: precision:   7.14%; recall:   2.70%; FB1:   3.92  14
       sportsteam: precision:  33.33%; recall:   1.43%; FB1:   2.74  3
           tvshow: precision:   0.00%; recall:   0.00%; FB1:   0.00  1
```

This important score is FB1 (balanced F-score), which across all types in 9.67%. Report your HMMs score on the dev data. It should be at least 9%.

## Part 2: Structured Perceptron (45 points)

In this section you will complete an implementation for a Structured Perceptron, which is structured much like a HMM. Remember, even though it is structured *like* a HMM, a SP is not a probabilistic model: it is more of a scoring function:

$$s(\mathbf{w}, \mathbf{t}) = s_t(t_n, < EOS >) + \sum_{i=1}^{n} \Big( s_t(t_{i-1}, t_i) + s_w(t+i, w_i) \Big)$$

Note that if you're using log-probabilities in your HMM, then your HMM viterbi implementation is identical to the viterbi implementation for a structured perceptron.

a I have written code for you in `part2.parta` which will try to train a SP on the training data, but it will not work until you complete the following method:

- `SP.sp_training_algorithm`. This method takes the word corpus and tag corpus as arguments and should implement the structured perceptron training algorithm. This method returns a pair of integers as output: the number of correctly predicted tags, and the number of total tags predicted.

Once you have completed this method, `part2.parta` will train the structure perceptron until it either converges with a convergence threshold of `1e-7`, or a max of 300 epochs happens. This make take quite a while (20 or so minutes), so feel free to play with these settings. In your report (and feel free to implement this in your code), tell me the following:

- How did you decide to stop the training process?
- Did you use any tricks like randomly shuffling the training data, or averaging weights?

b I have written code for you in `part2.partb` which will evaluate your models performance on the development data. Report the printout of your models performance in your report. Your FB1 score should be at least 15%.

## Part 3: Improvement (10 points)

Create a file called `part3.py`. In this file, you are welcome to recycle any code from `[part1,part2].py`, but be careful to follow the requested api:

a Try adding more features to your model. As long as each feature is a function of either $t_{i-1}$ and $t_i$, or of $t_i$ and $w_i$, you should be able to modify your viterbi algorithm to compute it. You're welcome to try other modifications other than adding features. In your report, describe each this you tried, and what its effect on training and development performance was.

b Create a function called `partb`. This function should take no arguments. It should load and run your best model on the *test* data and report your FB1 score. A small part of your grade will depend on how you did relative to the rest of the class.

*Suggestion*: I would save the bigram table and emission table to disk (they're numpy arrays so you can save numpy arrays to disk). I would save your best model to the same directory that `[part1,part2,part3].py` are located in, and turn your saved model in to gradescope. Your `part3.partb` function can load the model from disk instead of having to train a model from the training data on gradescope. This can save you a lot of time and potentially avoid a gradescope timeout!

## Submission

Please turn in all of the files that you modified to gradescope. In your report, you should include all of the items asked for by the questions. If an instruction in this prompt says to report something, you should include those elements in your pdf. Please do not handwrite these: typeset pdfs only. There will be separate links on gradescope for your code and your pdf. Your code will be autograded, while your report will be graded by us.