CS378 Assignment 3: Sequence Modeling and Parsing

Academic Honesty: Please see the course syllabus for information about collaboration in this course. While you may discuss the assignment with other students, **all code you write and your writeup must be your own!**

Goals There are two goals for this assignment. First, you will learn about structured inference techniques, including dynamic programming and beam search, and how to implement these efficiently. Second, you'll get experience looking at parse trees, analyzing syntactic phenomena, and reasoning about PCFGs.

Dataset and Code

Please use Python 3.5+ for this project.

The data is not freely available typically (it's licensed by the Linguistic Data Consortium), so the code and data bundle is only available on Canvas.

Data The dataset for this project is the Penn Treebank (Marcus et al., 1993). The Penn Treebank was originally split into 25 sections (0 through 24). The training set is sections 2 to 21, the dev set is section 22, and the test set is section 23. Sections 0, 1, and 24 are unused for historical reasons. We have already preprocessed these files: you are given training and dev set POS tagged sentences for Part 1. The tagged sentences are one word/tag pair per line with blank lines between sentences, and the trees are in the standard PTB bracket format. You are given readers for each, so you should not have to interact with these representations directly.

In Part 2, you can feel free to look at these trees, or you can come up with your own examples using the provided web demo.

Terminology Symbols in the grammar like S, NP, etc. are called *nonterminals*. Part-of-speech tags (NNP, VBZ, etc.) are a special type of nonterminal called *preterminals*. Actual words are called *terminals*. A tree therefore consists of a number of nonterminal productions of arity 1 or greater, n preterminals, and n terminals (leaves), where n is the number of words in the sentence.

Getting started Download the code and data. Expand the tgz file and change into the directory. To confirm everything is working properly, run:

```
python pos_tagger.py
```

This loads the data, instantiates a `BadTaggingModel` which assigns each word its most frequent tag in training data, and evaluates it on the development set. This model achieves 91% accuracy, since it can correctly tag all unambiguous words such as function words, so it's actually not a bad baseline!

Framework code The framework code you are given consists of several files. We will describe these in the following sections.

Part 1: Inference in Sequence Models (50 points)

In this part, you will experiment with inference in HMM part-of-speech taggers.

`pos_tagger.py` is the driver class. You should be familiar with the general structure by this point. The data pipeline involves calling `read_labeled_sents` from `treedata.py` to read POS tagged sentences out of the tagged data files (`train_sents.conll` and `dev_sents.conll`). `treedata.py` contains preprocessing and data reading code. You will be using `LabeledSentence`, which represents a sentence as a list of `TaggedToken` objects, each of which contains a string word and a string tag.

`models.py` contains two tagging models. `train_bad_tagging_model` trains an instance of `BadTaggingModel`, which assigns each word its most frequent tag in the training set (so "training" just entails counting word-tag pairs). `train_hmm_model` estimates parameters for the HMM and returns an instance of `HmmTaggingModel`.

Please read the comments in `HmmTaggingModel` to understand what is given to you as the output of the training procedure. Let $|U|$ denote the number of tags as in the Viterbi lecture note (to avoid colliding with the matrix T for transitions). Take note of the shape of each matrix: `init_log_probs` is a $|U| - 1$-length vector, `transition_log_probs` is a $|U| - 1 \times |U|$-sized matrix, and `emission_log_probs` is a $|U| - 1 \times |V|$-sized matrix.

Q1 Implement the Viterbi algorithm in the `viterbi_decode` function in `inference.py`. **Report performance in both accuracy and runtime. Your model must get at least 94% accuracy and evaluate on the development set in at most 250 seconds on a CS lab machine.**

Note that your model should be a correct implementation of the Viterbi algorithm: for an arbitrary `HmmModel` and a given sentence, it should return the highest probability path in all cases. You should not hardcode in anything specific to this particular tagset, English, or this particular HMM.

Q2 Implement beam search for the sequence model in the `beam_decode` function. This is used instead of Viterbi if you pass in the `--use_beam` argument.

`utils.py` contains a `Beam` class if you wish to use it. `Beam` maintains a set of at most `size` elements in sorted order by scores. Note that this implementation uses lists and binary search, meaning that it will not be as efficient as it would be if it used data structures like heaps. However, for most applications in NLP, particularly neural network models, manipulating the beam is not the code bottleneck, rather computing beam elements and their scores is.

Report performance in both accuracy and runtime for three to five different values of the beam size, particularly beam size 1. For beam size 3, you should get at least 94% accuracy and evaluate on the development set in at most 50 seconds. You should be able to get this working nearly as well as Viterbi depending on the beam size, so if you're seeing a major performance drop, you have a bug. For this part, we will only grade the results on beam size 3.

Autograder The autograder will run three tests. First, it will test the accuracy of **Viterbi (20 points)**. Second, it will test the accuracy of **beam search (20 points)**. Third, it will run a hidden test on a new HMM to check for correctness of **both Viterbi and beam search (10 points)**, using multiple different beam sizes including beam size 1. If you fail this third test, come up with a small example yourself and verify that your inference code is computing scores correctly and extracting the correct best path.

Part 2: Syntactic Parsing (50 points)

Q3 (25 points) In this part, you'll look at how parsers work in practice. You'll be using the Stanford CoreNLP dependency and constituency parsers[1] which are strong parsing models available as web demos. The representations should look familiar from examples in class, although the dependency parser produces *labeled* dependency trees. You can find documentation about the labels here: `https://universaldependencies.org/u/dep/`.

You can describe dependency trees by listing parent → child relationships in text or by including screenshots of these visualizations. If you really want to produce diagrams for the following question parts, consider using the `tikz-dependency` package.[2]

a) For the sentences *I ate spaghetti with chopsticks* and *I ate spaghetti with meatballs*, describe the following. (1) Which of the two interpretations does the **dependency parser** choose for each sentence? Is it correct? (2) Do the same analysis for the **constituency parser** and say whether it gets each sentence correct. (3) Are you surprised by the model's behavior here? Comment on what you see.

b) Consider the two sentences: *He likes stuffing* (where here we mean *stuffing* as the food item served at Thanksgiving) and *He likes stuffing his face with turkey*. Using the **dependency parser**, (1) report whether the analysis of each sentence is correct, and if incorrect, how it is incorrect. Be sure to look at both the structure and the POS tags. (2) How should the parents and children of *stuffing* differ in these two sentences?

c) Find a sentence that has at least five children for a single word in its **dependency parse**; perhaps try some different sentences to get a sense of how this might arise. Report the phrase involving that word and describe why this behavior arises here.

d) Find a new example that this **dependency parser** parses incorrectly. Include the example, its parse, and describe what is incorrect about the parse. Hint: you can think of what makes sentences ambiguous or try to find complicated sentences. You might have to look up meanings and usage of dependency labels to understand what they mean.

e) Construct (or find) an example of at least 8 words whose **constituency tree** starts as a balanced binary tree: the top production breaks the sentence exactly into two constituents of length $\frac{n}{2}$. Give the example and the top layer of the parse (the part that is balanced); you do not need to give the whole parse.

f) Construct (or find) an example of at least 8 words that contains at least 4 verbs[3] and the head of the sentence is the last word (ignoring punctuation). Give the example and indicate which are the verbs and which is the head. Use the dependency parser for this.

Q4 (15 points) Consider the following PCFG (bracketed numbers are probabilities):
NP → NP CC NP [0.3]
NP → NP PP [0.3]

[1] `https://corenlp.run/`

[2] `ctan.math.washington.edu/tex-archive/graphics/pgf/contrib/tikz-dependency/tikz-dependency-doc.pdf`

[3] MD or any tag starting with V. On other web demos, you may see AUX as well.

NP → NNS [0.4]
PP → P NP [1.0]

NNS → cats [1.0]
CC → and [1.0]
P → in [1.0]

Define a PCFG with these rules, the nonterminals {NP, PP, NNS, CC, P}, terminals {*cats, and, in*}, and root symbol NP.

For all question parts, provide justification so we can give partial credit as appropriate.

a) For the sentence *cats and cats*, how many valid syntactic parses (nonzero probability under this grammar) are there?

b) For the sentence *cats and cats in cats*, how many valid syntactic parses (nonzero probability under this grammar) are there?

c) The rules involving tag-word pairs in the grammar are called the lexicon. Suppose we smooth the lexicon so that all (word, tag) pairs have nonzero probability. For example, in this case [NNS → *and*] and [NNS → *in*] would be introduced to the grammar, as would similar extra rules for CC and P. Now, for the sentence *cats and cats*, how many valid syntactic parses (nonzero probability under this grammar) are there?

Q5 (10 points) Consider the following PCFG:
S → NP VP [1.0]
VP → V PP [1.0]
PP → P NP [1.0]
NP → NP CC NP [p]
NP → NNS PP [q] (note that this rule differs from the Q4 grammar)
NP → NNS [$1 - p - q$]

NNS → cats [1.0]
CC → and [1.0]
V → slept [1.0]
P → in [1.0]

Define a PCFG with these rules, the nonterminals {S, VP, PP, NP, NNS, CC, V, P}, terminals {*cats, and, slept, in*}, and root symbol S. As an example, this grammar can produce the sentence *cats slept in cats and cats in cats*. Two probabilities in this PCFG, p and q, are variables that we haven't defined yet. To have a legal PCFG, we must have $0 \leq p, q \leq 1$ and $p + q \leq 1$.

For all question parts, provide justification so we can give partial credit as appropriate.

a) If $p = 0$ and $q = 0$, what is the distribution over lengths of sentences produced by this grammar? Give your answer as a probability distribution over integers. What is its expected value?

b) If $p = 0$ and $q = 1$, what is the expected value of sentence lengths produced by this grammar?

c) If $q = 0$, what is the value of p that causes the expected NP length to diverge (i.e., no longer have a finite expected value)?

Deliverables and Submission

You will submit both your code and writeup to Gradescope. These are submitted as **two separate uploads** to Gradescope.

Written Submission You should upload to Gradescope a PDF or text file of your answers to the questions. This can be handwritten and scanned/photographed if that works best for you.

Note that you can submit the written assignment independently of the code. If you are unable to get the code fully working, please write up what you did and answer as many questions as possible, even partially, so we can assign you appropriate partial credit.

Please put your name on the assignment, and select a page with your name on it as the "question page" for any autograded question parts.

Code Submission Your code in `inference.py` will be evaluated by our autograder on its execution time and whether it meets the required accuracy values. **Note that you are uploading `inference.py` this time around, not `models.py`!**

Make sure that the following command works before you submit:

```
python pos_tagger.py --model HMM

python pos_tagger.py --model HMM --use_beam --beam_size 1

python pos_tagger.py --model HMM --use_beam --beam_size 3
```

References

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, June.