CS378 Assignment 3: Sequence Modeling and Parsing

Due date: Monday, March 9 at 11:59pm CST

Academic Honesty: Reminder that assignments should be completed independently by each student. See the syllabus for more detailed discussion of academic honesty. Limit any discussion of assignments with other students to clarification of the requirements or definitions of the problems, or to understanding the existing code or general course material. Never directly discuss details of the problem solutions. Finally, you may not publish solutions to these assignments or consult solutions that exist in the wild.

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 syntax data, analyzing syntactic phenomena, and thinking about design decisions in parsers.

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 both project parts 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) and the raw Wall Street Journal (WSJ) files (for Part 2). 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. Also, you'll just be looking at the trees here in Part 2, so the train/dev/test splits aren't really important to observe here, but we've kept the data like this anyway.

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 HMM part-of-speech taggers. You will need to interact with several elements of the framework code to make this happen.

`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.** You will complete the definition of this class to support decoding in two different ways.

Q1 (25 points) Implement the Viterbi algorithm in the `viterbi_decode` function. **(1) Briefly describe anything interesting about your implementation,** such as any particularly important runtime optimizations you made. **(2) 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.**

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 (25 points) 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.

(1) Briefly describe your implementation. (2) Report performance in both accuracy and runtime for three to five different values of the beam size, particularly beam size 1. For some beam size, you should get at least 94% accuracy and evaluate on the development set in at most 100 seconds. (3) Describe qualitatively how beam search compares to Viterbi overall in terms of accuracy and runtime. 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. Hardcode into your final solution what you feel your best "overall" (speed and runtime) beam size satisfying the requirements.

Again, your model should be a correct implementation of the beam search procedure: up to numerical stability and ties, it should execute the algorithm as specified with the given beam size.

Part 2: Syntactic Parsing (50 points)

In this part, you will be interacting directly with parse trees and getting experience with constituency parsing.

Looking at the data `treedata.py` contains a loader for the Penn Treebank, which reads the additional `alltrees_dev.mrg.oneline` file that is provided (a reformatted version of the section 22 development set). The main function will load in the treebank using the recursive `Tree` data structure and print ten trees. `render_pretty` prints a tree in a nicely indented format, which you may find easier to look at than raw treebank trees when making sense of the data.

Q3 (10 points) Understanding the syntactic function of grammar elements in context is crucial to designing good parsers. In this part, you should analyze the Wall Street Journal dataset using both code and looking at it manually. You can either use the raw data files or use the main function of `treedata.py` as described above to view and manipulate the data.

a) What are the three most common parents of PP in the WSJ dataset?

b) Look at the distribution of prepositional phrases based on their parent: they might differ by length, by the preposition they contain, or by syntactic properties further down in the PP. Comment on **at least two** differences you notice.

Q4 (20 points) In this part, you'll look at how dependency parsers work in practice. Use the AllenNLP dependency parser,[1] which is available as a web demo. Note that this demo seems to not work in Safari but should work in other browsers. The demo arranges parses with words in rectangles labeled to include their POS tag. Trapezoids with labels indicate arc labels and parent-child relationships. Unlike the diagrams we've seen, words are not necessarily kept in sentence order, though you can mouse over words to see where they appear in the sentence.[2]

You can describe dependency trees by listing parent $\rightarrow$ child relationships in text. If you want to produce diagrams for the following question parts, consider using the `tikz-dependency` package.[3]

a) For the sentence *Teacher strikes idle kids*, which of the two interpretations does the parser choose? Why might this be?

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*. What does the parser do with these two sentences? Are the analyses correct? How should the parents and children of *stuffing* differ in them?

c) Find a sentence that has at least five children for a single word (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.

[1] `https://demo.allennlp.org/dependency-parsing`

[2] Note that this demo has a bug where it sometimes parses a sentence out of order; check that the token order the parser is using is correct, and don't count this as an error for part (d) as it's simply a bug in the web tool. If this happens for a sentence you want to parse, feel free to use the Stanford parser instead: `http://nlp.stanford.edu:8080/parser/`

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

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. Some of these labels are documented in the universal dependencies documentation.[4]

Q5 (10 points) Consider the following PCFG (bracketed numbers are probabilities):
NP $\rightarrow$ NP CC NP [0.3]
NP $\rightarrow$ NP PP [0.3]
NP $\rightarrow$ NNS [0.4]
PP $\rightarrow$ P NP [1.0]

NNS $\rightarrow$ cats [1.0]
CC $\rightarrow$ and [1.0]
P $\rightarrow$ 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 $\rightarrow$ *and*] and [NNS $\rightarrow$ *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?

Q6 (10 points) Consider the following PCFG:
S $\rightarrow$ NP VP [1.0]
VP $\rightarrow$ V PP [1.0]
PP $\rightarrow$ P NP [1.0]
NP $\rightarrow$ NP CC NP [p]
NP $\rightarrow$ NNS PP [q] (note that this rule differs from the Q5 grammar)
NP $\rightarrow$ NNS [$1 - p - q$]

NNS $\rightarrow$ cats [1.0]
CC $\rightarrow$ and [1.0]
V $\rightarrow$ slept [1.0]
P $\rightarrow$ in [1.0]

[4] `https://universaldependencies.org/u/dep/`

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 distribution over lengths of sentences produced by this grammar? What is its expected value?

c) If $p = 0$ and $q = 0.5$, what is the distribution over lengths of sentences produced by this grammar? What is its expected value?

d) Now consider just the set of NPs that can be produced starting from a root NP symbol. 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

Your submission will be evaluated on several axes:

1. Writeup: correctness of answers, clarity of explanations, etc.
2. Execution: your code should train and evaluate within the time limits without crashing and give the required accuracy values

Submission You should submit the following files to Canvas **as a flat file upload (no zip or tgz)**:

1. A PDF or text file of your answers to the questions
2. `models.py`. **Do not modify or upload any other source files. If you change the Beam implementation or anything like that, copy yours into `models.py` and use it there instead.**

There is no code submission for Part 2. You do not need to include any blind test output this time around.

Make sure that the following commands work before you submit:

```
python pos_tagger.py --model HMM

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

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.