CS378 Assignment 3: Sequence Modeling and Parsing

Due date: Monday, March 11 at 5:00pm 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 discuss issues directly relevant to 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.

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. (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.**

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 more tuned data structures. 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.

Part 2: Syntactic Parsing (50 points)

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

Setup For Q4, you need to be able to run the backend parser, which is in Java. This should work out of the box on the CS lab machines. On your personal machine, you can download and install the latest Java Runtime Environment (JRE 1.8+) to run the command below.

The parser you will be using is implemented in Java courtesy of UC Berkeley. Making parsers fast is challenging; optimized implementations in Python need to use Cython and managing a large grammar can be slow. Even this reasonably efficient implementation may take 10-20 minutes to parse the whole development set depending on the grammar size.

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) Comment on two differences you notice in the distribution of prepositional phrases based on their parent. That is, how do PPs with one parent overall tend to differ from those with a different parent?

Q4 (20 points) In this part, you'll play around with a grammar annotation-based parser. To get started, run:

```
java -cp cs378-parser.jar edu.berkeley.nlp.assignments.parsing.PCFGParserTester\
  -path data/wsj/ -h 0 -v 0
```

You do not need to modify this command. This will load in the data from the `wsj/` directory, do some standard tree preprocessing (stripping out category modifiers like -TMP and empty elements like -NONE-), binarize the trees, estimate a PCFG, and parse the development set. The model prints out `[Current]` evaluation numbers for each sentence and an `[Average]` value at the end. The code takes the raw `wsj` directory and does the dev and test splitting for you, so you do not need to worry about that data handling at all. Parsing the development set will take a few minutes. To run quicker experiments, use `-maxTestLength 20` to cause evaluation to only run on trees of length 20.

(a) Try varying `h` and `v` to use grammars with more refined horizontal and vertical Markovization. Report values for at least three settings of these **on the entire development set**.[1]

(b) Describe the relative importance of these two types of Markovization: which one seems more critical for high performance? Give one intuitive reason this might be the case.

(c) Consider the second sentence in the development set: *The bill intends to restrict the RTC to Treasury borrowings only , unless the agency receives specific congressional authorization .*

[1] Parsing only short sentences can give you a handle on the process but gives a skewed sense of results overall, since short sentences are easier to parse.

Look at the parses of this sentence with $h = 0, v = 0$ and with $h = 2, v = 2$. First, compare each of these trees with the gold parse and describe the differences as intuitively as you can (how does the sentence interpretation change?). Compare the parses with each other and describe why you think $h = 2, v = 2$ gives better results in this case. (For help interpreting the categories, you can refer to a guide courtesy of Mihai Surdeanu.[2] Googling a grammatical concept will usually surface an explanation of what it is with examples.)

Q5 (10 points) Consider the following PCFG (bracketed numbers are probabilities):
NP $\rightarrow$ NP CC NP [0.4]
NP $\rightarrow$ NP PP [0.4]
NP $\rightarrow$ NNS [0.2]
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]

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*

[2] http://www.surdeanu.info/mihai/teaching/ista555-fall13/readings/PennTreebankConstituents.html

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.