Midterm project [updated]

Ethan Poole
LING 185A: Comp. Ling. I
Due: 11 May 2020

Instructions: Download `Midterm.zip` from the course website and unarchive its contents into the same directory on your computer. The import lines at the top of `Midterm.hs` import definitions from `ProbSLG.hs` and `Helpers.hs`, which you may then use in your answers. Before starting to code any of your answers, you should familiarize yourself with the contents of these two files.

`Checker.hs` imports your functions from `Midterm.hs`, passes corpus data through them, and prints the output. The functions in this file are for you to use in part 3; see below for details. Please do *not* spend time trying to understand the code in this file, as it involves concepts beyond the scope of this class!

Please submit your midterm as two files: a PDF file with your write up and a modified version of `Midterm.hs`.

Note: To make the instructions more readable, I do not include the type signatures here, but they are provided in `Midterm.hs`.

Introduction

Natural-language processing (NLP) involves using concepts from computational linguistics to work with raw linguistic data, such as text and audio. In NLP, a crucial prerequisite to performing any syntactic or semantic analysis is PART-OF-SPEECH TAGGING (POS-tagging). For example, in order to parse or interpret the sentences in (1), the program needs to know that *catfish* is a verb in (1a) and a noun in (1b). In NLP-terms, *catfish* needs to be "tagged" as a verb in (1a) and as a noun in (1b).

(1) a. Alex wanted to catfish someone.

b. Alex wanted to eat catfish.

The most prominent approach to POS-tagging is to utilize the immediately surrounding context in which a word occurs, in particular bigram sequences. For example, to figure out that *catfish* is a verb in (1a), the program might utilize the fact that tense markers like *to* are typically followed by verbs. This kind of procedure scales up and can provide surprisingly good accuracy.

Formally, this approach to POS-tagging utilizes a PROBABILISTIC STRICTLY-LOCAL GRAMMAR (PSLG), as defined in (2) and (3). Recall that an SLG encodes transitions that are bigram sequences. In a PSLG, those transitions, the starting symbols, and the final symbols are all associated with probabilities.

(2) A probabilistic strictly-local grammar is a four-tuple $\langle \Sigma, \text{start}_P, \text{fin}_P, \Delta_P \rangle$ where:

   a. Σ, the alphabet, is a finite set of symbols;

   b. $\text{start}_P : \Sigma \to [0,1]$ is the starting-symbol function;

   c. $\text{fin}_P : \Sigma \to [0,1]$ is the final-symbol function;

   d. $\Delta_P : (\Sigma \times \Sigma) \to [0,1]$ is the transition function;

   e. $\sum_{x \in \Sigma} \text{start}_P(x) = 1$; and

   f. for all $x \in \Sigma$, $\text{fin}_P(x) + \sum_{y \in \Sigma} \Delta_P(x,y) = 1$

(3) Given a PSLG $G = \langle \Sigma, \text{start}_P, \text{fin}_P, \Delta_P \rangle$, the probability of generating the string $x_1 x_2 \ldots x_n$ is:

$$\text{start}_P(x_1) \times \Delta_P(x_1, x_2) \times \cdots \times \Delta_P(x_{n-1}, x_n) \times \text{fin}_P(x_n)$$

In this midterm project, you will be building your own POS-tagger using PSLGs. That is the end goal, but in what follows, I have broken down the steps to that goal into smaller chunks.

1 *8 points*

First, you will need some basic functions to work with PSLGs. Please code up the following functions:

(4) a. `follows` such that `follows g x` returns the list of characters that can follow `x`, paired with the probability of such, according to the PSLG `g`.

Example usage:
`follows g1 "very"` $\Longrightarrow^*$ `[("very", 0.3), ("fat", 0.7)]`

b. `precedes` such that `precedes g x` returns the list of characters that can precede `x`, paired with the probability of such, according to the PSLG `g`.

Example usage:
`precedes g1 "very"` $\Longrightarrow^*$ `[("the", 0.2), ("very", 0.3)]`

c. `valP` such that `valP g str` returns the probability of `str` according to the PSLG `g`. Note that if a (P)SLG generates a string, then it does so in exactly one way, i.e. there are no derivational ambiguities, unlike FSAs.

Example usage:
`valP g1 ["the", "cat"]` $\Longrightarrow^*$ `0.25`

d. `valP'` such that `valP' g str` returns the probability of the substring `str` according to the PSLG `g`, i.e. ignoring whether the first and last characters of `str` are possible start and final symbols in `g`.

Example usage:
`valP' g1 ["very", "fat", "cat"]` $\Longrightarrow^*$ `0.35`

2 *5 points*

Probabilistic grammars are only useful insofar as they encode meaningful probabilities (unlike our toy examples in class). The standard way, especially in NLP, to furnish probabilistic grammars with meaningful probabilities is to use a corpus. A corpus is a collection of texts, audio, or video, which is typically naturally-occurring data, e.g. from newspapers and books.

Your next task is to write a Haskell function that builds a PSLG from a corpus:

(5) `buildProbSLG` such that `buildProbSLG corpus` returns a PSLG that is trained on `corpus`. That is, the PSLG's starting symbols, final symbols, transitions, and probabilities should be computed from the data in `corpus`.

In the resulting PSLG:

- the probability associated with a starting symbol s should be the likelihood that a sentence starts with s in the corpus;
- the probability associated with a final symbol f should be the likelihood that a sentence ends with f in the corpus[1]; and
- the probability associated with a transition (x, y) should be the likelihood that x is followed by y in the corpus.

Several easy-to-work-with toy corpora are provided in `Helpers.hs`.

Code notes: Corpora are represented with the type `(Corpus a)`, which is a list of expressions of type `(Sentence a)`, which themselves are lists of expressions of type `a`. PSLGs are represented with the type `ProbSLG`. Note that `ProbSLG` has a single value constructor that is also named `ProbSLG`, which just "encases" an ordinary tuple. This is a common design pattern in Haskell. I have used it here only so that I could provide a custom implementation of the **`show`** function for `ProbSLG` which is easier to read.

[1] Strictly speaking, this is a simplification, if you note (2f).

3 *10 points*

With those pieces in place, you are now in a position to build your own POS-tagger! In NLP, POS-taggers are trained on already-tagged corpora, i.e. corpora that have been manually tagged for parts-of-speech (traditionally by hand!). The *Brown Corpus* is one of the more famous corpora tagged for parts-of-speech, and it is the one that you will be working with here (technically, a very small subset of it). See the course website for a link to the full list of tags used in the Brown Corpus.

A POS-tagger works by using the already-tagged corpus to build a PSLG (or other kind of probabilistic grammar). That PSLG can then be used to assign parts-of-speech to the words in raw text. In other words, the PSLG can be used to predict what the parts-of-speech may be.

Your final coding task then is to build your own simple POS-tagger by coding up the following functions:

(6) a. `posProbSLG` that takes a parsed corpus of tagged words (type `Corpus TaggedWord`) and returns a PSLG for the POS-tags alone.

b. `tag` such that `tag corpus str` returns a list of possible fully-tagged versions of `str`, each associated with a probability, according to a PSLG trained on the corpus in `corpus` (i.e. run `corpus` through `posProbSLG`).

c. `tagBest` such that `tagBest corpus str` returns the most probable fully-tagged version of `str`, according to a PSLG trained on the corpus in `corpus` (i.e. run `corpus` through `posProbSLG`).

Code notes: A tagged word is represented with the type `TaggedWord`. Like `ProbSLG`, `TaggedWord` has a single value constructor named `TaggedWord`, which encases an ordinary tuple, whose first member is a word and whose second member is a part-of-speech tag.

Accuracy requirement: To receive full credit, your POS-tagger must achieve an accuracy level of at least 35% on the portion of the Brown corpus included in `Midterm.zip`.

In `Checker.hs`, I have provided several test functions that apply data from the Brown Corpus to your POS-tagger. (I/O in Haskell is complicated; hence why I have provided these functions). Run `ghci Checker.hs` to use the test functions:

- `checkPOS` checks `posProbSLG`.
- `checkTag` checks `tag`.
- `checkTagBest` checks `tagBest`.
- `checkAccuracy` checks the word-by-word accuracy of your tagger using `tagBest`.

There will be a variety of choices that you will have to make in developing your POS-tagger. For example, how does it start off? Does it really need to use the PSLG to tag *the* and other functional words? Here, there is room for creativity in how you approach these issues, and there is no single right answer.

Finally, it may be the case that you want to "clean up" or *sanitize* the corpus data, e.g. removing some punctuation or collapsing some tag distinctions. If you do so, you will need to code those sanitation procedures as functions and add them to `sanitize` (which, by default, is empty). `checkAccuracy` will apply your sanitation functions before feeding the corpus into your POS-tagger.

4 *5 points*

Your last task is to do a write-up about your POS-tagger. Explain how your POS-tagger works, what choices you made about its operation, etc. Discuss ways in which you think it could be improved. Your write-up should be between one to two pages of *prose*. You are more than welcome to—and in fact should—include relevant code snippets in your write-up to complement your prose, but they do not count towards the length requirement.