Final Project 2: Part-of-Speech Tagging

Due: Fri. 6/11/2021, 11:59pm Pacific

Natural Language Processing (NLP) involves using concepts from computational linguistics to work with raw linguistic data, such as text and audio. One common task in NLP is performing part-of-speech tagging ("POS-tagging"), as a prerequisite to any form of syntactic or semantic analysis. For example, in order to correctly parse the sentences *Time flies like an arrow* and *Fruit flies like a banana*, the program needs to know that the words *flies* and *like* have different grammatical categories in these two sentences. In NLP terms, these words need to be "tagged" as different parts of speech.

The most prominent approach to POS-tagging is to use the immediately surrounding context in which a word occurs, in particular bigram sequences. Formally, this looks a lot like the probabilistic variants of SLGs and FSAs that we learned about in Week 9. In this final project, you will try your hand at coding up a POS-tagger using a probabilistic strictly-local grammar (PSLG).

Instructions: Download `FinalProject2.zip`, unarchive its contents into the same directory on your computer, and rename `FinalProject02_Stub.hs` to `FinalProject02.hs`. (Please use this name exactly.) The import lines at the top of the stub file import definitions from `ProbSLG.hs` and `Brown.hs`, so you can use them exactly as you would if they were defined in the same file.

Please submit your project as two files on CCLE: a PDF with your write-up, and a modified version of `FinalProject02_Stub.hs`. Do not modify or submit `ProbSLG.hs` or `Brown.hs`.

Notes:

Have a look at `ProbSLG.hs`. Here are a few things to notice:

- Our Haskell type for PSLGs is defined as `type ProbSLG sy`. This is a four-tuple consisting of the following elements: (1) a list of symbols Σ; (2) a list of ordered pairs I, where the first element is a starting symbol and the second element is its starting probability; (3) a list of ordered pairs F, where the first element is a final symbol and the second element is its final probability; and (4) a list of triples T, consisting of two symbols and the probability of transitioning from the first to the second symbol.
- Unlike the toy examples that we've been using in class, the standard way in NLP to furnish a probabilistic grammar with meaningful probabilities is to use a corpus: a collection of naturally-occurring texts, e.g. from newspapers, books, or websites. Our Haskell type for corpora is type `Corpus a`, which is a list of expressions of type `Sentence a`, which themselves are lists of expressions of type `a`.
- In NLP, POS-taggers are typically trained by using a corpus that has already been manually tagged for parts-of-speech (traditionally by hand!). Our Haskell type for a tagged word is

type `TaggedWord`. Note that `TaggedWord` has a single value constructor that is also named `TaggedWord`, which just encases an ordinary tuple. The first member of this tuple is a word (type `String`) and the second member is a part-of-speech tag (also type `String`).

- I've provided a few examples of easy-to-work-with mini-corpora of various different types. The already-tagged corpora have type `Corpus TaggedWord`. There is also an example PSLG (`g1`), which roughly models the same distribution over sentences as in `corpus3`, but without the corresponding part-of-speech tags.
- I've provided some simple helper functions (`divide`, `getTag` and `getWord`) for help in working with probabilities and `TaggedWords`. Take a look at what they do.

No need to worry about the content of `Brown.hs` for now; this will be explained in Part 3.

1 Warm-up: Working with PSLGs (6 points)

Recall that a probabilistic strictly-local grammar (PSLG) is a four-tuple (Σ, I, F, T) where:

- Σ, the alphabet, is a finite set of symbols;
- $I : \Sigma \to [0, 1]$ is a function assigning initial probabilities, where $I(x) = P(X_1 = x)$;
- $F : \Sigma \to [0, 1]$ is a function assigning final probabilities, where

$$F(x) = P(X_{i+1} = \textsc{stop} \mid X_i = x); \text{ and}$$

- $T : \Sigma \times \Sigma \to [0, 1]$ is a function assigning transition probabilities, where

$$T(x, y) = P(X_{i+1} = y \mid X_i = x)$$

with the additional requirements that:

- $\sum_{x\in\Sigma} I(x) = 1$; and
- for all $x \in \Sigma$, $\sum_{y\in\Sigma} T(x, y) + F(x) = 1$.

To start, here are a few warm-up functions to get used to working with PSLGs. Notice that our type `ProbSLG sy` represents I, F, and T as lists of tuples, rather than as full functions. Additionally, we won't be using any special STOP symbol in our corpora, so you can think of $F(x)$ as representing the conditional probability of a string ending after the i-th symbol, given that the i-th symbol is x.

A. Write a function `follows :: Ord sy => ProbSLG sy -> sy -> [(sy, Double)]` that takes a PSLG and a symbol, and returns a list of symbols that can follow the given symbol, paired with the probability of those transitions in the given PSLG. If nothing can follow the given symbol, the result should be the empty list.

```
*FinalProject02> follows g1 "very"
[("very",0.3),("fuzzy",0.7)]
*FinalProject02> follows g1 "mat"
[("on",0.2)]
*FinalProject02> follows g1 "the"
[("cat",0.3),("very",0.2),("fuzzy",0.2),("mat",0.3)]
```

B. Write a function `bigrams :: [a] -> [(a,a)]` that converts a list of any type into a list of its bigrams.

```
*FinalProject02> bigrams [1,4,6,7,3]
[(1,4),(4,6),(6,7),(7,3)]
*FinalProject02> bigrams ["the", "very", "fuzzy", "cat"]
[("the","very"),("very","fuzzy"),("fuzzy","cat")]
*FinalProject02> bigrams ["cat"]
[]
```

C. Write a function `valP :: Ord sy => ProbSLG sy -> [sy] -> Double` that returns the probability of a given list of symbols under a given PSLG. Note that if a (P)SLG generates a string, then it does so in exactly one way; there are no derivational ambiguities, unlike for FSAs.

```
*FinalProject02> valP g1 ["the", "cat"]
0.24
*FinalProject02> valP g1 ["the", "fuzzy", "cat"]
9.6e-2
```

2 Building a PSLG from a corpus (4 points)

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

D. Write a function

```
buildProbSLG :: (Ord a, Eq a) => Corpus a -> ProbSLG a
```

which takes a corpus, and returns a PSLG that is trained on the given corpus. For example, `buildProbSLG corpus1` should return a `ProbSLG` whose symbols, starting symbols, final symbols, transitions, and probabilities are computed from the data in `corpus1`.

3 POS-tagging (40 points)

You're now ready to start building your POS-tagger! In NLP, a POS-tagger typically works by using an already-tagged corpus to build a PSLG (or other kind of probabilistic grammar) whose symbols are the part-of-speech tags in the training corpus. That PSLG can then be used to predict the part-of-speech tags for words in new, untagged corpora.

The Brown Corpus (`https://en.wikipedia.org/wiki/Brown_Corpus`) is one of the more famous corpora tagged for parts-of-speech. In `Brown.hs`, I've provided two very small portions of this corpus, parsed as our type `Corpus TaggedWord`. One of these "mini-corpora," `brownDev`, is the corpus that you will be using to train your POS-tagger— i.e., the corpus that you will use to build a PSLG to model the probabilities of sequences of part-of-speech tags. The second "mini-corpus," `brownTest`, is the one that you will use to test your POS-tagger. That is, when your tagger is working, it should be able to assign part-of-speech tags not only to the words of the corpus that it was trained on, but also to some proportion of the words in this new test corpus.

We'll break this down into a few steps, starting with a few simple helper functions.

E. Write a function

```
posProbSLG :: Corpus TaggedWord -> ProbSLG String
```

which takes a parsed corpus of tagged words (type `Corpus TaggedWord`) and returns a PSLG that models the probabilities of the part-of-speech tags alone. For example, `posProbSLG corpus3` should return a `ProbSLG String` whose symbols are `"D"`, `"N"`, `"Adj"`, `"Adv"`, and `"P"`.

F. Write a function `sanitize :: Corpus TaggedWord -> Corpus TaggedWord` which takes a parsed corpus of tagged words and removes punctuation, retaining only the lexical items (i.e., the symbols that actually have grammatical categories). For example, a symbol like `,` should be removed, but it would be best to keep the `'s` that is part of English possessives. You can use the punctuation tags in the Brown corpus as a guide here.

G. Write a function `mostFrequentTag :: Corpus TaggedWord -> String -> String` which takes a parsed corpus of tagged words and a given word, and returns the most frequent tag in the corpus for that word.

```
*FinalProject02> mostFrequentTag corpus5 "cat"
"B"
*FinalProject02> mostFrequentTag corpus5 "the"
"A"
```

H. Now here's the fun part! Write a function

```
tag :: Corpus TaggedWord -> [String] -> [(Sentence TaggedWord, Double)]
```

which takes a parsed corpus of tagged words and a sentence (list) of not-yet-tagged words, and returns a list of possible fully-tagged versions of the given sentence, each paired with a probability according to a PSLG trained on the given corpus. That is, the probability associated with each possible fully-tagged version of the given sentence should come from the PSLG that would be created by running the corpus through your function `posProbSLG`. Because a sentence of type `Sentence TaggedWord` is just a list of individual `TaggedWords`, the result of this function is a list of pairs consisting of a `Sentence TaggedWord` and the probability of that sentence under your trained PSLG.

I. Write a function

```
tagBest :: Corpus TaggedWord -> [String] -> Sentence TaggedWord
```

which takes a parsed corpus of tagged words and a sentence of not-yet-tagged words, and returns the fully-tagged version of the given sentence that has highest probability under the PSLG trained on the given corpus.

J. Write a function

```
testAccuracy :: Corpus TaggedWord -> Corpus TaggedWord -> Double
```

which takes a training corpus of tagged words, and a test corpus of tagged words with the same part-of-speech tags, and calculates the accuracy of your POS-tagger when trained on the training corpus and tested on the test corpus. Specifically, this function should make use of `tagBest` to attempt to tag the first five words[1] of every sentence in the test corpus with a PSLG trained on the training corpus. Then, for each one of these words, the function should compare the tag that your POS-tagger predicted against the actual tag in the test corpus, and compute the proportion of tags that it got correct.

In order to receive full credit, your POS-tagger should achieve at least 20% accuracy when trained on a "sanitized" version of `brownDev` and tested on a "sanitized" version of `brownTest` (i.e., versions of these mini-corpora with punctuation removed). In other words, you should find an accuracy score of 0.20 or higher when you run `testAccuracy sanitize(brownDev) sanitize(brownTest)`.

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— what tag should it assign to the first word in the sentence? Do you want to collapse any of the tag distinctions in the Brown corpus, or do any extra form of "sanitizing" beyond removing punctuation? There is room for creativity in how you approach these issues, and there is no single right answer.

K. Finally, please write up a short (2-4 page, single-spaced) explanation of how your POS-tagger works, and what choices you made in implementing it. As part of your write-up, you should address the following questions:

- Are there any systematic errors that your POS-tagger makes for certain words or certain types of words? Why do you think that it makes these errors, and what do you think could be done to improve them?
- Try running `testAccuracy sanitize(brownDev) sanitize(brownDev)` — you'll probably find that your accuracy has gone up. Why do you think this is? What insights do we gain from training and testing on separate portions of the corpus?
- How might you extend your existing POS-tagger to use a probabilistic *finite-state automaton*, rather than a probabilistic SLG? How do you think this would affect the accuracy of your model, and why? There's no need to actually try this— the point is simply to think about what would be involved.

[1] We're restricting the evaluation to the first five words of each sentence in order to speed things up. But you might still notice that this still takes a rather long time, unless you've done some form of memoization . . .