Homework 4: Transition-Based Dependency Parser

Due Tuesday March 14th, 2023 at 11:59pm

Introduction and Task

In this homework, you will be implementing components of a transition-based dependency parser, specifically methods to:

- determine the projectivity of a dependency parse (Question 1)
- carry out the `SHIFT` operation (2.a)
- carry out the `LEFTARC` and `RIGHTARC` operations (2.b)
- synthesize these operations by combining them into the set of correct configurations and actions for the parser (2.c)

The graded deliverables for this assignment are found in Questions: [**1, 2.a, 2.b, 2.c, 3**]. They will be weighted roughly equally, ~20 points per component.

Questions 1 and 2.a-c are programming questions. Question 3 is a written question prompting you to display your understanding of the differences between dependency and phrase-structure grammars.

The *optional* Bonus sections do not contain graded deliverables. They allow you to further explore how a transition-based parser can be configured and utilized for a real-world task, specifically:

- take in a dependency and carry out updates based on optimal action predictions
- a simplified implementation of A Fast and Accurate Dependency Parser using Neural Networks.

To start, let's explore

- dependency relations
- how NLP researchers format dependency-parsing data, and
- review transition-based parsing

Dependency Relations, Universal Dependencies Format and Transition-Based Dependency Parsing

Dependency Relations

In a dependency grammar, we establish binary relations between words, codifying connections by tagging a word as either a **head** or a **dependent**.

The head (also referred to as the **root**) is the core component of the sentence. Dependents, meanwhile, are classified based on the grammatical function they carry out with respect to the head (such as *indirect object*).

Universal Dependencies Format

This information should assist you with completing the different portions of Question 1.

The dependency parsing data we'll work with in this assignment are sourced from the Universal Dependencies (UD) project (Nivre et al., 2020). The `.colnll` files we'll hand off to the neural parser (down in the bonus section) are the result of this research and will power our training pipeline.

Here is what the UD entry for our example sentence looks like:

```
1     From  from  ADP   IN    _     3     case
2     the   the   DET   DT    _     3     det
3     AP    AP    PROPN NNP   _     4     obl
4     comes come  VERB  VBZ   _     0     root
5     this  this  DET   DT    _     6     det
6     story story NOUN  NN    _     4     nsubj
7     :     :     PUNCT :     _     4     punct
```

The first column indexes each token in the sentence. This must be greater than 0.

The second column lists the original form of the word, with the third column listing its lemmatized form.

The fourth lists the part-of-speech tag in UPOS format. The fifth column lists language parts-of-speech tags, if applicable. The subsequent numerical column lists the HEAD of the word, either pointing to an ID (denoting that the word with that ID is the current word's HEAD) or `0` (denoting tha the current word is the HEAD).

The final column contains the dependency relation to the HEAD in UDEP format. If the current word is the HEAD, this will take the value of `root`.

For the methods you need to program, the most salient columns are:

- token ID
- original token form
- part-of-speech
- ID of the head
- dependency relation

Intuitively understanding these different values will be helpful for evaluating the projectivity of a given parse. You can detect non-projective arcs based on the location of a given word's head relative to other arcs in a sentence.

Transition-Based Dependency Parsing

Note: this section draws heavily from SLP §18.2, which is worth reviewing in close detail before proceeding with the assignment.

This information should assist you with completing the different portions of Question 2.

The **stack** is what we'll build our parse on. The **buffer** is the group of tokens that have yet to be parsed. Both of these structures, in our code, will simply be lists containing the ID of the tokens we're working with. The stack is called `stack` and the buffer is called `wbuffer`.

Python-wise, you can consider the 'top' item in the stack to be `stack[-1]` and the 'second' item in the stack to be `stack[-2]`.

The **configuration** is the state of the parse at a given decision-point. In our code, this will be represented as a list called `configurations`, containing, per-entry, the buffer, the stack and the dependency tree (as a 3-element tuple).

The dependency tree itself is a condensed indicator of one token's dependency parse, and is a list called `arcs` containing, per-entry: its dependency relation, its head token's ID, and its own token ID (as a 3-element tuple).

At each decision-point, we have a few different options of which **transition** to apply to build the parse. We'll keep track of these transitions in our code using `gold_transitions`.

At a high-level, this is what carrying out a transition operation should look like:

1. append the latest configuration to `configurations`.
2. append the latest action to `gold_transitions`.
3. update `wbuffer`, `stack` and `arcs` accordingly with the proper indices + dependency relations.

The order of operations matters, as we want to capture the configurations and transition rules before making changes to `stack`, `wbuffer` and `arcs`.

Next, let's detail the programming deliverables.

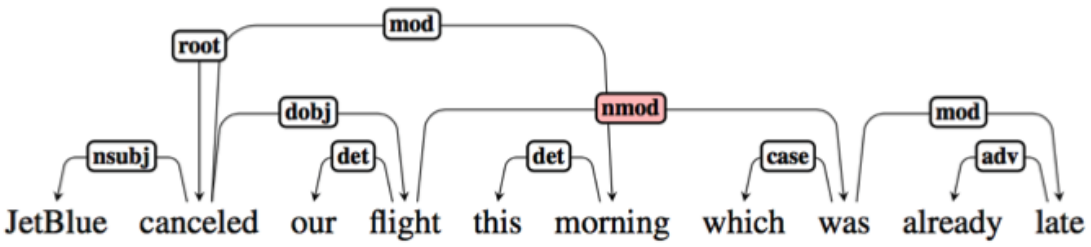
Programming Deliverables

Question 1

This question asks you to write a method to detect whether an inputted sentence has a projective dependency parse. A **projective** parse is one that contains paths from the head to every word that lies between the head and the dependent in the sentence.

When looking at the dependency tree for a sentence, we say a sentence is projective if its tree is drawn with no crossing edges. Meanwhile, a tree is non-projective if there exists at least one path from a given dependency to its head that crosses the arc that links another dependency to its head.

In the example we explored in lecture, the arc from *flight* to its modifier *was* is non-projective because there is not a path connecting *flight* to the words *this* and *morning*.



In this method, we've provided you with starter code for detecting non-projective parses such as this.

There is more than one way to solve this problem, so you are welcome to remove our starter code and try something different. No matter how you approach things, please do keep the `# BEGIN / # END SOLUTION` flags in place!

The input to this method will be a subset of the UD data for a given sentence, namely a list of 5-element tuples consisting of each token's: ID, original token form, part-of-speech, ID of the head, and dependency relation.

The method should return `True` if the sentence is projective, otherwise `False`.

Question 2.1

2.1 tasks you with implement the `SHIFT` operation, which removes the word from the front of the input buffer and pushes it onto the stack.

This function will be invoked to perform a *single* `SHIFT`.

Once `configurations` and `gold_transitions` are updated, remove `wbuffer[0]` and push it onto the stack.

Question 2.2

2.2 tasks you with implementing the remaining transition operators that operate on the top two elements of the stack, `LEFTARC` and `RIGHTARC`.

Both of these operations are known as **reduce** operations because they combine 2 elements on the stack then remove 1 of them (thus reducing the size of the stack).

- `LEFTARC`: assert a head-dependent relation between the word at the top of the stack and the second word. Then, remove the second word from the stack.
  - $s_1 \rightarrow s_2$
  - remove s_2
- `RIGHTARC`: assert a head-dependent relation between the second word on the stack and the word at the top. Then, remove the top word from the stack.
  - $s_2 \rightarrow s_1$
  - remove s_1

Question 2.3

2.3 tasks you with iteratively utilizing these methods to convert a full dependency parse into the correct transition actions. This involves visiting every token and deducing the correct actions.

We've provided you with starter code for the iteration. Your job is to fill in the proper way to check for and invoke a `RIGHTARC` operation.

While, like Question 1, there are other strategies for carrying out this method as a whole, we request that you stick to our starter code here. Also, please make sure your solution is entirely contained with the `# BEGIN / # END SOLUTION` flags.

For the `RIGHTARC` operation you need to fill in:

- check if s_2 and the s_1 are related in the dependency tree.
- if so, check if all the dependents of s_1 have already been assigned.
- if that's also true, then:
    - perform a `RIGHTARC` operation using your `perform_arc` function
    - update the `dep_arcs` dictionary
- otherwise:
    - perform a `SHIFT` operation using your `perform_shift` function.

Code Expectations

Like most NLP code, there are plenty of edge-cases when it comes to the dependency-parsing, such as the stack needing to be at least 2 elements large for a `RIGHTARC` to be applied.

We are not going to craft test cases to check that your method's "handle" these sorts of situations. We'll gravitate towards common-sense inputs/outputs similar to those you see in the "sanity check" methods throughout the notebook.

Additionally, while passing those "sanity checks" is not a bonafide guarantee your method is entirely well-formed, it is a solid indicator that your submission is functioning correctly. On the flip side, if you're not clearing those checks, that indicates your logic has problems that should be resolved!

Lastly, you don't need to import any additional libraries or use the Colab GPU to complete the core deliverables; base Python is sufficient for these questions.

Written Deliverable

Question 3

In the space provided in your Colab notebook (the markdown cell below `Q3 response`), please explain some **key differences** between the phrase-structure grammars and dependency grammars (~150 words).

We aren't looking for an exhaustive list, rather, a sufficiently detailed explanation about the situational advantages and major differences that each of these different formalisms offers.

To get you started, here are some considerations you might discuss:

- which of the two yields more accessible information following a parse?
- which of the two produces better results for languages that, unlike English, possess flexible word order?
- which of the two boasts more efficient computational complexity?
- what are the types of information we get following a parse (e.g. which would be better if you wanted to document parts-of-speech tags?)

It will be helpful to review the textbook readings on each of these concepts -- Chapters 17 (phrase-structure grammars) and 18 (dependency parsing) respectively.

Once you have completed the programming questions and the written component, you are welcome to submit.

You can optionally explore the Bonus sections for examples of how these operations can be utilized at scale to build a real-world parser!

How to Submit

- Submit your work to Gradescope
  - Download your Colab notebook as a `.ipynb` file
    - (File --> Download .ipynb)
  - Submit **HW4.ipynb**
    - Please do not include extraneous print statements, unnecessary cells, etc. in your final submission as this can cause issues for the Autograder.