11-411/11-611 Homework Assignment 2

NLP Teaching Staff

Due: February 9, 2023

1 Introduction

In this assignment, you will create a LEMMATIZER for English verbs. Your lemmatizer will turn inflected words like *topped*, *topping*, and *tops* into a normalized form called a LEMMA (*top* in this case) plus a morpheme boundary character (^) plus a suffix (*ed*, *ing* or *s*).

```
Input:  topped
Output: top^ed

Input:  topping
Output: top^ing

Input:  tops
Output: top^s
```

One focus of this assignment is WEIGHTS. In the last assignment, you used a neural model consisting of pretrained (and finetuned) weights. In later assignments, you will learn how to train the weights of models. In this assignment, you will manually construct and manually tune the weights of a graphical model (a wFST) in order to peform an NLP task.

In this assignment, you will construct a lemmatizer for English verbs using weighted finite state transducers (wFSTs) (discussed in class), using `wfst4str`, a Python library build on top of the `rustfst` library (a Rust port of the well-known OpenFST library).

1.1 Motivation

Many languages have INFLECTIONAL MORPHOLOGY, which results in multiple different words with the same lemma (e.g., *sing*, *sings*, *sang*, and *sung*). The lemma can also be called the CITATION FORM or DICTIONARY FORM. In English, we use the infinitive (*sing* as in *to sing*) as the lemma.

The infinitive is an especially useful lemma because, for some NLP tasks, it is necessary to obtain the infinitive. Consider taking sentence (1) and converting it to sentence (2). (1) is

a statement in the past tense. To make it into a question, you change the past tense verb *founded* into the lemma *found* and place the past tense auxiliary verb *did* at the front of the sentence.

1. Romulus *founded* Rome.
2. Did Romulus *found* Rome?

A lemmatizer is a program that takes a word form as its input and returns the corresponding lemma as its output. For example, given *founded*, it should return *found* and given *sang*, it should return *sing*.

1.1.1 Morphotactics

As discussed in class, MORPHEMES are meaningful pieces of words such as *found* and *ed*. MORPHOTACTICS is about the order of morphemes in a word. The morphotactics of English inflection is very simple (ignoring for the moment words like *sing* that inflect by changing a vowel as in *sang* and *sung*):

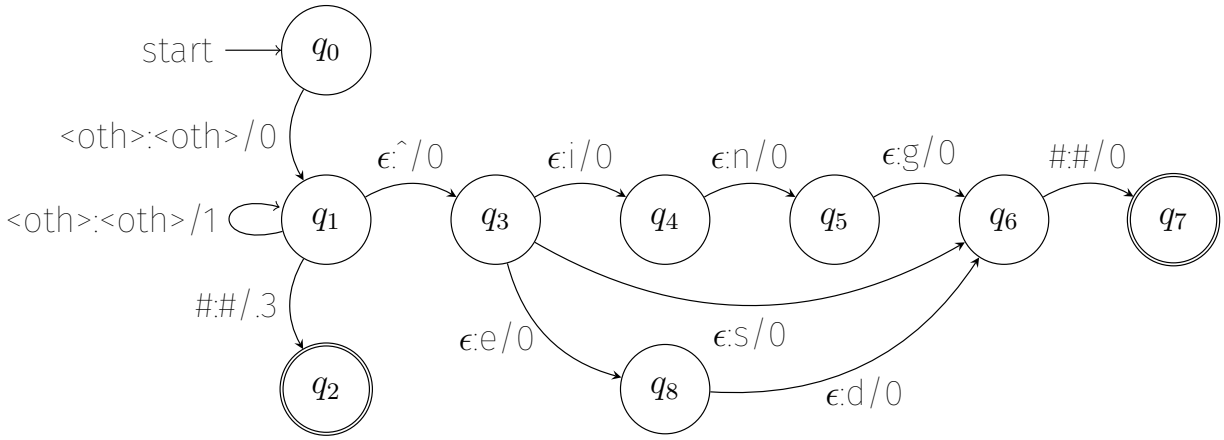
- There is at most one inflectional affix per word
- It is always a suffix

Thus, a model of English inflectional morphotactics consists of a transducer for lemmas concatentated with a transducer for the morpheme boundary symbol (**^**), concatenated with a transducer for the suffixes, concatenated with a word boundary symbol (**#**):

```
<stem> + ^ + {s, ing, ed} + #
```

(Where + indicates concatenation. The + symbol will not appear in your output.)

You have seen a version of this transducer in Lecture 5. When you are completing this assignment, you may implement the morphotactic transducer shown in that diagram and reproduced here:



1.1.2 Allomorphy

As discussed in class, a morpheme can sound different or be spelled differently in different contexts. For example, the third-person singular non-past suffix is `s` in `cats` but `es` in `fishes`. This phenomenon is called ALLOMORPHY. Allomorphy is often governed by rule. When dealing with written language, these patterns are called ORTHOGRAPHIC RULES or SPELLING RULES. In this assignment, we will be dealing with the rules below.

- **Silent e-deletion**. Delete a silent `e` when it is followed by `^` plus either `ed` or `ing`. Example: `mute^ed` becomes `mut^ed`.
- **E-insertion**. Insert an `e` in positions preceded by `s`, `z`, `sh`, `ch`, or `x` plus `^` and followed by `s`. Example: `fish^s` becomes `fish^es`.
- **Consonant doubling**. In words with one syllable, double some consonants at the end of the word when they are preceded by a vowel and followed by `^ed` or `^ing`. Getting the set of consonants that double right can be tricky. This rule is given to you, but you are free to change it. This rule should produce words like `topped` from `top^ed`.[1]
- **Y-replacement**. Replace `y` with `i` before `^ed` and with `ie` before `^s`. This rule will produce *pried* from `pry^ed`, and `parties` from `party^s`. (But it should not produce `plaies` from *play*^*s*. It should produce *plays*). You will need to figure out what pattern determines when you should not change a *y*.

As with the morphotactic transducer, we will write our allomorphic transducers so that they "apply down"—convert words from a more abstract representation to one closer to the surface representation. We do this because this generally make them easier to reason about and write. Since, as you learned in Lecture 5, FSTs are invertible, we can then simply invert each of these FSTs and obtain FSTs that "apply up"—from surface to abstract representations.

1.2 Weighted Finite State Transducers

A weighted finite state transducer (or wFST) is an FST (discussed in Lecture 6) in which each transition (or arc) is associated not just with an input symbol and an output symbol, but also with a WEIGHT. The weight of a wFST can be a real value, a boolean, an integer, or any number of other things depending on the SEMIRING of the wFST.

When operations are peformed on wFSTs, the weights of the new wFST are calculated based on the weights of the input wFST(s) according to principles that are common to all wFSTs, irrespective of semiring.

[1] Note that the consonant doubling rule is actually more complicated than this. For example, it applies to words that start with two consonants like `drop^ed` → `dropped`. You can safely extend the rule to handle these cases. However, it also sometimes applies in longer words (based on the position of stress). You should **not** try to cover these cases as you will lose in precision more than you gain in recall.

1.2.1 Semirings

Semirings are useful mathemtical objects from abstract algebra. They are important because they allow the same algorithms for wFSTs to generalize to different kinds of problems. A semiring is a tuple $\langle R, \oplus, \otimes, \bar{0}, \bar{1} \rangle$:

- R is a set
- $(R, \oplus)$ is a commutative monoid with the identity element $\bar{0}$
- $(R, \otimes)$ is a monoid with the identity element $\bar{1}$
- $\otimes$ distributes over $\oplus$
- Multiplication ($\otimes$) by $\bar{0}$ annihilates R.

In this assignment, we will start with the Tropical Semiring, in which R is the set of real numbers, $\oplus$ is min and $\otimes$ is addition. When you are computing a the weight of a path, you apply $\otimes$ to the weights of the transitions in the path (meaning that, in the Tropical Semiring, you simply add them.)

1.2.2 The AT&T Representation

The FST library we will be using, `wfst4str`, understands a way of representing wFSTs which we will call the "AT&T Representation." Files or strings in this representation consist of one or more entries of two types:

1. An entry with two fields, separated by whitespace and corresponding to a final state and the final weight of this state.
2. An entry with five fields, separated by whitespace and corresponding to the source state, the target state, the input symbol, the output symbol, and the weight.

Take, for example, the following wFST:

```
0 1 a b 1.0
1 2 b c 0.5
2 0.0
```

This would create a wFST with a transition from q_0 to q_1 with the label `a:b` and the weight 1.0 and a transition from q1 to q2 with the label `b:c` and the weight 0.5. q_2 is a final (accepting) state, with the weight 0.0.

Note that there are no leading or trailing spaces in the wFST notation. If you add leading or trailing whitespace, your wFST will not function properly.

1.3 Installing and using `wfst4str`

In this assignment, we will use a library called `wfst4str`. It is written in Rust but behaves as an ordinary Python library. It is built on top of another Rust library, `rustfst`, which is a Rust port of the famous (and famously non-portable) `OpenFst` library (written in C++). Aside from being written in a more sensible language, `rustfst` is also usually faster than `OpenFST`. `wfst4str` makes it easy to apply wFSTs to strings (and to get strings back). `rustfst` (and `OpenFST`) don't make a lot of assumptions about what the symbols transduced by the wFST are—they are just numbers—which means that you must do some extra work in order to do things we want to do, like apply a wFST to a string and get back the "best" output (the one with the lowest weight). `wfst4str` makes this rather easy. In return, one gives up some flexibility: one can only use the Tropical semiring and normal symbols can only consist of single Unicode characters (exceptions are special symbols like `<eps>`, the empty string, and `<oth>` for "other").

To install `wfst4str`, simply activate the appropriate virtual environment (in `conda`, `pyenv`, `virtualenv`, etc.) and enter:

```
pip install wfst4str
```

You can also install it without a virtual environment, on a per-user basis, but we do not recommend this as it leads to madness.

If you are using Google Colab (recommended) you can install `wfst4str` with:

```
!pip install wfst4str
```

Documentation for the most important `wfst4str` class is available on docs.rs. Consult it often.

1.4 Special symbols

There are a few symbols that are special to `wfst4str` which you should know about:

- `<eps>`, epsilon or the empty string
- `<oth>`, other (shorthand for every symbol except those implicity indicated on an outgoing transition from the same state). It must be explicitly expanded with `explode_oth`, which is called—in our assignment—with the argument `'#', '^'` so that `<oth>` will not match `^` or `#`.

We also use other symbols (`<c>`, `<v>`, `<g>`, etc.,) to refer to classes of symbols/letters (consonants, vowels, and glides/semivowels). However, these are not inherently meaningful to `wfst4str`. Instead, one calls the `sub` method to expand these classes into groups of transitions (see the consonant doubling example in the notebook).

1.5 Hints Regarding Weights

Weights are important when there is more than one path through a wFST for a given input string. For example, for the string `wish` there may be one path that yields `wish^s` and one that yields `wish^es` (inserting `e` when `^s` is preceded by `sh`). `wimps` would only map to `wimps`. **The path that inserts `e` (and, in general, the paths that make a change conditioned upon some context) should generally have a lower weight than the alternate paths (typically 0).** You can view weights as setting up case statements where the low-weight paths are tried first, following by the higher-weighted path. The highest weighted path is relevant just in case none of the other paths are.

Tuning weights can be tricky. **Initially, set all of your weights to zero, then bump some of them up following the heuristic given above.** `WeightedFST` objects have a method `outputs_by_weight(input)` which returns a list of (output, weight) tuples ordered by weight. Take the following example:

```
>>> import wfst4str
>>> SYMS = <a list>
>>> tactic_fst = wfst4str.WeightedFst()
>>> tactic_fst.set_input_symbols(SYMS)
>>> tactic_fst.set_output_symbols(SYMS)
>>> tactic_fst.add_state()
0
>>> tactic_fst.set_start(0)
>>> tactic_fst.populate_from_att(MORPHOTACTIC)
>>> tactic_fst.explode_oth({'#', '^'})
>>> tactic_fst.outputs_by_weight('study#')
[('study^s#', 4.0), ('study^ed#', 4.0), ('study^ing#', 4.0), ('study#',
↪  4.300000190734863)]
```

As should be the case, the weight of the three inflected forms is equal (4.0) but the weight of the uninflected form, which should be selected only in cases where none of the others would fit, is somewhat higher.

In general, the assertions are there to help you build your wFSTs and get them working properly. However, it is actually possible to pass the performance threshold without passing *all* of the assertions.

1.6 A Lemmatizer—Desiderata

What makes a good lemmatizer, and how will you know that your lemmatizer is good? It should have the following properties:

- When the input is a known word (in the lexicon) the output should match what is given in the lexicon. For example, `lemmatizer("sung")` should yield `"sing"`.
- When the input is a word that is part of the language which is not in the lexicon, the output should match the actual lemma of the word. For example,

```
lemmatizer("bepuzzling")
```

should equal `"bepuzzle"`.

- When the input is a made-up (NONCE) word, the output should respect the orthographic rules of the language (and match a native writer of the language's intuitions about what the lemma would be). For example,

  ```
  lemmatizer("squigged")
  ```

  should yield `"squig"`.

wFSTs can yield multiple outputs for a single input. We will take as your output the one with the lowest weight.

1.7 Putting it All Together

1.8 Evaluation

You will be evaluated on three validation data sets and three held-out data sets:

- In-vocabulary
- Out-of-vocabulary but attested ("real" words that are not in the lexicon)
- Nonce words (invented words)

You will receive full credit if your scores are above:

- 98% (in vocabulary)
- 65% (out of vocabulary)
- 75% (nonce)

For both the validation set and the held out test set.

2 Learning Objectives

At the end of this assignment, students should be able to:

- Differentiate between morphotactics and allomorphy in morphology
- Understand the task of sequence-to-sequence transduction
- Build wFSTs to deterministically transduce strings
- Use analysis of errors over a development set to tune a complex system
- Understand the role of development and test sets in constructing NLP systems

3 Task 1: Complete the Lemmatizer (90 marks)

You will be provided with a notebook with missing wFSTs marked by `TODO`. Your task is to fill these in with actual wFSTs in AT&T format so that you meet the performance thresholds set above. You will have to implement the following wFSTs:

- Morphotactic wFST (worked together in recitation)
- Silent e-deletion
- E-insertion
- Y-replacement

We `highly` recommend using Google Colab for this task. Simply upload the notebook and you should be able to run the cells without any issue. Once you finish this task, copy past the content of the cell marked with `TODO` in the `lemmatizer.py` file provided in the handout.

4 Task 2: Written Analysis (10 marks)

Create a PDF with your answers for the following questions, and clearly demarcate where each answer is on each page using Gradescope's interface.

1. Here are some other semirings:

Semiring	Set	$\bigoplus$	$\bigotimes$	$\bar{0}$	$\bar{1}$
Boolean	$\{false, true\}$	$\vee$	$\wedge$	$false$	$true$
Probability	$\mathbb{R}_+$	$+$	$\times$	0	1
Log	$\mathbb{R} \cup \{-\infty, +\infty\}$	$\bigoplus_{log}$	$+$	$+\infty$	0
Tropical	$\mathbb{R} \cup \{-\infty, +\infty\}$	min	$+$	$+\infty$	0

with $\bigoplus_{log}$ defined by: $x \bigoplus_{log} y = -log(e^{-x} + e^{-y})$

When using the probability semiring, the best path is the most probable one.

   (a) Why might the probability semiring be computationally inefficient or problematic (give two reasons)? Think about how numbers are represented by computers and what operations are most efficient. (2 marks)

   (b) What other semiring on this list might you use to represent probabilities with weights? How would you do it? How would you choose the best (most probable) path? (2 marks)

   (c) What would have to change to convert your wFSTs (from Task 1) to this representation? (2 marks)

2. It is likely that your transducers sometimes worked transducing in the original direction, but failed when they were inverted. Why does this happen? (4 marks)

5 Deliverables

This assignment will be submitted via Gradescope in two parts. Upload both deliveribles to Gradescope.

1. **Coding.** Paste the `TODO` items from `lemmatizer_handout_S23.ipnyb` into a file called `lemmatizer.py`, which is part of the handout. Upload `lemmatizer.py`
2. **Written.** Submit answers to the questions as a one-page PDF file.

Submissions are due February 9, 2023 at 11:59pm EST via Gradescope.