11-711: Algorithms for NLP

Homework Assignment #2: OpenFST

Out: October 3, 2015
Due: October 20, 2015

Overview

What is a Tokenizer?

A tokenizer is a common pre-processing tool for many Natural Language Processing tasks. Statistical parsers, machine translation systems, and the building of speech recognition systems and language models all involve a tokenizer. A tokenizer turns a string such as
`Your first assigment, homework 1, had lots of proofs on it.`
into a sequence of tokens such as
`Your` `first` `assignment` `,` `homework` `1` `,` `had` `lots` `of` `proofs` `on` `it` `.`

Intuitively, if we are building a system that "knows about" words and does things with them, and that system previously saw the token `assignment,` with a comma attached, if we then ask it about `assignment` without a comma, the system will treat it as an entirely new unseen word, which is probably not what we want. For example, a machine translation system might not be able to translate an unseen word. On the other hand, we might want to group the words `homework 1` into a single token if we want our system to handle them as a single concept. By tokenizing the input of such systems, we allow them to treat the same word[1] or words as the same. Formally, effective tokenization can reduce sparsity in statistical models – but we'll save that discussion for another class.

Tools we provide:

- `text2txtfst.py`: Converts a plain text file with one sentence per line into OpenFST (AT&T) text FST format including the special symbols <`s`>, <`/s`>, <`space`>, and <`newline`>.
- `ascii-syms.py`: Produces a symbol table of ASCII characters to integer IDs as expected by OpenFST's `--isymbols` option.
- `outtxtfst2tokens.py`: Converts an OpenFST output text FST into a sequence of tokens, one per line, that can be scored by `score.py`.
- `batch.sh`: Break data into batches for tokenization to reduce memory usage.

[1]What counts as "the same word" is a much deeper question, which we'll ignore here.

- `weight.awk`: Script for adding weight 1 to all transitions in your FST. This is important for `fstshortestpath`.
- `score.py`: Provides some statistics about a tokenizer on training and dev sets and also produces a score for competing with your classmates.

Data we provide:

- ABC (Toy Data): `make_abc.py` (generates 1 very long line [80MB] consisting of the letters abc; the line does *not* end with a newline nor space)
- News Commentary (Somewhat Clean Data): `nc.train.txt, nc.dev.txt, nc.test.txt` (about 60,000 lines each)
- Twitter (Real, Nasty Data): `twitter.train.txt, twitter.dev.txt, twitter.test.txt` (100,000 lines each)

Special characters:

Within OpenFST, we use the special arc labels:

- *space* becomes <`space`>
- *newline* becomes <`newline`>
- An epsilon transition is <`epsilon`>
- Your tokenizer should indicate the beginning of each token with <`t`> and the end of each token with <`/t`>

Assumptions:

You can assume that all data will contain ASCII characters (in the hex range 0x20–0x7F, decimal range 32–127). This excludes tabs and other special characters. Each line will always end with <`newline`>.

Hints:

- The FST visualizations produced by `fstdraw` can quickly become very large. Having a PDF viewer with infinite zoom is therefore useful. Both Linux and OSX have `xpdf`, which provides a convenient "zoom to selection" function.
- This exercise is based on `http://www.openfst.org/twiki/bin/view/FST/FstExamples`.
- Additional resources for OpenFST are available at `http://www.openfst.org/twiki/bin/view/FST/FstQuickTour` and `http://www.openfst.org`.
- You may find that as your tokenizers become increasingly complicated (especially in the last section), it takes longer to try out your ideas due to the FST becoming slower. If this is the case, we recommend that you select a small number of interesting sentences and try your new ideas on those. Then, after making several changes, you can measure the result on the whole data less frequently.

A few last things:

- Whenever you are asked a question that requires you to implement a FST, we expect you to write a FST or a script that generates a FST to solve that question. You should not modify any of the provided pre-processing or post-processing scripts.

1 Getting to Know OpenFST [20 Points]

In this section, we'll be using the ABC toy data set to keep your FSTs small enough to easily visualize, so that you can get a feel for how OpenFST works. Your entire input vocbulary will consist of the set {a b c}. We provide you with `abc.txtfst`, which can read in a single letter and annotate it as a token. For example, given `a`, it will produce <t> `a` </t>.

To answer this question, you will use commands similar to the following:

```
# Write your transducer in OpenFST (AT&T) text format called abc.txtfst
fstcompile --isymbols=abc.syms --osymbols=abc.syms abc.txtfst abc.fst
fstinfo abc.fst
fstdraw --isymbols=abc.syms --osymbols=abc.syms --portrait < abc.fst > abc.dot
dot -Tpdf < abc.dot > abc.pdf

# Time it on the example input
./make_abc.py > abc.input.txt
./text2txtfst.py < abc.input.txt > abc.input.txtfst
fstcompile --isymbols=abc.syms --osymbols=abc.syms abc.input.txtfst abc.input.fst
time fstcompose abc.input.fst abctok.fst abc.out.fst
fstinfo abc.out.fst
```

Answer all of the following for each of the problems in this section:

- *[1 point per question below]* a) Use `fstinfo` to determine how many states, arcs, and accepting states, this FSA has
- *[3 points per question below]* b) Use `fstdraw` to visualize this FSA – include the result as your answer
- *[1 point per question below]* c) Use `fstcompose` and the Unix `time` to transduce the toy input using your FST. Did the FST accept your input? If so, how long did it take? (Report total time, not user or system time)

1. [**5 points**] Answer a-c using the `abc.txtfst` provided to you. You will use it like `abctok.txtfst` is used in the commands above.
2. [**5 points**] Use `fstclosure` to extend your tokenizer to accept a string of any length (still using the same 3 letter alphabet) where each letter is its own token. Answer a-c using this FST.
3. [**5 points**] Use `fstdeterminize` to modify your previous non-deterministic FST to make a deterministic FST. Answer a-c again using this FST.

4. [**5 points**] Use `fstminimize` and `fstrmepsilon` to modify your previous deterministic FST to generate a deterministic FST with the minimal number of states. Answer a-c again using this FST.

2 Simple Tokenizers [25 Points]

In this section, you'll explore tokenizers with a larger alphabet (the ASCII character set instead of just ABC) and you'll be working with real data. Due to the larger number of characters here, you might find it convenient to automatically generate the .txtfst files in this section using a scripting language such as Python.

Use the Newswire **10k** data set (nc10k.train.txt / nc10k.dev.txt) to complete this section.

For all but the last question in this section, you will use the `score.py` script to answer the following "stats" questions about the tokenizers you construct:

- a) *[1 point per question below]* How many types and tokens does this tokenizer produce on the training set?
- b) *[1 point per question below]* On the dev set?
- c) *[1 point per question below]* What is the token-wise OOV rate of this tokenizer?
- d) *[2 points per question below]* What problems do you think this tokenizer would suffer from applied to a real task (speech recognition, machine translation, etc.)?

2.1 Build a tokenizer that makes every character its own token

For example, given `t w o` <space> `c a t s .` <newline>, you should produce <t> `t` </t><t> `w` </t><t> `o` </t> <t> `c` </t><t> `a` </t><t> `t` </t><t> `s` </t><t> `.` </t>.

[**5 points**] Answer the stats questions from above.

2.2 Build a tokenizer that splits tokens on spaces and newline characters only

For example, given `t w o` <space> `c a t s .` <newline>, you should produce <t> `t w o` </t> <t> `c a t s .` </t>.

This tokenizer should not separate any punctuation, etc.

[**5 points**] Answer the stats questions from above.

2.3 Modify the above space-delimiting tokenizer to separate all periods as a separate token

[**5 points**] Answer the stats questions from above.

2.4 Modify the above tokenizer that separates periods to recognize "New York" as a single token

Captalization matters here. For this question, you should return a single token for $\boxed{\texttt{New York}}$ iff the "N" and "Y" are capitalized. All other tokens should continue to be space-delimited; for example, we expect two tokens for $\boxed{\texttt{New}}$ $\boxed{\texttt{Hampton}}$. Use `fstshortestpath`. For `fstshortestpath` to work properly, add a weight of 1 to every transition. The included `weight.awk` will do this for you.

[**5 points**] Answer the stats questions from above.

2.5 Summary

[**5 points**] Discuss the tradeoffs that you observed above in terms of OOV rate and number of test tokens. (NOTE: You will be facing these tradeoffs – with some additional constraints – in the competition section of this homework below.)

3 Beyond Tokenizers [15 Points]

Tokenizers are confined to identifying meaningful breaks between word-like items in text; they never drop characters nor transform them (traditionally with the exception of whitespace). Transformations are the business of text normalizers, stemmers, and other such tools. In this section, we will briefly explore such tools to give you an appreciation for how larger NLP pipelines might function in an FST paradigm.

Use the Newswire **10k** data set (nc10k.train.txt / nc10k.dev.txt) to complete this section.

Note that in this section, you will be modifying characters besides whitespace – according to our definition of a tokenizer, this is not allowed; `score.py` checks for this by default. You will need to disable these checks in this section by passing `--no-checks` as the last argument to `score.py`.

3.1 Build a transducer that acts as a trivial stemmer, keeping only the first 3 characters of each token

Your stemmer transducer will read in the output of your tokenizer that splits on spaces only (you built this in Section 2.2 above), modify the tokens as described, and write them out in the same token format (using begin-token and end-token markers to indicate token boundaries). If a token is only of length 3 or less, you should return it unmodified. Use `fstcompose` to produce a single efficient machine that can be applied to new inputs.

(Note that this is only a trivial stemmer. More advanced stemmers such as the Porter stemmer exist and actually take lexical and morphological knowledge into consideration.)

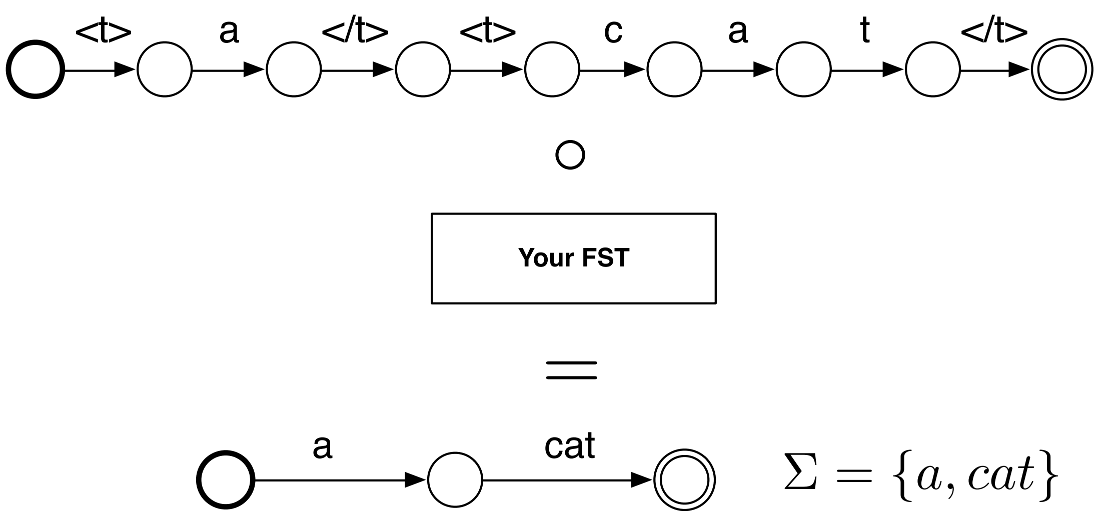
Use the `score.py` script to answer the following questions about your stemmer:

- a) [**2 point**] How many types and tokens does your tokenizer+stemmer machine produce on the training set?
- b) [**1 point**] On the dev set?
- c) [**1 point**] What is the token-wise OOV rate of your tokenizer+stemmer?
- d) [**2 points**] What problems do you think stemmers solve compared to a tokenizer alone when applied to a real task (speech recognition, machine translation, etc.)?
- e) [**2 points**] What problems do you think stemmers introduce compared to a tokenizer alone when applied to a real task (speech recognition, machine translation, etc.)?

3.2 Preparing Tokenizer Output for a Downstream Application

Imagine an application that takes a FSA as input where each of the FSA's arcs must be exactly one token (that is, you are no longer allowed to use `<t>` and `</t>` to delimit tokens). Such applications include machine translation systems, which often treat sentences as a list of words, where the words come from a discrete vocabulary (in fact, these systems internally map each word to a unique integer vocabulary ID since integer comparisons are much faster than string comparisons). In this section, you will be emitting one word token per arc – that is, your input vocabulary will be characters and your output vocabulary will be tokens. Here's an example:

$$\Sigma = \{a, c, t, <t>, </t>\}$$



$$\Sigma = \{a, cat\}$$

1. [**3 points**] Briefly describe a FST that converts the format we've been using above (tokens can span multiple arcs and are delimited by special markers) to the format described here (each token occupies exactly one arc).
2. [**2 points**] Notice that this transducer is infinitely large for words of arbitrary length. However, given an alphabet of 95 characters (what you've been using in this assignment) and a maximum word length of 5 characters, what is the upper bound on how many states the resulting machine could have, using your design above?
3. [**2 points**] Again, using your design from above, how many states would the machine have if we already know the recognizable token vocabulary V in advance? (As we would if we were preparing input for a typical machine translation system; You can assume any tokens in the input that are not in the token vocabulary V are mapped to an unknown token `<UNK>`). Hint: Your answer may be an expression rather than a specific number.

4 Some Friendly Competition [40 Points]

You will be competing to keep the number of tokens produced on the training data small while also keeping your out-of-vocabulary (OOV) rate low. The objective function you will be competing on is:

$$\texttt{Recall} = \frac{\displaystyle\sum_{\substack{m\in\texttt{tok(test)}\\ \text{iff } m\in\texttt{tok(train)}}} |m|}{\displaystyle\sum_{w\in\texttt{devOrig}} |w|} \qquad \texttt{Coherence} = 1 - \frac{\displaystyle\sum_{t\in\texttt{tok(test)}} 1}{\displaystyle\sum_{w\in\texttt{test}} |w|}$$

$$\texttt{Score} = \frac{2\cdot\texttt{Recall}\cdot\texttt{Coherence}}{\texttt{Recall}+\texttt{Coherence}}$$

`Recall` measures the proportion of tokens in the tokenized test set that are also found in the tokenized training set, weighted by the length of the token. `Coherence` measures the number of token splits in the tokenized test set compared to the maximum possible number of token splits. Reducing test set OOVs improves `Recall` while reducing the number of test set tokens by merging more characters into larger tokens improves `Coherence`. `Recall` and `Coherence` are combined as a harmonic mean. Intuitively, this scoring function simulates trying to keep recognition rates high (you don't want too many out-of-vocabulary items in your test set) while not breaking apart lexical items to the point where they no longer have any resemblance to their semantic (or grammatical) meanings.

We provide you with training, development, and test sets; **you may use only this data in this competition**. For purposes of competing with your classmates, you will be scored on a fourth unseen data set (blind test), which you will not have access to. Your tokenizer *may not* drop nor transform any characters from the input except spaces and newlines; you *may* include spaces and newlines inside tokens if you wish (e.g. `New York`), but whitespace characters have no impact on score, so you could equivalently output `NewYork`.

4.1 Tokenizing Clean Data: Newswire Text

In this section, you will use the Newswire data – *not the shortened 10k version.*

1. [**10 points**] In a paragraph, describe your strategy, the techniques you used to obtain better scores, and why you believe these techniques might have helped.
2. [**10 points**] Impress us with your tokenizer.

4.2 Tokenizing Real Data: Tweets

1. [**10 points**] In a paragraph, describe your strategy, the techniques you used to obtain better scores, and why you believe these techniques might have helped.
2. [**10 points**] Impress us with your tokenizer.

How to Submit Your FSTs

In addition to submitting your solutions to these exercises, you will also submit your two tokenizers *in OpenFST text format* at `http://demo.clab.cs.cmu.edu:8081/turnin/`. Again, you should submit your **tokenizer only**, before composing it with the input – we will be composing it with a blind test set, which you never see. You should submit a single `.tar.gz` file containing exactly the following two files *nc.txtfst* and *twitter.txtfst*. Your tar file should be called `submission.tar.gz` and should *not* contain any internal directory structure. You can create such a tarball using:

```
tar -cvzf submission.tar.gz nc.txtfst twitter.txtfst
```

Evaluation

We will evaluate your submissions on our side by:

1. Running `weight.awk` to make sure all transitions have weight 1. This is a no-op if your text FST already has weight 1 for every transition.
2. Converting your text FST to a binary FST using the `ascii.syms` file as both input and output symbols
3. Composing the blind test data (which you will never see) with your text FST
4. Running the result through `fstshortestpath`
5. Converting the result back to text using `outtxtfst2tokens.py`
6. Ensuring that you did not drop nor alter any characters besides space
7. Evaluating it using `score.py`

Please test your tokenizer before submitting. We cannot grade tokenizers that do not produce output.