Overview

What is a Tokenizer?

Tokenization is a first step in most NLP applications. While computers represent text as sequences of characters, most language processing systems (e.g., parsers, machine translation systems, information extraction systems, etc.) need to operate on words. We call the process of breaking sequence of characters into words tokenization. For example, a tokenizer turns a string such as

Your first assignment, 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

Often, a token is an English word, but it can also be a word root, an affix (e.g. +ing), a punctuation symbol, an emoticon, a number, or it could be a sequence of "words" that form a single unit, like New York or Homework 1. Thus, tokenization can split an input text stream arbitrarily (and possibly perform some amount of normalization). However, in most cases, the splits will happen at word boundaries or punctuation characters.

Although segmenting a stream of characters into token-like words may seem relatively straightforward at first, in practice, good tokenization decisions are crucial for good performance in downstream NLP applications.

Your Tokenizer

In this assignment you will develop a tokenizer for English using finite state transducers (FSTs) that produces tokens that correspond roughly to words, and deals appropriately with punctuation, numbers, acronyms and the like. We will not consider multiword tokens in this assignment.

You will be using OpenFST (http://www.openfst.org/) to build a hand-built tokenizer. OpenFST is a set of open source tools and C++ libraries developed by Google for manipulating and using weighted finite-state transducers (WFSTs). PyFST is a Python interface to OpenFST.

1What counts as a word is a deep question which we’ll ignore here.
developed by Victor Chahuneau (http://victor.chahuneau.fr/) when he was an MLT student. In this homework we will be using it primarily to design unweighted FSTs, that is, WFSTs that use only a weight of 1. Note that OpenFST uses weights in the log space, so weights of 1 will be represented as a 0 (we will discuss in lecture that different algebras have different definitions of 1 and 0).

While OpenFST lets you construct FSTs from the command line or using a simple text format, txtfst\(^2\), which was developed by AT&T (the precursor to OpenFST was developed by many of the same authors when they worked at Bell Labs), we recommend developing your FSTs using PyFST.

Your tokenizer will work by transforming streams of characters into whitespace delimited text. For example, for the input:

Mr. O’Neill thinks the boys’ stories about Chile’s capital aren’t amusing.
I said "Don’t go!"
Shares of AT&T rose by 4.7%.
He read it on al-Jazeera?
That costs just $19.95.
I went... to New York and New Hampton and San Francisco.

You should produce the output:

Mr. O’Neill thinks the boys’ stories about Chile’s capital aren’t amusing.
I said "Don’t go!"
Shares of AT&T rose by 4.7%.
He read it on al-Jazeera?
That costs just $19.95.
I went... to New York and New Hampton and San Francisco.

Development

Here is what your development pipeline should look like:

1. Write code for your tokenization FST. This will involve writing a python program to generate the FST.

2. Determinize, minimize, etc. your FST.

3. Run your tokenizer on input and evaluate the output; if you are unhappy with it, go back to (1).

Special characters and indicating token boundaries

In addition to standard ASCII symbols for letters, numbers, punctuation, we include special arc labels:

- \(<space>\) represents the space symbol;
- \(<newline>\) represents the newline symbol;
- \(<\text{epsilon}>\) represents an \(\varepsilon\)-transition;

\(^2\)You should be able to figure out the format easily by looking at the examples we provide, but there is also an official definition (http://www2.research.att.com/~fsmtools/fsm/man4/fsm.5.html)
Scripts we provide

- **fst_helpers.py**: Contains helper functions that convert sentences to FSTs, FSTs to sentences, prepare FSTs for submission, and produce some statistics about FSTs. See usage examples in Problem 1.

- **score.py**: provides some statistics about a tokenizer on training and dev/test sets and also produces scores for competing with your classmates.

PyFST demo scripts we provide (check the directory pyfst_demo)

- **ABC_Demo.py**: a python script that generates the FST used in Problem 1;

- **ABC_Demo.ipynb**: the same script in the form of the iPython notebook;

Data we provide

- 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)

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 return feeds of all sorts. Each line will always end with `<newline>`.

Hints

- Please review the OpenFST/PyFST recitation slides if you are lost.

- The FST visualizations produced by OpenFST can quickly become large. If you have a PDF viewer with infinite zoom, using fstdraw to create PDFs or large FSTs may be 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](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](http://www.openfst.org/twiki/bin/view/FST/FstQuickTour) and [http://www.openfst.org](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.

- If you use PyFST, make sure that you use the same symbol table that you created in the very beginning (see examples in the folder **pyfst_demo**).
• You do not have to directly produce the final version of the FST. You can build smaller blocks and combine them using operations of union, closure, composition, etc. For information on how to do this in PyFST, please refer to the API description: http://pyfst.github.io/api.html#module-fst.

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 vocabulary will consist of the set \{a,b,c\}. We provide you with ABC Demo.py (or ABC Demo.ipynb), which shows how to create a simple FST that can read in a single letter and annotate it as a token. Given ab, it will produce a b.

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

```python
import fst
from fst_helpers import *

sigma = fst.SymbolTable('<epsilon>')</n
sample_input = input_to_fsa('abc' * 100000, sigma)

my_fst = fst.Transducer(sigma, sigma)
# Write some code to build your transducer here

fst_info(my_fst) # Part a), information about size of the FST
save_as_pdf(my_fst, 'my_fst.pdf') # Part b), create an image of the FST

time_composition(sample_input, my_fst) # Part c), Compose sample input data with FST
```

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

• [1 point per question below] a) Use `fst_info` to determine how many states, arcs, and accepting states, this FSA has

• [3 points per question below] b) Use `save_as_pdf` to visualize this FSA – include the result as your answer (LaTeX command: \includegraphics{my_fsa.pdf}).

• [1 point per question below] c) Use `time_composition` to transduce the toy input using your FST. Did the FST accept your input? If so, how long did it take?

1. [5 points] Answer a-c using the demo FST in ABC Demo.py provided to you.

2. [5 points] Extend the ABC demo FST by using `abc = abcclosure()` 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 \( \text{abc} = \text{abc.determinize()} \) to modify your previous non-deterministic FST to make a deterministic FST. Answer a-c again using this FST.

4. **[5 points]** Use \( \text{abc.minimize()} \)\(^3\) 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.

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 provided `score_nc10k` function 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? On the dev set?
- **b) [1 point per question below]** What are the type- and token-wise OOV rates of this tokenizer?
- **c) [1 point per question below]** How well does the tokenizer perform in terms of precision, recall, and f-score?
- **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.)?

Invoke the `score_nc10k` function as follows:

```python
from fst_helpers import *
from score import *
my_fst = fst.Transducer(ascii_alphabet)
# Bunch of code to build your fst
score_nc10k(my_fst)
```

The output will contain token and type counts on both the training and dev sets, token and type OOV rates, and precision, recall, and f-score metrics for both the training and dev sets. The number of tokens in a data set is the number of words, counting repeats, in the data set. Similarly, the number of types is the number of unique words in the data. Thus, for example the sentence, “the cat sat on the mat” contains 6 tokens, but only 5 types.

The OOV (out-of-vocabulary) rate counts the number of tokens and types that exist in the development data but not in the training data. Generally tokenizers do well on words they saw in their training data, but may not perform optimally on words they’ve never seen before. As such, OOVs are generally bad, and a lower count is better.

Precision and recall both measure the overlap between the tokens produced by your tokenizer and the reference system. If your precision is higher than your recall, you’re probably undersegmenting. If your recall is higher, you’re probably oversegmenting. The f-score is the harmonic

\(^3\) Note: no “\( \text{abc} = \ldots \)” . This is because closure and determinize are **constructive** operations, while minimize is a **destructive** operation. See [http://pyfst.github.io/user_guide.html#operations](http://pyfst.github.io/user_guide.html#operations) for more details.
mean of precision and recall, and gives a balanced indication of the quality of your tokenizer in a single number.

Note that this function may take several minutes to complete. During development, you might find it more convenient to create smaller train/test sets, and call the `score(my_fst, train_in, train_gold, dev_in, dev_gold)` function instead. Be careful in your handling of newlines. The number of newlines in the output should be equal to the number of newlines in the input, and they should never occur inside of tokens. Spaces may be preserved or dropped, whichever you find easier, but should not occur inside of tokens, except in the case of New York in Problem 2.4.

2.1 Build a tokenizer that makes every character its own token

For example, given two cats,

\[
\text{you should produce } \begin{array}{c}
\langle t \rangle \ t \langle /t \rangle \\
\langle t \rangle \ o \langle /t \rangle \\
\langle t \rangle \ c \langle /t \rangle \\
\langle t \rangle \ a \langle /t \rangle \\
\langle t \rangle \ s \langle /t \rangle \\
\langle /t \rangle
\end{array}
\]

[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 two cats,

\[
\text{you should produce } \begin{array}{c}
\langle t \rangle \ t\langle \text{two} \rangle \ c\langle \text{ats} \rangle \\
\langle /t \rangle
\end{array}
\]

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

Capitallization matters here. For this question, you should return a single token for 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 New Hampton.

[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 nochecks=True as an extra argument to score.nc10k.

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:
\[ \Sigma = \{ a, c, t, <t>, </t> \} \]

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 try to get your tokenizer as close as possible to the “gold” tokenized data as possible. For this task, your objective function will be the f-score, as determined by `score.py`. The f-score is the harmonic mean of two quantities, precision and recall. Raising either will boost your f-score and get you closer to the top!

**Precision** measures the proportion of tokens in your tokenized output that also appear in your output. **Recall** measures the proportion of tokens in the tokenized gold data that are correctly produced by your tokenizer. Mathematically, if the gold corpus is a bag of words \( G \) and your FST outputs a hypothesis tokenization \( H \), the precision \( P = \frac{|G \cap H|}{|H|} \) and the recall \( R = \frac{|G \cap H|}{|G|} \). The f-score \( F \) is then defined as \( F = \frac{2PR}{P+R} \). All three quantities are between 0 and 1, and can be thought of as different measurements of what percentage of tokens your FST correctly produces.
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. \texttt{New York}), but whitespace characters have no impact on score, so you could equivalently output \texttt{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 electronically at \texttt{http://demo.clab.cs.cmu.edu:8265/turnin/}. Each tokenizer should have both a SymbolTable file ending in \texttt{.sym} and an in OpenFST text format FST ending in \texttt{.txtfst}. You should submit your tokenizer only, before composing it with the input – we will be composing it with a blind test set, which you will never see. You should submit a single \texttt{.tar.gz} file containing exactly the following four files: \texttt{nc.sym nc.txtfst twitter.sym} and \texttt{twitter.txtfst}. Your tar file should be called \texttt{submission.tar.gz} and should not contain any internal directory structure.

You can use the provided functions \texttt{save symbol table(sigma, filename)} and \texttt{save as txtfst(fst, filename)} to generate the required \texttt{.sym} and \texttt{.txtfst} files. You can then create a tarball containing these four files using:

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

### Evaluation

We will evaluate your submissions on our side by:

1. Converting your text FST to a binary FST
2. Composing the blind test data (which you will never see) with your text FST
3. Running the result through \texttt{shortestpath}
4. Converting the result back to text

5. Ensuring that you did not drop nor alter any characters besides space

6. Evaluating it using `score.py`

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