CS114 Spring 2015

Homework 1: Finite State Transducers

Due January xx, 2015

This assignment is mostly on finite state automata and will require you to create automata in Python.[1] There are a total of three problems, and we have provided some code to get you started, as well as a few utilities that will make it easier for you to debug and test your code.

Some minimal test cases are provided as well. Before you turn in your work, you should at least pass all of the test cases we provided although it is neither sufficient nor necessary for getting full credit. You should add your own test cases to help yourself debug your program.

Problem 1: Regular Expressions

1. Implement a function that determines whether a given string is a valid email address or not. (`is_email_address()` in `recognizer.py`)
2. Implement a function that determines whether a given string is a valid phone number or not e.g (123)-456-7890 or 123-456-7890 or 123-4567890 or other valid patterns. (`is_phone_number()` in `recognizer.py`)

Problem 2: Soundex

The Soundex algorithm is a phonetic algorithm commonly used by libraries and the Census Bureau to represent people's names as they are pronounced in English. It has the advantage that name variations with minor spelling differences will map to the same representation, as long as they have the same pronunciation in English. Here is how the algorithm works:

Step 1: Retain the first letter of the name. This may be uppercased or lowercased.

Step 2: Remove all **non-initial** occurrences of the following letters: `a, e, h, i, o, u, w, y`. (To clarify, this step removes all occurrences of the given characters *except* when they occur in the first position.)

[1] The assignment is adapted from Jordan Boyd-Graber's course at UMD.

Step 3: Replace the remaining letters (except the first) with numbers:

- `b, f, p, v` $\rightarrow 1$
- `c, g, j, k, q, s, x, z` $\rightarrow 2$
- `d, t` $\rightarrow 3$
- `l` $\rightarrow 4$
- `m, n` $\rightarrow 5$
- `r` $\rightarrow 6$

If two or more letters from the same number group were adjacent in the *original* name, then *only* replace the first of those letters with the corresponding number and ignore the others.

Step 4: If there are more than 3 digits in the resulting output, then drop the extra ones.

Step 5: If there are less than 3 digits, then pad at the end with the required number of trailing zeros.

The final output of applying Soundex algorithm to any input string should be of the form `Letter Digit Digit Digit`. Table 1 shows the output of the Soundex algorithm for some example names.

Input	**Output**
Jurafsky	J612
Jarovski	J612
Resnik	R252
Reznick	R252
Euler	E460
Peterson	P362

Table 1: Example outputs for the Soundex algorithm.

Construct an FST that implements the Soundex algorithm. Obviously, it is non-trivial to implement a single transducer for the entire algorithm. Therefore, the strategy we will adopt is a bottom-up one: implement multiple transducers, each performing a simpler task, and then compose them together to get the final output. One possibility is to partition the algorithm across three transducers:

1. **Transducer 1**: (`letters_to_numbers`) Performs steps 1-3 of the algorithm, i.e, retaining the first letter, removing letters and replacing letters with numbers.

2. **Transducer 2**: (`truncate_to_three_digits`) Performs step 4 of the algorithm, i.e., truncating extra digits.

3. **Transducer 3**: (`add_zero_padding`) Performs step 5 of the algorithm, i.e., padding with zeros if required.

Note that each of these three transducers will have characters as input/output symbols.

To make things easier for you, we have provided the file `soundex.py` which is where you will write your code. It already imports all needed modules and functions (including `fsmutils.py`). It also creates three transducer objects—as dictated by the bottom-up strategy outlined above—such that all you should have to do is to figure out the states and arcs required by each transducer. It also contains code that allows you to input a single name on the command line to get the output.

Note that while we have provided you with sample unit tests containing some names, it might be very useful to test your code on other names. For comparison purposes, you may use one of the many Soundex calculators available online to create more test cases.

Problem 3: English Morphology

English often requires some spelling changes at the morpheme boundary between the stem and the affixes that indicate inflection verb. Specifically, we will build a finite-state transducer to handle the K-insertion rule in English.

If a verb ends with vowel + c, then we add k before adding in the -ed and -ing suffixes. For example

panicked ↔ panic + ed ↔ panic+past form
panicking ↔ panic + ing ↔ panic+present participle form
lick ↔ lick + ed ↔ lick+past form
lick ↔ lick + ing ↔ lick+participle form

As you can see in the examples, the morphological parsing process can be done in two steps (although you may skip the intermediate step). So you will need to build two separate FSTs and compose them. The lexicon

FST only has to include two inflectional morphemes (-ed and -ing) and five words: want, sync, panic, lilac, and lick. The other FST will handle the actual K-insertion rule. This method is very similar to the example given in the chapter.

The final FST will be used as a morphological parser to transduce the surface form (e.g panicked) to the lexical form (panic + past form). The inverted FST will be used to transduce the lexical form to the surface form.

Turning in Your Assignment

Submit your completed code to Latte. We are using a grading script to grade the work, so please make sure:

- You submit each file individually. Do not zip or tar the files.
- You do not change filenames, function names, or the API.
- Do not add `print` statements to the code you turn in
- Add your name as a comment to all files you turn in

NLTK FST

We've provided an a FST implementation from NLTK. However, the documentation is less than perfect. Therefore, in this section, we will give you a brief introduction to everything that you will need to understand how to build finite state transducers in Python.

To start building FSTs, you need to first import the `fst` module into your program's namespace. Then, you need to instantiate an `FST` object. Once you have such an object, you can start adding states and arcs to it. Listing 1 shows how to build a very simple finite state transducer—one that removes all vowels from any given word.

Feel free to try out the example to see how it works on some of your own input. There are a few points worth mentioning:

1. The Python `string` module comes with a few built-in strings that you might be able to use in this assignment for purposes of iteration as used in the example on line 23. These are:
   - `string.letters` : All letters, upppercased and lowercased
   - `string.ascii_lowercase` : All lowercased letters

Listing 1: A 1-state transducer that deletes vowels

```
# import the fst module
import fst

# import the string module
import string

# Define a list of all vowels for convenience
vowels = ['a', 'e', 'i', 'o', 'u']

# Instantiate an FST object with some name
f = fst.FST('devowelizer')

# All we need is a single state ...
f.add_state('1')

# and this same state is the initial and the final state
f.initial_state = '1'
f.set_final('1')

# Now, we need to add an arc for each letter; if the letter is a vowel
# then then transition outputs nothing but otherwise it outputs the same
# letter that it consumed.
for letter in string.ascii_lowercase:
    if letter in vowels:
        f.add_arc('1', '1', (letter), ())
    else:
        f.add_arc('1', '1', (letter), (letter))

# Evaluate it on some example words
print ''.join(f.transduce(['v', 'o', 'w', 'e', 'l']))
print ''.join(f.transduce('e x c e p t i o n'.split()))
print ''.join(f.transduce('c o n s o n a n t'.split()))
```

- `string.ascii_uppercase` : All uppercased letters

2. States can be added to an FST object by using its `add_state()` method. This method takes a single argument: a unique string identifier for the state. Our example has only one state (line 13). Furthermore, there can only be **one** initial state and this is indicated by assigning the state identifier to the FST object's `initial_state` field (line 17). However, there may be multiple final states in an FST. In fact, it is almost always necessary to have multiple final states when working with transducers. All final states may be so indicated by using the FST object's `set_final()` method (line 18).

3. Arcs can be added between the states of an FST object by using its `add_arc()` method. This method takes the following arguments (in order): the starting state, the ending state, the input symbol and, finally, the output symbol. If you wish to use single characters as input or output symbols, you must enclose them in parentheses (lines 25 and 27).

   However, if you wish to use entire words as input or output symbols, you must enclose the word in **square brackets** (not in parentheses). For example, if you wish to add an arc that takes the string *ten* as input and returns the number string *10* when going from state 1 to 2, you should use:

   ```
   f.add_arc('1', '2', ['ten'], ['10'])
   ```

   ϵ's may be indicated by an empty set of parentheses or square brackets, depending on the context. (line 25).

4. An FST object can be evaluated against any input string by using its `transduce()` method. Here's how:

   (a) If your transducer uses **characters** as input/output symbols, then the input to `transduce()` must be a **list of characters**. You may either directly input a list of characters (line 30) or you may convert a string to a list of characters by spacing out its characters and calling its `split()` method (lines 31 and 32).

   (b) If your transducer uses **words** as input/output symbols, then the input to `transduce()` should be a **list of words**. Again, you can either explicitly use a list of words or call the split method on a string of words separated by whitespace. For example, say your FST maps from strings like *ten* and *twenty* to number strings *10* and *20*, then to evaluate it on the input string *ten twenty*, you should use either:

```
f.transduce('ten twenty'.split())
```

OR

```
f.transduce(['ten', 'twenty'])
```

Provided Utilities

To make it easier for you to solve the programming problems, we have provided 3 handy utilities in the included python file `fsmutils.py`. These utility functions will help you to test each transducer that you build and compose multiple transducers together.

1. The first is `composechars()`, which allows you to compose any number of transducers (that use single characters as input strings) and evaluate it on any input string[2]. For example, if you have created three transducers `f1`, `f2` and `f3` and you wish to evaluate their composition on the input string `S`, then you should use the following code:

   ```
   from fsmutils import composechars
   output = composechars(S, f1, f2, f3)
   ```

   The above function call computes $(\texttt{f3} \circ \texttt{f2} \circ \texttt{f1})(\texttt{S})$. i.e., it will first apply transducer `f1` to the given input `S`, use the output of this transduction as input to transducer `f2` and so on and so forth. It will raise a generic exception if one or more input transducers do not work correctly. Note that since all transducers for this function use single characters as the input symbols, `S` must be a list of characters.

2. The second utility function is `composewords()` which allows you to compose transducers that use words as input symbols, instead of single characters. The usage is similar to `composechars()` but the input string `S` must be a list of words in order to be used with this function.

[2]Note that this function only performs composition in a practical sense and does not actually create a single composed transducer. However, for this assignment, the former is more than sufficient.

3. The final utility function is `trace()`. Given any single transducer `f` and a string `S`, this function will print the entire path taken through `f` when using `S` as the input. This can prove extremely invaluable for debugging any transducer. It may be used as follows:

```
from fsmutils import trace
trace(f, S)
```