CMSC 723: Computational Linguistics I

Assignment 2: Finite State Adventures

Jimmy Lin (Instructor) and Melissa Egan (TA)

Due: **September 30, 2009**

Introduction

This assignment is on finite state automata and transducers. There are three problems. The first requires no programming, and can be done on paper. For both programming problems, you are provided with some code to get started, as well as a few utilities that will make it easier for you to debug and test your code.

FSTs in NLTK

Although NLTK does come with a fully functional finite state transducer (FST) module, 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 NLTK.

To start using NLTK for 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 (provided along with the assignment as `devowelizer.py`) 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 (the string `'abc...xyzABC...XYZ'`)

Listing 1: A 1-state transducer that deletes vowels

```python
# import the fst module
from nltk_contrib.fst 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
# Outputs 'vwl'
print ''.join(f.transduce(['v', 'o', 'w', 'e', 'l']))
# Outputs 'xcptn'
print ''.join(f.transduce('e x c e p t i o n'.split()))
# Outputs 'cnsnnt'
print ''.join(f.transduce('c o n s o n a n t'.split()))
```

- `string.ascii_lowercase` : All lowercased letters (the string `'abc...xyz'`)
- `string.ascii_uppercase` : All uppercased letters (the string `'ABC...XYZ'`)

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 14). 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 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 31) or you may convert a string to a list of characters by spacing out its characters and calling its `split()` method (lines 33 and 35).

(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'])
```

To make it easier for you to solve the two programming problems, we have provided three 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.

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.[1] 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 (`f3` ∘ `f2` ∘ `f1`)(`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.

[1]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.

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.

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, ['v', 'o', 'w', 'e', 'l'])
trace(f, 'e x c e p t i o n'.split())
```

Problem 1 (10 points)

Solve Problem 2.8 from the textbook. Please make sure to justify your answer by using examples.

Problem 2 (35 points)

Background: 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.)

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` $\rightarrow 2$

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

Table 1: Example outputs for the Soundex algorithm.

- `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 the 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.

Construct an FST in NLTK 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**: Performs steps 1-3 of the algorithm, i.e, retaining the first letter, removing letters and replacing letters with numbers.
2. **Transducer 2**: Performs step 4 of the algorithm, i.e., truncating extra digits.

3. **Transducer 3**: 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 iterate over an input file containing names (one per line) when redirected into the standard input:

```
python soundex.py < input.names
```

When you run the above command, the output should be something like:

```
Jurafsky --> J612
Jarovski --> J612
Resnik --> R252
Reznick --> R252
Euler --> E460
Peterson --> P362
```

For this problem, please turn in a file `soundex.py` that implements the Soundex algorithm and has the above input/output behavior, along with any explanation you feel is appropriate. We will run your program in exactly the manner specified above to check its output. We will also run your program with a held-out testset (i.e., a different set of names) to see if your implementation is correct.

Notes:

(i) You may use any number of transducers as long as the algorithm works.

(ii) While we have provided you with a sample input file 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.

Problem 3 (55 points)

Background: In the Hindi language, numerals can be spelled out just like they are in English. For example, the numeral 1211 is written as `one thousand two hundred eleven` in English and एक हज़ार दो सौ ग्यारह in Hindi. This Hindi string can be transliterated into English as `ek hazaar do sau gyaarah`. The following table shows the translations (both in Hindi and in transliterated English) for the numerals 1 to 20.

1: एक (`ek`)	2: दो (`do`)	3: तीन (`teen`)	4: चार (`chaar`)
5: पाँच (`paanch`)	6: छह (`chaha`)	7: सात (`saat`)	8: आठ (`aath`)
9: नौ (`nau`)	10: दस (`das`)	11: ग्यारह (`gyarah`)	12: बारह (`barah`)
13: तेरह (`terah`)	14: चौदह (`choudah`)	15: पंद्रह (`pandrah`)	16: सौलह (`saulah`)
17: सत्रह (`satrah`)	18: अट्ठारह (`attharah`)	19: उन्नीस (`unnees`)	20: बीस (`bees`)

Next we have the translations for the various numerical quantifiers:

hundred: सौ (`sau`)
thousand: हज़ार (`hazaar`)

There also exists a quantifier in Hindi that does not exist in English. This quantifier (लाख़ or `laakh`) is applied to numbers with order of magnitude greater than 10^5 and replaces the American convention of using `hundred thousand` for such numbers. For example, while the numeral 120415 is written as `one hundred twenty thousand four hundred fifteen` in English, it is written as एक लाख़ बीस हज़ार चार सौ पंद्रह (`ek laakh bees hazaar chaar sau pandrah`) in Hindi.

Construct an FST in NLTK that can translate any given numeral into its corresponding Hindi string. For the sake of convenience, assume that the following is true:

- The output of the FST should be the English transliteration of the Hindi string.
- Any given numeral N will *always* satisfy the following equation:
  $N = a \text{ x } 10^5 \ + \ b \text{ x } 10^3 \ + \ c \text{ x } 10^2 \ + \ d$
  where:

  $0 \leq a \leq 9$

  $0 \leq b \leq 20$

$0 \leq c \leq 9$
$0 \leq d \leq 20$

The sole purpose of these restrictions is to make sure that you will only ever need to use the translations provided above, e.g., you do not need to know the translation of any numeral higher than 20 to solve this problem.

Here are some examples you can test your code on:

Input	**Output**
5	`paanch`
102	`ek sau do`
1000	`ek hazaar`
120312	`ek laakh bees hazaar teen sau barah`

Just like for Problem 2, we have provided a file called `hindinums.py` with boilerplate code. You should add your code to this file. This file also contains driver code that will allow you to iterate over a list of numbers provided via standard input as follows:

```
python hindinums.py < input.numbers
```

When you run the above command, the output should be something like:

```
5 --> paanch
102 --> ek sau do
1001 --> ek hazaar ek
120312 --> ek laakh bees hazaar teen sau barah
9216 --> nau hazaar do sau saulah
```

For this problem, please turn in a file `hindinums.py` that implements the translation algorithm and has the above input/output behavior, along with any explanation you feel is appropriate. We will run your program in exactly the manner specified above to check its output. We will also run your program with a held-out testset (i.e., a different set of numbers) to see if your implementation is correct.

Notes:

(i) Note that the above equation allows the number 0 but there is no corresponding Hindi translation provided. For this assignment, you may output an empty translation for the number 0.

(ii) For this problem, we will not tell you how many transducers to use. You should figure that out on your own. We can also provide you with a couple of cryptic clues which might be helpful:

1. Characters can be words too.
2. Fixed length makes for simpler code.