Natural Language Processing **Summer 2017 Module 5**

Assignment 1

The goal of this assignment is to learn the implementation of one technique for determining distances between words or phrases called *Levenshtein Distance*. You will combine this with an approximation for the sound of names in Standard American English (SAE), called *Soundex*, which is described below, to find groups of similar names.

Requirements

Your implementation will find groups of similar-sounding names. There are 3 major requirements for your implementation.

Requirement 1: Accept parameters passed on command line. There will be two arguments as sys.argv. In order, they are: the names file; the number of clusters.

The names file is a plain-text file of names. The full version of this file is provided as `propernames`, and it contains a list of common American first names, one name per line. (Many of the names are historically Western European, but some originate from other parts of the world. On any OS X computer, these names are available at `/usr/share/dict/propernames`; however, you must use the version attached to this assignment in order to ensure consistency of results.) A function for reading this file is provided in the source file `name_cluster_manager.py`, but you are free to change it.

The number of clusters is an integer between 2 and 372.

Proper Name(s)	**Soundex Name**
California	C416
David	D130
Google	G240
Robert Rupert	R163
Tigger	T260

Table 1: Sample proper names and their American Soundex equivalents

Requirement 2: Convert each name read to the American Sounxex equivalent. The conversion procedure, from proper name to Soundex name, is as follows:

A. Make the first letter of the Soundex name equal to the uppercase first letter of the proper name.
B. Replace subsequent letters in the Soundex name with their (numeric) sound equivalent, specifically:

a. Letters `b`, `f`, `p`, or `v` in the proper name become “1” in the Soundex name.
b. Letters `c`, `g`, `j`, `k`, `q`, `s`, `x`, or `z` in the proper name become “2” in the Soundex name.
c. Letters `d` or `t` in the proper name become “3” in the Soundex name.
d. The letter `l` in the proper name becomes “4” in the Soundex name.
e. Letters `m`, or `n` in the proper name become “5” in the Soundex name.
f. The letter `r` in the proper name becomes “6” in the Soundex name.

Repetitions of consecutive characters in the Soundex name are not permitted. American Soundex does not consider vowels, or consonants ‘h’, ‘w’ or ‘y’ when encoding after the first character. Finally, American Soundex is limited to 4 characters. If the Soundex name is fewer than 4 characters, ‘0’ characters are appended in order to make it exactly 4 characters long. In Table 1, find some names and their Soundex equivalents.

Requirement 3: Cluster the Soundex names using AgglomerativeClustering as implemented in scikit-learn with an arbitrary number of clusters (between 2 and 372) and with all other parameters set to their default values. Use Levenshtein Distance as the distance between observations (names) when clustering. Output the resulting clusters, one per line, with spaces separating each name.

Implementation Provided

For this implementation, you are given source code in two python files: `name_cluster.py` and `name_cluster_manager.py`. The file `name_cluster.py` contains a “main” function which creates an instance of the class `name_cluster_manager` and calls two functions on it: `cluster_names()` and `print_names()`. While it could easily be incorporated into the `name_cluster_manager.py` source file, it has been refactored out to support grading. You are not allowed to change the source code in this file.

The source file `name_cluster_manager.py` contains a partial implementation for the manager class, meaning it has some of the functions you need to satisfy the requirements above, and it is missing others. Specifically, it reads the first argument as the `propernames` file and assumes the second argument is the number of clusters to form. It also does not perform any error-checking.

You are also provided two data files, `propernames` and a subset called `test_names`. The X file may be used to test your implementation. When executed as follows — i.e. to produce 2 clusters from test_names — the following is the expected result:

```
$ python name_clusterer.py test_names 2
John Jack Jim James
Roxana Roxane Roxanne Roxie
```

Finally, you are provided a source code file, `check_sound.py`, which checks the Proper Name / Soundex Name pairs in Table 1. If implemented as recommended, the following is the output:

```
Soundex works for Google
Soundex works for Robert
Soundex works for Tigger
Soundex works for David
Soundex works for California
Soundex works for Rupert
All soundex tests passed!
```

Implementation Required

You must complete the `name_cluster_manager.py` implementation and add additional classes for submission. Two classes are recommended for implementation, as described below, because the decomposition into these classes represents a reasonable object-oriented solution for this assignment. If implemented, these classes will be used by functions of the `name_cluster_manager` class.

Class `sound`: This class may contain a function called `get_soundex()` which accepts a string representing a name and returns the soundex equivalent for the argument.

Class `linguistic_distance`: This class may contain a function called `levenshtein()` which accepts two strings and returns the Levenshtein distance between the two strings.

Any implementation should comply with the Style Guide for Python Code, found online at https://www.python.org/dev/peps/pep-0008/, for the Style portion of the grade for this assignment.

Submission

Submit the source code for the `name_cluster_manager.py` file. Add any additional required source code, including additional classes such as `sound.py` and `linguistic_distance.py`. You may also add any comments in a README file (text, PDF or Word document) to help the grader understand or execute your implementation.

Grading

Your grade for this assignment will be determined as follows:

- 75% = Implementation: your class implementations must run successfully with the source files and data provided. It must produce the expected results, a sample of which appears in the Implementation section above. Any deviation from the expected results results in 0 credit for implementation.
- 15% = Decomposition: in the eyes of the grader, your solution follow the suggestions above or otherwise must represent a reasonable object-oriented decomposition to this problem.
- 10% = Style: your code must be readable to the point of being self-documenting; in other words, it must have consistent comments describing the purpose of each class and the purpose of each function within a class. Names for variables and functions must be descriptive, and any code which is not straightforward or is in any way difficult to understand must be described with comments. These points and more are described in the Style Guide for Python Code.

Late assignments will not be accepted.