**INLS 609-270 - Assignment 1**

You name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\* Please submit your answers as a PDF attachment via Sakai. Include your data and code as a **separate** zip file attachment.

\* General discussion is allowed, but you should compose with your solution **independently**.

**Question 1: Word Association (30 points)**

Implement a program that finds the top 100 word associations in a given collection of Amazon reviews measured by pointwise mutual information we introduced in class.You may want to (1) remove some punctuations (e.g., comma, quotation marks, and so on); (2) consider only those ordered word pairs that appear within a fixed size of text windows (e.g., window size = 5 consecutive words) for at least a certain number of times (e.g., 50 times). You might want to start by taking the basic statistics, such as the frequency of each word (number of reviews that a word appeared in) and the frequency of each word pair (number of reviews a pair of consecutive words appeared in). Examine the word associations you’ve found. Do they all make sense? Can you see something interesting or undecipherable?

**Question 2: Feature Selection (30 points)**

Implement a program that finds 100 single words (so-called “unigrams”) that are most associated with sentiment labels (label = 1: positive; label = 0: negative) in the given collection of Amazon reviews using Chi-square (χ2) we introduced in class. Examine the words you’ve found. For each of these words, can you tell which sentiment (positive or negative) this word is strongly associated with? Do you see “mysterious” words that do not clearly associate with positive or negative sentiment?

**Question 3: Spell Correction (40 points)**

Spelling correction is a common functionality provided by most search engines. The basic idea can be simplified as matching an out-of-vocabulary string to a word in vocabulary that is the closest in spelling (for example, "carrolina" -> " carolina"). In other words, we need a function that measures the similarity between two strings. Of course, for those who knew it, a natural measure of string similarity/distance is the Levenshtein Edit Distance (http://en.wikipedia.org/wiki/Levenshtein\_distance). You can find lots of implementations at http://en.wikibooks.org/wiki/Algorithm\_Implementation/Strings/Levenshtein\_distance. The problem in practice, however, is that the computation of edit distance is very costly, especially for long strings. Computationally it takes O(m\*n) to find the edit distance between two strings with m and n letters (which can be pretty ugly when m and n are large). Spelling something like "*Parastratiosphecomyia sphecomyioides*" correctly is a challenge, and figuring out "*Parastratioschecomia*" responds to "*Parastratiosphecomyia*" isn't easier.

Alternatively, if we make use of a smart representation of a word/string, we can potentially reduce the time complexity with an approximation. One such approximation is to break a long string into **overlapping** **tri-grams**. That means you can represent a string with a **set** of tri-grams. That says, "*parastratioschecomia*" becomes {par, ara, ast, str, tra, rat, ati, tio, ios, osc, sch, che, hec, eco, com, omi, mia}, and "*Parastratiosphecomyia*" becomes {par, ara, ast, str, tra, rat, ati, tio, ios, osp, sph, phe, hec, eco, com, omy, myi, yia}. If we quickly compare the two sets, we can see they are highly similar, high enough to draw the conclusion that the two original strings are similar. Even sweeter, it takes a time complexity of O(m+n) to compute the similarity of two sets, instead of O(m\*n).

You are provided with a dictionary (all words starting with "a" in wiktionary). Please implement a function so that for any input string, you can return the top 10 words in the dictionary that are the most similar to it. Use the Jaccard similarity that we introduced in class. Include the results for the following 5 strings:

abreviation

abstrictiveness

accanthopterigious

artifitial inteligwnse

agglumetation

Compare the results of your implementation with the results generated by the edit distance (you may use an existing implementation or package). Are you getting a good approximation?

One way to further tune the itemset representation of strings is to vary the length of the "*n*-grams" in your set. As you might already have guessed, an *n*-gram means *n* consecutive letters in a string. Try to use overlapping bigrams (2-grams), 4-grams, and 5-grams instead of tri-grams, and compare the results with the results using tri-grams. Which length of *n*-gram seems to work best for this task?