CS 341, Spring 2011
Programming Assignment: Finding Similar Sentences

All students taking CS341 are required to successfully complete this programming assignment by 5:00pm on Thursday, April 7. Please upload your solution to DropBox folder on `coursework.stanford.edu` (more details below).

Collaboration policy: This assignment should be done individually. It is okay to discuss general algorithm ideas with others. But please do not discuss anything specific to the code for this programming assignment or to your implementation with anyone else. Please also do not look at anyone else's code (including source code found on the internet), or show your code to anyone else. If you have questions about the assignment or would like help, please email us at `cs341-spr1011-staff@lists.stanford.edu`. The collaboration policy stated above applies only to this programming assignment. For the research project you will be doing, you are welcome (and encouraged) to talk to anyone about your work and use any open-source/etc. code you find on the internet (with attribution).

Feel free to use any programming language you like but we think that using C++ or Java is best (maybe Python). To run the program over the full data you will need a machine with about 4GB memory (you can use `corn.stanford.edu` to run your computations). Our C++ implementation took around 10 minutes to run on a fast computer and used 3GB of main memory.

1 Similarity of Sentences

In this problem set, you will implement some algorithm to find pairs of sentences that are copies or near copies of one another. Some useful methods are described in Chapter 3 of *Mining of Massive Datasets* by A. Rajaraman and J. Ullman, including locality-sensitive hashing and indexed based methods (Section 3.9).
You can get the chapter here: `http://i.stanford.edu/~ullman/mmds/ch3.pdf`.

Your goal is to quickly find pairs of sentences that are, at the word level, edit distance at most 1. Edit distance is defined for character strings in Section 3.5.5, but in this exercise you will think of sentences as strings and the words of which they are comprised as the characters. Thus, two sentences S_1 and S_2 they are at edit distance 1 if S_1 can be transformed to S_2 by adding, removing, or substituting a single word.

For example, consider the following sentences, where each letter represents a word:

- S_1: `A B C D`
- S_2: `A B X D`
- S_3: `A B C`
- S_4: `A B X C`

Then the following pairs of sentences are at word edit distance 1 or less: (S_1, S_2), (S_1, S_3), (S_2, S_4), (S_3, S_4).

The input data file has been provided for you at `http://snap.stanford.edu/class/cs341-2011/cs341-sentences-08.txt.zip`. This is a 500MB zipped (1.43GB unzipped) file that contains 9.4 million lines (sentences). File only contains sentences with 10 or more words in it(this information might helpful to you on algorithm devising phase). Each line of the file contains a single sentence. The first field in the file is the sentence id which is then followed by the words of the sentence. We already removed all punctuation, so all words in the sentence are separated by a single space.

1.1 Data Preprocessing

You should read the input file, parse it, and convert the words of the sentences to unique integer ID's. Replacing words with their integer ID's will later allow you to more efficiently work with the data. A simple strategy for this conversion is to simply read over the input data and build a hash table that maps words to unique ID's.

In the example below each word is replaced by its word id.

- *Input*1: `I love big data mining`
- *Input*2: `You love big data base`
- *Output*1: `1 2 3 4 5`
- *Output*2: `6 2 3 4 7`

Also, you should remove duplicate sentences from the file. Doing so is fairly simple; hash sentences to buckets and examine buckets for duplicates. To make the result deterministic, eliminate the sentence with the larger ID whenever you encounter two identical sentences.

Implementation tip: When testing/debugging your code use small examples and small datasets before you run it on the full data set. This will considerably speed up your progress.

Generally it will also be useful for you to save intermediate outputs of each step so that you do not need to parse/generate all the data from scratch.

1.2 The Hard Part

Now, you need to implement an algorithm that finds all pairs of sentences that are at edit distance exactly 1, assuming you have already eliminated duplicates. There are far too many pairs of sentences for you to consider all pairs, even if you make extensive use of parallelism. Thus, you need to use some method of similarity search, and you should think carefully about how you do this. It is OK to miss a small fraction of the similar pairs of sentences if it speeds up your algorithm. Some things to think about:

1. Shingling plus minhashing doesn't work too well for edit distance. For example, if we use 3-shingles of words, the sentences A B C D and A B X D have Jaccard similarity 0, even though their edit distance is only 1.

2. A possible way to reintroduce the technology we learned for Jaccard similarity or distance is to regard (temporarily) sentences as sets of words rather than lists of words.

3. Another approach is to modify the index- and length-based techniques of Section 3.9 to handle edit distance.

1.3 Output Generation

Each line of the output file `cs341-sentences.out` should contain two numbers separated by space: `"<I> <J>\n"`, which means that sentence IDs I and J are at word edit distance of 1 or less. You need to save each pair only once (i.e., only save when $I < J$). You do not need to save pairs of identical sentences.

Implementation tip: You can quickly check whether two sequences I and J (assume I is longer than J, $|I| \geq |J|$) are edit distance 1 apart. First if I and J differ in length for more than 1 element, then they are surely more than edit distance 1 apart. Second, we can then do the following: first we traverse over the common prefix of I and J and the check for two cases: addition/delection of a element and substitution of a element: (addition) skip 1 element of I and then share the rest of the elements with J (since J is shorter this is the same as inserting an element in J); (2) skip 1 element in both I and J (this corresponds to substitution) and then check that I and J share the rest of the elements.

2 Submission Instructions

Please submit your solution by uploading a zip file to your DropBox folder on `coursework.stanford.edu`. Upload all your code and the output file `cs341-sentences.out` that contains IDs of pairs of sentences that are at word edit distance of 1 or less. Each line of the output file should contain 2 *space* separated numbers: `<I> <J>\n`, which means that sentence ids I and J are at word edit distance of 1 or less. You need to save each pair only once (i.e., only when $I < J$). You can also include a README if there are notes that you would like us to look at. A good solution should should run and produce good results.

3 Contact

This programming assignment is (by design) more open-ended than most assignments you might have seen in other classes (including CS246, CS221 and CS229). If you have questions, do not understand parts of it, find parts of it ambiguous, or need help with C++/Java/Python, do not hesitate to email us at `cs341-spr1011-staff@lists.stanford.edu` to ask for help. In case you have questions about the lecture notes or want clarifications pertaining to the programming assignment, we will also have two office hours from 9-10am on Friday, April 1 and 9-10am on Tuesday, April 5 in Gates B26B.