Zhiping Xiu, HW2

1:

A):

Features:

<Daily Average Post Count, Follower Count, Followee Count, Monthly Average Online Time (hours)>

Possible values:

Daily Average Post Count: {float point number} >= 0

Follower Count, {integer} >= 0

Followee Count, {integer} >= 0

Monthly Average Online Time: {float point number} >= 0

This set of features can be used to evaluate twitter user's activity level.

For example:

We can use this features to group users to high active, medium active or low active. And then we can group peoples to this groups accordingly. We know facebook have tons of post everyday, we can use this grouping to see if where are all this post are come from, like are they come from a small group of very talkative people, or large group of average talkative people.

B):

One possible distance measure between two twitter users based on feature set mentioned in A can be described as follow: **(we use A, B, C and D to represent Daily Average Post Count, Follower Count, Followee Count and Monthly Average Online Time respectively)**

I want to define the metric as:

$$d(U_1, U_2) = W_k * |A_1B_1 - A_2B_2| + W_j * |C_1D_1 - C_2D_2|$$

Explain: A * B, which means daily average post count * follower count, this product can represent how much you can influence other people, let's say you post 5 post every day, and you have 500 follower, then the product is 2500, you might say that you influence power is 2500 * Wk, the more you post and the more people are following you, you get more chance to influence other people with your post.

Similarly, C * D can be roughly interpreted as the degree that you can be influenced, if you have many followee who post on twitter every day, and you got a lot of time online, you are more possible to contribute to the community by reading other's article, (this directly contribute nothing, but this can bring side effect, like comments or "like").

When it's about to compare the difference of activity between two users, we might want to evaluate both how much they can influence others, and how much they can "listen" to others. So we get the absolute value of difference in both to degree, and apply a proper weight to upon them.

This distance measure is metric, here comes the proof:

We need to proof reflexivity, non-negative, symmetry, triangle inequality.

Reflexivity: $d(U_1, U_1) = W_k * 0 + W_j * 0 = 0$

Non-negative: since we are collecting the absolute value, >=0 is promised

Symmetry: similar to non-negative, collecting absolute value can promise symmetry

Triangle inequality: Consider it this way: The value of A and B has no influence on the value of C and D. So we can combine AB to one degree, and CD to another one, then the metric reduced to a Manhattan Distance of AB and CD. Which follow the Triangle inequality rule. (With this, we actually don't need to prove the other three features: reflexivity, non-negativity and symmetry)

C):
No, Do not treat values for all three elements equally.
Because possible values for each elements have different scale. For example, people tend to have 100,000 hairs on their head and weighted 70 kg, and about 5.9 feet height. If we treat all three elements equally and calculate Euclidean distance, weight and height is no longer matters in front of overwhelmingly large number of hairs.

Before we start to design a metric, we first arbitrarily consider the data:
Child usually to have lower weight and height than other group.
Teen would be higher than child in weight and height, and lower than the other two.
Middle-aged and elderly can have similar height and weight.
Elderly can have less hair than other three groups.

We want to apply weight to three features, so that hairs can have less weight than the other two. So the group of points representing each kind of classification would looks like ellipsoid. And it is nature to think higher people tend to have heavier weight than shorter people, we can expect the ellipsoid might be not parallel to corresponding axes.

So, I will want to use Mahalanobis distance for this cause, since it's weighted and tilting is allowed for the data ellipsoid.

D):
The paper introduced a way to calculate the metric between two string S_a and S_b, and a method to get the actual editing operation required to get string A to string B.

Consider the DNA string, which is constructed by only 4 possible "letters": A, G, C, T. The influence would be there are much higher chance to find derived letters between S_a and S_b. Or using the paper's word: there would be much common to see "line" between two strings. This would cause the distance expected usually smaller than 26 letters string comparison.

According to the paper, the metric nature won't be influenced by the string composition, so we don't need to worry about that either. But, considering a little change of the composition of DNA may hugely change it's characteristic, apply a metric designed for a string on this is controversial. Maybe we can improve this by adjusting the parameters of the algorithm, according to the dataset.

2:

A):

To test levenshtein_distance() and dictionarywords(), open Python console under /code directory.

```
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from spellcheck import *
>>> levenshtein_distance('tinker', 'banker')
2
>>> dictionarywords = load_dictionary('3esl.txt')
>>> find_closest_word('tincker', dictionarywords)
'tinker'
```

B):

To test this, call following command from /code directory:

```
python spellcheck.py 'spellcheck_input.txt' ‘3esl.txt'
result saved to corrected.txt
```

Results of different calls will be stored at the same file, split by double newline character ‘\n \n’.

So far, the correction results is not so good.

C):

To test measure_error(), open Python console under /code directory.

```
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from spellcheck import *
>>> typos, truewords = load_typo_tuple('wikipediatypoclean.txt')
>>> dictionarywords = load_dictionary('3esl.txt')
>>> measure_error(typos, truewords, dictionarywords, 1, 1, 1)
(wait for couple of minutes)
>>> 0.20067516879219804
```

The function is upgraded using the method described in question 3-B to achieve shorter responding time.

3:

A):

Run measure_error on data wikipediatypo.txt using 3els.txt, on my macbook pro would spend about 6000s (without using the method described in 3-B, please notice when you try that on problem 2-C, you are using the code that upgraded by 3-B. The exact number is near 6100 seconds, but I forget to write that done).

Try 64 kind of parameter combinations would spent less than (but similar to) 108 hours. Considering parameter like 0, 0, 0 would be easier to calculate, that's why I said it's a little less than 108 hours, but it's not going to change a lot.

B):

Searching the dictionary exhaustively every time for checking one word in testing set could be very time consuming. For this sake, we can split the dictionary to narrow the searching scale, to save time. And each time when we are looking for a word similar to a certain typo, we only search the subset that most likely to contain the answer.

So, I divided the dictionary to 26 parts, each parts include the words that start with certain letter. For example dictionary[0] will give you the sub dictionary containing all the word that start with 'a'. In this way, only a small part of dictionary has been searched for every typo.

The shortcoming of using this method is, it's not promised to return the exact answer as searching the dictionary exhaustively. For example a typo "toute" might be more expected to be translated to "route" (Levenshtein distance = 1) than "touts" (Levenshtein distance = 3). We have to trade accuracy for scalability (time-wise).

And then, we can further cut off the irrelevant testing set, wikipediatypo.txt include about 1.5k words that do not exist in 3els.txt. Testing those words that don't exist in our training set is meaningless, since a wrong answer is promised. So we can use wikipediatypoclean.txt instead. Which only include the words whose correct spelling is an entry in the dictionary file 3els.txt. Using this subset and the dictionary dividing method mentioned earlier, we can shorten the time usage for each measure_error to about 70 seconds, 4 seconds for some iteration whose input is 0, 0, 0 or 0, 1, 0. Then we can finish the experiment in 1 hours.

To test find_closest_word_fast(), open Python console under /code directory.

```
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from spellcheck import *
>>> dictionarywords = load_dictionary('3esl.txt')
>>> find_closest_word_fast('tincker', split_dictionary_by_alphabet(dictionarywords),
1, 1, 1)
'tinker'
```

Please don't try invalid words on this function, like '~@":{#', which is not start with an English letter, in practical usage like measure_error, we have try-catch mechanism to handle those situation, but we don't have it inside this function since we presume all the input are already valid.

C):

In this experiment, I did almost exactly like 3-B said, with only one thing different. You can try it on any dataset you like, for 1 hour runtime you can use wikipediatypoclean.txt, but the dataset I used to produce the result is described as follow:

Instead of using wikipediatypoclean.txt given with the assignment, I created a wikipediatypocleaner.txt, which is similar to wikipediatypoclean.txt, except it not only included the word starting with a~c. wikipediatypoclean.txt only has the word starting with 'a', 'b' and 'c', wikipediatypocleaner.txt, however, include all the typo tuple whose correct spelling owned by both wikipediatypoclean.txt and 3els.txt. In this way, the time would be a couple hours longer than 1 hour, but the waiting time is till reasonable, and we can get a more reasonable answer.

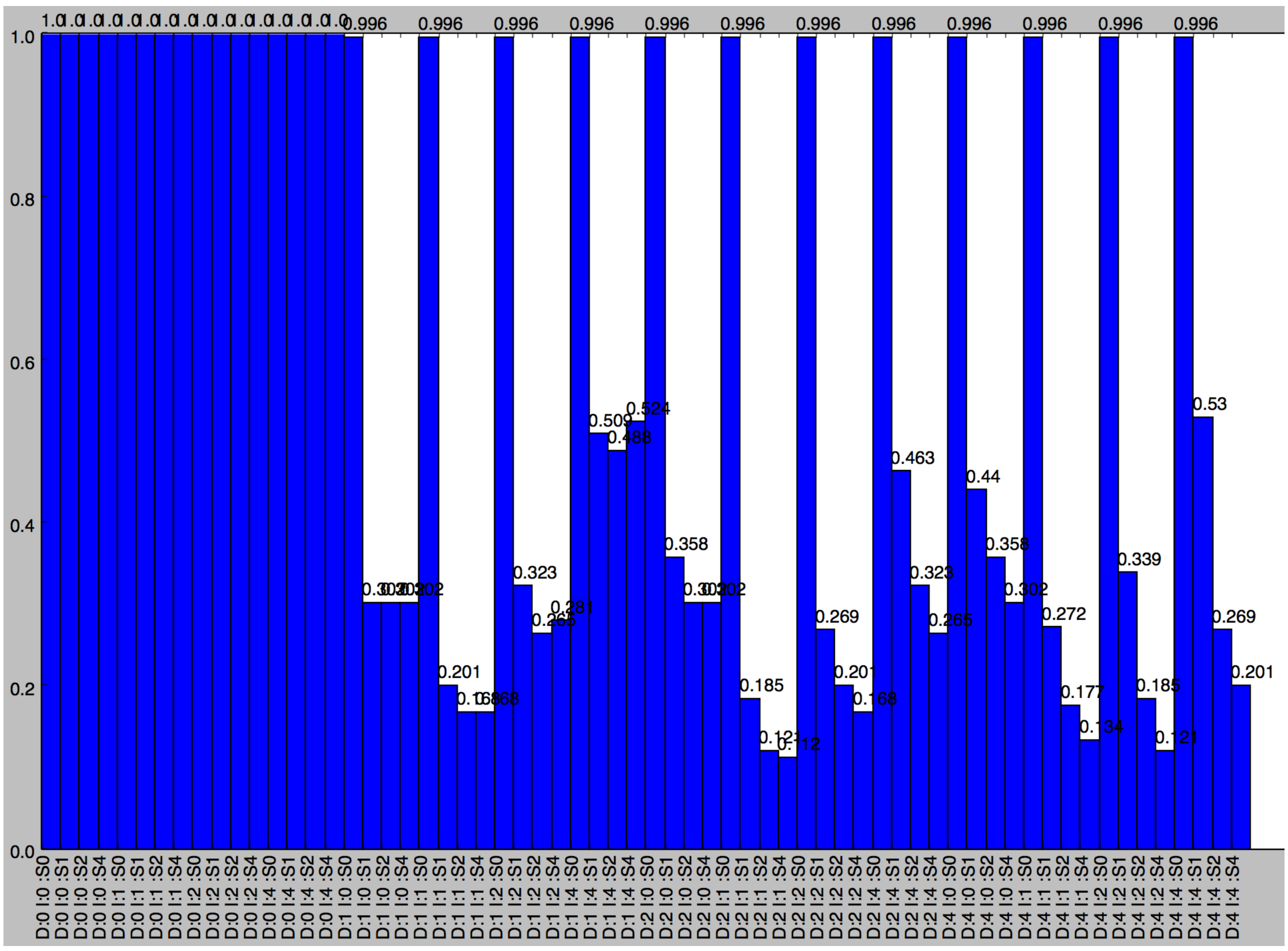
To test this new method by yourself, open Python console under /code directory.

```
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from spellcheck import *
>>> try_best_cost_combination()
```

This will use the default file combination described in front, if you want to change parameters to get a faster response (within one hours):

```
>>> try_best_cost_combination('wikipediatypoclean.txt', '3esl.txt', 'best_cost_combination_record.txt'):
```

Then wait about 3 hours (10957.1081541 seconds). The result will be output to best_cost_combination_record.txt.



Then, you can get a bar chart according to the result:

```
>>> from spellcheck import *
>>> lst, legend =
read_best_cost_combination_record('best_cost_combination_record.txt')
>>> plot_bar_chart(lst, legend)
```

Each bar on the chart is map to one kind of combination of three parameters, deletion_cost (D), insertion_cost (I), and substitution_cost (S). And the height of each bar corresponding to the error rate. I'm rounding the numbers for better appearance. You can checkout the full output in file best_cost_combination_record.txt.

As you can see from the chart, the optimal combination is deletion_cost = 2, insertion_cost = 1, substitution_cost = 4, with that combination you can get about 11.2% error rate with our input data.

4:

A):

Calculate the manhattan distance every time could be time consuming, but we can pre computed all the possible combination and store the result to a [26 * 26] list for future use. In my program, function get_keyboard_distance_table() was intended to calculate that list. And the list is already calculated and hard coded as global variable for another function qwerty_substitution_cost(c1, c2), which is used to calculate the qwerty substitution cost between two different letter.

Function qwerty_levenshtein_distance(string1, string2, deletion_cost=1, insertion_cost=1) is very similar to function levenshtein_distance(), except every time we want to get a substitution cost, instead of getting it from parameter directly, we use qwerty_substitution_cost(c1, c2) to calculate it using current letter considered. Since substitution cost is no longer given directly, we don't need to set it as one of the parameters.

To test qwerty_levenshtein_distance() and qwerty_find_closest_word_fast(), cd to /code directory, enter python console:

```
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from spellcheck import *
>>> levenshtein_distance('tinker', 'banker')
2
>>> qwerty_levenshtein_distance('tinker', 'banker')
4
>>> dictionarywords = load_dictionary('3esl.txt')
>>> qwerty_find_closest_word_fast('tinler',
split_dictionary_by_alphabet(dictionarywords))
'tinker'
```

B):
To test this experiment by yourself, open Python console under /code directory.

```
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from spellcheck import *
>>> qwerty_try_best_cost_combination()
```

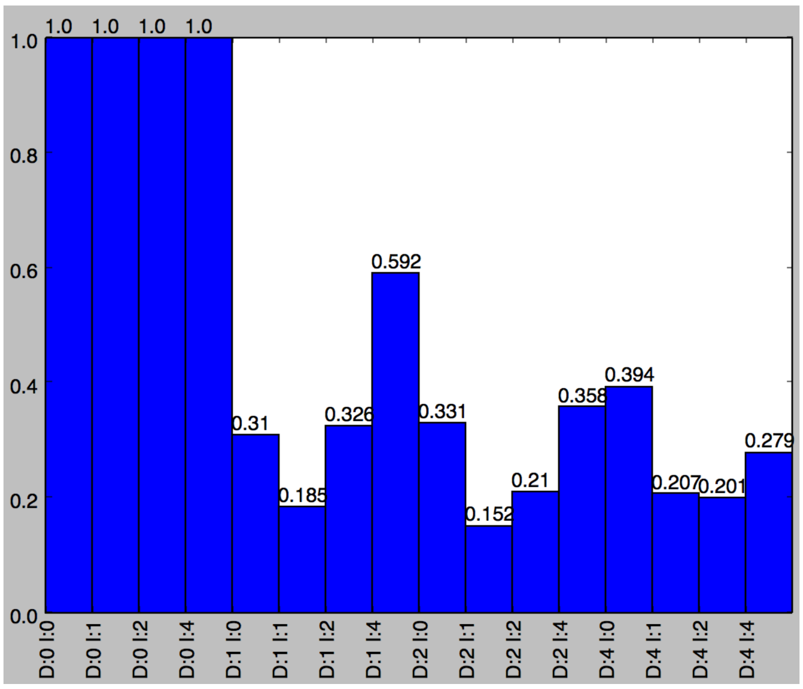
Then wait about 3 hours (10957.1081541 seconds). The result will be output to best_qwerty_cost_combination_record.txt

Similarly, you can watch the output by:

```
>>> from spellcheck import *
>>> lst, legend = qwerty_read_best_cost_combination_record()
>>> plot_bar_chart(lst, legend)
```

As you can see from the chart, the optimal combination is deletion_cost = 2, insertion_cost = 1, with that combination you can get about 15.2% error rate with our input data. You can checkout the full output in file best_qwerty_cost_combination_record.txt.

The result showing that qwerty_levenshtein_distance is not better levenshtein_distance. A guess is: The typo in the data set we are using is not highly related to qwerty_levenshtein_distance. Maybe our assumption is not suitable for this dataset.



There are another possible reason: the dataset we are using only included all the possible typo, but didn't include the appearance rate of each kind of typo. For example, one people can typo "hello" to "hwllo" many time a day (which is sensitive by qwerty_levenshtein_distance), but only typo "atlas" to "atlos" (which is sensitive by normal levenshtein_distance) once, those two typo are weighted same in our dataset. If we give hwllo higher weight since it appear more often than atlos, we maybe find qwerty_levenshtein_distance more accurate.

<u>Please notice:</u>
<u>The code is split to two file, spellcheck.py and helper.py</u>

<u>spellcheck.py is only for ML logic</u>
<u>helper.py handle file reader, writer, and other practical function</u>