Problem 1:

1. Feature Set for Facebook Account:

#of Friends:

#of Posts:

#of Pictures:

This feature set can be extended with many more features, however, I think these three will give sufficient information about a facebook account to begin with. Based on these features, I conjecture I can properly categorize people into groups of very active facebook users, normal facebook users, and non-active users. They can all be encoded in positive integers, part of the real numbers. The range on friends will be somewhere between 0 and 5000 (fb’s limit). The range of posts will go from 0 to technically infinity, although more realistically, the upper limit will be in the range of a few thousand posts.

1. In order to measure distance, I will define a classical Euclidean distance between the elements with features {#ofFriends, #ofPosts,#ofPics}. The Euclidean Distance between two points (x,y) given these three features will be expressed as:

Where x1 corresponds to #ofFriends, x2 to #ofPosts, x3 to #ofPics. For the Euclidean distance measure of 3dimensional points it can be shown that it is a metric. First, d(x,y) = 0 for all points in this space that have equally valued features (coordinates) are equal. When looking at the facebook account as a point in a 3-dimensional space, this is obvious, since all points with the same coordinates have a distance of 0 to each other and the same distance to the origin (reflexivity). As we are taking the square of the individual differences, our difference measure will always be ≥ 0 (non-negative). Furthermore, the “point in a 3D coordinate system” analogy also supports the third requirement of a metric, that is: d(x,y) = d(y,x). The distance of one point to another does not depend on the direction we are taking: Euclidean distance measures the shortest distance between two points. Finally, the triangular property applies, too. To verify this, let’s look at the general proof of the triangular property on a metric defined as the shortest distance in n-dimensional space:

Macintosh HD:Users:mrmaster:Desktop:Screen Shot 2015-10-09 at 10.04.43 AM.png

Given 3 points x,y,z we need to prove that d(x,z) ≤ d(x,y) + d(y,z). Let for 1 ≤ k ≤ n. This means we look at the maximum distance in a single (the k-)dimension. This brings the problem down to a one-dimensional one. Now,

**q.e.d.**

1. Expected range of values for these elements are:
   * Height in feet: 1.5 – 9.0 ft (excluding unborn babies)
   * Weight in Kg: 3.0 – 250 kg (excluding the really obese people)
   * Number of hairs: 0 – 300.000

I don’t think it would make sense to weight all features equally. The number of hairs (or more precisely follicles) is constant throughout life and thus not an indicator of age. The only thing you loose when you “loose hair” is the actual hair strands that come out of the follicles. Thus, I would decide to completely neglect the number of hairs and just look at the combination of weight and height to determine a sample person’s category. However, I do not think that this will provide accurate measures for the person’s classification. There are just too many possible combinatinos for height and weight at all ages that I don’t think this could work reliably. If I had to work with this data, I would try to define an Euclidean distance with a weight on the height and “hope for the best”.

1. If we just look at a simple strand of DNA consisting of a sequence such as “atgc” then it would be possible to measure the distance to another string. However, the distance measure we have used so far doesn’t give any information on which part of the string is incorrect. In addition, since there are only 4 letters used in the DNA-alphabet, it would be hard to differentiate between the individual possible combinations. Two strings with the same distance to a sample could have a very different structure, after all. Thus, it would be nearly impossible to match with high accuracy between a set of two equally sized strings. The metric would need to be adapted to take into account the lower number of available letters and the fact that the dictionary words we could potentially use might only differ in one single letter, and not like with the spellchecker bigger parts or even the entire word. Having only 4 letters available really is the reason why this most likely wouldn’t work.

Problem 2:

1. The program is implemented in Python. By default, it will only run the specified tests for problem 2. For all other code (problem 3,4) you will have to uncomment the corresponding sections.

Problem 3:

1. Note: I conduct all testing on my Dell Precision Work Tower with 64GB RAM and an Intel XEON E5 processor. First, I test 100 random samples; these take 75.06 seconds to run through. I get 40.0% error. The wikipediatypo.txt file consists of 4223 typos. Thus, I conjecture that the entire process would take around 3169.8 seconds ((4223/100)\*75.06). I repeat this experiment on the entire wikipediatypo.txt dataset. I measure an effective runtime of 2915.84 seconds with an error rate of 50.5%. If I were to do the grid search on the entire data set, this would take (64\*2915.84) = 186613.76 seconds (2.1 days). 10 fold cross validation would mean running the algorithm on 9/10th of the data, 10 times. Thus, the total runtime would be (64\*(2915.84\*9/10)\*10) = 1679523.84 seconds, or 19.44 days.
2. In 1 hour, I can run 5213 (4223\*3600/2915.84) tests on my machine. To maximize the number of parameter combinations I can test, I decide not to do cross validation, but rather focus on optimizing the measure\_error output. 5213 trials would leave 81 tests per combination. Looking at the difference in measured error between the 100 runs and the full runs (from before) I see that this is not sufficient. I decide to run a test to see how the error converges over time. I decide to take a differential rolling window approach with 20 tries in the window. If these 20 tries are all within 1% range, then I stop. The code for this rolling window is included but commented out in the source file provided. I end up with a suggested 232 tries. A plot of the average error for each sample with 232 trials is shown in Figure 1. This gives 232 trials per parameter combinations to get an accurate estimate of performance. First I speculate that setting any of the costs to 0 doesn’t decrease the error rate, but rather increases it. Thus, I decide to validate this by running the experiment once with all 64 parameters: the ideal combination remains the same as the one with only 27 combinations and the combinations with 0s perform worse than the others. This reduces the possible number of combinations from 64 to 27. 5213/27 = 193. This is already very close to the number we had determined as possible number of trials per combination of parameters. I decide to use this number to calculate the measured error for all 27 possible combinations. So in the end, I am varying the individual costs in the set {1,2,4} on 193 trials each. This will result in an experiment that will (on average) last 1 hour and in the end return the setting combination with the minimal measured error. I use the full dictionary and a random subset of typos and truewords for the experiment. The random set is selected at each restart of the program. The set of typos and truewords remains the same for all parameter combinations.

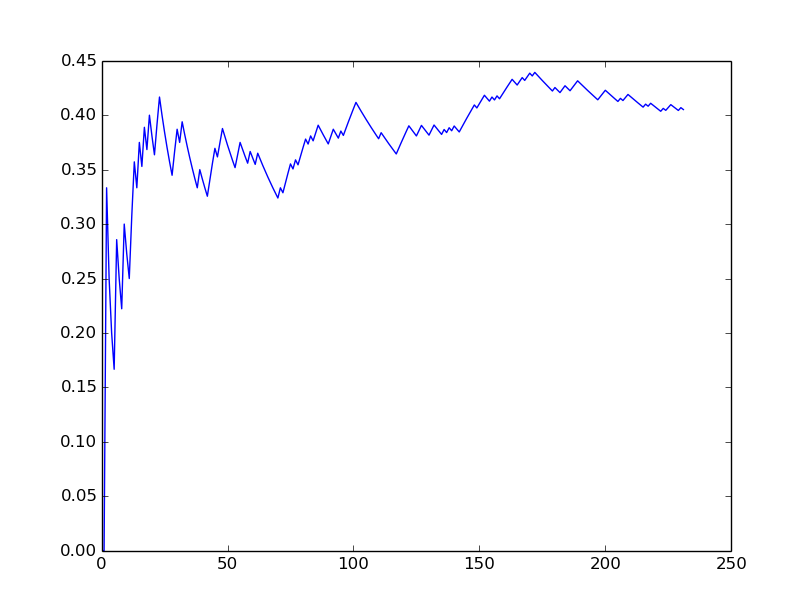
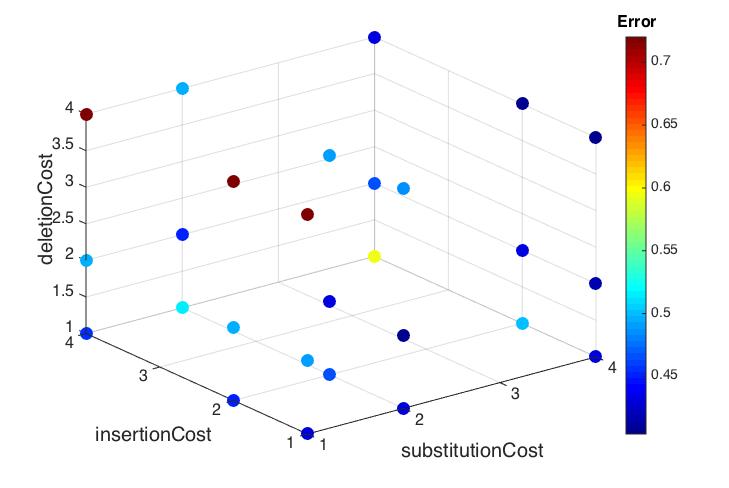


Figure Error vs. number of tries

1. I run the experiment on the full Wikipedia file for one hour. From the results, we can see that with this method we determined the ideal setting to be at substitutionCost = 2, deletionCost = 2, and insertionCost = 1. A 3D scatter plot of the error as a function of these 3 parameters is shown in Figure 2. The color of each dot corresponds to the error. The darkest dot is the one with the lowest error rate and thus the best.

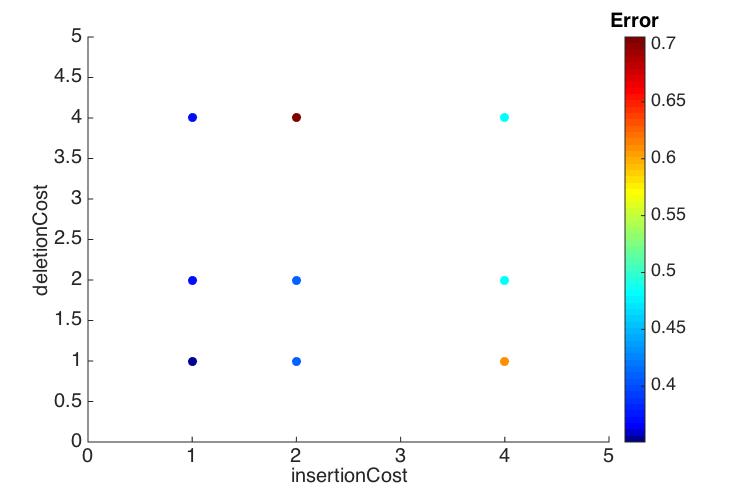


**Normalized Error Rate**

Figure Error Graph

Problem 4:

1. The function is implemented in the python file. I decide to use a matrix representing the keyboard with origin at “1”. “Shifted” rows are aligned in the matrix. E.g. the cost from “a” to “z” is 1.
2. In this setup, there are 2 variables to vary in a set of 4. This would give me 4^2 (=16) different combinations. First, I need to see how the introduction of the qwerty distance function affected runtime performance. For this, I time a trial consisting of 160 tests (10 words per combination). This time, I measure a significantly worse time-performance. In total, it takes 1180 seconds to run this experiment. This scales up to only 488 tests per hour. I decide to neglect setting any of the parameters to 0 again, reducing the number of possible combinations to 9. This gives me 488/9 = 54 trials per combination. I run this experiment for approximately one hour (3708.82 seconds). The ideal setting returned, is: insertionCost = 1 and deletionCost = 1 with an average error of 35%. Again, we plot the error corresponding to the to varied parameters. The result is shown in Figure 3. In terms of mean error, the qwerty\_distance measure for substitution cost is a better approach as it lowers the average error by around 10-15% percent. However, this comes at a significant tradeoff in performance, where we can only run a fraction of the previous tests in the same time. Thus, one would have to very precisely consider the kind of application that one is implementing the spell checker for, before deciding which one to implement.



**Normalized Error Rate**

Figure Error Graph 2