Problem One
A) Feature set <number of friends, number of photos, number of videos, number of statuses>

This feature set consists of a variety of indicative features that can be measured using real numbers. Number of friends, number of photos, number of posts, and number of videos can all be measured using any integer greater than or equal to 0. This feature set is useful is gauging what type of Facebook user it is as it clearly represents user activity and engagement.

B) Given the above feature set, a metric could be the total number of likes the profile has. This measurable could be clearly compared across profiles, and would be a good indicator of how popular a given user is. If a user receives more likes than another user when taking into account pictures, videos, and posts, then this is a holistic view of general online popularity.

C) No it does not make sense to put people in a Euclidean 3-space. I would place the most weight on number of hairs on head. This metric is not dependent on gender, because no matter what it is likely that a child has more hair than an elderly person; this trend is clear across gender lines. That being said, the distribution would definitely be more extreme for the male gender. I would then weigh height and weight at the next tier, as they are less indicative of age. While it is true a shorter, lighter person is probably a child, this is not as clear of a pattern and is a bit more difficult to discern across gender.

D) As explained in the paper, one could assess the distance between strings based on how many edits are necessary to make them identical. The three different means to edit a given string are insertion, deletion, and replacement. Weighted equally, these methods can determine the distance between two strings by counting how many times each needs to be applied. This metric could also be used to discern the distance between two DNA strands. Because DNA strands are encoded with 4 bases, each represented with a letter, their representations take the same format as a standard string. Because of this, the method described in the paper could be used as it stands to determine the distance between two DNA strands. In order to see how two DNA strands' bases compare, this algorithm could assess the distance between their two string representations.

Problem 2
****PLEASE REFER TO THE README TO SEE HOW TO RUN THE FOLLOWING***

A) Implemented in file spellcheck.py

B) Implemented in file spellcheck.py
Can be run using command: python spellcheck.py <ToBeSpellCheckedFileName> 3esl.txt

C) Implemented in file spellcheck.py
Can be run using command: python spellcheck.py <Typo/CorrectDuoFileName> 3esl.txt 1
The last argument 1 indicates that measure error will be run. It is important that wikipediatypo.txt, etc. are used for this command, as the program runs under the assumption that each line consists of the typo and the corresponding correct word.

Problem 3

A) Because the wikipediatypo.txt file is extremely long, 4223 lines, I took sample sets of the file to determine how long it would take to check the entire file. I first created a test.txt document that consisted of 10 lines (10 pairs, 20 words) of the original wikipediatypo.txt document. Using the time function, it took 12.2094750404 seconds to run measure_error on the 10 lines with the entire 3esl dictionary. Then, I edited the text.txt file to have 20 lines (20 pairs, 40 words) of the original wikipediatypo.txt file. Using the time function, it took 26.0232889652 seconds to run measure_error on the 20 lines with the entire 3esl dictionary. Through these two calculations, one can extrapolate that the inputted file's size (#words) and the time it takes to measure error is relatively linear. With this information, I did the following math calculation.

4223 / 10 = 422.3 sample sets of size ten within the entire file
(422.3 sets) * (12.2094750404 sec) = 5,156.0613096 seconds
5,156.0613096 seconds / 60 = 85.93435516 minutes
85.93435516 min / 60 = 1.4322392527 hours

Thus, it would take approximately 1.4 hours to run measure_error on the entire wikipediatypo.txt file using the 3esl dictionary.

A variation on this process would be testing how long it takes to run all 64 parameter combinations for insertion, deletion and substitution costs among the values in the set {0, 1, 2, 4}. Because the above calculation represents running measure_error using one possible combination within this value set, this process would take approximately 64 times longer.

1.4322392527 hr * 64 = 91.663312173 hours
91.663312173 hr / 24 = 3.8193046739 days

Thus, it would take approximately 3.8 days to run measure_error on wikipediatypo.txt using the entire 3esl dictionary, and testing all 64 parameter combinations.

Section done for v1 of HW: Another variation of this process would be testing how long is takes to run 10-fold cross validation on the wikipediatypo.txt file and testing all 64 parameter combinations. Using this method, you would divide the wikipediatypo.txt file into 10 equal sections. Then, you would run measure_error using all 64 combinations on 90% of the file to determine the best parameter combination. After, you would measure_error on the remaining 10% of the file using the chosen best parameter combination. This process would repeat 10 times, cycling through each 10% of the file. Given this method, I did the follow math computation to determine approximately how long it would take.

4223 * .10 = 422.3 lines (10% of the file)
4223 lines *.90 = 3,800.7 lines (90% of the file)

3,800.7 lines * (12.2094750404 sec / 10 lines) = 4,640.4551786 seconds
4,640.4551786 seconds * 64 combos = 296,989.13143 seconds

422.3 lines * (12.2094750404 sec / 10 lines) = 515.60613094 seconds

296,989.13143 sec+ 515.60613096 sec = 297,504.73756096 seconds
297,504.73756096 sec * 10 = 2,975,047.3756 seconds
2,975,047.3756 sec / 60 = 49,584.122927 minutes
49,584.122927 min / 60 = 8,264.0204878 hours
8,264.0204878 hrs / 24 = 344.33418699 days

Cross-validation would take much longer, but it would provide more accurate data because the training data and testing data are separated. That being said, a way to expedite this process would be to divide wikipediatypo.txt alphabetically, and test it against a subset of the dictionary that corresponds to the given set's alphabetical order. This would be quicker because it would iterate through a smaller dictionary, but it would compromise the test results because it is not fully testing our model.

B) In designing this experiment, I wanted to ensure that it would run under an hour and properly represent the entire model. Because of this, the entire fixed dictionary and all parameter combinations from set {0, 1, 2, 4} are tested. It is important to properly vet closest_word() and determine how many false classifications the algorithm produces. By tapering the dictionary to better fit the sample set, the experiment would produce deceivingly low error measurements. Further, I cannot logically cut down the parameter set and would rather test more permutations, so I decided to utilize the entire set. I will use wikipediatypo.txt because it uses real-life data, and does not scrub the words whose correct answer is not in our dictionary. Choosing wikipediatypoclean.txt would deceivingly skew our results, as we are deliberately affecting our data set to better fit the given dictionary. Our model should function under zero assumption about the relationship between our typo file and dictionary file. In turn, it makes sense to use wikipediatypo.txt.

While cross validation would be a rational approach because it separates training and testing data, it would be a very time consuming process as indicated in the previous calculation. In turn, this method would have to utilize a smaller sample size, compromising the breadth of words that the function would run on. In turn, I chose to have a greater sample size, larger breadth and forego the benefits of cross validation. Below is the calculation I used to determine the largest sample size that measure_error could evaluate using the entire fixed dictionary that is still under an hour.

1 hour = 60 min
60 min * 60 = 3600 seconds

64x = 3600 seconds
x = 3600/64
x = 56.24 seconds/iteration

The above calculation finds that given this hour constraint, each iteration can take max 56.24 seconds.

takes approximately 15 seconds per 10 lines of wikipediatypo.txt

approximately 55 seconds allocated per iteration

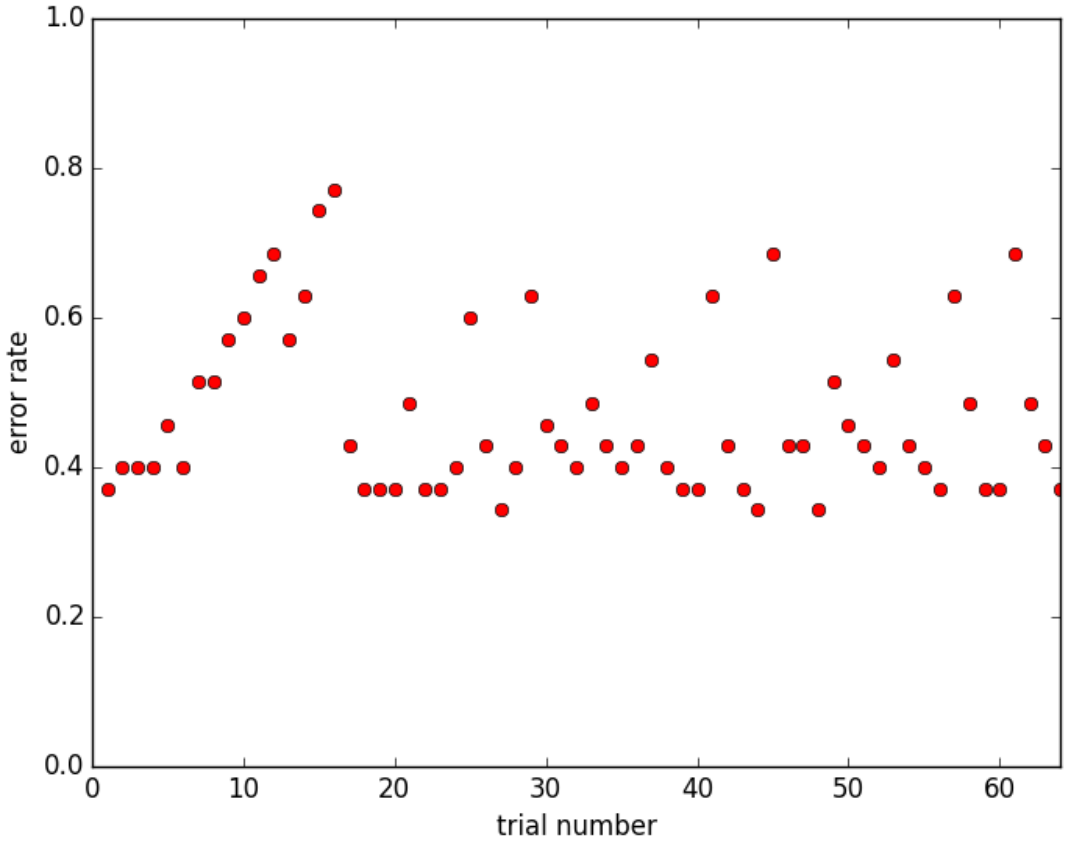
(15/10) x = 55
15x = 550
x = 36.6666 lines
Thus, sample size will be 35 lines.

The above calculation takes the time constraint and determines that the sample size can be a maximum of 36.666 lines. I approximated this further to be 35 lines (35 pairs, 70 words).

Based on these computations, the experiment is as follows.

1. **64** iterations testing all combinations of parameters **{0, 1, 2, 4}**
2. Sample: **35 lines of wikipediatypo.txt**
3. Entire fixed dictionary: **3esl.txt**
4. Run time **< 1 hour**

C)



Here is a graph of my results from the experiment I designed in Part B. The x-axis represents the trial number, and the y-axis represents the corresponding error rate given the parameters used

during that trial. I did not include the exact parameters used per trial on the x-axis because I thought it would look visually cluttered and would take away from the graphical representation. That being said, below are the corresponding parameters used per trial. Based on this experiment and sample data, the best parameter is: (2, 3, 3) with the error rate 0.342857142857.

Trial Number: Test Parameter

1. (1, 1, 1)
2. (1, 1, 2)
3. (1, 1, 3)
4. (1, 1, 4)
5. (1, 2, 1)
6. (1, 2, 2)
7. (1, 2, 3)
8. (1, 2, 4)
9. (1, 3, 1)
10. (1, 3, 2)
11. (1, 3, 3)
12. (1, 3, 4)
13. (1, 4, 1)
14. (1, 4, 2)
15. (1, 4, 3)
16. (1, 4, 4)
17. (2, 1, 1)
18. (2, 1, 2)
19. (2, 1, 3)
20. (2, 1, 4)
21. (2, 2, 1)
22. (2, 2, 2)
23. (2, 2, 3)
24. (2, 2, 4)
25. (2, 3, 1)
26. (2, 3, 2)
27. (2, 3, 3)
28. (2, 3, 4)
29. (2, 4, 1)
30. (2, 4, 2)
31. (2, 4, 3)
32. (2, 4, 4)
33. (3, 1, 1)
34. (3, 1, 2)
35. (3, 1, 3)
36. (3, 1, 4)
37. (3, 2, 1)
38. (3, 2, 2)
39. (3, 2, 3)
40. (3, 2, 4)

41. (3, 3, 1)
42. (3, 3, 2)
43. (3, 3, 3)
44. (3, 3, 4)
45. (3, 4, 1)
46. (3, 4, 2)
47. (3, 4, 3)
48. (3, 4, 4)
49. (4, 1, 1)
50. (4, 1, 2)
51. (4, 1, 3)
52. (4, 1, 4)
53. (4, 2, 1)
54. (4, 2, 2)
55. (4, 2, 3)
56. (4, 2, 4)
57. (4, 3, 1)
58. (4, 3, 2)
59. (4, 3, 3)
60. (4, 3, 4)
61. (4, 4, 1)
62. (4, 4, 2)
63. (4, 4, 3)
64. (4, 4, 4)

Problem 4

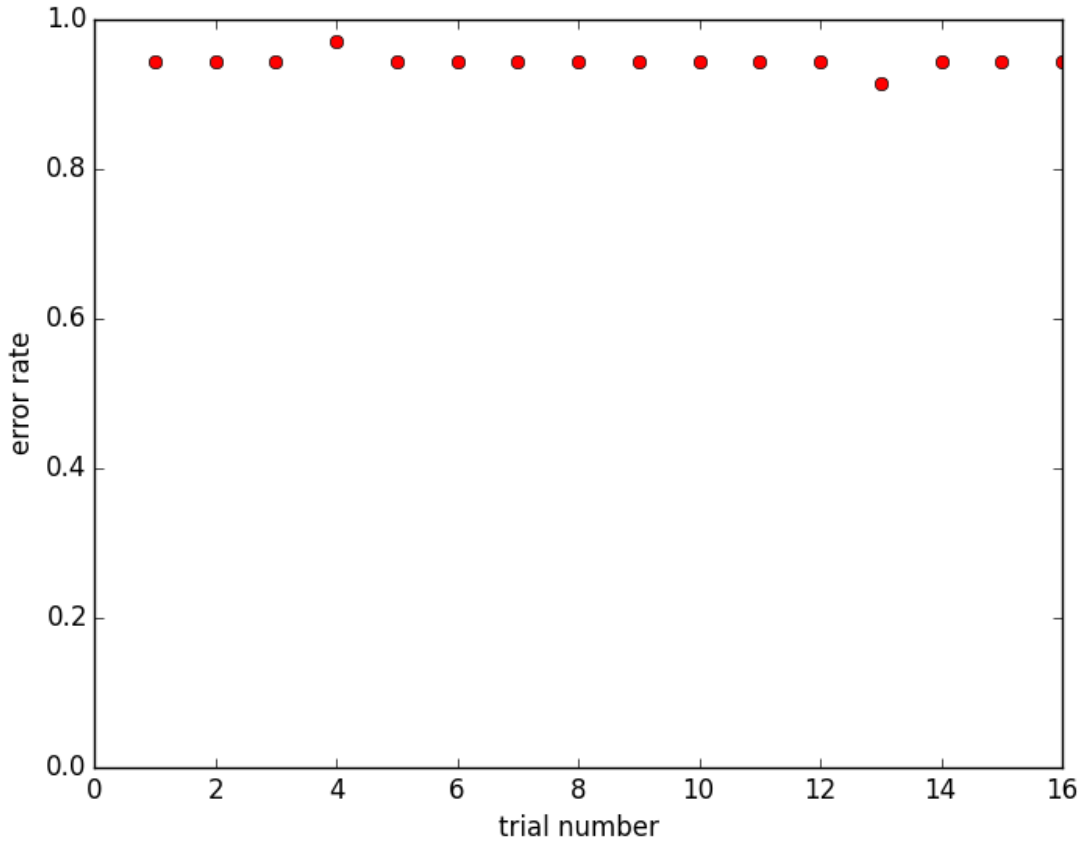
A) Implemented in file spellcheck.py
Can be run using command: python spellcheck.py <Typo/CorrectDuoFileName> 3esl.txt 2
The last argument 2 indicates that the qwerty version of the distance method should be used. It is important that wikipediatypo.txt, etc. is used for this command, as the program runs under the assumption that each line consists of the typo and the corresponding correct word.

B) As described in 2b, the experiment's conditions are as follows:

1. **16** iterations testing all combinations of parameters **{0, 1, 2, 4}**
2. Sample: **35 lines of wikipediatypo.txt**
3. Entire fixed dictionary: **3esl.txt**
4. Run time **< 1 hour**

One iteration with 35 lines (35 pairs, 70 words) of wikipediatypo.txt takes approximately 180 seconds using the qwerty levenshtein distance function. We are keeping details of the experiment as similar as possible to the one described in 2B. That being said, while the parameter set remains the same, this experiment will only test permutations of size 2, totally 16 permutations. This is because substitution cost will vary based on the Manhattan Distance function, so the permutations will only apply to insertion and deletion costs. The following computation shows that even with the increase in time per iteration, the reduction in total iterations keeps this experiment under one hour.

180 sec * 16 = 2880 sec
2880 sec / 60 = 48 min



Here is a graph of my results from the experiment I designed in Part B that utilizes the qwerty distance method. The x-axis represents the trial number, and the y-axis represents the corresponding error rate given the parameters used during that trial. I did not include the exact parameters used per trial on the x-axis because I thought it would look visually cluttered and would take away from the graphical representation. That being said, below are the corresponding parameters used per trial. Based on this experiment and sample data, the best parameter is: (4, 1) with the error rate: 0.914285714286.

1. (1, 1)
2. (1, 2)
3. (1, 3)
4. (1, 4)
5. (2, 1)
6. (2, 2)
7. (2, 3)
8. (2, 4)
9. (3, 1)
10. (3, 2)

11. (3, 3)
12. (3, 4)
13. (4, 1)
14. (4, 2)
15. (4, 3)
16. (4, 4)

Based on the above error rates and distribution, within my given experiment levenshtein distance preforms better than qwerty levenshtein distance. That being said, there are notable issues within my experimental design that I delve into further in the next section.

Post Experiment Analysis:
I decided to keep the details of my experiment exactly the same as the one run in Part B in order to have more comparable results. Meaning, I kept the sample size, inputted file, dictionary, and parameters the same. The only notable difference was that I only utilized permutations of size two, since substitution cost utilized the qwerty distance function.

While this approach was thought out, and rooted in the value of controlled environments, the high error rates across all parameter combinations indicate I overlooked an important intricacy. Given the Manhattan distance function, substitution cost could range from 1-12. Since the parameter list was (1, 2, 3, 4), the max insertion and deletion cost ever could be was 4. With this, a majority of the time insertion cost or deletion cost will be a more optimal solution that substitution cost. This discrepancy led to widely incorrect replacements, and in turn led to higher error rates. If I were to run another experiment, I would test parameters (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12). The bottleneck here, obviously, is time. This would take 2^{12} iterations. Given the 35 line sample set, that would be a significant investment in time.

180 seconds for 35 lines
Meaning, 180 / 35 = 5 sec/line

2^{12} iterations = 4,096
4096 iterations (5 seconds) = 20,480 seconds
20,480 sec / 60 = 341.3 minutes
341.3 min / 60 = 5.688 hours

Based on the above computation, it would take approximately 6 hours to run all parameter combinations on just one word. With our present file set, it would be 35 times that. While this is quite an investment in time, I anticipate the error rates would decrease drastically, as various combinations would be more suitably weighted against the Manhattan Distance component. Another solution to these high error rates is switching from Manhattan Distance to Euclidean Distance. This would dilute the costliness of substitution, and also would likely produce more accurate results.