# EECS 349 (Machine Learning) Homework 2

Xinyi Chen

## Problem 1

### A)

| Features | Weight | Height |
|---|---|---|
| Description | User's weight in kilograms | User's Height in centimeters |
| Range | 0 to 300 | 0 to 300 |

Vector: X = <Weight, Height>

This can capturing if user is slim, normal or overweight.

### B)

Metric: X = <Weight, Height>

$$d(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

Because:

d(x,y) = 0 iff x = y

d(x,y) >= 0

d(x,y) = d(y,x)

d(x,y) + d(y,z) >= d(x,z)

So it is a metric.

### C)

No, we can't treat these three elements values equally. Because numbers of hairs is way more bigger than other two elements, if we do this, number of hairs will be the main factor to cluster people. It doesn't make any sense.

I would change the vector into <height in feet, weight in kilograms, number of **thousands** of hairs on their head> and then put it in a Euclidean 3-space where we treat values for all three elements equally.

**D)**

We can consider four bases of DNA as four characters, A(adenine), T(thymine), C(cytosine) and G(guanine), so every strand of DNA can be considered as a string made up of these four characters(e.g. ATCGATAC is a strand of DNA). In this way distance between two strands of DNA can be considered as distance between two strings, so we can use the "the String to String Correction Problem"  way to determine two distance between two strands of DNA.

**Problem 2**

**A)**

Please open *runtests.py* and run *hw2_problem_2A_find_closest_word()* and *hw2_problem_2A_levenshtein_distance().* I have put these line at the bottom of file, delete '#' to run them.

**B)**

Please run : python *spellcheck.py* <ToBeSpellCheckedFileName> *3esl.txt*

**C)**

Please open *runtests.py* and run *hw2_problem_2C().* I have put this line at the bottom of file, delete '#' to run *hw2_problem_2C()*.

**Problem 3**

**A)**

Since I have built a dictionary words BK-Tree to speed up my searching and comparing process, which is 6 times faster, so my program takes only 1858.7 seconds(31 minutes) to run *measure_error* in *wikipediatypo.txt* using the *3esl.txt* dictionary. So run this for 64 times will take 1984 minutes.

**B)**

I'm going to pick *wikipediatypoclean.txt* as my dataset, and *3esl.txt* initial from A-C part as my dictionary. Because *wikipediatypoclean.txt* has only 634 examples and all the corrected words are initial from A-C, so I only need to build a dictionary words BK-Tree within 4500 nodes which initial from A-C, it is way more less than A-Z.

**C)**

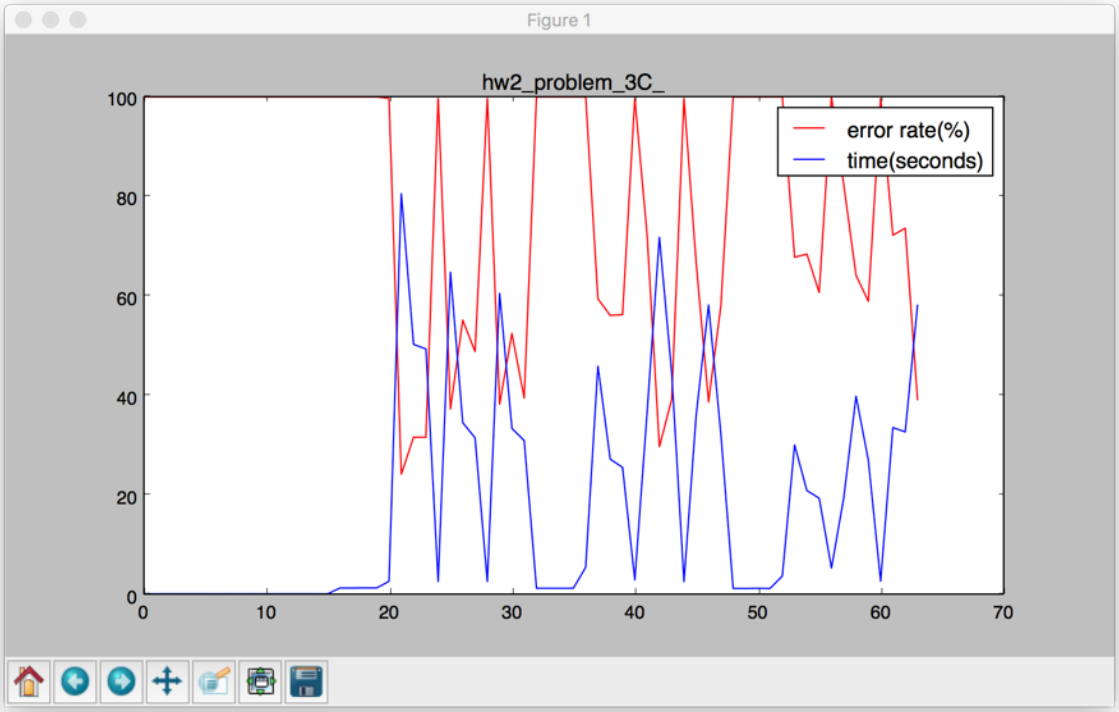
Please open *runtests.py* and run *hw2_problem_3C()*. I have put this line at the bottom of file, delete '#' to run *hw2_problem_3C()*.

Results: (red line is the best one)

| No. | deletion_cost | | insertion_cost | substitution_cost | error_rate | runtime |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 100.00% | 0.27s | |
| 1 | 0 | 0 | 1 | 100.00% | 0.27s | |
| 2 | 0 | 0 | 2 | 100.00% | 0.27s | |
| 3 | 0 | 0 | 4 | 100.00% | 0.27s | |
| 4 | 0 | 1 | 0 | 100.00% | 0.29s | |
| 5 | 0 | 1 | 1 | 100.00% | 0.27s | |
| 6 | 0 | 1 | 2 | 100.00% | 0.29s | |
| 7 | 0 | 1 | 4 | 100.00% | 0.28s | |
| 8 | 0 | 2 | 0 | 100.00% | 0.28s | |
| 9 | 0 | 2 | 1 | 100.00% | 0.27s | |
| 10 | 0 | 2 | 2 | 100.00% | 0.29s | |
| 11 | 0 | 2 | 4 | 100.00% | 0.31s | |
| 12 | 0 | 4 | 0 | 100.00% | 0.30s | |
| 13 | 0 | 4 | 1 | 100.00% | 0.28s | |
| 14 | 0 | 4 | 2 | 100.00% | 0.30s | |
| 15 | 0 | 4 | 4 | 100.00% | 0.28s | |
| 16 | 1 | 0 | 0 | 100.00% | 1.45s | |
| 17 | 1 | 0 | 1 | 100.00% | 1.44s | |
| 18 | 1 | 0 | 2 | 100.00% | 1.47s | |
| 19 | 1 | 0 | 4 | 100.00% | 1.47s | |
| 20 | 1 | 1 | 0 | 99.68% | 2.80s | |
| 21 | 1 | 1 | 1 | 24.29% | 84.17s | |
| 22 | 1 | 1 | 2 | 31.70% | 50.38s | |
| 23 | 1 | 1 | 4 | 31.70% | 50.04s | |
| 24 | 1 | 2 | 0 | 99.68% | 2.77s | |
| 25 | 1 | 2 | 1 | 37.38% | 64.45s | |

| | | | | | |
|---|---|---|---|---|---|
| 26 | 1 | 2 | 2 | 55.21% | 34.85s |
| 27 | 1 | 2 | 4 | 48.90% | 31.43s |
| 28 | 1 | 4 | 0 | 99.68% | 2.78s |
| 29 | 1 | 4 | 1 | 38.33% | 60.71s |
| 30 | 1 | 4 | 2 | 52.52% | 32.70s |
| 31 | 1 | 4 | 4 | 39.59% | 31.01s |
| 32 | 2 | 0 | 0 | 100.00% | 1.43s |
| 33 | 2 | 0 | 1 | 100.00% | 1.41s |
| 34 | 2 | 0 | 2 | 100.00% | 1.44s |
| 35 | 2 | 0 | 4 | 100.00% | 1.39s |
| 36 | 2 | 1 | 0 | 100.00% | 5.78s |
| 37 | 2 | 1 | 1 | 59.46% | 45.61s |
| 38 | 2 | 1 | 2 | 56.15% | 28.33s |
| 39 | 2 | 1 | 4 | 56.31% | 24.86s |
| 40 | 2 | 2 | 0 | 99.68% | 2.71s |
| 41 | 2 | 2 | 1 | 72.24% | 37.50s |
| 42 | 2 | 2 | 2 | 29.81% | 71.34s |
| 43 | 2 | 2 | 4 | 39.27% | 44.75s |
| 44 | 2 | 4 | 0 | 99.68% | 2.61s |
| 45 | 2 | 4 | 1 | 66.40% | 35.52s |
| 46 | 2 | 4 | 2 | 38.80% | 57.01s |
| 47 | 2 | 4 | 4 | 58.04% | 34.31s |
| 48 | 4 | 0 | 0 | 100.00% | 1.35s |
| 49 | 4 | 0 | 1 | 100.00% | 1.45s |
| 50 | 4 | 0 | 2 | 100.00% | 1.44s |
| 51 | 4 | 0 | 4 | 100.00% | 1.47s |
| 52 | 4 | 1 | 0 | 100.00% | 4.24s |
| 53 | 4 | 1 | 1 | 67.82% | 31.59s |
| 54 | 4 | 1 | 2 | 68.45% | 22.19s |
| 55 | 4 | 1 | 4 | 60.73% | 20.20s |
| 56 | 4 | 2 | 0 | 100.00% | 5.63s |
| 57 | 4 | 2 | 1 | 82.33% | 19.67s |
| 58 | 4 | 2 | 2 | 64.20% | 40.45s |
| 59 | 4 | 2 | 4 | 58.99% | 26.98s |
| 60 | 4 | 4 | 0 | 99.68% | 2.65s |
| 61 | 4 | 4 | 1 | 72.24% | 34.18s |
| 62 | 4 | 4 | 2 | 73.66% | 33.53s |
| 63 | 4 | 4 | 4 | 39.27% | 59.23s |

[Finished in 1114.3s]




X axis represents result number(0-63), red line represents error rate(%), and blue line represents runtime(seconds).

As we can see in the graph, the best accuracy performance(red line) is No. 21 result, when insertion, deletion and substitution costs all equals 1, and the best error rate is 24.29%, however it's not the best time performance point. There are some points that take less than 1 second to run, but accuracy is too poor. So the best performance combination is 1, 1, 1.

**Problem 4**

**A)**

Please open *runtests.py* and run *hw2_problem_4A()*. I have put this line in at bottom of file, delete '#' to run *hw2_problem_4A()*.

**B)**

Please open *runtests.py* and run *hw2_problem_4B()*. I have put this line in at bottom of file, delete '#' to run *hw2_problem_4B()*.

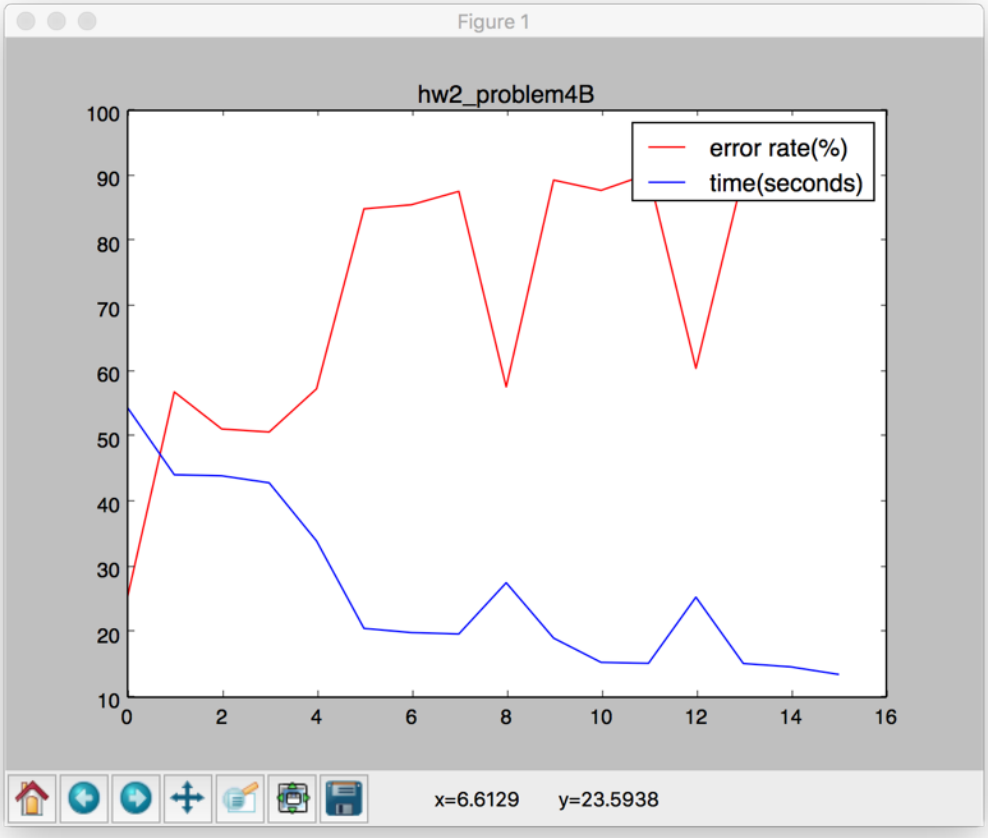
Because in problem 3, when one of the costs is 0, the error rate will be almost 100%, so I won't test 0 this time. I vary insertion and deletion costs among the values in the set {1, 2, 3, 4}

**C)**

Result: (red line is the best one)

| No. | deletion_cost | insertion_cost | error_rate | runtime |
|---|---|---|---|---|
| 0 | 1 | 1 | 25.08% | 54.67s |
| 1 | 1 | 2 | 56.94% | 44.24s |
| 2 | 1 | 3 | 51.26% | 44.08s |
| 3 | 1 | 4 | 50.79% | 43.00s |
| 4 | 2 | 1 | 57.41% | 34.09s |
| 5 | 2 | 2 | 85.02% | 20.67s |
| 6 | 2 | 3 | 85.65% | 20.03s |
| 7 | 2 | 4 | 87.70% | 19.82s |
| 8 | 3 | 1 | 57.73% | 27.67s |
| 9 | 3 | 2 | 89.43% | 19.18s |
| 10 | 3 | 3 | 87.85% | 15.47s |
| 11 | 3 | 4 | 90.38% | 15.32s |
| 12 | 4 | 1 | 60.57% | 25.46s |
| 13 | 4 | 2 | 90.22% | 15.29s |
| 14 | 4 | 3 | 91.17% | 14.77s |
| 15 | 4 | 4 | 89.27% | 13.63s |

[Finished in 416.5s]




X axis represents result number(0-15), red line represents error rate(%), and blue line represents runtime(seconds).

As we can see in the graph, the best accuracy performance(red line) is No. 0 result, when insertion and deletion costs all equals 1, and the best error rate is 25.08%, however it’s not the best time performance point. The best time performance point is No.15, it takes 13.63 seconds, but the error rate is 89%. So the best performance combination is 1, 1.

*levenshtein_distance* is better than *qwerty_levenshtein_distance*, because accuracy is higher and runtime is almost the same.