EECS 349 (Machine Learning) Homework 2

How to submit your homework

1. Create a **PDF document** containing answers to the homework questions. Show your reasoning in your answers.
2. Submit source code for the program you write.
3. Compress all of the files specified into a .zip file.
4. Name the file in the following manner, *firstname_lastname_hw2.zip*. For example, *Bryan_Pardo_hw2.zip*.
3. Submit this .zip file via Canvas by the date specified on Canvas.

Problem 1 Defining Metrics (3 points):

A. (1/2 point) Define a feature set for a Facebook account so that you can represent any account as a vector of a fixed number of features that can be encoded as real numbers. What are the features? What values can they take? What is this feature set useful for capturing about facebook account holders?

B. (1 point) Define a metric on the vector space from part A. Prove it is a metric.

C. (1/2 point) Assume each point in your space represents a person encoded as 3-element vector: <height in feet, weight in kilograms, number of hairs on their head>.

Give an expected range of values for each of these elements. The goal is to cluster people into 4 groups "child, teen, middle-aged, elderly". Does it make sense to put people in a Euclidean 3-space where we treat values for all three elements equally? Why or why not? How would you design a metric to cluster people appropriately? Explain your reasoning.

D) (1 point) A strand of DNA can be encoded as a sequence of four bases—adenine, guanine, cytosine, and thymine. Consider the metric for strings defined in "the String to String Correction Problem" (a paper available on the course calendar). Explain how this could be used to determine the distance between two strands of DNA. Does the metric need to be adapted in any way, or could it be used, essentially as-is? Back up your statement.

Problem 2 Making a Nearest-Neighbor Spell Checker (3 points):

The combination of a distance measure for words and the word list from the dictionary are all you need to make a simple nearest-neighbor spell-corrector. Define a word as a text string. Let $L=\{l_1,l_2,\ldots l_n\}$ be the list of words and let w be the word you're spell-checking. Define $d(a,b)$ as the Levenshtein distance (Figure 1) between strings a and b. Then, the closest word in the dictionary to our word is the one with the lowest distance to that word.

$$l_* = \underset{l \in L}{\operatorname{argmin}}(d(l,w))$$

If we assume that the closest word in the dictionary is the one to use for correcting, then this is all we need for a spell checker. Given a sentence like "My doeg haz gleas", simply find the closest dictionary word to each of the words in the sentence. If the distance to the closest word is 0, the word must be spelled right. If the closest dictionary word is not 0 steps away, then substitute the closest dictionary word in and you'll get "My dog has fleas". Or so we hope.

Figure 1 shows pseudocode of the Levenshtein distance for strings. This variant is the one by Wagner and Fischer. It gives a measure of distance between any two strings. Refer to the paper '*The String-to-string Correction Problem'* from the assigned reading in the course calendar for more details on this.

```
int LevenshteinDistance(char s[1..m],
                        char t[1..n],
                        deletionCost,
                        insertionCost,
                        substitutionCost)
{
  // for all i and j, d[i,j] will hold the Levenshtein distance between
  // the first i characters of s and the first j characters of t
  // NOTE: the standard approach is to set
  // deletionCost = insertionCost = substitutionCost = 1


  declare int d[0..m, 0..n]    // note that d has (m+1)x(n+1) values
  for i from 0 to m
    d[i, 0] := i*deletionCost  // dist of any 1st string to an empty 2nd string
  for j from 0 to n
    d[0, j] := j*insertionCost // dist of any 2nd string to an empty 1st string

  for j from 1 to n
  {
    for i from 1 to m
    {
      if s[i] = t[j] then
        d[i, j] := d[i-1, j-1]   // no operation cost, because they match
      else
        d[i, j] := minimum(d[i-1, j]   + deletionCost,
                           d[i, j-1]   + insertionCost,
                           d[i-1, j-1] + substitutionCost)
    }
  }

  return d[m,n]
}
```

Figure 1. Pseudo code for Levenshtein distance: Wagner and Fischer algorithm

Getting a Dictionary

The 12dicts project (http://wordlist.aspell.net/12dicts/) has several variant dictionaries of English. You should download Version 6 of the data set (also known as 12dicts-6.0.2.zip).

The Data

The file *wikipediatypo.txt* is included with this homework. This is a list of common spelling mistakes in the Wikipedia and is used to correct typographical errors throughout Wikipedia. See (http://en.wikipedia.org/wiki/Wikipedia:Typo) for information on spellchecking the Wikipedia. Each line in *wikipediatypo.txt* contains a common misspelled word followed by its associated correction. The two words (or phrases) on each line (error, correction) are separated by a tab. Note, some corrections may contain blanks (e.g. a line containing “aboutto” and “about to”).

The file *wikipediatypoclean.txt* is a subset of *wikipediatypo.txt* that contains only words whose correct spelling is an entry in the dictionary file */American/3esl.txt* from the 12dicts data set. It is in the same format as *wikipediatypo.txt.* Note this uses only words starting with a portion of the alphabet.

The file *syntheticdata.txt* has the same format as the other two files, but the spelling errors were created using a synthetic distribution. Note this uses only words starting with a portion of the alphabet.

EECS 349 (Machine Learning) Homework 2

Programming Hints

Some helpful packages to import to do this assignment:

```
import csv #this would be for reading our text files

import numpy as np #this would be for computing levenshtein_distance
(multi-dimensional arrays are easier with these)
import matplotlib.pyplot as plt #this would be for making many of the
plots required in this assignment
```

A. (1 point) Write a function that finds the closest word (string) in a dictionary (a list of strings) to a given input string. Call it *find_closest_word*. This function should call the *levenshtein_distance* function that you write. Both of these functions should be included in the same file "spellcheck.py".

```
def find_closest_word (string1, dictionary):

      #write some code to do this, calling levenshtein_distance, and
return a string (the closest word)

      return closest_word


def levenshtein_distance(string1, string2, deletion_cost,
insertion_cost, substitution_cost):

      #write some code to compute the levenshtein distance between two
strings, given some costs, return the distance as an integer

      return distance
```

B. (1 point) Write a Python command-line program ("spellcheck.py") that takes two ASCII text files as input: the first is the file to be spell-checked. The second is a dictionary word list with one word (or phrase) per line (we're thinking of *3esl.txt*). It should output a file in the current directory called *corrected.txt*. Treat every contiguous sequence of alphanumeric characters in the input file as a word. Treat all other characters (e.g. blanks, commas, '#', tabs, etc.) as word delimiters. The file corrected.txt should be the spell-corrected input file, where each word in the input file has been replaced by the nearest dictionary word. Call this file *spellcheck.py*. Make sure it can be called as:

邻接的字母与数字是word，其他(空格，逗号，#，tab等)都是分隔符

```
python spellcheck.py <ToBeSpellCheckedFileName> 3esl.txt
```

C. (1 point) Write a function, called *measure_error* that takes 3 lists of strings, called *typos, truewords*, and *dictionarywords*. respectively. The *i*th *typo* corresponds to the ith *trueword*. For each typo *i*, the function should find the closest dictionary word with the function *find_closest_word* and then compare the closest dictionary word to the *i*th *trueword* . If the closest dictionary word and the *trueword* are the same, then the error for that example is 0. If they are different, then this was a misclassification and the error is 1. The output of this function should be the error rate (number of errors divided by the number of typos).

```
def measure_error(typos, truewords, dictionarywords):

       #find whether the corrected typo using the dictionary matches
the true word. 0 if it doesn't, 1 if it does. Count them all up and
return a real value between 0 and 1 representing the error_rate

       return error_rate
```

EECS 349 (Machine Learning) Homework 2

Problem 3 Parameter Picking (2 points):

A. (1/2 point) How long does it take to run *measure_error* on the data in *wikipediatypo.txt* using the *3esl.txt* dictionary? Hint: use the time function for this. For example:

```
import time

start = time.time()

# your code here

print (time.time() – start)
```

Different metrics can change the performance of a nearest-neighbor classifier. One obvious way to change your metric is to vary the insertion, deletion and substitution costs of your *levenshtein_distance* function. One obvious thing to try would be a grid search of the space and see which settings give you the best results. If you were to vary insertion, deletion and substitution costs among the values in the set {0, 1, 2, 4}, this would give 64 parameter combinations. How long would that take, given you use the files specified in A? How about if you used 10-fold cross validation on your data?

B. (1/2 point) Design an experiment to determine the best values for insertion, deletion and substitution costs that can be run by your system in under 1 hour. Say what values of each parameter you'll try. Say whether you're doing cross validation. If so, how many folds? Pick a dictionary (or part of a dictionary) and data that will let you complete the grid search described in part A of this problem. Explain the reasoning for your choices.

C. (1 point) Perform the experiment you designed in part B. Show a graph (or graphs) that give the results. What is the best combination of insertion, deletion and substitution costs?

Problem 4 A Better Distance Measure? (2 points):

A. (1 point) We discussed an edit distance measure in class that uses the distance on the QWERTY keyboard as the substitution cost function. Ignore capitalization (things on the same key have distance 0). Distance between keys is Manhattan distance, measured by difference in rows + difference in columns. Let *d(x,y)* be the distance function. Example values are *d('Q','q')* = 0, *d('G','B')* = 1, and *d('Q','d')* = 3, since 'd' is 2 columns to the right and 1 row down from Q. Consider only alphanumeric characters. Treat all other characters (e.g. blanks, commas, '#', tabs, etc.) as word delimiters that you don't have to measure distances on. In other words, you don't have to calculate *d('1','!')*.

Write an edit distance that uses this substitution cost function in place of a fixed substitution cost.

```
def qwerty_levenshtein_distance(string1, string2, deletion_cost,
insertion_cost):

      #write some code to compute the levenshtein distance between two
strings, given some costs, return the distance as an integer

      return distance
```

B. (1 point) Rerun the experiment from problem 3, this time using qwerty_levenshtein_distance. This means you're only optimizing two parameters: deletion_cost and insertion_cost. You may need to try different ranges of values than you did in question 3. What set of values did you try? Show a graph (or graphs) with results. Don't forget to label dimensions. What is the best combination of insertion, deletion and substitution costs? Is qwerty_levenshtein_distance better than levenshtein_distance? Why or why not?