EECS 349 (Machine Learning) Homework 2

WHAT TO HAND IN

You are to submit the following things for this homework:

1. A PDF document containing answers to the homework questions.
2. The source code and (in the case of C/C++/C#) executable for the program you write.

HOW TO HAND IT IN

To submit your homework:

1. Compress all of the files specified into a .zip file.
2. Name the file in the following manner, firstname_lastname_hw2.zip. For example, Bryan_Pardo_hw2.zip.
3. Submit this .zip file via blackboard.

DUE DATE: the start of class on Mon 17-Oct-11

The Levenshtein Distance Measure

```
int LevenshteinDistance(char s[1..m], char t[1..n])
{
  // 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


  declare int d[0..m, 0..n]     // note that d has (m+1)x(n+1) values
  declare int deletionCost = insertionCost = substitutionCost = 1;

  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

Figure 1 (above) 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 String-to-string Correction Problem" for more details on this.

EECS 349 (Machine Learning) Homework 2

The Dictionary Word List

The 12dicts project (http://wordlist.sourceforge.net/12dicts-readme.html) is to create a list of words approximating the common core of the vocabulary of American English.

The 3esl list from this project contains words and phrases listed in 3 ESL (English as a Second Language) dictionaries. The list is composed of all words from the smallest of the 3 sources, plus all words contained in both of the others. There are roughly 23,000 entries. One word (or phrase) per line.

This list is included as a text file with this homework in the file *3esl.txt*.

The Test/Train 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 Wikipedia:Typo for information on and coordination of spellchecking work in the Wikipedia. Each line in this file 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 *3esl.txt*.

The file *syntheticdata.txt* has the same format as the other two files, but the spelling errors were created using a synthetic distribution.

Your task: Building, training, and testing a spell-corrector

The combination of the distance measure and the word list from the dictionary are all you need to make a simple 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 between words 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 closets dictionary word is not 0 steps away, then substitute the dictionary word in and you'll get "My dog has fleas". Or so we hope.

There are, of course, catches: a word may be properly spelled but not in the dictionary (such as a proper noun or the British spelling). If the word is misspelled, there is still a question about what the right replacement word is. The "closest" dictionary word is defined by the distance measure you use…and it may be that the word you think of as the "right" one is not the "closest" one, according to the distance measure. So you may need to change the distance measure somehow.

In this homework, you'll build a simple spell-corrector. You'll then build an "improving" spell-corrector by using a simple hill-climbing algorithm to update the distance measure. Then, you'll rethink the distance measure. Finally, you will compare your spell-correctors and report statistics on their performance.

EECS 349 (Machine Learning) Homework 2

Your Program

Your program must be written in C/C++/C#, or MatLab. Executable requirements for the varying languages are outlined below. We are not responsible for debugging code written in other languages or other operating systems. Your source code must be well commented so that can be easily understood.

C/C++/C#

If your program is written in C/C++/C#, you must hand in all your source code AND an executable file. The source code must be able to be compiled in **Visual Studio 2010 Express** on a 64-bit Windows 7 PC. The executable must run in the 64-bit Windows 7 PC as well. Visual Studio 2010 Express is downloadable at http://www.microsoft.com/visualstudio/en-us/products/2010-editions/express.

MATLAB

If your program is written in MATLAB, you must hand in all your source code. The program must run in **MATLAB 2011a** on a 64-bit Windows 7 PC. Our Wilkinson lab and T-lab have MATLAB 2011a installed on all Windows machines.

Note: MATLAB users can read our input files with ease by using the 'textscan' command.

Problem 1: Make a baseline spelling corrector for words (4 points)

A) (1 point) Write a distance measure function that accepts two strings as input and outputs the Levenshtein distance between them. Call it ***LevenshteinDistance***.

B) (1 point) Write a function that finds the closest word (string) in a dictionary of strings to a given input string. Call it ***FindClosestWord***.

C) (2 points) Write a command-line program 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 program ***SpellCorrect***. Make sure it can be called as:

```
Usage: SpellCorrect(<ToBeSpellCheckedFileName>, <3esl.txt>, <corrected.txt>)
```

Problem 2: Make error-measuring tools for the spelling corrector (2 points)

A) (1 point) Write a function, called ***MeasureError*** that takes 2 containers (e.g. vectors, arraylists, arrays, matlab cell arrays etc.) called *typo* and *trueword*. The *i*th typo corresponds to the *i*th trueword and the values come from the *i*th line in the training data file. For each typo *i,*the function should find the closest dictionary word with the function ***FindClosestWord*** 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).

B) (1 point) Write a function called ***MakeNFolds***. This will take as input a file formatted like *wikipediatypo.txt* , and an integer input *numFolds* that can take a value from 2 to 100. It will partition the data set into *numFolds* equal-size folds. (Look up "partition set" if you don't know what that means). It will then output *numFolds* **pairs** of files into the current directory. Each pair will have a test file and a training file. The training file will

contain 1 fold of the data. The test file will contain the remaining *numFolds*-1 folds of the data. For example, if the input file is called *input.txt* and *numFolds* = 3, the result would be the following set of files:

input1train.txt, input1test.txt, input2train.txt, input2test.txt, input3train.txt, input3test.txt

***NOTE: it is typical to test on the small set (1 fold) and train on the large set (n-1 fold). We are reversing it in this lab because MATLAB is slow and training a hill climber on the larger set may take too long.*

Problem 3: Create a hill climber to learn a better spelling corrector (3 points)

Update the *LevenshteinDistance* function so that *deletionCost*, *insertionCost* and *substitutionCost* are all real-value input parameters that range from 0 to 1. Call this ***LevenshteinDistance2***. Now do the following.

A) (2 points) Write a hill-climber that finds the best values for *deletionCost*, *insertionCost* and *substitutionCost*. Evaluation of each combination of values will be done with a function ***MeasureError2***, which will call ***LevenshteinDistance2.*** Call this function ***HillClimber***.

B) (1 point) Explain any design choices you made for the hill climber. Specific things to mention are: Which next-states are reachable in one step from the current state? How many are reachable in one step? Do you do simulated annealing? Do you randomly pick a new place to start every time you do hill-climbing? Are steps always the same size? What is that step size?

HINT: When debugging your hill climber, work on a subset of the data. For example: a dictionary of the "a" words and a labeled training set of 30 typos whose trueword starts with "a".

HINT: For this lab, it is OK to use a fixed size grid of points for hill climbing (For example, use a fixed step size of 0.1). Then, a step is moving by one integer value for one of the 3 parameters you are learning.

HINT: When hill climbing, it might be good to keep an array for your parameter space that stores the mean error you've found for a particular combination of parameters on your current training set. That way, if you re-visit a point, you don't have to re-calculate error.

HINT: Your hill climber may be better off if you add simulated annealing. Or even if you use a fixed probability of not picking the "best" neighbor as the place you move next.

HINT: What is your termination condition for the hill climber? Do you stop after a maximum number of steps? If you take the hints above, your space will have 1000 points maximum in it. So, if hill climbing is to be of value, you should stop long before 1000 steps.

Problem 4: Define a better distance metric (2 points)

A) (1 point) Describe (but don't code up) an update of the Levenshtein distance measure to somehow capture the relative likelihood of someone mistyping one string for another. One easy way to do that would be to replace the *substitutionCost* scalar with a function *substitutionCost(c1, c2) that* determines the substitution cost between two characters c1 and c2. (For example, for someone using a QWERTY keyboard, the string "grog" should be closer to "frog" than "grog" is to "prog." Or maybe you want to deal with phonetic spelling mistakes like "hear" vs. "here".) Give your argument for why your distance measure DOES make words that are easier to mistype for each other also closer using the distance measure.

B) (1 point) Show why your measure from part (A) is or is not a metric.

Problem 5: Evaluating Results (4 points)

A) (1 point) How quickly (as measured in number of hill-climbing steps) does the hill climber converge to its maximal performance on each of the 3 training files? Illustrate this with one graph for each file, where the best sample error **on a single training set of 30 typos** is plotted as a function of the number of steps. How long does one hill-climbing step take (in minutes and seconds)?

B) (1 point) Select one of the 3 training files. Run the hill-climber on this file using 30-fold cross validation. For each fold, train on the training set. Then, record the mean sample error **on the test set** of each fold at the end of training. Evaluate how well standard ***LevenshteinDistance*** *(where all costs equal 1)* and ***LevenshteinDistance2*** *(with the best trained costs by the hill climber for each training set).* Plot a histogram of the distribution of sample error for each distance measure. Report whether you feel the distribution from 5B resembles a normal distribution.

C) (1/2 point) Explain the design choices you made so that you could generate this data. What data did you use to test, to train? What dictionary was used to train and to test?

Hint: This may take a long time, even after applying some time-saving code tricks. Therefore, you may have to make experimental design choices to speed things up. One example would be to train on data from "wikipediatypoclean.txt" and a smaller version of the dictionary (say….only the letters A through F). You'd still need to TEST after hill-climbing on each fold using the full dictionary, though.

D) (1 point) Select a statistical test to determine whether the sample error means of your spell-corrector with the two distance measures (standard Levenshtein and the one with hill-climber learned weights) are significantly different at the 95% confidence level. Explain why you chose that test. Apply that test to the data from step (B). Report the results. If, for some reason, you feel the tests discussed in class are not applicable, consider using the sign test (http://en.wikipedia.org/wiki/Sign_test) to determine whether the difference between the sample error medians is significant.

E) (1/2 point) Which of the three labeled training data sets is most reflective of a real-world scenario for your spell-corrector? Does learning to reweight address the majority of the sample error for this data set? If not, what was it about this data that the system could not deal with? How would you do to change the spell checker to deal with the remaining error so that your system could be deployed in the real world?

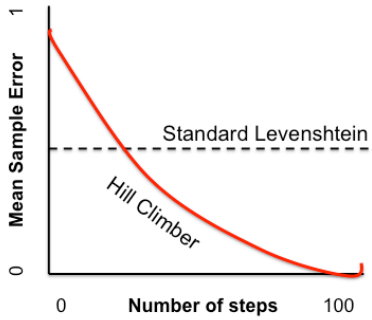

Figure 2. An example learning curve graph