Northeastern University, Khoury College of Computer Science

CS 6120 — Assignment 3

Due: February 23, 2026 (100 points)

Name and Student E-mail

Autocorrect is a productivity tool that helps you quickly adjust any user spelling mistakes in typing, tapping, or taking dictation. For example, if you type in the word **I am lerningg**, chances are very high that you meant to write **learning**. You use auto-correct every day on your cell phone and computer. In this assignment, you will implement the base algorithm that underlies most modern systems today. Of course, your implementation differs from production applications as they have been optimized with improvements and heuristics throughout the years. But it is still good `:-)` The following questions will give us some hands-on experience with *autocorrect*.

In particular, you will need to (1) get a word count given a corpus, (2) get a word probability in the corpus, (3) manipulate and edit strings, (4) filter strings according to certain criteria, and (5) implement the minimum edit distance to compare strings, find the optimal path for the edits. These steps are broken up into three questions in this homework, and you will fill out this code template. At its heart, we will be implementing a *dynamic programming* solution, the basis of many computer science problems that are complex in nature. Feel free to test your solution out using the full Shakespeare Data, called `shakespeare-edit.txt`, which you can download at the course website. There, you will also find unit test data in the file `shakespeare-7k.txt`, which can help you sanity check your code.

Question 1: Download and Process the Data

Implement the function `process_data` which

1. Reads in a corpus (text file)
2. Changes everything to lowercase
3. Returns the dictionary of words and probability of occurrence

The words should not have any punctuation or numbers in them. The function signature looks like the following.

```
def process_data(file_name):
    """
    Input:
        filename: A file_name which is found in your current
                  directory. You just have to read it in.
    Output:
        wordprobs: a dictionary where keys are all the processed
                    lowercase words and the values are the frequency
                    that it occurs in the corpus (text file you read).
```

For all words $\{w_i\}$, the above function signature outlines how we arrive at the dictionary of $w_i \rightarrow P(w_i)$, where the probability of w_i appearing is $P(w_i)$.

Question 2: Identifying Probable Words

We identify four possible edits to our word in seeing which other words are most similar. That is, there are four ways we can operate on a string through its characters in order to change a word. These are:

- `delete_letter`: given a word, we can change it by **removing one character**.
- `switch_letter`: given a word, we can change it by **switching two adjacent characters**.
- `replace_letter`: given a word, we can **replace one character by another different letter**.
- `insert_letter`: given a word, **we can insert an additional character**.

Taking advantage of lazy evaluation and other computational tricks, write a function that determines all the possible words that are of a distance of exactly *two* edits away. These words must appear in your corpus and they must be ordered according to their overall probability (calculated in the prior question.)

The function signature looks like this:

```
def probable_substitutes(word, probs, maxret = 10):
    """
    Input:
        word - The misspelled word
        probs - A dictionary of word --> prob
        maxret - Maximum number of words to return
    Returns:
        [(word1, prob1), ... ]
    """
```

Question 3: Computing the Minimum Edit Distance

Now that you have implemented your auto-correct, how do you evaluate the similarity between two strings? For example: 'waht' and 'what'. Also how do you efficiently find the shortest path to go from the word, 'waht' to the word 'what'? You will implement a dynamic programming

system that will tell you the minimum number of edits required to convert a string into another string.

Implement the minimum edit distance function with the following function signature.

```
def min_edit_distance(source, target, ins_cost = 1,
                      del_cost = 1, rep_cost = 2):
    '''
    Input:
        source: starting string
        target: ending string
        ins_cost: integer representing insert cost
        del_cost: integer representing delete cost
        rep_cost: integer representing replace cost
    Output:
        D: matrix of size (len(source)+1 , len(target)+1)
           with minimum edit distances
        med: the minimum edit distance required to convert
             source to target
    '''
```

Recall in class the creation of the matrix of edit distances. We can build this matrix by starting out with the following initialization:

$$
\begin{aligned}
D[0,0] &= 0 \\
D[i,0] &= D[i-1,0] + del\_cost(source[i]) \\
D[0,j] &= D[0,j-1] + ins\_cost(target[j])
\end{aligned}
$$

Subsequently, the dynamic programming problem can build the matrix entries with the following per cell operations:

$$
D[i,j] = min \begin{cases} D[i-1,j] + del\_cost \\ D[i,j-1] + ins\_cost \\ D[i-1,j-1] + \begin{cases} rep\_cost; & if src[i] \neq tar[j] \\ 0; & if src[i] = tar[j] \end{cases} \end{cases}
$$

Submission Instructions

Submit your Python file with the above function signatures called `assignment3.py` to Gradescope.