CMPE 130: Advanced Algorithm Design **SJSU Spring 2019**
Professor Gheorghi Guzun Due: March 12, 2019

Project Report: Autocorrect and Autocomplete

Grace Mao, Joseph Lee, and Thinh Manh

Abstract

Communicating over a digital medium often requires the digitization of the written word into digital text. From articles to proposals to everyday "texting", the presence of digital typing is ubiquitous and thus is the need to type accurately, especially given the relative size of thumbs to the size of letters on a mobile phone. Accuracy can be ensured through the use of a spell checker, or autocorrect feature. Digital typing can be augmented further with the ability to predict partial, current words (autocomplete) as well as future words (predictive text). We implement autocorrect (restricted Damarau-Levenshtein distance) and autocomplete (prefix tree) using Python.

To implement autocorrect, we take in an input of a partially spelled or misspelled word and calculate the restricted Damarau-Levenshtein distance to a dictionary of English words stored in a hash structure. The dictionary of English words is generated from a corpus of the entirety of Wikipedia in 2010 in plaintext; approximately 6.5GB. The shortest distance will represent the shortest "edit distance" from our input string to a valid English word; and "edit" being a single-character edit (insertions, deletions, substitution). In addition, we also consider adjacent character transpositions.

Autocomplete could be implemented similarly to autocorrect, but partial words might we mapped to unwanted outputs via Levenshtein distance. Assuming the input words have accurate initial characters, we implement autocomplete using a prefix-trie generated from our dictionary of English words that dynamically suggests words upon user input.

Both of these features are implemented in Python using the pynput package to dynamically read in and process a user's input.

Introduction

Problem Definition. We investigate the problem of autocorrection and autocompletion in natural language text input, specifically for the English language. The advent of the touch-type keyboard in mobile devices has only increased not only the need to just spellchecker, but an efficient one. The problem involves determining a metric for the closeness of two strings of words, specifically between a user input string and an English language word string.

Applications. Efficient autocorrect and autocomplete have many various usages in natural language processing as well as DNA and protein sequencing. Determining a metric for similarity between sequences of characters, or strings, is central to comparing such strings. For example, DNA is made of building blocks called bases. Sequences of these building blocks encode the function of cells. However, if the bases are incorrectly ordered, even by one, the resulting function of the cell is affected. Therefore, it is necessary to have a robust, efficient method to detect such errors. With the advent of CRISPR genetic engineering technologies, these string error detection and correction methods are robust and pervasive at the cutting edge. Yet, a more general usage of said methods are in everyday text editors, text messaging, and emails. Some shortcomings and limitations of existing string error detection and correction methods are in very domain specific areas, such as different languages with more complex alphabets.

Algorithms. The algorithm we used is the restricted Damerau-Levenshtein "edit distance" algorithm. This is very similar to the fundamental Levenshtein distance algorithm, a string metric for measuring difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other. However, because we are dealing with user input, the restricted Damerau-Levenshtein distance includes the error of adjacent transpositions as a unit distance edit from the original string. This does cause the restricted Damerau-Levenshtein distance to violate the triangle inequality. We discuss more about this later.

Methods

Forming the frequency table. First and foremost before being able to check the closeness of partially spelled or misspelled words, we generated a dictionary of valid words. Our initial dictionary of valid English words was from a list of roughly 300,000 English words. However, comparison against just a list of valid english words yielded results that included uncommonly used english words. Therefore, we determined a need to add frequency of occurrence as a parameter in deciding which words to choose. Therefore, we constructed a frequency table of english words sourced from a text corpus of a snapshot of Wikipedia from 2008. This corpus included approximately one billion words totaling roughly 7GB in a single .txt file.

There were roughly 60,000 instances of words that occurred 10+ times. We implement a frequency table using a dictionary in Python, which is a hash table data structure. This allowed for $O(1)$ constant time retrieval of keys (valid english words) and comparison based on values (frequency of occurrence). We chose a dictionary because it has fast access time for an index and the order did not particularly matter.

```
however 791748
always 127042
included 301059
an 3956409
strain 10650
supporting 51907
market 157276
economy 75547
private 169769
property 110981
unrestrained 460
egoism 405
bases 25710
right 209079
might 102567
such 977413
as 8178754
without 263546
neither 38518
```

A snapshot of the frequency table.

Autocorrect. For autocorrect, we used the restricted Damerau-Levenshtein distance, otherwise known as the Optimal String Alignment (OSA) algorithm. The Damerau-Levenshtein distance is a string metric for measuring the edit distance between two sequences. Informally, the Damerau-Levenshtein distance between two words is the minimum number of operations (consisting of insertions, deletions or substitutions of a single character, or transposition of two adjacent characters) required to change one word into the other. This differs from the classical Levenshtein distance by including transpositions among its allowable operations in addition to the three classical single-character edit operations (insertions, deletions and substitutions). In In his seminal paper, Damerau stated that more than 80% of all human misspellings can be expressed by a single error of one of the four types. In natural languages, strings are short and the number of errors (misspellings) rarely exceeds 2. In such circum-

stances, restricted and real edit distance differ very rarely, so the prior can be used with confidence as the implementation is straight forward and the running time is more efficient.

To express the Damerau–Levenshtein distance between two strings a and b, a function $d_{a,b}(i,j)$ is defined, whose value is a distance between an i–symbol prefix (initial substring) of string a and a j–symbol prefix of b.

The restricted distance function is defined recursively as,

$$d_{a,b}(i,j) = \min \begin{cases} 0 & \text{if } i = j = 0 \\ d_{a,b}(i-1,j) + 1 & \text{if } i > 0 \\ d_{a,b}(i,j-1) + 1 & \text{if } j > 0 \\ d_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} & \text{if } i,j > 0 \\ d_{a,b}(i-2,j-2) + 1 & \text{if } i,j > 1 \text{ and } a[i] = b[j-1] \text{ and } a[i-1] = b[j] \end{cases}$$

where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise.

Each recursive call matches one of the cases covered by the Damerau–Levenshtein distance:

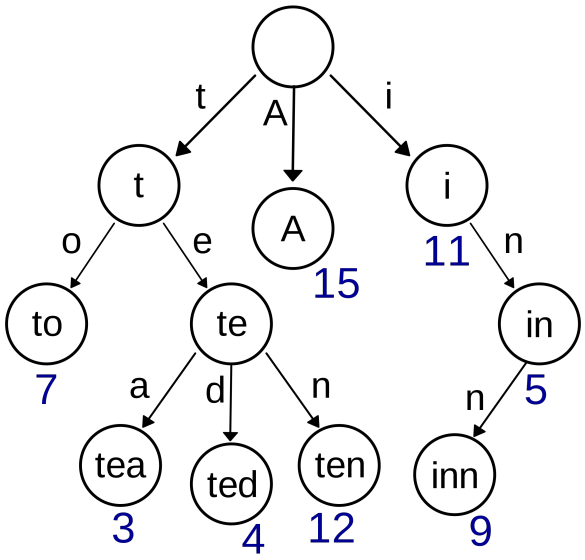
- $d_{a,b}(i-1,j) + 1$ corresponds to a deletion (from a to b).
- $d_{a,b}(i,j-1) + 1$ corresponds to an insertion (from a to b).
- $d_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)}$ corresponds to a match or mismatch, depending on whether the respective symbols are the same.
- $d_{a,b}(i-2,j-2) + 1$ corresponds to a transposition between two successive symbols.

The Damerau–Levenshtein distance between a and b is then given by the function value for full strings: $d_{a,b}(|a|,|b|)$ where $i = |a|$ and $j = |b|$ denote the lengths of a and b.

		a	n		a	c	t
	0	1	2	3	4	5	6
a	1	0	1	2	3	4	5
	2	1	1	1	2	3	4
c	3	2	2	2	2	2	3
a	4	3	3	3	2	2	3
t	5	4	4	4	3	3	2

An example of a table generated by restricted Damerau-Levenshtein distance.

Autocomplete. For autocomplete, we used the keys of our frequency table to construct a prefix tree. A prefix tree or "trie" is a kind of search tree—an ordered tree data structure with associative data, such as strings, as nodes. A node's position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Therefore, we are able to generate all words that begin with any prefix by descending down the tree.



A example of a prefix tree.

User Interface. To create a user interface to take dynamically run the optimal string alignment algorithm on partially spelled or misspelled user input, we use the Pynput Python package. This library allows one to control and monitor input devices, such as the mouse and keyboard. The user interface generates 3 autocorrect suggestions, i.e. the 3 closest valid English words to the current input, as well as a single autcomplete suggestion, which is the most frequent English word from a list of candidates generated by using the current input as a prefix. It also supports period punctuation, backspacing and carriage returns. This can further be extended to cache all user input for future features. Pressing the spacebar autocorrects to the nearest word if the current input is not a valid English word.

```
hello wor
3: for
2: or
1: war
0: world
```

A example of the user interface

Results

We evaluated our solution with testing on anecdotal usage and user feedback. Our feedback was generally positive but some suggestions were made for improvements including speed of usage and better suggestion results. We also built a timing framework; our autocorrect procedure runs in a few milliseconds for random inputs, which is good enough.

Analysis and Related Work

Related Work.

The project provides wide range of support to Natural Language Processing for Machine Learning. It helps the machine learning collect the right data and reduce amount of time to process. The Machine Learning model can use autocomplete to predict the words are not complete. For example, the book or a piece of information is only containing first part of the word so it need to predict the word and make the information complete. We read three related articles:

In "Spell Checking Techniques in NLP: A Survey" by Gupta et. al, the authors introduce concepts of typographical errors that are more common than others, as well as n-gram based error correction techniques, which consider groups of words instead of one word at a time. Some other techniques include rule-based techniques, probabilistic techniques, and neural network machine learning techniques. Our approach is much simple and these techniques probably offer some kind of advantage given all their complexity. However, our robust approach seems to provide adequate results for usage.

"A Survey of Spelling Error Detection and Correction Techniques" by Mishra et. al discusses a usage of the Levenshtein distance as well as the the Hamming algorithm and the longest common subsequence problem. However, the Hamming algorithm can only compare strings of equal length, which is not so good for our purposes. Longest common subsequence as a metric is not as good as the Damerau-Levenshtein distance. This paper describes how considerable work has been done in the area of English and related languages, but that there is very little work done in Hindi language.

"Development of a Spelling List" by M. McIlroy explores the idea of a spelling checker used on minicomputers. It considers the need to make a dictionary of words as compact as possible. Its results show that methods such as stripping prefixes and suffixes reduces the list below one third of its original size, hashing discards 60 percent of the bits that remain, and data compression halves it once again. This paper tells how the spelling checker works, how the words were chosen, how the spelling checker was used to improve itself, and how the (reduced) list of 30000 English words was squeezed into 26000 16-bit machine words. This idea could be used to improve our method by reducing the number of words checked against.

There are different methods to implement spell checker. There is edit based similarities, comparing two strings by minimum edit needed to transform a string to another. Levenshtein algorithm and hamming algorithm are edit based. It's good for short strings but it's inefficient to long strings. Longest common subsequence is sequence based and it compares two strings by subsequence. It is also good for short strings and bad for long strings.

Algorithm Analysis.

Autocorrect

The most important algorithm in this application is the *restricted Damerau-Levenshtein* (RDL) distance algorithm, also known as the Optimal String Alignment algorithm. We use the RDL distance as a string metric for determining the distance between two strings; for our purpose, namely, a partially spelled word and the closest words. The input for OSA is two strings and the output is the minimum number of edits necessary to get from one string to the other. The running time must traverse each string against the other resulting in a running time of $O(mn)$ where m and n are the lengths of the input strings, respectively.

Pseudo-code for Optimal String Alignment (OSA).

```
algorithm OSA-distance is
    input: strings a[1..length(a)], b[1..length(b)]
    output: distance, integer

    let d[0..length(a), 0..length(b)] be a 2-d array of integers, dimensions length(a)+1, length(b)+1
    // note that d is zero-indexed, while a and b are one-indexed.

    for i := 0 to length(a) inclusive do
        d[i, 0] := i
    for j := 0 to length(b) inclusive do
        d[0, j] := j

    for i := 1 to length(a) inclusive do
        for j := 1 to length(b) inclusive do
            if a[i] = b[j] then
                cost := 0
            else
                cost := 1
            d[i, j] := minimum(d[i-1, j] + 1,     // deletion
                               d[i, j-1] + 1,     // insertion
                               d[i-1, j-1] + cost)  // substitution
            if i > 1 and j > 1 and a[i] = b[j-1] and a[i-1] = b[j] then
                d[i, j] := minimum(d[i, j],
                                   d[i-2, j-2] + cost)  // transposition
    return d[length(a), length(b)]
```

Autocomplete

The most appropriate data structure for autocomplete problem is the prefix tree or Trie. This algorithm is finding the common prefix and storing them in the Trie. The inputs will take prefix of any word and output the complete list of words with such a prefix. The running time for this data structure is $O(n)$ for insert and search since it iterate through every node of the Trie until it reach to leaf node.

Pseudo-code for autocomplete:

```
Class TrieNode:
    init():
      end<-False
      children={}
  Class Trie:
  insert(word):
    Curr <- root
    For letter in word:
      Node <- curr letter of the children
      If node ==None:
        Set node = TrieNode()
        Curr.children[letter] =node
      curr<- node
    Set curr.end to True
  search(word):
    curr<-root
    Loop letter in word:
      Node <- curr letter of the children
      If node ==None:
        Return false and stop
      Set curr<-node
      Return curr.end and stop
  All_word_beginning_with_prefix(prefix):
    Set cur<- root
    Let c loop through prefix:
      Set cur node = children node get from c
      Check if cur node is None:
        If true return and stop the function
    Use all_words with parameter prefix
```

Conclusions and Lessons Learned

We are able to finishing the project in timely manner. In this project, we successful take input words and calculate the Levenshtein distance to find the shortest "edit distance" , and we produce the correct word (insertion, deletion or substitution). Autocomplete is another part of the project, and it also takes the same input as autocorrect. However, the word need to be partial correct in order to proceed the algorithm. The Trie is used as algorithm for this part, and it produce successful run. The user interface was a new venture and we learned how to take and process input dynamically from a user.