Joseph Perez

Prof. Kruger

15 December 2022

CPE 593

Text Input Prediction

Text input prediction is a widely used technology that helps improve the speed and accuracy of typing on devices such as smartphones and tablets. It is based on the idea that people tend to use certain words and phrases repeatedly in their writing, and that it is possible to predict the words and phrases that a person is likely to type next based on any characters or words that have been entered so far. To implement a text input prediction program, we can use a combination of data structures and algorithms to store, process, and retrieve data and formulate predictions.

This implementation uses a trie data structure where each node represents a single character in a word. For example, in the word "hello", the trie would have a root node with five child nodes, one for each of the letters in the word. One issue with using a trie to search for the completions of input prefixes is that these searches are inefficient. A text prediction algorithm must be as close to real-time as possible. In order to make this process more efficient, we will be using a trie consisting of nodes containing hashmaps that store any keys in use. This makes traversing a trie until the end of a given prefix near instantaneous, and dramatically increases the speed of the search for any word completions. While these modifications may not make a noteworthy difference in storing and retrieving English words, in languages such as Japanese where there are sometimes thousands of possible characters that must be iterated over to further traverse our trie, these modifications could cut down search times substantially.

Before constructing the trie, we must first begin by processing a sample text. Using a Python script and relying on Pythons Natural Language Tool Kit library, we process this sample text and output two files: words.txt and grams.txt. Words.txt contains all the words in the sample text stripped of any punctuation and formatted to be one word per line in the document. All the words in words.txt are also converted to lower case to prevent any distinction between identical words that do or do not start with a capital letter. In grams.txt, we generate a list of the n-grams that we want to use to enable contextual analysis in the program. N-grams are sequences of words at a length of n that lead to the suggestion of the next possible word. For example, in this sentence, a few 2-grams would be “for example”, “example in”, “in this”, and so on. The larger the value of n we use when generating these grams, the more accurate our contextual analysis will be. However, the higher n is, the more memory our program will take as these n-grams must be stored in our trie.

The program first constructs the trie by reading from words.txt line by line. It adds each word to the trie, starting at the root node and working down the tree one character at a time. For example, when adding the word "hello" to the trie, the program would first add a child node for the letter "h" to the root node, then a child node for the letter "e" to the "h" node, and so on until the entire word is added to the trie and the last node’s frequency is set to one. A frequency greater than zero allows the program to denote where the end of a word is in the trie. If the node for the given character already exists, we traverse the trie instead of creating a new node. If the word being inserted already exists in the trie, the program increments the frequency of the word’s end node.

Once the trie has been initialized with words, the program then starts reading from grams.txt. Grams.txt is formatted so that each line contains a gram followed by the gram’s associated words. An associated word is the word that could be suggested based upon the user inputting the given gram. The two are separated by a “.” which the program uses to tokenize each line. When inserting a gram, the program traverses the preconstructed trie by the gram’s associated word. The gram is inserted at each node we cross through. If the gram already exists in the node, we increment the frequency of the gram.

After the trie has been fully constructed, it can now be used to make predictions. In the case of finding a word completion, the program starts at the root node of the trie and follows the path of nodes corresponding to the characters the user has already typed. For example, if the user has already typed "hel", the program would start at the root node, find the "h" node, then the "e" node, and finally the "l" node. At this point, the program would have reached a point in the trie where it can begin suggesting words. By recursively searching the nodes below this point for chains of nodes that lead to a frequency of greater than one, we append any resulting word and their frequencies to a vector and then use quicksort to order them by greatest frequency to lowest frequency. The program then outputs the top 5 results to the user. In the case of suggesting the next possible word given an input gram, we traverse the entire trie for chains of nodes that contain the given gram. When a chain is detected, we append each character to a string until we reach the end point of a word. This string along with the gram’s frequency in the word is then appended to a vector where the results will later be sorted.

This program, while functional, has a flaw in that it cannot distinguish between sentence endings and beginnings. It was later noticed that this issue could be resolved by including capital letters in any words or grams inserted into the trie. In addition, there are a few other improvements that could be made to the program. One possibility is to store strings in the nodes of the trie rather than individual characters, which would reduce the number of nodes. These strings could contain parts of the inserted word, branching off at points where different words diverge. Another potential change is to split the trie into two separate tries—one containing words and the other containing grams. Doing a depth first search to find all words in a trie can become time-consuming as the number of unique words increases. However, having a trie filled with nodes that contain individual words within a gram would make it much faster to search for a matching associated word given a gram. However, this modification would require substantially more memory, as there are thousands of possible ways to start a gram. It would be worth considering whether this tradeoff is worth it.