COMP30018 Knowledge Technologies

Implementing local edit distance with the TRE library

Daniel Porteous 696965

August 2016

1 Introduction

The algorithm examined in this report is local edit distance, namely with the standard Levenshtein distance scoring mechanism. The library in question is the TRE library, an efficient local edit distance implementation written in C, operated with a Python wrapper. The scope of the problem is beyond non-trivial, with an extremely large amount of data. In analysing $\simeq$1.3 million locations and $\simeq$300 thousand tweets, the number of Levenshtein distance calculations that need to be performed numbers around $\simeq$400 billion. In each of these location/tweet combinations, there might be checks for matches of various edit distances. The final tool is fairly efficient and robust, though can capture too many matches, solutions for which are discussed in the report.

2 Algorithm choice

Local edit distance treats a tweet as a monolithic entity, as opposed to tokenising like would be done for a global edit distance implementation (the same applies for multi-words location names). Treating the data as such has its advantages and disadvantages. Local edit distance is sometimes not as efficient because it must consider the whole tweet and all permutations of it n-edit-distance from the original. This is obviously more complicated than considering permutations of just small, distinct tokens. However the increased complexity comes with its benefits. Local edit distance excels at finding matches for locations with multiple words:

1. Going to WashingtnFc →Washington DC (Edit distance = 3)

as well as spelling errors that cross space boundaries:

2. Family visiting from tob oston →Boston (Edit distance = 2)

To do the former with global edit distance requires examining all the permutations of the tokens in respect to the length of the location being searched for, which diminishes the aforementioned advantages of global edit distance considerably. The latter task is even harder, as adjacent tokens will have to be considered on a per-character basis anyway.

2.1 TRE library

The TRE implementation of local edit distance features a per-search time complexity of $\mathcal{O}(n^2m)$[1], where m is the length of the regex query (location) and n is the length of the string being searched (tweet). While quadratic complexities are generally undesirable, the quadratic factor represents the location which is (usually) quite small compared to the length of the tweet. As such, the quadratic factor won't scale too heavily.

[1] [Lau] Laurikari 2016

3 Pre-processing

3.1 Basic operations

Some basic operations were first performed to clean up the data sets. While these steps involved tokenisation, the actual TRE algorithm did not. The small tweets file was comprised of 65969 words, which by the end of basic pre-processing was trimmed down to about ≃40000 (consider that each tweet had 4 unimportant tokens, the numbers at the start and the datetime at the end).

1. Removing unnecessary tokens from tweets, namely the two numbers at the start and the datetime at the end.

2. Removing punctuation and converting all characters to lower case for both locations and tweets (Twitter users don't capitalise locations consistently enough to utilise it).

3. For locations, removing tokens that are predominantly numbers and then removing duplicates. In the location set this removes 60158 items (consider all the wells).

4. Removing locations that are too short (4 chars) or long (70 chars). This removes 26995 items, but has obvious trade-offs in accuracy loss, anoter .

The code implemented these steps one by one for demonstration purposes, though many could be done simultaneously in a single pass for improved efficiency.

More heavy pre-processing steps could be performed taking into account Twitter conventions, for example that **rt:** means retweet or @ precedes a username, which could likely be safely ignored (occasionally a place will have an associated Twitter account).

3.2 Removal of common words

This pre-processing step was tested with mixed results. It involved the following steps:

1. Get the standard dictionary of words in UNIX systems from /usr/share/dict/words.

2. Iterate through the list, removing all locations in US-loc-names.txt from the dictionary. This leaves a dictionary of common words without location names.

3. Using this dictionary, remove all common words from all tweets, ideally leaving behind only location words.

This method significantly decreased the number of words in the tweets from ≃40000 (ie. after previous pre-processing) by 33869 to ≃6000.

While it may seem like this would hinder matching of multi-word locations (eg. *Kaby Lake*) because of the common word *lake*, the US-loc-names.txt dictionary actually has many places with names such as *Lake* alone. This prevented such issues from occurring as often as would be expected, making it a surprisingly valid method. Unfortunately, some other examples did suffer due to this procedure, for example removing the token *San* from *San Francisco*. This meant that it would only match on an edit distance of 3 *Francisco*. Nonetheless, the improvement in run time was significant. For the small tweet file and all 1.3 million locations (picking up edit distances from 0-3 inclusive), runtime with this improvement sat at ≃35 minutes, whereas without it required ≃165 minutes.

3.3 Inner-word matching

Something that turned out to be a much larger problem than expected was the substantial group of false matches where a small location substring (eg. Ash) was found within a larger location (eg. Washington DC). Local edit excels at this task (consider that the previous example is a perfect 0-cost match) which, while often desirable, was frequently annoying in this context. An effective solution was to modify the regex query from `/query/` to `/\bquery\b/`. While this still found small locations at the start and end of larger locations, it eliminated a large amount of the problematic cases. This is obviously in exchange for a loss in accuracy,

another noteworthy performance metric, as some (very small) amount of correct matches would be missed.

4 Precision analysis

In analysing the performance of the algorithm, precision[2] is primarily considered. A measure such as recall is unfeasible to calculate due to the enormous amount of manual work required to find all misspelled locations. Accuracy has been previously touched on.

4.1 Precision metric

The notion of a correct match is difficult to define. An algorithm such as this cannot intuitively tell whether a match is truly correct. Indeed, as far as the algorithm is concerned, each one of these matches **is** correct because it is within the given edit distance of one of the locations in the dictionary. As such, the following precision measures were ascertained by selecting a random subset of all the matches and deciding whether the matched location is what the user intended, for example *Berkley* →*Berkeley*, or not, for example *Coupon* →*Boston*. (Note also that calculating recall is effectively unfeasible due to the enormous amount of manual work required to find all misspelled locations in question).

4.2 Results

As expected, there are an enormous amount of false positives (incorrect matches). Inversely, at least by manual inspection there seem to be very few missed mistakes. This makes sense because the results capture so many variations with an edit distance of even 2, let alone 3. The vast amount of false positives are a result of a couple of factors, namely the huge amount of location names, especially small ones. More aggressive culling of small location names could be a way around this error, as well as a more sophisticated approach to the removal of common words.

A function was written which produces 200 random matches at a given edit distance. These are manually-verified precision measures for said 200 tweets at edit distances of 1-3 (files for which are included with the code):

1. 6% (12/200)
2. 2% (2/200)
3. 0% (0/200)

We see that as we increase edit distance, precision goes down. These results are to be expected (though to be taken cautiously over such a small sample size), as with each increase in the edit distance a far greater range of locations could match, the vast majority of which won't be correct matches, especially considering the wide range of cases that local edit distance handles better than other algorithms.

5 Conclusion

It goes without saying that the vast majority of potential spelling corrections that these algorithms find are not what the user had in mind. A good few steps towards improving these would be to aggressively cut down the locations dictionary down to the most commonly talked-about places, as well as more intelligent elimination of non-location words from the tweets in question. Real improvements would ideally come from more complicated technologies such as natural language processing, in which the algorithm could understand where a location is most likely to appear in a tweet, if at all. Anything that can bring the computer closer to attaining wisdom/understanding will be essential in the face of such a computationally enormous task.

References

[Lau] Ville Laurikari. *Github: laurikari/tre*. URL: `https://github.com/laurikari/tre`.

[2] precision is defined in terms of $CorrectMatches/NumberofMatches$.