COMP30008(KT) Project 1 Report

Task:

A file containing Twitter tweets and a file containing the names of US locations are given. In the tweets file, information of tweet_id and tweet_text of each tweet are given. Using each location in the location file as a query, we are supposed to find all the tweets that mentions the name of the location; spelling errors are allowed.

Method:

I separated all the words in the tweet_text in each tweet. All the words are converted to lower cases and all the non-alphabetic characters are removed.

Using all the words from all the tweets, a trie is built. Each word is associated with the tweet_id of the tweet that contains it. Multiple tweets may contain the same word. A set (a python data structure that is similar to list, used to store unique objects) of tweet_ids is therefore stored at the node representing each shared word.

The ascii name of each location provided in the location text file is used for the sting search. The name has normally more than one word. Each word is then used for a search in the trie. A set of tweet_ids is collected for each word search. And the intersection of all the sets is the set of tweet_ids whose corresponding tweets contain all the words.

Approximate string matching is achieved by using the global edit distance method. A maximum allowed distance is asked to be provided at the start of the programme. Whenever a tweet contains words whose combined global edit distance are non-larger than the maximum_allowed_distance from the location name, its tweetID will be recorded and displayed. This is implemented as follows:

```
e.g.  (hypothetical)
address name: Goose Hill
maximum allowed distance: 2

# find out the tweets that contains a word whose global edit
# distance is non-larger than 2 as compared to 'Goose'
Goose:
dist0_tweet_ids_G: 1000, 1005
dist1_tweet_ids_G: 1001, 1002
dist2_tweet_ids_G: 1003

# the same procedure is done for Hill
Hill:
dist0_tweet_ids_H: 1000
dist1_tweet_ids_H: 1005, 1002, 1008, 1003

# find the intersection sets between the two lists of sets, keep
# track of the combined distance
dist0_tweet_ids_G ∩ dist0_tweet_ids_H:
      dist0_tweet_ids: 1000
dist0_tweet_ids_G ∩ dist1_tweet_ids_H:
      dist1_tweet_ids: 1005
dist1_tweet_ids_G ∩ dist0_tweet_ids_H:
      dist1_tweet_ids: none
dist1_tweet_ids_G ∩ dist1_tweet_ids_H:
```

```
        dist2_tweet_ids: 1002
    dist2_tweet_ids_G ∩ dist0_tweet_ids_H:
        dist2_tweet_ids: none
    dist2_tweet_ids_G ∩ dist1_tweet_ids_H:
        # this intersection operation is not performed because the
        # resulting set has a combined distance of 3, which is larger
        # than 2
```

Effectiveness:

Pros:

Doing a global edit distance search in a trie is much faster than doing the search by looping through all the words in all the tweets. Many of the words are shared by many tweets, and their distances from the target word are therefore only calculated once. Moreover, the trie also aids in the speed of the global-edit-distance-search function, by using Steve Hanov's algorithm[1]. The programme runtime is shown below

```
Working on the training_set_tweets_small.txt and US_small.txt:
        with max_allowed_distance=0, it takes 5.95s
        with max_allowed_distance=1, it takes 97.26s
        with max_allowed_distance=2, it takes 478.68s

Working on the training_set_tweets.txt:
        each location query takes 90s if the max_allowed_distance=1
```

Example queries (from training_set_tweets_small.txt):

```
Latin Hill :
     with the distance of 2:
       6201654205
Return :
     with the distance of 0:
       5260328290 5537843907 5686760293 5639940679 5014901995
4808394644 5759205287 5816052854 4083602103 5754605433
```

Cons:

The problem with this algorithm is that it does not take into account whether the words of a location name are mentioned together or apart.

e.g.
one of results returned is:

```
First Church :
     with the distance of 0:
       3374002874
```

However, the tweet of tweet_id 3374002874 is actually:

```
@NinjaRehab Okay, first of all Jesus went to temple. Second, he
walked with his church and taught them all the time.  Be part of a
church.
```

'First' and 'church' are indeed mentioned in the tweet, but they are mentioned separately. The 'church' mentioned in this tweet is most probably not the 'First church' that we are looking for.

Scalability:

---

[1] Steve Hanov's algorithm can be found here: http://stevehanov.ca/blog/index.php?id=114

As the programme functions on a per query basis, the increase in the size of the location file will only incur linear growth in the programme runtime.

A mere increase in the tweet file size will incur nothing much more than a linear growth in runtime as well.

Bad scalability occurs when the max_allowed_distance increases or when the number of words of each location name suddenly increases. This is due to the bottleneck function[2] which deals with finding the intersection of all sets. This function runtime grows exponentially with the growth of any of these two factors.

Challenges:

Finding the intersection of multiple sets of ids is a non-trivial task for me. Different queries have different number of words. And for each word, it has an uncertain number of sets of tweet_ids, categorised by their edit distances. As a result, to programme a function that needs to find the intersection of a dynamic number of lists, inside which has, again, dynamic number of sets, is indeed not so straightforward for me.

Suitability & Possible Improvements:

In consideration of the tweets-file's huge size, the implementation of trie in this method has significantly improved the runtime as compared to searching through each tweet individually and treating each query and each tweet as a single string. Though the latter method will not have the problem as outlined in the Effectiveness section above, the improvement in runtime, in my opinion, outweighs the incurred inaccuracy in results.

If I am allowed more time, I may also store in the trie the information about the position of each word in its tweet. This way, I am then allowed to solve the above mentioned problem.

Another method that I am tempted to try is to have the all the tweets, treated as a single string, stored in a suffix array. Treating each location name as a single string as well, and it is searched in the suffix array. The approximate string matching can be implemented by traversing the whole suffix array top down while keeping count of the distances of each route. When one route has its distance exceeding the maximum allowed distance, that route is abandoned. And the remaining successful routes are returned. The indices of the found string-matches are then used to backtrack to find their corresponding tweet_ids.

---

[2] the function is "process_search_result(output, allowed_errors)" located in dis_search.py