Term Project Report - T-61.5060

Rakshith Shetty (466945)
rakshith.shetty@aalto.fi

I. Preliminaries :

The term project involved tasks related to implementing Nearest neighbour search algorithms on a database of tweets. Distance between two tweets x and y is measured using the angle distance as defined in Eq 1.

$$angle(x, y) = \arccos \frac{|x \bigcap y|}{\sqrt{t(x)}\sqrt{t(y)}} \quad (1)$$

where $t(x)$ and $t(y)$ are the number of words in tweets x and y respectively.
We are tasked with programming brute-force search, optimized exact search and approximate search algorithms to search nearest neighbours of 1000 query tweets.

Implementation setup: I used C-programming language to complete most of the tasks in the project. Plotting was done using Matlab. I borrowed some source code for basic building blocks like reading a word, building hash table for the vocabulary, from open-source codebase from Google [1] , which I had used before. Rest of the code was written on my own. I ran all my experiments on a Octa-core Linux machine with 8GB RAM with each core running at 1.6Ghz clock speed, accessible at at *rhea.ics.hut.fi*. Further instructions on how to use the code can be found in the README file in the source directory.

II. Task 1: Preprocessing

To complete this task, I run through the whole database once collecting all unique words and counting their occurrences. Each word is represented as an entry in a array of structures and hash table is used to quickly retrieve the word in this array. Once the whole vocabulary is built it is sorted in the descending order of its frequency. This whole process took about **33.4** seconds. Then it is dumped into a file and plotted in matlab. I found that there are a total of **8672048** unique terms in the database and the database contains **14707260** tweets with the maximum length of a tweet being **45** terms.

Let n_t be the number of tweets term t occurs in. Figure 1, shows the distribution of n_t, i.e number of terms that occur exactly in n_t tweets. This figure is plotted in log-log scale, i.e both x and y axis is in log scale. As we can see there are a lot of terms which occur in a small number of tweets. For eg. there are about 6277522 terms which occur in only one tweet! That's about 72% of the database. Rest of the terms seem to be concentrated around 100-1000 tweets.

Figure 2 shows the cumulative of this distribution. Cumulative of this distribution is defined as

$$Cum(n_t) = Terms\ occuring\ in\ n_t\ tweets\ or\ less$$

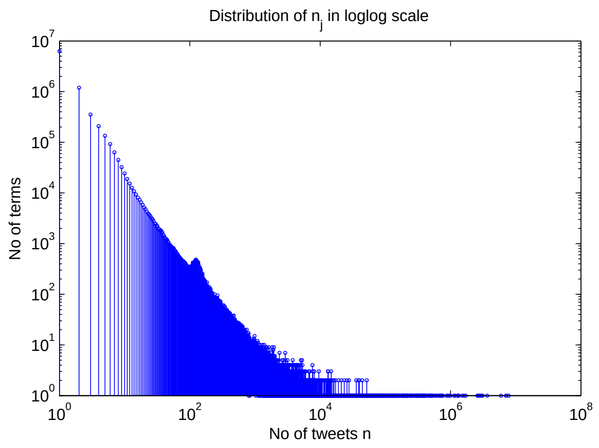

Figure 1. Distribution showing number of terms occuring in n_t tweets

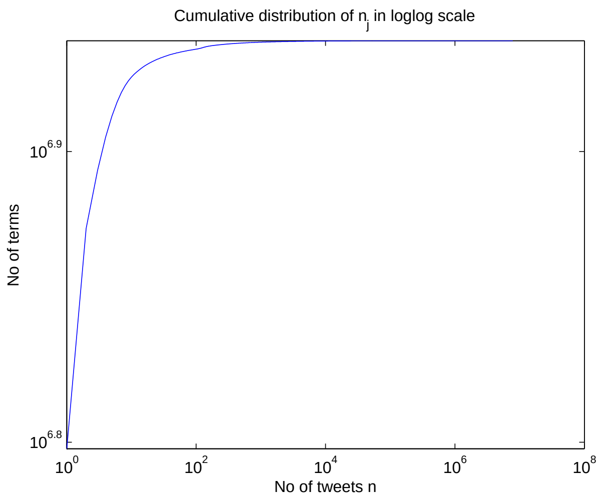

Figure 2. Cumulative distribution of n_t

Cumulative distribution also tells the same story, most of the terms occur in only 1 or 2 tweets. This could be because of misspellings, non-standard english used in tweets, and also possibly due to pre-processing done on the data. For eg. there is a word 'cantgetawayevenatcolleg' which is either a result of user who tweeted this trying to save on characters by cutting on white-space or a result of preprocessing on the data. There are also a lot of terms which seem gibberish, for eg, cnzupo, gwlebmcqa, zyrguvqw etc which make them unique.

Figure 3 shows the plot of number of tweets term t occurs in. This shows us that there are few frequent words which occurs in a lot of tweets. Top 5 frequently occurring terms are t, co, http, ebola, and u. There are about 20 terms which occur in more than a million tweets and about 2000 terms

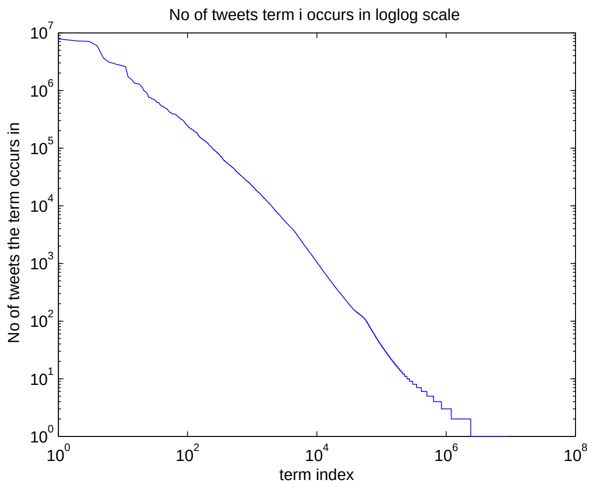

Figure 3. Plot of number of tweets a word t occurs in

which occur in more than 10000 tweets.

III. Task 2: Brute Force Search

Before performing brute force search I did a preprocessing on the database and queries where all words are replaced with its position index in the vocabulary. This is done in-order to avoid doing string comparison millions of times and instead compare the word indices which is equivalent. This preprocessing step was quite fast as I used hash table to search for a word in the vocabulary and retrieve its index. It takes about **40.32** seconds to complete this task for the entire database. After this process all the database tweets are written into a binary file along with their lengths to make it easy to retrieve it later. For eg the tweet '*a co crimea divid doe east hasten http polit secess split t threaten ukrain via w west xtcnwl youtub*' is represented as '0 19 5 1 900 2269 319 207 32669 2 373 15577 3092 0 688 12 41 94 220 2394526 327'. Here first entry, 0, is the tweet id and second, 19, is tweet length. Rest are indexes of the words present in this tweet. It is to be noted that this preprocessing step is done before all the search algorithms discussed in this report.

After this brute-force search is run on the whole database looking for the nearest neighbours of all the 1000 queries. All the queries are kept in the memory. Database is processed one entry at a time from the disk. For each database entry, its distances to all the 1000 queries is computed at onetime, and minimum distance array is updated for all queries. Hence in one sweep of the database we will have all nearest neighbours for all the 1000 queries. This process took $\boxed{\mathbf{1976}\ seconds}$ to complete the search for all 1000 queries.

I also repeated the process on the databases obtained by using dimensionality reduction techniques. Figure 4 a) shows the execution times per tweet vs d for databases containing d frequent terms. This execution time seems to be approximately log-linear in d. I have plotted time per query as some query tweets gets completely removed in the reduced database (esp. for small d) and to get a fair comparison it is better to look at $total\ time\ taken/no\ of\ queries$. Similarly Figure 4 b) and Figure 4 c) show the average execution time per query for d-infrequent and d-random databases respectively. It is to be noted that for d=100, 400 all queries in the infrequent and random databases get removed and hence I put execution time as 0.

IV. Task 3: Exact nearest neighbour search.

In this section I describe the algorithm I designed for doing exact nearest neighbour search and the speedups obtained.

A. Algorithm

I used two main ideas to design a fast exact search algorithm for this problem (from the hints). First the fact that to find the nearest neighbours we only need to look at tweets in the database which have atleast one term in common with the query. Secondly we can use the length constraints to do early termination.

1) Exploiting constraint to have atleast one common term: To exploit the first fact, I decided to build a database which stores for each term t in the vocabulary, $List(t)$ the list of all tweets it appears in . For eg, term 't' appears in about 7 million tweets so the list for 't' is 7 million long.

Also each of these lists are sorted in ascending order of tweet lengths. Like I noted before the maximum length of a tweet in the database is 45 terms. Hence it is safe to assume there are atmost 45 unique tweet lengths in the database. Even within tweets of the same length the tweets are stored in ascending order of their id. This is very important and is used later to do efficient search in these lists.

All the lists are stored in a single binary file by concatenating them serially. Let's refer to this file as 'Map' file. Another header file is built which contains for each term t, $Offset(t)$ the offset to the beginning of $List(t)$ in the Map file. It also contains an array $LenBlk(t, i)$ of size 50 for each term, where

$$LenBlk(t, i) = offset\ of\ first\ tweet\ of\ length\ i\ in\ List(t)$$

Hence to retrieve the list of tweets of length 2 containing the term w, we need to read from offset $Offset(w)+L(w, 2)$ in the Map file. Let $List(w, l)$ denote the sub-list of $List(w)$ which contains only tweets of length l.

Once we have these two files built we are ready to do the nearest neighbour search. Now given a query Q, we just need to look at tweets in the union of $List(t)$ of each term t in the query. Formally search space of Q is

$$SearchSpace(Q) = \bigcup_{i=1}^{length(Q)} List\left(Q(t)\right)$$

One more thing to observe is that we don't need to actually retrieve the database tweets and compute the distances anymore. If a database tweet Dt has I terms in common with query Q of length l_q, then it will appear in I of the lists $List\left(Q(1)\right), List\left(Q(2)\right), \cdots List\left(Q(l_q)\right)$. Finally among database tweets of length l, nearest neighbour to Q is the one which appears in highest number of sub-lists $L(Q(t), l)$. Hence we can go over l_q sublists of fixed length l at a time, find the most repeated tweet in these l_q lists. Then

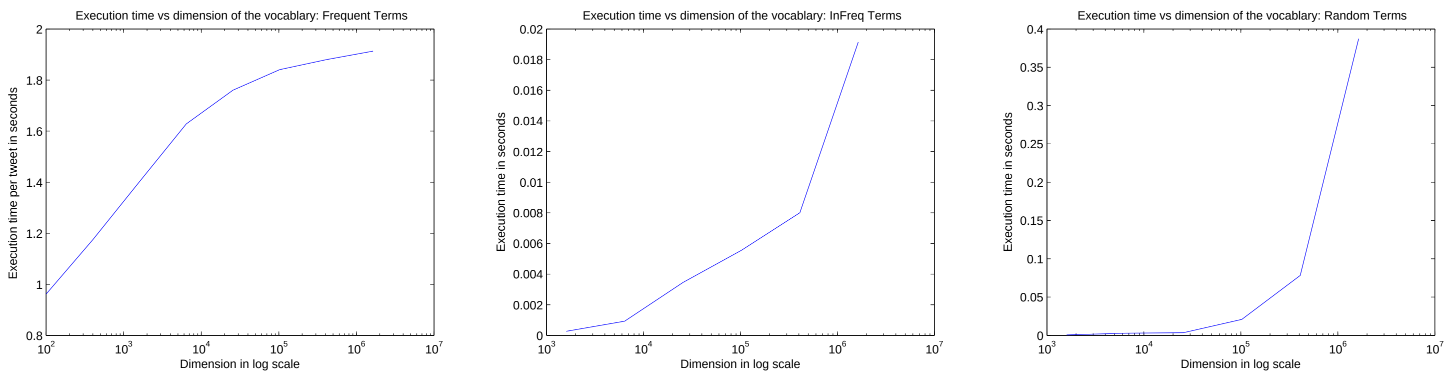

Figure 4. Execution time vs d for brute force search using a) d-Frequent b) d-Infrequent and c) d-Random databases

combining results from all lengths , we obtain the nearest neighbour. This is shown formally in Algorithm 1.

Major chunk of the computation in this algorithm is in finding the tweet which occurs in highest number of lists of fixed length. This problem is equivalent to finding union of n sorted lists, except we don't really need the union but instead need to keep track of the entry which repeats the most when computing the union. For this I designed the algorithm given in Algorithm 2 taking inspiration from the Priority Queue algorithm described here [2]. In summary these are the steps involved here:

- At each iteration find the smallest entry min_el at the head $(List(i)(0))$ of all the l_q lists.
- The number of lists that have this minimum entry at the head is the number of repetitions of this entry. Let this be R, and let $minL$ be the set of lists that have min_el at their head.
- Also keep track of the second smallest entry while finding the minimum, min_2.
- Now if $R > R_{max}$, update $R_{max} = R$, and $NN_{len} = min\_el$
- Remove min from all the lists in $minL$. There are R of them. Advance their pointer till their top entry is $\geq min_2$.
- Repeat till all the lists are exhausted or if $R_{max} > l_q$ or $R_{max} > len$

2) Exploiting length constraints: Lets say that we have obtained current minimum distance d_{min}, and it was with a database tweet of length l_{min} which yielded intersection I_{min}. Now we can put a constraint on the minimum number of intersections needed from the next block of tweets of length l to get a smaller distance as follows:

$$\begin{aligned} d_{min} &> d \\ \arccos\left(\frac{I_{min}}{\sqrt{l_{min}}\sqrt{l_q}}\right) &> \arccos\left(\frac{I}{\sqrt{l}\sqrt{l_q}}\right) \\ \frac{I_{min}}{\sqrt{l_{min}}\sqrt{l_q}} &< \frac{I}{\sqrt{l}\sqrt{l_q}} \\ I &> I_{min}\sqrt{\frac{l}{l_{min}}} \end{aligned} \tag{2}$$

Hence equation 2 gives a lower bound on the number of intersections needed from the next block of tweets we are going to look at. Note that since we are searching in ascending order of tweet lengths $l_{min} < l$

Algorithm 1 Computing nearest neighbour using tweet lists

```
for i = 1: l_q
  List(i) = loadList(Q(i));
end
best_sim = 0

/* MaxLength is defined as 50 */
for len = 1:MaxLength
  /* Process one length block at a time
   * and find the tweet which repeats
   * appears in most lists.
  */
  [Intersection , I_max_tweet] =
    computeMaxRepeat(len, l_q , List(1: l_q,len))

  similarity = Intersection/sqrt(len)
  if similarity > best_sim
    best_sim = similarity
    nearest_neighbour = I_max_tweet
  end
end
```

But we also know that I is lower bounded by

$$I < \min(l, l_q) \tag{3}$$

This lower bound can be further extended internally into the Algorithm 2, by observing that at any iteration, I is upper bounded by $active\_lists$. Thus, we can terminate the search in sublists of length l, $List(:, l)$, when the lower bound in Eqn 2 exceeds the upper bound in Eqn 3, as shown in Eqn 4. This termination condition gives great speed improvements as lot of length blocks get skipped or terminated early.

$$I_{min}\sqrt{\frac{l}{l_{min}}} > \min(l, l_q, active\_lists) \tag{4}$$

B. Results

First task was to build the Map and Map header indexes for the whole database. This is a memory intensive task and I had program the whole thing to run completely from the disk. Hence pre-processing time greatly increased. It took about 2.5

Algorithm 2 Computing most repeated entry in n lists

```
function computeMaxRepeat(len,l_q, Lists)
done = 0
% Initialize pointer to all lists
for i = 1:l_q
    p[i] = 0;
    end[i] = length(Lists(l_q))
end

max_repeat = 0;
NN = 0;

while !done
    done = 1
    min_el = infinity
    second_min = infinity
    R = 0
    active_lists = 0
    % go through all lists and find the current minimum element
    for i = 1:l_q
        % Check if this list is exhausted
        if p[i] < end[i]
            done = 0
            active_lists++
            if Lists(i)(p[i]) == min_el
            % this list also has the same element as the current minimum.
            % This represents an intersection
                minL[R] = i
                R++
            elseif Lists(i)(p[i]) < min_el
                second_min = min_el
                min_el = Lists(i)(p[i])
                R = 0
                minL[R] = i
                R++
            elseif Lists(i)(p[i]) < second_min
                second_min = Lists(i)(p[i])
            end
        end
end
    % now update the current best
    if max_repeat < R
        Rmax = R
        NN = min_el
    end
    % now update the pointers
    for i = 1:R
        while (p[i] < end[i]) && Lists(i)(p[i]) < second_min
          p[i]++
        end
    end
    % Some exit conditions  to terminate the search
    if (active_lists < max_repeat) || (max_repeat == l_q) || (max_repeat == len)
        break
    end
    end
    return NN, Rmax
end
```

hrs to build these two indexes for the entire database, but cpu consumption of this program was only about 10%, and process was mostly waiting on disk access.

Once the database is built I ran the exact nearest neighbour search on the full database which took $\boxed{186.75\ seconds}$. This is a speedup of about **10.58 times** over brute force search. I also repeated these experiments over dimensionality reduction databases. Figure 5 shows the speedup factor plotted for all the three dimensionality reduction databases. For d-Frequent database the speedup is significantly higher for small d, and settles down to a factor of ~10.8 as d increases. For d-Infrequent databases speedup factor seems to grow with the database size. And for random database it is fluctuating around 200. Better performance on infrequent and random databases could be explained by the fact that the algorithm performs best when there are some very small lists which will get exhausted quickly and then early termination conditions kick in.

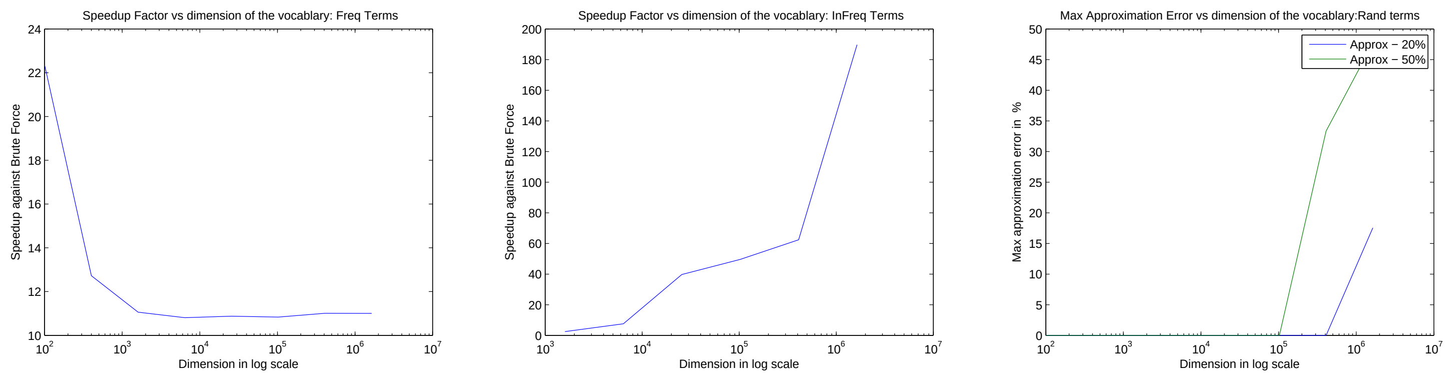

Figure 5. Speedup factor vs d for Exact search using a) d-Frequent b) d-Infrequent and c) d-Random databases

V. Task 4: Approximate nearest neighbour search.

A. Algorithm

My implementation of the approximate search algorithm is identical to the exact search implementation except for a change in the length constraints. It is easy to see that we could incorporate the approximation factor α into the length constraints as below

$$\begin{aligned} d_{min} &> \alpha d \\ d_{min} &> \alpha \arccos\left(\frac{I}{\sqrt{l}\sqrt{l_q}}\right) \\ \cos\left(\frac{d_{min}}{\alpha}\right) &< \frac{I}{\sqrt{l}\sqrt{l_q}} \\ I &> \sqrt{l \cdot l_q}\cos\left(\frac{d_{min}}{\alpha}\right) \end{aligned} \tag{5}$$

Thus for the approximate search I use this new lower bound given in Eqn 5 instead of Eqn 2 and the termination condition now becomes

$$\sqrt{l \cdot l_q}\cos\left(\frac{d_{min}}{\alpha}\right) > \min\left(l, l_q, active\_lists\right) \tag{6}$$

Apart from this change in termination condition rest of the algorithm remains same as exact search. It is to be noted that the approximation factor here bounds the maximum approximation error and not the mean error.

B. Results

Running approximate nearest neighbour search on entire database took $\boxed{166.6\ seconds}$ for 20% error $(\alpha = 1.2)$ and $\boxed{126.98\ seconds}$ for 50% error $(\alpha = 1.5)$, which is a speedup of about **11.86 times** and **15.56 times** respectively over brute force search. I also repeated these experiments over dimensionality reduction databases. Figure 6 shows the speedup factors for both approximate search results and for all three dimensionality reduction databases. As we can see for Frequent terms this again follows the same pattern as exact search. Also 50% approximate search is faster than 20% approximate search as expected. Figure 7 shows the maximum approximation error over all queries for d-frequent, d-Infrequent and d-Random databases. We can clearly see that the maximum error never crosses the bound as expected. Figure 8 shows the mean approximation error for the three datasets. This plots expose one shortcoming in this approximate algorithm that the mean error is quite small (~4% for 50% bound and ~1% for 20% bound). Thus the algorithm doesn't fully utilize the error margin provided and could run much faster if it did. This of-course could be simplistically fixed by scaling up the lower bound, but this would require tuning the scaling factor and would probably be query/database dependent.

VI. Conclusion

In this report I discussed the implementation of nearest neighbour search algorithms on a twitter database. As was seen exploiting the length constraints we could design very efficient algorithms which is able to search nearest neighbours for 1000 queries in around 3 minutes. Approximation algorithm was designed as a simple extension of exact search algorithm described. Although it was faster than the exact search, it was found to not use the error bound fully and could be made faster by futher tuning of the approximation factor. I wanted to try out locality sensitive hashing or other truly randomized algorithms but ran out of time. I think more efficient approximate algorithms could be designed by explicitly taking into account the distribution vocabulary frequencies. On other note, The scale of the problem forced me to explore memory efficient programming techniques and use of structured binary files for efficient disk access.

References

[1] https://code.google.com/p/word2vec/
[2] http://www.programcreek.com/2013/02/leetcode-merge-k-sorted-lists-java/

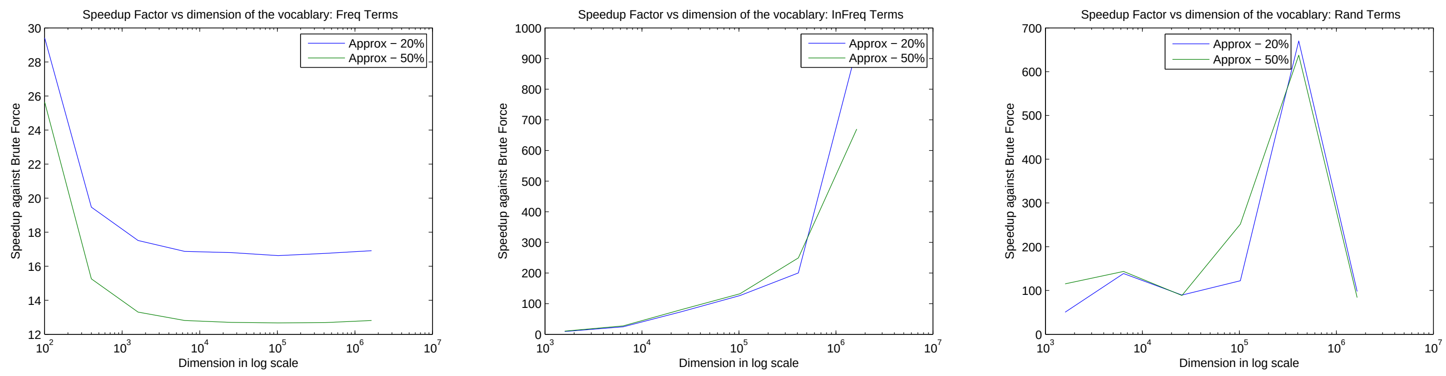

Figure 6. Speedup factor vs d for Approximate search using a) d-Frequent b) d-Infrequent and c) d-Random databases

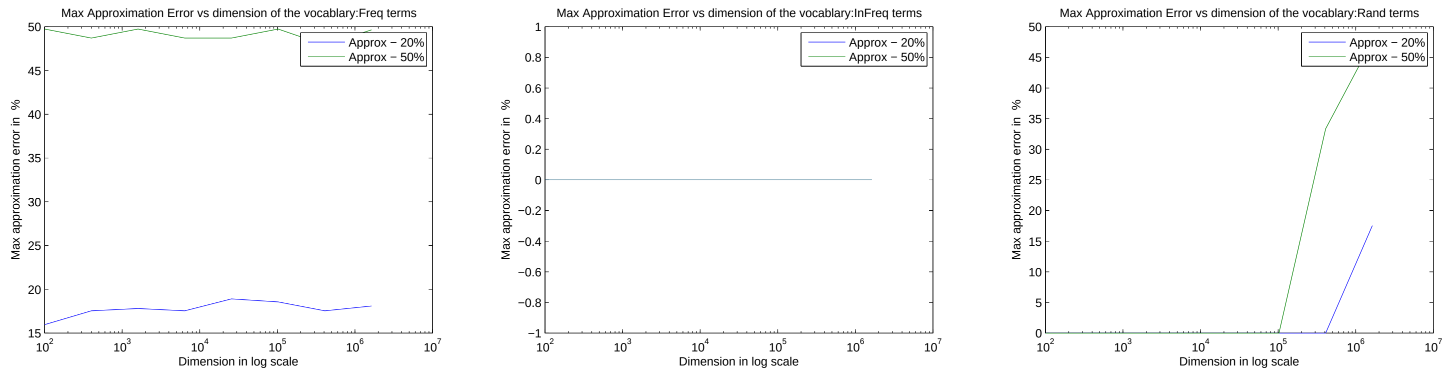

Figure 7. Maximum approximation Error vs d for a) d-Frequent, b) d-Infrequent and c) d-Random databases

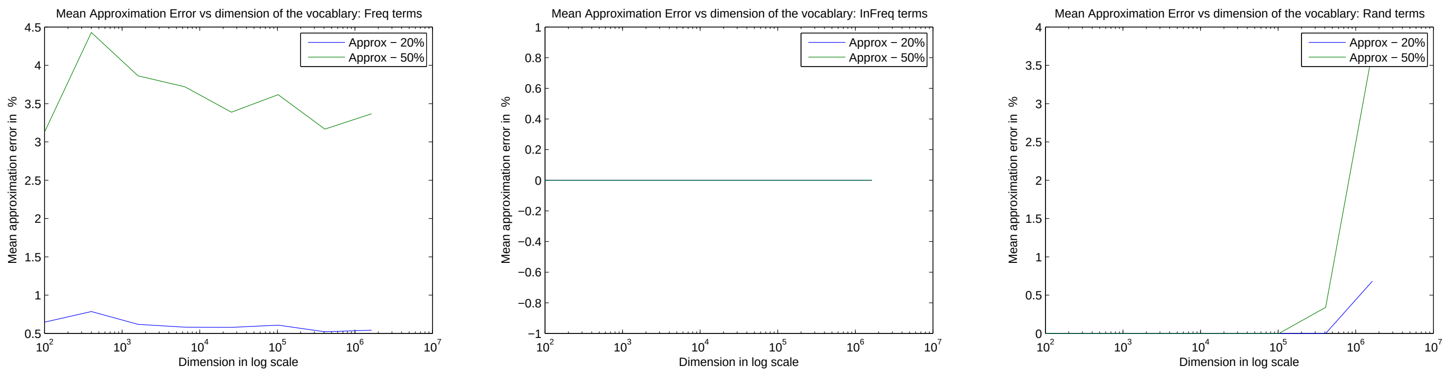

Figure 8. Mean approximation Error vs d for a) d-Frequent, b) d-Infrequent and c) d-Random databases