

Michael Lan
Lingfei Zeng
Xunjie Zhu

Stage2 - The Design Stage.

- Design Description:

There are three types of users who will use this system. We will begin with the first user type: 1) users who select a topic and bring up a cluster of tweets related to that topic.

- ◊ 1) Tweet clustering based on semantic similarity

- ◊ 1.1) Semantic Similarity Measurement

We aim to cluster tweets into different groups using Hierarchical Clustering. To do this, we require semantic similarity scores between two tweets (or sentences, documents. We may use these terms interchangeably.). We propose two methods for calculating the similarity between two sentences: (1) Edit distance based approach (2) TF-IDF based approach.

- ◊ 1.1a) Extended Edit Distance Algorithm

We utilize an extended **Edit Distance** algorithm integrated **Word2Vec**[1] embedding to compute the distance between sentences in the tweets, and by grouping by shortest distance we divide tweets into different groups.

In details, we describe tweet sentence S_1 as $S_1 = \{a_1, a_2, \dots, a_n\}$ and sentence S_2 as $S_2 = \{b_1, b_2, \dots, b_m\}$. S_1 consists of n words and S_2 consists of m words. a_i is the i th word of S_1 and b_j is the j th word of S_2 . We use $\text{Sim}(S_1, S_2)$ to describe the similarity between S_1 and S_2 , with the range of 0 (no relation) to 1 (semantic equivalence). For edit distance, we utilize Levenshtein Distance algorithm to extend to the word sets in sentences. We define the Levenshtein distance between S_1 and S_2 is $L(n, m)$, where $0 \leq i \leq n$ and $0 \leq j \leq m$. And we have:

$$L(i, j) = \max(i, j) \quad \text{if} \quad \min(i, j) = 0,$$

$$L(i, j) = \min \begin{cases} L(i-1, j-1) + c(a_i, b_j) \\ L(i, j-1) + 1 \\ L(i-1, j) + 1 \end{cases} \quad \text{otherwise.}$$

Where the function $c(a_i, b_j)$ is defined as

$$c(a_i, b_j) = \begin{cases} 0 & (a_i = b_j) \\ 1 & (a_i \neq b_j) \end{cases}$$

We also need to consider the circumstances that if two different words have the same or similar meaning (e.g., “cat” and “kitty”), in which case edit distance defines them as a mismatch. We address this issue by introducing Word2vec to measure semantic similarity between words. Word2vec is an unsupervised deep learning algorithm that maps each word to a vector $\in \mathcal{R}^n$ such that the semantically similar words are mapped to the vectors that are close to each other in the geometry space. In our settings, given any two words, e.g., a_i and b_j in the above, we first apply word2vec to map them to two vectors, \vec{v}_1 and \vec{v}_2 , and then use the euclidean distance, $\|\vec{v}_1 - \vec{v}_2\|$, to measure the cost of the replacement. We may normalize the cost into a relatively small range.

$$c(a_i, b_j) = \|\vec{v}_1 - \vec{v}_2\|$$

where $\vec{v}_1 = \text{word2vec}(a_i), \vec{v}_2 = \text{word2vec}(b_j)$

◇ 1.1b) TF-IDF Weighting

The other method for calculating the similarity is using the TF-IDF algorithm. Similar to the above Edit distance based algorithm, the TF-IDF based method also needs to represent the words to vectors. In addition, the TF-IDF based method also needs to represent the documents as vectors. The traditional method to represent word and document vectors is Bag of Words. The bag-of-words feature vector is the sum of all one-hot vectors of the words, and therefore has a zero value for every word that did not occur. In the weighted variation, it is a weighted sum according to term frequency or TF-IDF scores. But the problem is that the document vector constructed by this method is high dimensional and sparse(because there are thousands of words in the corpus but a tweet can at most contain 140 words, so most of the values in the document vector is 0). In order to solve this problem, we use **Word2Vec**[1] to get a dense, low-dimensional embedding vector of each words. Then, we will use two divergent algorithms to measure the similarity of documents, the first one is based on extended Edit Distance and the second one is based on TF-IDF weighting.

After getting vectors of words, we can use weighted sum of word vectors to represent document vectors, then use cosine distance of document vectors to measure the similarity between them. We first introduce the concept of TF-IDF.

TF is a short-cut for term frequency, which means how frequently a term occurs in a certain document. IDF is a short cut for Inverse Document Frequency, which measures how important the term is. Let $TF(i, j)$ be the term frequency of $word_i$ in $documents_j$, $count(i, j)$ be the times that $word_i$ appears in $document_j$, C be the total number of words in all documents. N be the total number of documents, $df(i, j)$ be the number of documents containing $word_i$. The formula of TF-IDF are shown as follows.

$$TF - IDF(i, j) = TF(i, j) * IDF(i, j)$$

$$TF(i, j) = \frac{count(i, j)}{C}$$

$$IDF(i, j) = \log \frac{N}{df(i, j)}$$

We can represent each document as a matrix with C rows, each row i of the matrix is the embedding vector of the $word_i$ if $word_i$ is in the document or a vector consisting of zero if $word_i$ is not in the document. Then, we use TF-IDF method to calculate a weighting vector, the vector contains C rows, each row of the vector is the TF-IDF value of corresponding words. The paragraph vector of a document is the dot product of document matrix and weighting vector. Then, we can use cosine similarity of two vectors measure the distances. For any two vectors a, b

$$cos(a, b) = \frac{a \cdot b}{|a| \cdot |b|}$$

◇ 1.2) Clustering Algorithm

After completing similarity measurement, we need to choose a clustering algorithm to cluster the tweets. Since we did not know the exact number of categories, so clustering algorithms based on partitions such as K-means are not suitable. Hierarchical Clustering would be a better choice. We choose agglomerative clustering algorithm.

Agglomerative clustering algorithm treats each tweet as a singleton cluster at first and repeatedly merge pair of clusters with minimum distance until the pairwise distances between all remaining clusters are lower than certain number. The function which is used to measure the distance between two different cluster C_i and C_j is usually called linkage function. There several types of linkage function: single linkage, complete linkage and average linkage. Assume $distance(m, n)$ is the semantic distance between tweet m in cluster C_1 and tweet n in cluster

C_i . Single linkage is the minimum of the distance(m, n) for all $m \in C_i, n \in C_j$, Complete linkage is the maximum of the distance(m, n) for all $m \in C_i, n \in C_j$, Average linkage is the average of the distance(m, n) for all $m \in C_i, n \in C_j$. The formula of above three linkage functions are shown as follows:

$$\begin{aligned} single(C_i, C_j) &= \min_{\forall m \in C_i, n \in C_j} distance(i, j) \\ complete(C_i, C_j) &= \max_{\forall m \in C_i, n \in C_j} distance(i, j) \\ average(C_i, C_j) &= \frac{\sum_{\forall m \in C_i, n \in C_j} distance(i, j)}{|C_i||C_j|} \end{aligned}$$

- ◇ 2a) Recommending similar tweets and hashtags based on structural similarity

Next, we will describe the other 2 user types: 2) users, searching for a hashtag H, who want to find tweets related to H, but do not contain H and 3) users who want to find hashtags similar to a tweet that does not contain those hashtags. In case 2), the Tweet Search job algorithm will be invoked. In case 3), the Hashtag Search job algorithm will be invoked.

Before any search jobs are invoked, a directed bipartite network G with two different types of nodes, tweets and hashtags, will be created. Instead of storing them as strings, we give a unique ID to each tweet and hashtag that allows for efficient identification. A tweet will have an outgoing edge to a hashtag if it contains it. The graph G^2 has as its nodes every pair of tweets and every pair of hashtags in G. In Algorithm 3, we apply SimRank on G^2 to find a similarity score for every pair of tweets. The more outgoing edges two tweets have to the same hashtags, the more similar they are. If this network does not contain the tweet or the hashtag that the user inputs, then the search algorithms will return that no results were found.

The structural similarity, based on SimRank, is evaluated using these two equations, which will be explained below: Let A and B be tweets. $O(v)$ are the outgoing neighbors of v. The similarity $s(A, B)$ is:

$$s(A, B) = \frac{C_1}{|O(A)||O(B)|} \sum_{i=1}^{|O(A)|} \sum_{j=1}^{|O(B)|} s(O_i(A), O_j(B))$$

C is a constant between 0 and 1. It is the decay factor and is the only adjustable parameter of the algorithm. The reason for using C will be explained later.

Let c and d be hashtags. $I(v)$ are the ingoing neighbors of v . The similarity $s(c,d)$ is:

$$s(c, d) = \frac{C_2}{|I(c)||I(d)|} \sum_{i=1}^{|I(c)|} \sum_{j=1}^{|I(d)|} s(I_i(c), I_j(d))$$

The similarity between two tweets is the average similarity of their hashtags, $O(A)$ and $O(B)$. The similarity between two hashtags is the average similarity of tweets that contain them. Let A be either a hashtag or a tweet. The starting scores for the algorithm are the nodes (A,A) , which are given similarity score of 1, while all other nodes are given a similarity score of 0. In the next step, the similarity scores for hashtag nodes with an ingoing edges from tweet nodes (A,A) and tweet nodes with outgoing edges to hashtag nodes (x,x) are computed. These scores are calculated using $C*s(A,A)$, because two nodes that share a node by 1 degree of similarity as less similar than two nodes which are the same tweet. This iteration occurs until step K , such that the scores from step $K-1$ are the same as the scores at step K .

Similarity scores were shown to always exist, are unique, and are symmetric for any two nodes. Since $s(a,b)$ and $s(b,a)$ are symmetric, so only need to store one of their scores. Algorithm 4, the tweet search job, finds node pairs, which contain the searched tweet, that are most similar to the searched tweet. Then, algorithm finds the hashtags contained in the input tweet, and stores them in a set A . It also finds the hashtags contained in these similar tweets, and stores them in a set B . Finally, it returns the relative complement $B-A$, the desired output. Algorithm 5, the hashtag search job, works in the same way, but switches the roles of tweets and hashtags.

Many of the hashtags that are returned will too general to characterize specific topics. For instance, many tweets may contain the hashtag $\#news$, which may often be 2 degree of separations from $\#sports$ and $\#fashion$. At first, this might imply that $(\#sports, \#fashion)$ have high similarity because there are many (not directed) paths of length 4 connecting them, meaning they're often separated by 4 degrees. But their similarity is not only determined by the number of connections, but the score of those connected nodes. A tweet with both $\#sports$ and $\#news$, and a tweet with both $\#news$ and $\#fashion$, may have $\#news$ in common, but not much else. These means these tweets are often not very similar to each other, so they are weighted very lightly when determining the score of $(\#sports, \#fashion)$. Thus SimRank takes these very general hashtags into account. It uses the idea of mutual reinforcement to find which hashtags can specifically

characterize tweet topics.

This algorithm does not aim to find phrases which distinguish one topic from another; rather, it seeks to bridge communities by finding their similarities.

◇ 2b) Time Complexity of Algorithm 3 to 5

Assume we have constructed the graph G^2 by using graph G . Now let $d = \text{the average of } |I(a)| * |I(b)| \text{ over all nodes in } G^2$. Let $K = \text{average number of iterations the algorithm performs}$. Let n be the number of nodes in G^2 . Using a ‘naive’ implementation of Sim-Rank, average running time of Algorithm 3 is $O(K*n*d)$. Its worst case mainly depends on K .

Algorithms 4 and 5 are essentially the same, but switch the roles of tweets and hashtags. $P = \text{the number of tweet (for Algorithm 5) or hashtag (for Algorithm 4) nodes of } G^2$. WLOG, we will analyze Algorithm 4 and say Algorithms 4 and 5 have the same running time. First, it finds all the node pairs containing the input, which takes $O(P)$; if the input is not in the graph, it returns “No results were found.”

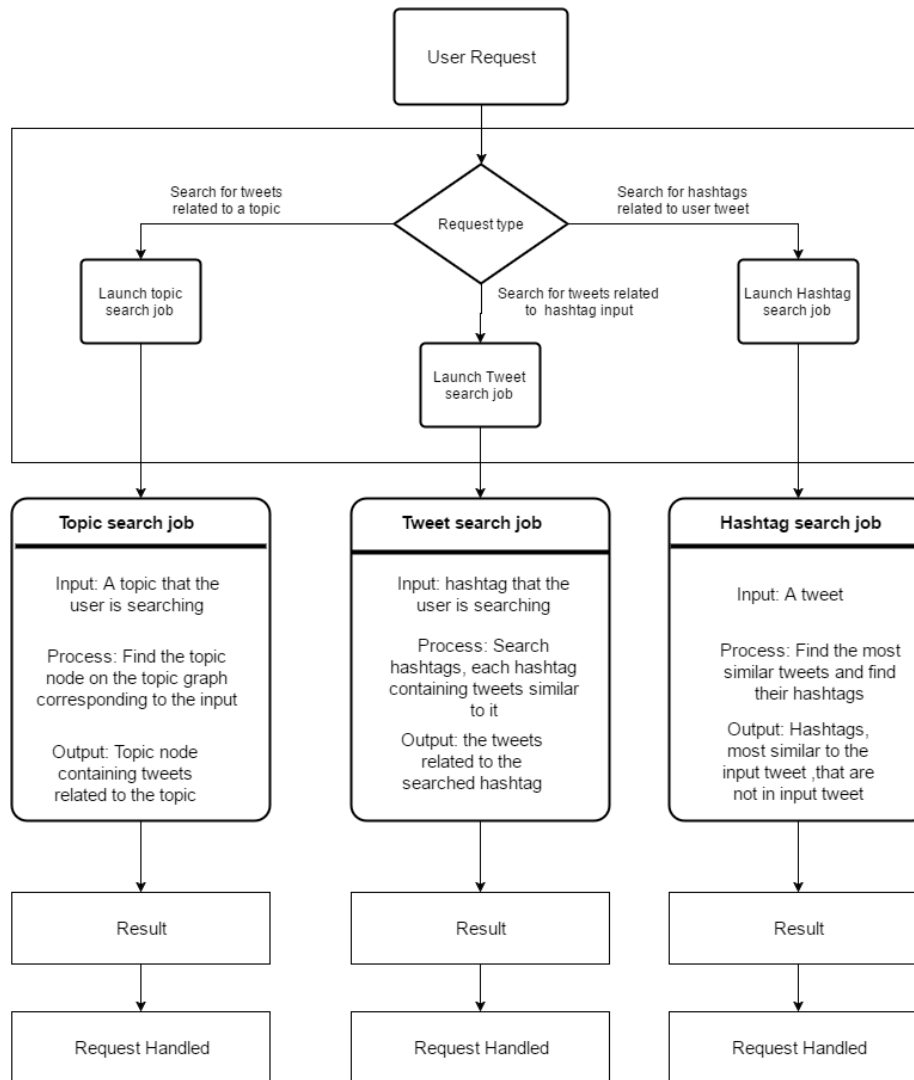
Let R be the length of the node scores hash table. The algorithm sorts the scores, and Mergesort has an average run time of $O(R*\log(R))$. Next, it finds all the neighbors of the input. We do not have an average case, so we look at the worst case, which is $O(n-P)$.

Most of the running time is due to computing the relative complement. Assuming the set contains no duplicate elements, this step compares two sets, so it takes $O(P*(P-1))$ * (the comparison cost); the comparison cost is constant because we are not comparing lengthy strings, but IDs. The total running time is: $O(P + R*\log(R) + (n-P) + P*(P-1)) = O(P*(P-1))$, which is polynomial.

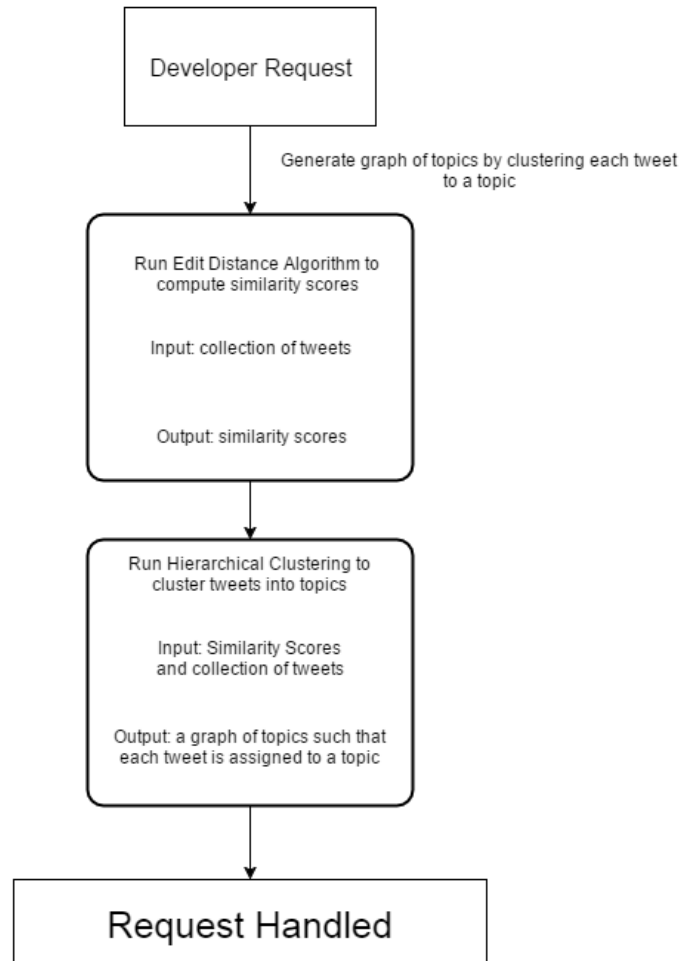
◇ 2c) Space Complexity of Algorithm 3 to 5

In Algorithm 3, we store `node_scores` in a hash table, so its space complexity is $O(n)$. In Algorithm 4, we have two hash tables of hashtags, which each cost $O(n-P)$ to store. We have one hash table of tweets, which costs $O(P)$ to store. So the total space complexity of Algorithm 4 is $O(n-P + P) = O(n)$. WLOG, the total space complexity of Algorithm 5 is $O(n)$.

Flow Diagram For User Requests



Flow Diagram For Clustering Tweets



- High Level Pseudo Code: refer to Algorithms 1 to 5

Algorithm 1: Edit Distance

input : A sentence S_1 with n words as $S_1 = \{a_1, a_2, \dots, a_n\}$ and a sentence S_2 with m words $S_2 = \{b_1, b_2, \dots, b_m\}$
output: The edit distance between S_1 and S_2 $L(n, m)$

```

1  for  $i = 0$  to  $n$  do
2    |  $L(i, 0) = i$ 
3  end
4  for  $j = 0$  to  $m$  do
5    |  $L(0, j) = j$ 
6  end
7  for  $i = 1$  to  $n$  do
8    | for  $j = 1$  to  $m$  do
9      |  $L(i, j) = \min \{L(i-1, j) + 1, L(i, j-1) + 1, L(i-1, j-1) +$ 
10     |    $c(a_i, b_j)\}$ 
11    | end
12 end
13 return  $L(m, n)$ 

```

Algorithm 2: Hierarchical Clustering Algorithm

input : An integer n represents the number of initial clusters, a clusterMatrix stores the sentence vector of every cluster, an array, a queue(implemented by binary heap) stores indices and distances between two cluster

output: clustered nodes

```
1 for  $i = 1$  to  $n$  do
2   |   initiaDistance( $i, i$ ) = 0;
3   |   for  $j = i + 1$  to  $n$  do
4   |   |   initiaDistance( $i, j$ ) = distance( $i, j$ ) queue add
5   |   |   entry(initiaDistance( $i, j$ ),  $i, j, 1, 1$ )
6   |   end
7 end
8 while  $n \geq \text{miniumClusters}$  do
9   |   entry = queue.pop();
10  |   merge(entry.cluster1, entry.cluster2, entry.distance, entry.distance,
11  |   |   clusterMatrix);
12  |   for  $i = 1$  to  $n$  do
13  |   |   if  $i \neq \text{entry.cluster1}$  and  $\text{clusterMatrix}[i].\text{size} \neq 0$  then
14  |   |   |    $\text{first} = \min(\text{entry.cluster1}, i)$ ;
15  |   |   |    $\text{second} = \min(\text{entry.cluster2}, i)$ ;
16  |   |   |   newDistance = getDistance(initialDistance,
17  |   |   |   |   clusterMatrix(first), clusterMatrix(second));
18  |   |   |   queue add entry(newDistance, first, second,
19  |   |   |   |   clusterMatrix(first).size, clusterMatrix(second).size)
20  |   |   end
21  |   end
22  |    $n = n - 1$ ;
23 end
```

Algorithm 3: Node pair similarity scores pseudocode

input : A bipartite graph G_2 of node pairs

output: A hash table of similarity scores for every node pair in G_2

```
1  Create hash table node_scores;
2  for  $(a,b)$  in  $nodes(G_2)$ : do
3      if  $a==b$  then
4          | Node_scores[ $(a,b)$ ] := 1
5      else
6          | Node_scores[ $(a,b)$ ] := 0
7      end
8  end
9  stop_signal := on;
10 while stop_signal == on do
11     stop_signal := on;
12     for  $(a,b)$  in  $nodes(G_2)$  do
13         if  $a \neq b$  then
14             | {COMMENT: Let  $N(v)$  denote the neighbors of node  $v$  }
15             |  $R_k(a,b) :=$ 

$$\frac{C_1}{|N(a)||N(b)|} \sum_{i=1}^{|N(a)|} \sum_{j=1}^{|N(b)|} R_{k-1}(N_i(a), N_j(b))$$

16             | end
17             if  $R_k(a,b) \neq R_{k-1}(a,b)$  then
18                 | stop_signal := off
19             end
20             node_scores[ $(a,b)$ ] :=  $R_k(a,b)$ 
21         end
22 end
23 Return node_scores
```

Algorithm 4: Tweet Search job pseudocode

input : A hashtag H that the user searches for, bipartite graph G_2 ,
hash table `node_scores`
output: Tweets, without hashtag H , which are similar to tweets with
hashtag H

```
1 Create hash table related_hashtags;  
2 stop_signal := on;  
3 for  $(a,b)$  in hashtag_nodes(G_2) do  
4   if  $a == H$  then  
5     stop_signal := off;  
6     related_hashtags[b] :=  $R_k(a,b)$   
7   end  
8 end  
9 if stop_signal == on then  
10  | Return "No results were found."  
11 end  
12 mergesort on related_hashtags by decreasing value;  
13 Similar_hashtags := 100 elements of related_hashtags with highest value;  
14 Create array H_tweets;  
15 for  $(x,y)$  in neighbors(H) do  
16   if  $x == y$  and  $x$  not in H_tweets then  
17     | Add  $x$  to H_tweets  
18   end  
19 end  
20 Create array H_similar_tweets;  
21 for hashtag in H_similar_hashtag do  
22   for  $(x,y)$  in neighbors(hashtag) do  
23     if  $x == y$  and  $x$  not in H_tweets and  $x$  not in similar_tweets then  
24       | Add  $x$  to H_similar_tweets  
25     end  
26   end  
27 end  
28 Return H_similar_tweets
```

Algorithm 5: Hashtag Search job pseudocode

input : Hash table node_scores, input_tweet, bipartite graph G_2
output: Hashtags that not in input_tweet but are in tweets similar to input_tweet

```
1 Create hash table related_tweets;  
2 stop_signal := on;  
3 for (a,b) in tweet_nodes( $G_2$ ) do  
4   if a == input_tweet then  
5     stop_signal := off;  
6     related_tweets[b] :=  $R_k(a,b)$   
7   end  
8 end  
9 if stop_signal == on then  
10  Return "No results were found."  
11 end  
12 mergesort on related_tweets by decreasing value;  
13 Similar_tweets := 100 elements of related_tweets with highest value;  
14 Create array input_tweet_hashtags;  
15 for (x,y) in neighbors(input_tweet) do  
16   if x == y and x not in input_tweet_hashtags then  
17     Add x to input_tweet_hashtags  
18   end  
19 end  
20 Create array similar_hashtags;  
21 for tweet in similar_tweets do  
22   for (x,y) in neighbors(tweet) do  
23     if x == y and x not in input_tweet_hashtags and x not in  
       similar_hashtags then  
24       Add x to H_similar_hashtags  
25     end  
26   end  
27 end  
28 Return similar_hashtags
```

- Algorithms and Data Structures: In Algorithm 1, the major algorithm is Edit Distance in Dynamic Programming. The data structure is two-dimensional array, and the time complexity is $\mathcal{O}(mn)$. In Algorithm 2, we use Agglomerative Hierarchical Clustering Algorithm to cluster the documents. The time complexity of this algorithm is $\mathcal{O}(n^2 \log n)$

The core of Algorithm 3 is the SimRank algorithm. The core of Algorithms 4 and 5 is Mergesort. Before computing the graphs, each tweet is stored as a pair of a unique identifier and an array of its hashtags. This pair will be stored as a hash table. The graphs are represented as adjacency

lists (array of arrays). Each node is a pair, so it will be stored as a hash table. Collections of hashtags and arrays, linked to their similarity scores, are stored in hash tables. Similar tweets and hashtags are stored in sets, a type of array.

References

- [1] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In Proceedings of the 26th International Conference on Neural Information Processing Systems, NIPS'13, pages 3111-3119, USA, 2013. Curran Associates Inc.
- [2] Jeh, Glen, and Jennifer Widom. "SimRank: a measure of structural-context similarity." In Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 538-543. ACM, 2002.