Computing Large-Scale Similarities : Distributed Locality Sensitive Hashing

Abstract

Many applications from various domains such as web search, mobile browsing, and Natural Language Processing rely on finding the nearest neighbors of a given text string from a large database. Due to the very large scale of data involved (e.g., users' queries from commercial search engines), computing nearest neighbors is the best way to understand users' intents. However, it is a non-trivial task as the computational complexity grow significantly with the number of queries. To address this challenge, we exploit Locality Sensitive Hashing (a.k.a, LSH) methods and propose novel variants in a distributed computing environment (specifically, Hadoop). We identify several optimizations which improve performance, suitable for deployment in very large scale settings. The experimental results demonstrate our proposed variants of LSH achieve the robust performance with better recall compared with "vanilla" LSH, even when using the same space determined by the number of hash tables.

1 Introduction

Every day, hundreds of millions of users visit commercial search engines to pose queries on topics of their interest. Such queries are typically just a few key words intended to specify the topic that the user has in mind. To provide users with a high quality service, search engines such as Bing, Google, and Yahoo require intelligent analysis to realize users' implicit intents. The key resource that they have to help tease out what is meant is their large history of requests, in the form of large scale query logs. A key primitive in learning users' implicit intents is the computation of nearest neighbors (queries) for a user given query. Computing nearest neighbors is useful for many search-related problems on the Web and the Mobile such as finding related queries (Jones et al., 2006; Jain et al., 2011; Song et al., 2012), finding near-duplicate queries (Lee et al., 2011), spelling correction (Ahmad and Kondrak, 2005; Li et al., 2012), and diversifying search results (Song et al., 2011); and many Natural Language Processing (NLP) tasks such as paraphrasing (Petrovic et al., 2012; Ganitkevitch et al., 2013), distributional similarity (Ravichandran et al., 2005; Agirre et al., 2009; Turney and Pantel, 2010), and creating sentiment lexicons from large-scale Web data (Velikovich et al., 2010).

In this paper, we focus on the problem of finding nearest neighbors for a given query from very large scale query logs available from a commercial search engine. Given the importance of this question, it is important to design algorithms that can scale to many queries over huge logs, and allow online and offline computation. However, computing nearest neighbors of a query can be very costly. Naive solutions that involve a linear search of the set of possibilities are simply infeasible in these settings. Even though distributed computing environments such as Hadoop make it feasible to store and search large data sets in parallel, the naive pairwise computation is still infeasible. The reason is that the total amount of work performed is still huge, and simply throwing more resources at the problem is not effective. Given a log of hundreds of millions queries, most are "far" from a query of interest, and we should aim to avoid doing many "useless" comparisons that confirm that queries are indeed far from it.

In order to address the computational challenge, this paper aims to find nearest neighbors by doing a *small* number of comparisons—that is, sublinear in the dataset size—instead of brute force linear search. In addition to *small* number of comparisons, we aim to retrieve neighboring candidates with 100% precision and high recall. It is important that false positive rate (ratio of "incorrectly" identifying queries as neighbors) is penalized more severely than false negative (ratio of missing "true" neighbors).

In the case of seeking exact matches for queries, effective solutions are based on storing values in a hash table and mapping in via hash functions. The approximate generalization of this approach is the framework of Locality Sensitive Hashing, where queries are more likely to collide under the hash function if they are more alike, and less likely to collide if they are less alike. The methods we propose in this paper meet our criteria by extending Locality Sensitive Hashing (Indyk and Motwani, 1998; Charikar, 2002; Andoni and Indyk, 2006; Andoni and Indyk, 2008) in novel ways. In particular, we apply the framework within a distributed system, Hadoop, and take advantage of its distributed computing power.

Our work makes the following contributions:

1. We begin with a "vanilla" LSH algorithm based on the seminal research of Andoni and Indyk (2008). To best of our knowledge, this is the first paper that applies this algorithm to NLP applications.

2. We propose four novel variants of vanilla LSH motivated by the research on Multi-Probe LSH (Lv et al., 2007). We show that two of our variants achieve significantly better recall than the vanilla LSH by using the same number of hash tables. The main idea behind these variants is to intelligently probe multiple "nearby" buckets within a table that have a high probability of containing the nearest neighbors of a query.

3. We present a framework on Hadoop that efficiently finds nearest neighbors for a given query from a commercial large-scale query logs in sublinear time.

4. We discuss the applicability of our framework on two real-world applications: finding related queries and removing (near) duplicate queries. The algorithms presented in this paper are currently being implemented for production use within a large search provider.

2 Problem Statement

We start with user query logs C having query vectors collected from a commercial search engines over some domain (e.g. URLs); similarity between queries is to be measured using cosine between the corresponding vectors. The problem we formulate here is given a set of queries Q and similarity threshold τ, we are interested in developing a batch process to return a *small* set T of candidate neighbors from C for each query $q \in Q$ such that the followings are satisfied: 1) $T = \{l \mid s(l,q) \geq \tau, l \in C\}$, where $s(q_1, q_2)$ is a function to compute a similarity score between query feature vector q_1 and q_2 ; 2) T achieves 100% precision with "large" recall. That is, our aim is to achieve a large recall, while using a scalable efficient algorithm.

The exact brute force algorithm to solve the above problem would be to compute similarities between each $q \in Q$ and all queries in C, and return all pairs that have similarities higher than the threshold τ. Computing similarities for all pairs is computationally infeasible on a single computer even if the size of Q is of the order of few thousands and the size of C is hundreds of millions. Even in a distributed setting such as Hadoop, the resulting communication needed between machines makes this strategy impractical. This in turn highlights "linear" computational complexity is not always acceptable to a certain problem domains.

Our aim is to empirically study a set of locality sensitive hashing techniques that enable us to return a set of candidate neighbors while performing a much smaller (theoretically *sublinear*) set of comparisons. In order to tackle this scalability problem, we explore the combination of distributed computation using a map-reduce platform (Hadoop) as well as locality sensitive hashing (LSH) algorithms. We explore a few commonly known variants of LSH and suggest several novel variants that are suitable to the map-reduce platform. The methods that we propose meets the practical requirements of a real life search engine backend, and demonstrates how to use locality sensitive hashing on a distributed platform.

3 Proposed Approach

We describe a distributed Locality Sensitive Hashing framework based on map-reduce. First, we present vanilla LSH algorithm based on the seminal work of Andoni and Indyk (2008). To best of our knowledge, this is the first paper that applies this variant of locality sensitive hashing algorithms to NLP applications.

The algorithm in Andoni and Indyk (2008) improves the existing research in LSH and Point Location in Equal Balls (PLEB) (Indyk and Motwani, 1998; Charikar, 2002). PLEB was applied for noun clustering (Ravichandran et al., 2005) and speech tasks (Jansen and Van Durme, 2011; Jansen and Van Durme, 2012). Recent prior work on new variants of PLEB (Goyal et al., 2012) for distributional similarity can be seen as implementing a special case of Andoni and Indyk's LSH algorithm.

We first present four new variants of vanilla LSH algorithm motivated by the technique of Multi-Probe LSH (Lv et al., 2007). A significant drawback of vanilla LSH is that it requires a large number of hash tables in order to achieve good recall in finding nearest neighbors, making the algorithm memory intensive. The goal of Multi-probe LSH is to get significantly better recall than the vanilla LSH by using the same number of hash tables.

3.1 Vanilla LSH

The LSH algorithm relies on the existence of an family of locality sensitive hash functions. Let H be a family of hash functions mapping $\mathbb{R}^D$ to some universe S. For any two query terms p, q; we chose $h \in H$ uniformly at random; and analyze the probability that $h(p) = h(q)$. Suppose d is a distance function (cosine distance for us), $R > 0$ be a distance threshold $R > 0$ and $c > 1$ be an approximation factor. Let $P_1, P_2 \in (0, 1)$ be two probability thresholds. The family H of hash functions is called a (R, cR, P_1, P_2) locality sensitive family if it satisfies the following conditions:

1. If $d(p, q) \leq R$, then $Pr[h(p) = h(q)] \geq P_1$,
2. and if $d(p, q) \geq cR$, then $Pr[h(p) = h(q)] \leq P_2$

An LSH family is generally interesting when $P_1 > P_2$. However, the difference between P_1 and P_2 can be very small. Given a family H of hash functions with parameters (R, cR, P_1, P_2), the LSH algorithm is devised by amplifying the gap between the two probabilities P_1 and P_2 by concatenating several functions. In particular, LSH algorithm concatenates K hash functions to create a new hash function $g(\cdot)$ as: $g(q) = (h_1(q), h_2(q), \ldots, h_K(q))$. A larger value of K leads to larger gap between probabilities of collision between close neighbors (i.e. distance less than R) and neighbors that are far (i.e. distance more than cR); the corresponding probabilities being P_1^K and P_2^K respectively. This amplification ensures a high precision of the algorithm by making the probability of dissimilar queries having the same hash value very small.

Symbol	Description
N	# of query terms
D	# of features i.e. all clicked unique urls
K	# of hash functions concatenated together $g(q) = (h_1(q), h_2(q), \ldots, h_k(q)))$ to generate the index of a table
L	# of tables generated independently with $g_j(q)$ index, $\forall 1 \leq j \leq L$
F	# of bits flipped, $\forall 1 \leq j \leq L$
τ	τ threshold
Recall	fraction of similar candidates retrieved
Comparisons	Avg # of pairwise comparisons per query

Table 1: Major Notations

In order to increase the recall of the LSH algorithm, the algorithm of Andoni et al. then uses L hash tables, each constructed using a different $g_j(\cdot)$ function, where each $g_j(\cdot)$ is defined as $g_j(q) = (h_{1,j}(q), h_{2,j}(q), \ldots, h_{K,j}(q)))$; $\forall 1 \leq j \leq L$. We list the major notations of LSH in Table 1.

3.2 LSH for Cosine Similarity

In this paper, we are interested in cosine similarity, and use the LSH family defined by Charikar (2002). For two queries; $p, q \in \mathbb{R}^D$, the cosine similarity between them is $\left(\frac{p.q}{\|p\|\|q\|}\right)$. The LSH functions for cosine similarity is defined as follows: if $\alpha \in \mathbb{R}^D$ is a random vector, then a corresponding hash function h_α can be defined as $h_\alpha(p) = \text{sign}(\alpha \cdot p)$. Typically, a negative sign is represented as 0 and positive sign as 1, and indices of buckets in the hash tables (i.e. the range of each g_j) are K bit vectors. In order to create a random vector α, we exploit the intuition in

```
Preprocessing: Input is $N$ queries with their respective feature vectors.
  • Select $L$ functions $g_j$, $j = 1, 2, \ldots, L$, setting $g_j(q) = (h_{1,j}(q), h_{2,j}(q), \ldots, h_{K,j}(q))$,
    where $\{h_{i,j}, i \in [1, K], j \in [1, L]\}$ are chosen at random from the LSH family.
  • Construct $L$ hash tables, $\forall 1 \leq j \leq L$. All queries with the same $g_j$ value
    ($\forall 1 \leq j \leq L$) are placed in the same bucket.
Query: $M$ test queries. Let $q$ denote a test query.
  • For each $j = 1, 2, \ldots, L$
      – Retrieve all the queries from the bucket with $g_j(q)$ function as the index of the bucket
      – Compute cosine similarity between query q and all the retrieved queries. Return all the
        queries which have similarity affinity within a similarity threshold ($\tau$).
```

Figure 1: Locality Sensitive Hashing Algorithm

(Achlioptas, 2003; Li et al., 2006) and sample each coordinate of α from $\{-1, +1\}$ with equal probability. Practically, each coordinate of α is generated using a hash function that maps that coordinate to $\{-1, +1\}$ (this is termed "hashing trick" in (Weinberger et al., 2009)). This hashing trick is important and useful as we do not need to explicitly store the huge random projection matrix of size $D \times K \times L$.

Figure 1 describes the algorithm for both creating the data structure, as well as for querying it. In preprocessing step, the algorithm takes as input N queries along with the associated feature vectors. In our setting, each query is represented using an extremely sparse and high dimensional feature vector that is constructed as follows: for query q, we take all the webpages (urls) that any user has clicked on when querying for q. Using this representation, we then generate the L different hash values for each query q, where each such hash value is again the concatenation of K hash functions. These L hash values per query are then used to create L hash tables. Since the width of the index of each bucket is K with each coordinate being $0/1$ bit, each hash table contains at most 2^K buckets. Each query term is then placed in their respective buckets for each of the L hash tables.

In order to retrieve near neighbors, for each of the M test queries, we first retrieve all the query terms which appear in the buckets associated with each of the M test queries. Next, we compute cosine similarity between each of the retrieved terms and the input test queries and return all those queries as neighbors which are within a similarity threshold (τ).

In this work, we have implemented the above algorithm in a map-reduce setting (Hadoop). In Section 4.3, we show that the map-reduce implementation scales to hundreds of millions of queries.

3.3 Reusing Hash Functions

In Section 3.2, we showed that vanilla LSH requires $L \times K$ hash functions. However generating hash functions is computationally expensive as it takes time to read all features and evaluate hash functions over all those features to generate a single bit. To minimize the number of hash functions computations, we use a trick from Andoni and Indyk (2008) in which hash functions are reused to generate L tables. K is assumed to be even and $R \approx \sqrt{L}$. We generate $f(q) = (h_1(q), h_2(q), \ldots, h_{K/2}(q)))$ of length $k/2$. Next, we define $g(q) = (f_a, f_b)$, where $1 \leq a < b \leq R$. Using such pairing, we can thus generate $L = \frac{R(R-1)}{2}$ hash indices. This scheme requires O($K\sqrt{L}$) hash functions, instead of O(KL). For rest of this paper, we use the above trick to generate L hash tables with bucket indices of width K bits.

3.4 Multi Probe LSH

As we discussed in Section 3.3, generating hash functions can be computationally expensive. Since the memory required by the algorithm also scales linearly with L, the number of hash tables, it is desirable to have a small number of tables to reduce the memory footprint. The memory footprint of vanilla LSH is what makes it impractical for real applications. Here, we first describe four new variants of the vanilla LSH algorithm motivated by the intuition in Multi-probe LSH (Lv et al., 2007). Multi-probe LSH obtains significantly higher recall than the vanilla LSH while using the same number of hash tables. In order to achieve this, the main intuition utilized in Multi-probe LSH is that in addition to looking at the hash bucket that a test query q falls in, it is also possible to look at the neighboring buckets in order to find its near neighbor candidates. Multi-probe LSH in (Lv et al., 2007) suggests exploring the neighboring buckets in order of the Hamming distance from the bucket in which q falls. They then empirically show that these neighboring buckets contain the near neighbors with very high probability. Although Multi-probe LSH is achieves a higher recall for the same number of hash tables, it will also require more probes since it searches for multiple buckets within a table. The main advantage of searching in multiple buckets over generating more number of tables is that it takes more memory and time to generate more tables in pre-processing.

The original Multi-probe LSH algorithm is developed for Euclidean distance, the details are described in (Lv et al., 2007). However, the Euclidean distance implementation does not immediately translate to our setting of cosine similarity. For example, in generating other the list of other buckets to look into, (Lv et al., 2007) utilizes the distance of the hash value to the boundary of the other bucket—this makes sense only when the hash value is a real number and not a $0/1$ bit, as it is for us. However, utilizing the same intuition, we present four variants of Multi-probe LSH for cosine similarity:

- **Random Flip Q:** The first variant is our baseline. In this, we first compute the initial LSH of a test query q, which gives the L bucket ids. Next, we create alternate bucket ids by taking each of the L bucket ids and then creating alternate candidate buckets by flipping a set of coordinates randomly in the LSH of the test query q.

- **Random Flip B:** The second variant is another baseline similar to the previous one. Instead of just flipping the bits for only the test query, here we flip bits for both the test query and all the queries in the database.

Data	N	D
AOL-logs	0.3×10^6	0.7×10^6
Qlogs001	6×10^6	66×10^6
Qlogs010	62×10^6	464×10^6
Qlogs100	617×10^6	2.4×10^9

Table 2: Query-logs statistics

- **Distance Flip Q:** The third variant is a more intelligent version of first variant. Instead of randomly flipping some coordinates of the test query q, we select a set of coordinates based on the distance of q from the random hyperplane (hash function) that was used in to create this coordinate. The distance of the test query q from the random hyperplane is the absolute value which we get before applying the sign function on it (see Section 3.2), i.e. is $\mathrm{abs}(\alpha \cdot q)$ if the random hyperplane is α.

- **Distance Flip B:** The fourth version is similar to third one, however here we flip bits for both the test query and for the queries in the database (i.e. this is the intelligent version of the second baseline).

4 Experiments

We evaluate our distributed large-scale approximate near neighbor framework by conducting several experiments on publicly available query logs as well as a large-scale query log collected from a commercial search engine.

4.1 Data

The public dataset that we demonstrate results on is adapted from the query logs of the AOL search engine (Pass et al., 2006) (AOL-logs dataset). Moreover, we show results on a large query log (hundreds of millions of queries) sampled from a commercial search engine. We started by collecting a dataset over multiple days, and containing $N = 600$ million unique queries: this does not represent an exhaustive set of queries posed to the search engine. We then created multiple datasets from this corpus by subsampling it at various rates. Qlogs001 represents a 1% sample of queries from the above corpus, Qlogs010 represents a 10% sample and Qlogs100 represents the entire corpus. For each query, a high

τ	AOL-logs		Qlogs001	
	Comparisons	Recall	Comparisons	Recall
0.7	57	.63	1052	.67
0.8		.84		.81
0.9		.98		.96

Table 3: Varying τ with fixed $K = 16$ and $L = 10$ on AOL-logs and Qlogs001.

L	AOL-logs		Qlogs001	
	Comparisons	Recall	Comparisons	Recall
1	7	.28	106	.36
10	57	.63	1052	.67
28	152	.77	2908	.78
55	297	.89	5648	.84

Table 4: Varying L with fixed $K = 16$ on AOL-logs and Qlogs001 with $\tau = 0.7$.

dimensional and sparse, feature vector was created over the domain of urls (the size of the domain is a few billion). The feature vector of a query q contains the webpages (urls) that have received a user click for this search query q. The weight of an url feature for a query q depends on the click through rate of this url for the query q. As a pre-processing step, we remove all the queries that have less than or equal to five clicked urls. Table 2 summarizes the statistics of our query-log datasets.

Test Data : We conduct all the experiments using 2000 random queries sampled from the query logs as the test set of queries. For evaluation, we compute the true similar candidates for all the 2000 test queries by calculating cosine similarity between each test query and all the queries in the rest of the dataset. For most of the experiments, we set the similarity threshold $\tau = 0.7$, meaning that the algorithm needs to retrieve candidates that have cosine similarity larger than or equal to 0.7.

4.2 Evaluation Metrics

We use two metrics for evaluation: recall and number of comparisons. Recall of an LSH algorithm is the fraction of *true* similar candidates (found using candidates returned by computing exact cosine similarity) that is retrieved by the LSH algorithm. The number of comparisons performed by an algorithm is computed as the average number of pairwise comparisons that is done per test query. The goal of this paper is to maximize recall and minimize the number of comparisons.

4.3 Evaluating Vanilla LSH

In the first experiment, we vary the similarity threshold parameter τ to be in $\{0.7, 0.8, 0.9\}$ while fixing $K = 16$ and $L = 10$ for the AOL-logs and Qlogs001 datasets. Table 3 shows that as expected, finding near-duplicates, when $\tau = 0.9$, is easier than finding nearest neighbors when $\tau = 0.7$. For, rest of this paper, we fix $\tau = 0.7$ as we are interested in both near-duplicates and nearest-neighbors for our test queries.

In the second experiment, we vary L to be in $\{1, 10, 28, 55\}$ while fixing $K = 16$ on the AOL-logs and Qlogs001 datasets. Recall that L denotes the number of hash tables and K is the width of the index of the buckets in the table. Increasing K results in increase of the precision by reducing the false positive candidates, but the L also need to be correspondingly increased in order the maintain a good recall (i.e. reduce false negatives). Table 4 shows that increasing L leads to better recall, but at the expense of performing more comparisons on both the datasets. In addition, having a large L means generating large number of random projection bits and hash tables which is both time and memory intensive. Hence, we fix $L = 10$, which leads to a reasonable recall with a tolerable number of comparisons.

In the third experiment, we vary K to be in $\{4, 8, 16\}$ while fixing $L = 10$ on AOL-logs and Qlogs001 datasets. As expected, Table 5 shows that increasing K reduces the number of comparisons and worsens recall on both the datasets. This is intuitive as the larger value of K leads to larger gap between probabilities of collision between close queries and far queries (see Section 3.1). Hence, we fix $K = 16$ to have few number of comparisons.

In the fourth experiment, we fix $L = 10$ and $K = 16$, values that were found to be reasonable using the previous experiments, and then increase the size of training data. Table 6 demonstrates that as we increase the training data size, the number of comparisons done by the algorithm also increase. This result indicates that K needs to be tuned with respect to a specific dataset, as a larger K will reduce the

K	AOL-logs		Qlogs001	
	Comparisons	Recall	Comparisons	Recall
4	112,347	.98	2,29,2670	.96
8	11,008	.90	221,132	.88
16	57	.63	1,052	.67

Table 5: Varying K with fixed $L = 10$ on AOL-logs with $\tau = 0.7$.

Data	Comparisons	Recall
AOL-logs	57	.63
Qlogs001	1,052	.67
Qlogs010	10,515	.64
Qlogs100	105,126	-

Table 6: Fixed $K = 16$ and $L = 10$ on different sized datasets with $\tau = 0.7$.

probability of dissimilar queries falling within the same bucket. K and L can be tuned by randomly sampling small set of queries. In this paper, we randomly select 2000 queries to tune parameter K.

Table 7, containing the result of our fifth experiment, shows the best K and L parameter settings on datasets with different sizes.[1] On our biggest dataset of 600 million queries, we set $K = 24$ and $L = 10$. These parameter settings require on an average only 464 comparisons to find approximate nearest neighbors compared to exact cosine similarity that involves brute force search over all 600 million queries in the dataset.

4.4 Evaluating Multi-Probe LSH

In the first experiment, we compare flipping the bits in query only. We evaluate two approaches: Random Flip Q and Distance Flip Q. We can make sev-

Data	Comparisons	Recall
AOL-logs ($K = 16$)	57	.63
Qlogs001 ($K = 16$)	1,052	.67
Qlogs010 ($K = 20$)	695	.53
Qlogs100 ($K = 24$)	464	-

Table 7: Best parameter settings (Based on minimizing number of comparisons and maximizing recall) of K with fixed $L = 10$ on different sized datasets.

Method	Random Flip Q		Distance Flip Q	
F	Comparisons	Recall	Comparisons	Recall
1	108	.65	106	.72
2	159	.66	*155*	**.75**
5	311	.70	303	.79
10	557	.75	552	.81
16	839	.82	839	.82

Table 8: Flipping the bits in the query only with $K = 16$ and $L = 10$ on AOL-logs with $\tau = 0.7$.

Method	Random Flip B		Distance Flip B	
F	Comparisons	Recall	Comparisons	Recall
1	204	.71	192	.80
2	433	.73	*405*	**.86**
5	1557	.86	1475	.93
10	4138	.94	4059	.96
16	5922	.96	5922	.96

Table 9: Flipping the bits in both the query and the database with $K = 16$ and $L = 10$ on AOL-logs with $\tau = 0.7$.

Method	Distance Flip Q		Distance Flip B	
Data	Comps.	Recall	Comps.	Recall
AOL-logs ($K = 16$)	155	.75	405	.86
Qlogs001 ($K = 16$)	2980	.76	7904	.84
Qlogs010 ($K = 20$)	1954	.64	5242	.72
Qlogs100 ($K = 24$)	1280	-	3427	-

Table 10: Best parameter settings (Based on minimizing number of comparisons and maximizing recall) of K with fixed $L = 10$ and $F = 2$ on different sized datasets with $\tau = 0.7$.

[1] Recall on Qlogs100 the precision/recall cannot be computed, as it was computationally intensive to find exact similar neighbors.

how lbs in a ton	**coldwell banker baileys harbor**	**michaels**	**trumbull ct weather**
how much lbs is a ton	coldwell banker sturgeon bay wi	maichaels	trumbull ct weather forecast
number of pounds in a ton	coldwell banker door county	machaels	weather in trumbull ct
how many lb are in a ton	door county wi mls listings	mechaels	weather in trumbull ct 06611
How many pounds are in a ton?	door county realtors sturgeon bay	miachaels	trumbull weather forecast
how many pounds in a ton	DOOR CTY REAL	michaeils	trumbull ct 06611
1 short ton equals how many pounds	door county coldwell banker	michaelos	trumbull weather ct
how many lbs in a ton?	door realty	michaeks	trumbull ct weather report
how many pounds in a ton?	coldwell banker door county horizons	michaeels	trumbull connecticut weather
How many pounds are in a ton	door county coldwell banker real estate	michaelas	weather 06611
how many lb in a ton	coldwell banker door county wisconsin	michae;ls	weather trumbull ct

Table 11: Sample 10 similar neighbors returned by Distance Flip B with $L = 10$, $K = 24$, and $F = 2$ on Qlogs100 dataset.

eral observations from Table 8: 1) As expected, increasing the number of flips improve recall at expense of more comparisons for both Distance Flip Q and Random Flip Q. 2) The last row of the Table 8 shows that once we flip all the K bits ($F = 16$), Distance Flip Q and Random Flip Q converge to the same algorithm. 3) Our results show that Distance Flip Q has significantly better recall than Random Flip Q with similar number of comparisons. In second row of the table with $F = 2$, Distance Flip Q has nine points better recall than Random Flip Q.

In the second experiment, we compare flipping the bits in both query and the database. We can make similar observations from Table 9 as made in the first experiment. In the second row of the Table with $F = 2$, Distance Flip B has thirteen points better recall than Random Flip B with similar number of comparisons. Comparing across second row of Table 8 and 9 shows that flipping the bits in both query and the database has better recall at the expense of more comparisons. This is expected as flipping both means that we can find queries at distance two (one flip in query, one flip in database), hence more queries in each table when we probe. Note, we also compared distance based flipping with random flipping on different sized data-sets, and found that distance based flipping is always significantly better in terms of recall as compared to random flipping.[2]

We select $F = 2$ as the best parameter setting with goal of maximizing recall by restricting comparisons to minimum. For better recall at expense of more comparisons, $F = 5$ can also be selected. However, results in Table 8 and 9 indicate that $F > 5$ does *not* increase recall significantly and at the same time leads to more number of comparisons.

In the third experiment, we show the results of both variants of distance based Multi Probe that is Distance Flip Q and Distance Flip B on different sized datasets. Table 10 shows the results with parameter $L = 10$, $F = 2$, and data-size dependent K (same settings of K for different sized datasets as used in the last experiment of Section 4.3). As observed in the last experiment, flipping bits in both query and the database is significantly better in terms of recall with more number of comparisons. The second and third row of the table respectively shows that flipping bits in both query and the database has eight points better recall on both Qlogs001 and Qlogs010 datasets. With the goal of maximizing recall with some extra comparisons, we select Distance Flip B as our final algorithm. Distance Flip B maximizes recall with small number of tables and comparisons. On our entire corpus (Qlogs100) with hundreds of millions of queries, Distance Flip B only requires $3,427$ comparisons compared to hundreds of millions of comparisons by exact brute force algorithm. Distance Flip B returns 9 neighbors on average per given query; averaged over 2000 random test queries.[3]

In the fourth experiment, Table 11 shows the qualitative results for some arbitrary queries. These results are found by applying our system (Distance Flip B with parameters $L = 10$, $K = 24$, and $F = 2$) on Qlogs100. The second column in Table 11 shows that the returned approximate similar neighbors can be useful in finding related queries (Jones et al., 2006; Jain et al., 2011). The third col-

[2] Due to limited space, we omit those results.

[3] As many queries can be long, hence such queries only have few neighbors.

umn shows an example, where we can find several popular spelling errors. The first and last column show examples of near-duplicate queries (Lee et al., 2011).

5 Related Work

There has been many papers in last decade that have focused on approximate, streaming, and randomized algorithms on many NLP problems. Prior work on LSH for noun clustering (Ravichandran et al., 2005) applied original version of LSH based on Point Location in Equal Balls (PLEB) (Indyk and Motwani, 1998; Charikar, 2002). The disadvantage of original LSH algorithm is that it involves generating large number of permutations ($L = 1000$, generating permutation is similar to generating more tables) and sorting bit vectors of large width ($K = 3000$). To address that issue, Goyal et al. (Goyal et al., 2012) proposed a new variant of PLEB that is faster than the original LSH algorithm but that still requires large number of permutations ($L = 1000$). In addition, their work can be seen as an implementing a special case of Andoni and Indyk's LSH algorithm.

The next major difference between our research and existing work is that the existing work deals with approximating cosine similarity by Hamming distance (Ravichandran et al., 2005; Van Durme and Lall, 2010; Van Durme and Lall, 2011; Goyal et al., 2012). In that problem setting, the goal is to minimize both false positives and negatives. In our work, we focus on minimizing false negatives with zero tolerance for false positives.

(Zadeh and Goel, 2013) developed a distributed version of the LSH algorithm, for the Jaccard distance metric, that scales to very large text corpora by virtue of being implemented on a map-reduce, and by using clever sampling schemes in order to reduce the communication cost. Our work is for the cosine similarity metric, and uses bit flipping in a distributed manner to reduce the number of hash tables in LSH and hence the memory.

6 Discussion

In this section, we highlight several applications that can take significant advantage of the approximate Distance Flip B algorithm presented in this paper. One of the interesting applications of the near-neighbor finding is to understand specific intents behind the user query. Given a user's query, Bing, Google, and Yahoo often delivers direct display results that summarize expected contents of the query. For instance, when a query “f stock price” is issued to search engines, the quick summary of the stock quote with a chart is delivered to the user as the part of the search engine result page. Such direct display results are expected to reduce the number of unnecessary clicks by providing the user with the appropriate content early on. However, when the query ”f today closing price” is issued to search engines, the three major search engines fail to deliver the same direct display experience to the user query, although its query intent is strongly related to “f stock price”. By employing an algorithm similar to Distance Flip B, we can build a synonym database, which will help trigger the same direct display among related queries.

Yet another application is removing duplicated instances in a set of suggested results. When a query set is retrieved from a repository and presented to users, it is important to remove similar queries from the set so that the user is not distracted by duplicated results. Given a set of we can apply Distance Flip B algorithm to build a lookup table of near-duplicates in order to find the “duplicated query terms” efficiently. As “duplicateness” of among query terms typically requires “higher” degree of similarity (relatively easier problem) than “relatedness”, we can tune parameters (K, L, F) based on a specific τ (e.g $\tau = 0.9$) from training samples. The fourth column in Table 11 illustrates several duplicate examples– “trumbull weather ct” and “weather in trumbull ct”.

7 Conclusion

To the best of our knowledge, this is the first paper that applies the vanilla LSH algorithm of Andoni et al. to the NLP domain. We also proposed four novel variants of LSH that are aimed to reduce the number of hash tables used. Two of our variants achieve significantly better recall than the vanilla LSH by using the same number of hash tables. We also present a framework on Hadoop that efficiently finds nearest neighbors for a given query from a commercial large-scale query logs (consisting of hundreds of millions of queries) in sublinear time.

References

Dimitris Achlioptas. 2003. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687.

Eneko Agirre, Enrique Alfonseca, Keith Hall, Jana Kravalova, Marius Paşca, and Aitor Soroa. 2009. A study on similarity and relatedness using distributional and wordnet-based approaches. In *NAACL '09: Proceedings of HLT-NAACL*.

Farooq Ahmad and Grzegorz Kondrak. 2005. Learning a spelling error model from search query logs. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Alexandr Andoni and Piotr Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions.

Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*.

Moses S. Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, STOC.

Juri Ganitkevitch, Benjamin Van Durme, and Chris Callison-Burch. 2013. Ppdb: The paraphrase database. In *North American Chapter of the Association for Computational Linguistics (NAACL)*.

Amit Goyal, Hal Daumé III, and Raul Guerra. 2012. Fast large-scale approximate graph construction for NLP. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and and Computational Natural Language Learning*.

Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC.

Alpa Jain, Umut Ozertem, and Emre Velipasaoglu. 2011. Synthesizing high utility suggestions for rare web search queries. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM.

Aren Jansen and Benjamin Van Durme. 2011. Efficient spoken term discovery using randomized algorithms. In *Proceedings of the Automatic Speech Recognition and Understanding(ASRU)*.

Aren Jansen and Benjamin Van Durme. 2012. Indexing raw acoustic features for scalable zero resource search. In *Proceedings of the InterSpeech*.

Rosie Jones, Benjamin Rey, Omid Madani, and Wiley Greiner. 2006. Generating query substitutions. In *Proceedings of the 15th International Conference on World Wide Web (WWW)*. ACM.

Chi Hoon Lee, Alpa Jain, and Larry Lai. 2011. Assisting web search users by destination reachability. In *Proceedings of the ACM International Conference on Information and Knowledge Management*.

Ping Li, Trevor J. Hastie, and Kenneth W. Church. 2006. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 287–296. ACM.

Yanen Li, Huizhong Duan, and ChengXiang Zhai. 2012. A generalized hidden markov model with discriminative training for query spelling correction. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*.

Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases (VLDB)*.

Greg Pass, Abdur Chowdhury, and Cayley Torgeson. 2006. A picture of search. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*. ACM Press.

Sasa Petrovic, Miles Osborne, and Victor Lavrenko. 2012. Using paraphrases for improving first story detection in news and twitter. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Deepak Ravichandran, Patrick Pantel, and Eduard Hovy. 2005. Randomized algorithms and NLP: using locality sensitive hash function for high speed noun clustering. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*.

Yang Song, Dengyong Zhou, and Li-wei He. 2011. Post-ranking query suggestion by diversifying search results. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. ACM.

Yang Song, Dengyong Zhou, and Li-wei He. 2012. Query suggestion by constructing term-transition graphs. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining (WSDM)*. ACM.

Peter D. Turney and Patrick Pantel. 2010. From frequency to meaning: Vector space models of semantics. *JOURNAL OF ARTIFICIAL INTELLIGENCE RESEARCH*, 37:141.

Benjamin Van Durme and Ashwin Lall. 2010. Online generation of locality sensitive hash signatures. In *Proceedings of the ACL 2010 Conference Short Papers*.

Benjamin Van Durme and Ashwin Lall. 2011. Efficient online locality sensitive hashing via reservoir counting. In *Proceedings of the ACL 2011 Conference Short Papers*.

Leonid Velikovich, Sasha Blair-Goldensohn, Kerry Hannan, and Ryan McDonald. 2010. The viability of web-derived polarity lexicons. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1113–1120, New York, NY, USA. ACM.

Reza Bosagh Zadeh and Ashish Goel. 2013. Dimension independent similarity computation. *The Journal of Machine Learning Research*, 14(1):1605–1626.