CS 246: Problem Set 1

Tony Hyun Kim

January 26, 2012

1 MapReduce

1.1 Friend Recommender

The source code of my FriendRecommend system can be found in the attachment to the assignment. My algorithm relies on a chain of two Map-Reduce steps where:

- `Map1`: For a user i with friends $\{f_j\}$, emit all unique pairs (f_j, f_k) that are connected through user i in the (key, value) form $((f_j, f_k), i)$.
- `Reduce1`: For each mediated connection (of length 2) between users (j, k), count the number of shared friends M_{jk}. Note: the pair (j, k) is filtered in this step if there is a direct connection between j and k by using a special flag from `Map1`.
- `Map2`: The idea is to re-group the previous output by user ID. For each input $((j, k), M_{jk})$, emit $(j, (M_{jk}, k))$ and $(k, (M_{jk}, j))$.
- `Reduce2`: For each user j, iterate through potential friend candidates $\{(M_{jk}, k)\}$. Here, I implemented a priority queue that keeps track of the top $N = 10$ candidates, sorted in decreasing order by M_{jk}.

List of 10 pairs of users that are not directly connected but have the most common friends. Note that this information emerges nicely from the intermediate output of the above algorithm. I use a short Matlab script to scan for the largest `NumberOfCommonFriends`.

```
<User1> <User2> <NumberOfCommonFriends>
18739 18740 100
31506 31530 99
31492 31511 96
31511 31556 96
31519 31554 96
31519 31568 96
31533 31559 96
31555 31560 96
31492 31556 95
31503 31537 95
```

1.2 Finding paths of length 5 in an undirected graph

The following MapReduce algorithm will perform a join operation on the graph paths provided by two input files here identified by `f1` and `f2`. Beginning with a list of first-degree edges, we will need to perform multiple passes of the following pseudocode to determine all paths of length $L = 5$. I use Matlab-based indexing.

```
Map(
  n   // A list of nodes in a single path (each line of input files).
  f   // File identifier is 'f1' if path comes from file 1; 'f2' if from file 2.
  )
  emit(n[1],  (f,n[2:end]))                 // Path in original order
  emit(n[end],(f,reverse(n[1:(end-1)])))    // Path in reverse order

Reduce(
  key // Index of first node in path
  vs  // (File identifier, rest of path)
  )
  for i = 1:(vs.len-1)
    for j = (i+1):vs.len
      if(vs[i][1] != vs[j][1]) // Join segments only from different files
        jpath = [reverse(vs[j][2]) key vs[i][2]]
        if(!duplicates(jpath)) // Filter out duplicate nodes in 'jpath' (i.e. loops)
              write(jpath[1],jpath[2:end]) // Write 'jpath' to output file
```

1.3 Paths of length L

Let the files `f1` and `f2` contain paths of lengths l_1 and l_2 respectively. A single MapReduce step (as outlined above) produces an output that contains the paths of length $l_1 + l_2$. This fact allows us to determine the minimum number of MapReduce jobs required to compute all paths of length L.

We begin with a file `f1` that contains paths of length 1. In the first iteration of the MapReduce algorithm, we feed two copies of `f1`, and obtain a file `f2` that contains paths of length 2. In the subsequent MapReduce step, we may choose to join `f1` and `f2`, or `f2` and `f1` yielding edges with lengths 3 or 4 respectively. In general, subsequent steps can join any pair of intermediate paths that have been computed up to that point.

If the desired edge length is a power of 2, it is clear that a minimum of $m = \log_2(L)$ steps are required. If L is not a power of two, the following code snippet computes the required number of steps m:

```
MinimumSteps(L)
  m = floor(log2(L))
  L = L - 2^m
  while(L!=0)
    k = floor(log2(L))
    L = L - 2^k
    m = m + 1
  return m
```

For example, it takes $m = 3$ MapReduce steps to compute $L = 5$.

1.4 Matrix multiplication AA^T

Let the matrix A be divided into blocks with n blocks along the rows and m blocks along the column. Denote by $A_{I,J}$ the (I, J)-th sub-block. We wish to compute

$$C = A \cdot A^T \quad \Longleftrightarrow \quad C_{I,K} = \sum_J A_{I,J} \cdot (A_{K,J})^T, \tag{1}$$

where $(A_{K,J})^T$ indicates the (K, J)-th sub-block of the original matrix A, which is followed by a transpose.

We will compute $C = A \cdot A^T$ by the following three MapReduce steps. Note that the first step simply parses the matrix A into sub-blocks and may be omitted if the input file is properly formatted along the block structure. The following two steps are conceptually similar to the two-step matrix multiplication algorithm discussed in the course textbook in Section 2.3.10.

```
Map1(
    i,j,aij // An element in the original matrix A
    n,m     // Block counts
    )
    [I J i' j'] = GlobalToBlockIndex(i,j,n,m) // Block index I,J; Sub-block index i',j'
    emit((I,J),(i',j',aij))

Reduce1(key,vs)
    emit(key,vs) // Identity function

Map2(
    I,J,AIJ // Block index I,J and the corresponding block matrix
    )
    for k = 1:n
        emit((I,J,k),('A',AIJ))
    AIJT = AIJ' // Take the transpose of AIJ.
    for k = 1:n
        emit((k,J,I),('At',AIJT))

Reduce2(key,(A AT)) // Order of A and A' may be reversed. Order appropriately
    emit(key,A*AT)  // Sub-block multiplication possible in a single node

Map3(
    (I,J,K),(AAT)
    )
    emit((I,K),AAT)

Reduce3(key,vs)
    emit(key,sum(vs)) // Result is the (I,K)-th block of C = A*A'
```

1.5 Matrix multiplication tradeoffs

First, we note that given n blocks along the rows and m blocks along the columns, our block indices (I, J) ranges: $I = 1, 2, \cdots, n$ and $J = 1, 2, \cdots, m$. With this in mind, we can compute the number of mappers

and reducers in the three steps as a function of n and m.

- `Map1`: Not applicable. One mapper per nonzero entry of A assuming standard matrix formatting.
- `Red1`: nm reducers. One for each matrix sub-block.
- `Map2`: nm reducers. One for each matrix sub-block.
- `Red2`: n^2m reducers. Each sub-block $A_{I,J}$ is multiplied n times, and there are nm sub-blocks in all.
- `Map3`: Same as `Red2`; n^2m mappers.
- `Red3`: nm reducers. One for each sub-block of output.

Each element of the matrix A is copied n times over the network. This occurs in the second MapReduce step, where the matrix sub-block is duplicated n times to separate nodes.

With nm sub-blocks, each block contains N^2/nm entries, where N is the size of the original matrix A. Since my block multiplication algorithm never requires the local storage of more than two sub-blocks (except for `Red3` which must perform a single-pass accumulation over the partial multiplication results), the local memory requirement scales as $1/nm$.

2 Association rules

2.1 A-priori algorithm

I have implemented two Java programs that generate association rules for item sets of size 2 and 3, namely:

- `FrequentItemsets`: is a three-pass MapReduce program that generates the supports for frequent singletons, pairs and triplets.
- `AssociationRules`: takes the output of `FrequentItemsets` (*i.e.* the supports for singletons, pairs and triplets) and generates all possible association rules and their confidences.

A list of top five rules in decreasing order of confidence for item sets of size 2 and 3:

```
wilt   --> thou (conf = 1.0000)
shalt  --> thou (0.9992)
mayest --> thou (0.9907)
didst  --> thou (0.9906)
art    --> thou (0.9867)
hast,thine  --> thou (1.0000)
shalt,thine --> thou (1.0000)
go,shalt    --> thou (1.0000)
out,shalt   --> thou (1.0000)
do,shalt    --> thou (1.0000)
```

The confidence is dominated by association rules that involve `thou`. This is clearly an artifact of the fact that the token `thou` occurs with high support (independent of other words).

2.2 Alternate evaluations of association rules

2.2.1 Lift measure

List the top 5 rules in decreasing order of the lift measure for item sets of size 2 and 3. Note that the lift measure is symmetric in A and B, *i.e.* $\text{lift}(A \to B) = \text{lift}(B \to A)$. In the following list, I present 10 distinct A, B pairs. Each entry should be interpreted as both $A \to B$ and $B \to A$.

```
book  <--> written   (lift = 67.6397)
about <--> round     (50.9435)
gold  <--> silver    (50.1028)
burnt <--> offerings (47.4101)
cut   <--> off       (45.4186)
spake  <--> moses,saying (40.4831)
christ <--> jesus,lord   (35.3443)
christ <--> god,jesus    (34.1711)
thus   <--> israel,saith (30.4210)
jesus  <--> christ,lord  (29.8515)
```

2.2.2 Conviction measure

An obvious disadvantage of the lift measure is that it is symmetric in A and B which precludes any causal interpretation between A and B. On the other hand, the conviction measure defined as

$$\text{conv}(A \to B) = \frac{1 - S(B)}{1 - \text{conf}(A \to B)} = \frac{1 - S(B)}{1 - \frac{S(A \cup B)}{S(A)}} \tag{2}$$

is clearly asymmetric in A and B.

Additionally, the conviction measure penalizes the rule $A \to B$ if the support for B is independently high [through the numerator $1 - S(B)$]. Hence the conviction measure is not arbitrarily inflated if the set B simply enjoys large support in the given dataset as was the case for the confidence measure.

2.3 Relationship between rules

2.3.1 $i_1 \to i_3$

This potential rule certainly satisfies the support requirement, since $\{i_1, i_3\}$ is a subset of $\{i_1, i_2, i_3\}$. However, it **may or may not** have the threshold confidence.

2.3.2 $i_1, i_2, i_4 \to i_3$

This **may or may not** be an association rule. There is no guarantee that the itemset $\{i_1, i_2, i_3, i_4\}$ has the necessary support.

2.3.3 $i_3 \to i_5, i_6$

This **may or may not** be an association rule. While $\{i_3, i_5, i_6\}$ has the necessary support (it is a subset of $\{i_3, i_4, i_5, i_6\}$), it is not guaranteed that the proposed rule has sufficient confidence.

2.3.4 $i_3, i_4 \to i_5$

Yes, this proposed rule **must** be an association rule. The itemset i_3, i_4, i_5 occurs with sufficient support since it is a subset of i_3, i_4, i_5, i_6. Furthermore, $\text{conf}(i_3, i_4 \to i_5) \geq \text{conf}(i_3, i_4 \to i_5, i_6)$ so the confidence threshold is also satisfied.

2.3.5 $i_2, i_5, i_7 \to i_6$

This **may or may not** be an association rule depending on why $i_2, i_5 \to i_6, i_7$ is not an association rule. If the latter fails because of insufficient support, then the proposed rule will also fail. However, if the latter only has insufficient confidence, then it is possible (though not guaranteed) that $i_2, i_5, i_7 \to i_6$ will exceed the threshold confidence.

3 Locality sensitive hashing

3.1 Necessary condition

We wish to show that for a similarity function $\text{sim}(x, y)$ to possess a locality-sensitive hashing scheme, the function $d(x, y) = 1 - \text{sim}(x, y)$ must satisfy the triangle inequality:

$$d(x, y) + d(y, z) \geq d(x, z) \tag{3}$$

for all x, y and z.

We begin with a probabilistic interpretation of the function $d(x, y)$:

$$\begin{aligned} d(x, y) &= 1 - \text{sim}(x, y) \\ &= 1 - \mathbb{P}\left[h(x) = h(y)\right] \\ &= \mathbb{P}\left[h(x) \neq h(y)\right]. \end{aligned} \tag{4}$$

We thus see that Eq. 3 is equivalent to the following statements:

$$\mathbb{P}[h(x) \neq h(y)] + \mathbb{P}[h(y) \neq h(z)] \geq \mathbb{P}[h(x) \neq h(z)] \tag{5}$$

$$\mathbb{P}(A) + \mathbb{P}(B) \geq \mathbb{P}(C) \tag{6}$$

where we have defined the events A, B and C as:

$$A = [h(x) \neq h(y)], \qquad B = [h(y) \neq h(z)], \qquad C = [h(x) \neq h(z)]. \tag{7}$$

Let us enumerate the total probability using the events A, B and C:

A	B	C	$\mathbb{P}$
0	0	0	$p_0 = \mathbb{P}(\bar{A} \cap \bar{B} \cap \bar{C})$
0	0	1	$p_1 = \mathbb{P}(\bar{A} \cap \bar{B} \cap C) = 0$
0	1	0	$p_2 = \mathbb{P}(\bar{A} \cap B \cap \bar{C}) = 0$
0	1	1	$p_3 = \mathbb{P}(\bar{A} \cap B \cap C)$
1	0	0	$p_4 = \mathbb{P}(A \cap \bar{B} \cap \bar{C}) = 0$
1	0	1	$p_5 = \mathbb{P}(A \cap \bar{B} \cap C)$
1	1	0	$p_6 = \mathbb{P}(A \cap B \cap \bar{C})$
1	1	1	$p_7 = \mathbb{P}(A \cap B \cap C)$

Note that the probabilities p_1, p_2 and p_4 are identically zero, since their associated event (the appropriate intersection of A, B and C) is the null event. For instance, the event $\bar{A} \cap \bar{B} \cap C$ requires simultaneously $h(x) = h(y)$, $h(y) \neq h(z)$ and $h(x) = h(z)$ which is clearly impossible.

From the table, we then find:

$$\begin{aligned} \mathbb{P}(A) &= p_5 + p_6 + p_7 \\ \mathbb{P}(B) &= p_3 + p_6 + p_7 \\ \mathbb{P}(C) &= p_3 + p_5 + p_7 \end{aligned}$$

from which the desired inequality of Eq. 6 immediately follows.

3.2 No locality-sensitive hashing scheme for Overlap similiarity

Let $A = \{x\}$, $B = \{x, y\}$ and $C = \{y\}$. It then follows that:

$$\begin{aligned} \mathrm{sim}_{\mathrm{Over}}(A, B) &= 1 \quad \rightarrow \quad d(A, B) = 0 \\ \mathrm{sim}_{\mathrm{Over}}(B, C) &= 1 \quad \rightarrow \quad d(B, C) = 0 \\ \mathrm{sim}_{\mathrm{Over}}(A, C) &= 0 \quad \rightarrow \quad d(A, C) = 1 \end{aligned}$$

Hence the overlap similiarity metric does not satisfy the triangle inequality $d(A, B) + d(B, C) \geq d(A, C)$ and hence cannot possess a locality-sensitive hashing scheme.

3.3 No locality-sensitive hashing scheme for Dice similiarity

Again consider $A = \{x\}$, $B = \{x, y\}$ and $C = \{y\}$. We have

$$\begin{aligned} \mathrm{sim}_{\mathrm{Dice}}(A, B) &= 2/3 \quad \rightarrow \quad d(A, B) = 1/3 \\ \mathrm{sim}_{\mathrm{Dice}}(B, C) &= 2/3 \quad \rightarrow \quad d(B, C) = 1/3 \\ \mathrm{sim}_{\mathrm{Dice}}(A, C) &= 0/1 \quad \rightarrow \quad d(A, C) = 1 \end{aligned}$$

which again fails the triangle inequality for d.

4 LSH for approximate near neighbor search

The "trick" to this problem is simply to be careful with notation, and to recall the algebraic properties of the logarithm function. The basic facts are as follows:

1. The dataset $\mathcal{A}$ is a set of n points in a metric space with distance measure d.
   (a) Define $z \in \mathcal{A}$ to be a specified "query point."
   (b) Assuming there exists a point $x \in \mathcal{A}$ such that $d(x, z) \leq \lambda$, our goal is to retrieve a point $x' \in \mathcal{A}$ with $d(x', z) \leq c\lambda$.
2. The family of hash functions $\mathcal{H}$ is $(\lambda, c\lambda, p_1, p_2)$-sensitive.
3. The family $\mathcal{G}$, formed as a k-way `AND` of the elements of $\mathcal{H}$, is then $(\lambda, c\lambda, p_1^k, p_2^k)$-sensitive.
   (a) With $k = \log_{1/p_2} n$, we may simplify p_2^k as follows: $p_2^k = (1/p_2)^{-\log_{1/p_2} n} = 1/n$.
4. Finally, we take $L = n^\rho$ random members $g_1, g_2, \cdots, g_L$ of $\mathcal{G}$ where $\rho = \frac{\log 1/p_1}{\log 1/p2}$ (or equivalently $1/p_1 = (1/p_2)^\rho$), and hash all the points of $\mathcal{A}$ using all g_i's.

4.1 An upper-bound on false positives

Let

- $W_j = \{x \in \mathcal{A} \mid g_j(x) = g_j(z)\}$, *i.e.* the subset of $\mathcal{A}$ that map under g_j to the same bucket as $g_j(z)$.
- $T = \{x \in \mathcal{A} \mid d(x, z) > c\lambda\}$, *i.e.* the set of points that do not match our search criterion. Ideally, the elements of T should not share any hash bucket with z.

We wish to show that

$$\mathbb{P}\left[\sum_{j=1}^{L} |T \cap W_j| > 3L\right] < \frac{1}{3} \tag{8}$$

i.e. the probability that we have more than $3L$ false positives in our LSH scheme (the elements of T that did in fact map to one of the buckets $g_i(z)$) is at most 1/3.

Consider the random variable

$$\sum_{j=1}^{L} |T \cap W_j| = L \cdot |T \cap W_1| \tag{9}$$

where we have used the symmetry of the problem with respect to the g_j's.

Now, let us examine the random variable $|T \cap W_1|$. Let $x \in T$ be fixed. Since $\mathcal{G}$ is $(\lambda, c\lambda, p_1^k, 1/n)$-sensitive and $d(x, z) > c\lambda$, the probability that $x \in W_1$ is at most $1/n$. It then follows that $|T \cap W_1|$ is a binomial distribution with a mean that is bounded as: $\mathbb{E}(|T \cap W_1|) = np < n \cdot 1/n = 1$. Using this result, Eq. 9 yields that $\mathbb{E}(\sum_{j=1}^{L} |T \cap W_j|) < L$.

We then apply the Markov inequality to Eq. 8

$$\mathbb{P}\left[\sum_{j=1}^{L} |T \cap W_j| > 3L\right] < \mathbb{P}\left[\sum_{j=1}^{L} |T \cap W_j| > 3 \cdot \mathbb{E}\left(\sum_{j=1}^{L} |T \cap W_j|\right)\right] < \frac{1}{3} \qquad \square$$

4.2 An upper-bound on false negatives

Let $x^* \in \mathcal{A}$ be a point such that $d(x^*, z) \leq \lambda$. We wish to show that

$$\mathbb{P}\left[g_j(x^*) \neq g_j(z)\ (\forall 1 \leq j \leq L)\right] < \frac{1}{e} \tag{10}$$

Since $\mathcal{G}$ is $(\lambda, c\lambda, p_1^k, 1/n)$-sensitive and $d(x^*, z) \leq \lambda$, the probability that $g_j(x^*) = g_j(z)$ for any particular j is at least p_1^k. Equivalently, the probability that $g_j(x^*) \neq g_j(z)$ for any particular j is at most $1 - p_1^k$; and it follows that the probability that $g_j(x^*) \neq g_j(z)$ for all j is at most $\left(1 - p_1^k\right)^L$.

Using our previous definitions of $k = \log_{1/p2} n$ and $\rho = \frac{\log(1/p_1)}{\log(1/p_2)}$ and $L = n^\rho$, we have

$$\begin{aligned}
\mathbb{P}\left[g_j(x^*) \neq g_j(z)\ (\forall 1 \leq j \leq L)\right] &\leq \left(1 - p_1^k\right)^L = \left(1 - \left(\frac{1}{p_1}\right)^{-\log_{1/p_2} n}\right)^L \\
&\leq \left(1 - \left(\frac{1}{p_2}\right)^{-\rho \log_{1/p_2} n}\right)^L = \left(1 - \left(\frac{1}{p_2}\right)^{\log_{1/p_2}(1/L)}\right)^L \\
&\leq \left(1 - \frac{1}{L}\right)^L < \frac{1}{e}.
\end{aligned}$$

In the last line, we have used the limit definition of the exponential function (*i.e.* $e^x = \lim_{m\to\infty}\left(1 + \frac{x}{m}\right)^m$) and the fact that the expansion for finite $m \geq 1$ is a lower bound for the series limit. □

4.3 (c, λ)-ANN has contant probability of success

In the (c, λ)-ANN algorithm, we retrieve at most $3L$ data points from the buckets $g_j(z)$, $(1 \leq j \leq L)$ and report the closest one as a (c, λ)-ANN. Let us estimate then the probability of error.

Note that we assume that there is at least one point $x^* \in \mathcal{A}$ with $d(x^*, z) \leq \lambda$. Clearly, if there are more points that satisfy the distance criterion, the probability of success will be higher.

The algorithm can fail in two ways:

- The element x^* is hashed to an incorrect bucket, *i.e.* $g_j(x^*) \neq g_j(z)$ for all j. This event was considered previously in Section 4.2. The probability of this "false negative" event is bounded at e^{-1}.
- The element x^* is hashed to a correct bucket, but too many false positives (more than $3L$) from the set $\sum_{j=1}^L |T \cap W_j|$ leads x^* not to be considered in our first $3L$ candidates. This "false positives" scenario was considered in Section 4.1.

The corresponding probability of failure is

$$\mathbb{P}_{\text{fail}} < e^{-1} + \left(1 - e^{-1}\right) \cdot \frac{1}{3} \approx 0.58,$$

and hence the reported point is an actual (c, λ)-ANN with constant probability. □

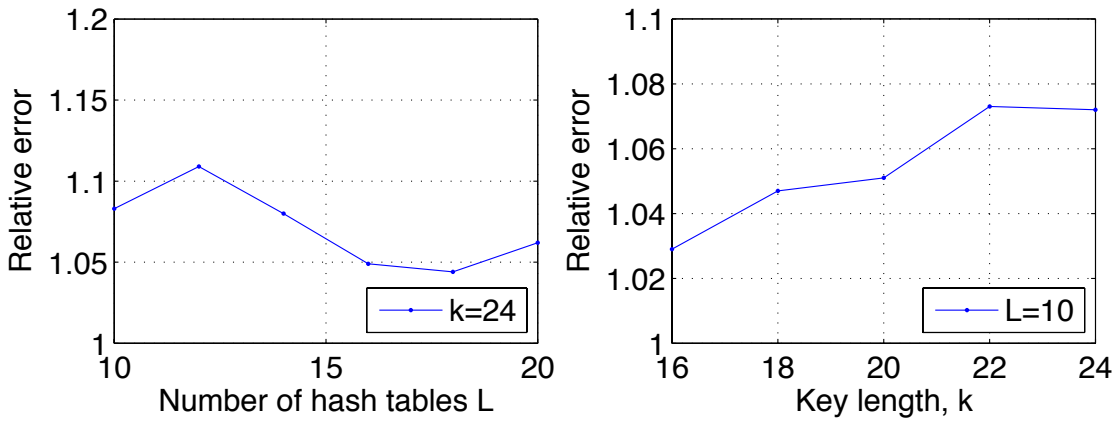

Figure 1: The relative error between LSH and exhaustive search results as a function of L and k. We compare the distances of 3 nearest neighbors under LSH and linear search. A relative error of 1 is ideal.

4.4 (c, λ)-ANN on a dataset of images

4.4.1 Running time

On my laptop, the average search time for LSH is 16 ms, while the linear (exhaustive) search takes 216 ms.

4.4.2 Error as a function of L and k

Fig. 1 shows the relative error of LSH compared to the linear search as a function of L and k. In the best case, the error should approach 1.

4.4.3 Top 10 nearest neighbors using the two methods

In Fig. 2, I show the top 5 (sorry, top 10 doesn't fit into the page as nicely...) LSH and linear search results for three different image patches. I used $L = 10$, $k = 16$ since that parameter set yielded the best performance as shown in Section 4.4.2. In most cases, the visual agreement between the two methods are quite good. (In fact the quantitative distance measures are also comparable.)

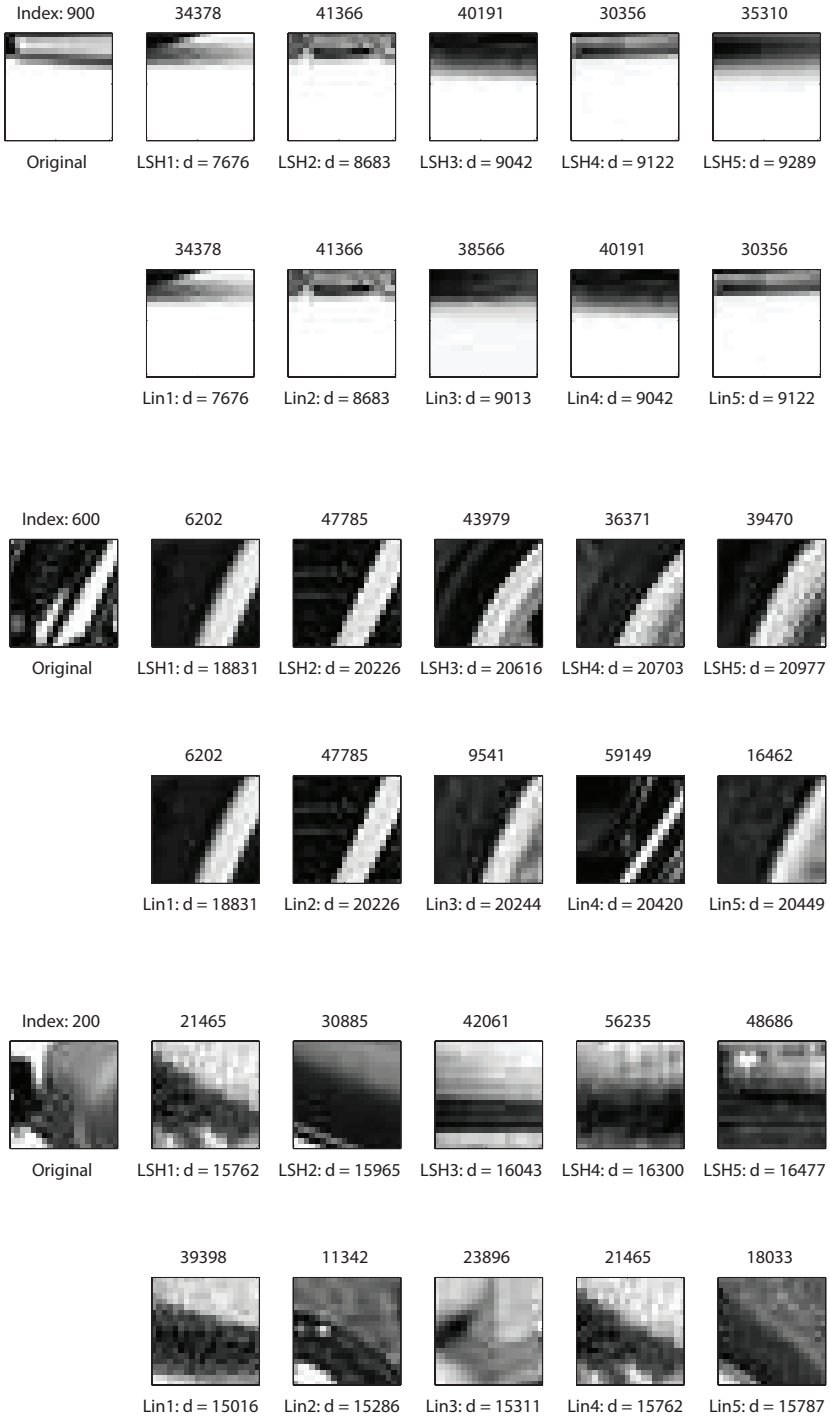

Figure 2: Visual comparison of top 5 nearest neighbors according to LSH and linear (exhaustive search). Three query images (indices of 900, 600 and 200) are shown.