APPROXIMATE NEAREST NEIGHBOR SEARCH
PROJECT REPORT

J. BAÑULEOS, M. CROSSKEY, R. LAREAU-DUSSAULT, O. MBODJI,
H. YAPLE, K. ZELEKE, S. KUMAR

Contents

1. Introduction 1
2. State-of-the-Art Hashing Methods 2
2.1. Creating hash table 2
2.2. Search Methods 5
3. Project's Contributions 7
3.1. Mixture of Gaussians 7
3.2. Our algorithm 8
4. Conclusion 10
4.1. Future work 10
Appendix A. Clustering error bound 10
Appendix B. Proof 2 12
References 15

1. Introduction

Driven by rapid advances in many fields including Biology, Finance and Web Services, applications involving millions or even billions of data items such as documents, user records, reviews, images or videos are not that uncommon. Given a query from a user, fast and accurate retrieval of relevant items from such massive data sets is of critical importance. Each item in a data set is typically represented by a feature vector, possibly in a very high dimensional space. Moreover, such a vector tends to be sparse for many applications. For instance, text documents are encoded as a word frequency vector. Similarly, images and videos are commonly represented as sparse histograms of a large number of visual features. Many techniques have been proposed in the past for fast nearest neighbor search. Most of these can be divided in two paradigms: Specialized data structures (e.g., trees), and hashing (representing each item as a compact code). Tree-based methods scale poorly with dimensionality, typically reducing to worst case linear search. Hashing based methods are popular for large-scale search but learning accurate and fast hashes for high-dimensional sparse data is still an open question.

In this project, we focus on fast approximate nearest neighbor search in massive databases by converting each item to a binary code. Localities Sensitive Hashing (LSH) is one of the most prominent methods that uses randomized projections to generate simple hash functions. However, LSH usually requires long codes for good performance. The main challenge of this project was how to learn appropriate hash functions that take input data distribution into consideration. This would lead to more compact codes, thus reducing the storage and computational needs significantly. The project focused on understanding and implementing state-of-the-art hashing methods, developing the formulation for learning data-dependent hash functions assuming a known data density, and experimenting with medium to large scale datasets. Ultimately, we wanted to find an accurate algorithm with $\mathcal{O}(n)$ time.

2. State-of-the-Art Hashing Methods

We were primarily concerned with the efficiency of two components of the state-of-the-art algorithms, construction time for the hashing code and the search time. Although an algorithm with construction time $\mathcal{O}(n)$ exists, an algorithm with search time $\mathcal{O}(n)$ does not exist.

There is a cover tree method presented in [1], which claims to have achieved our goal, a nearest neighbor search algorithm with $\mathcal{O}(n)$ time. The problem found with this work was the presence of the constant $c \sim 2^d$ (where d is the dimension of the data) in the search time of $\mathcal{O}(c^{16}n)$ and construction time of $\mathcal{O}(c^6 n \log(n))$. Trees will never be effective for high dimension data sets, so we did not consider them.

We implemented and compared tree types of hashing construction methods (linear, nonlinear, and clustering), and three types of search methods (pairwise, lookup tables and permutation). Moreover, we compare with a cover trees algorithm [].

CITATION NEEDED

We compared the running times of the state-of-the-art algorithms we implement to each other while we use the brute force, pairwise comparison, as a baseline. The brute force algorithm runs in $\mathcal{O}(n^2)$ time. With the brute force algorithm, when searching 100 000 points of dimension 128, it takes more than 12 hours to find the nearest neighbor. Hence, the need for a faster algorithm is clear. Of course, a faster algorithm must sacrifice some accuracy.

2.1. Creating hash table.

Creating hash table. In particular, we focus on hashing methods. A hashing function is a function $h : \mathbb{R}^d \to \mathbb{R}^m$, where $m < d$. A hashing function should be such that $\Pr[h(x) = h(y)] = \text{sim}(x, y)$, where $\text{sim} : \mathbb{R}^d \times \mathbb{R}^d \to [0, 1]$ is a similarity function.

We implement:

- Linear hashing method (see figure 1):

In this case, the similarity function is linear. For example if $\text{sim}(x, y) = x^T y$, then a good hashing function (see [2]) consists of projecting data onto m random hyperplanes and using the sign of the projection to determine the ith binary value. So each component of the hashing function looks like $h_i(x) = v^T x$, $i = 1 \dots m$, $v \in \mathcal{N}(0, \mathbf{1})$.

It takes $\mathcal{O}(mn)$ to create the hash table with the linear hashing method.

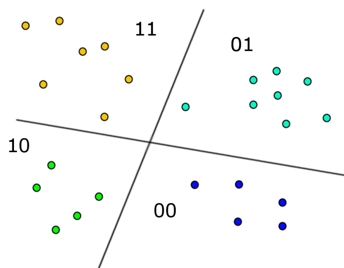

FIGURE 1. Linear hashing

- Nonlinear hashing function:

If the similarity is a non-linear function, $\text{sim}(x, y) = \kappa(x, y) = \phi(x)^T \phi(y)$, where ϕ is nonlinear. We must use a non-linear hashing method h to keep the property $P(h(x) = h(y)) = \text{sim}(x, y)$.

The function ϕ is usually unknown. Some algorithms have been constructed (see [4] and [9]) to avoid this problem. In this project we used three of them; the first two use κ to evaluate ϕ and the other replaces ϕ by a known function z such that $\text{E}[z(x)^T z(y)] = \phi(x)^T \phi(y)$.

In all these methods we want to approximate the hashing function $h(x) = v^T \phi(x)$, $v \sim \mathcal{N}(0, \mathbf{1})$. This is the hashing function in the space $\phi(x)$ with the linear similarity $\text{sim}(\phi(x), \phi(y)) = \phi(x)^T \phi(y)$.

The first method ([4]) approximates v by a linear combination of $\phi(x)$, $v = \sum \alpha_i \phi(x_i)$, such that $h(x) = v^T \phi(x) = \sum \alpha_i \kappa(x_i, x)$.

[4] explains how to choose v such that $v = \sum \alpha_i \phi(x_i) \sim \mathcal{N}(0, \mathbf{1})$. First we choose p points in our data set from which we will choose our $\phi(x)$ to create a v. We can choose these points randomly or choose those points to be the centers of p clusters in our data set (those center were approximated with the MATLAB function KMEANS).

With this method, we create the table in $\mathcal{O}(p^3 + pn)$ time.

Of all the hashing functions, this method gave the most accurate experimental result.

This nonlinear function is compared to the linear and clustering methods (see figure 4 and 5).

This method may seem slow, but once we have the clusters defined it is not that slow.

The second method is similar to the first one, but we take a little less trouble to evaluate κ. It is not significantly faster and gave us worse results that the first one.

The third method [9] uses a function $v(x)$ to evaluate $\phi(x)$. This method uses Bochner's theorem, that tells that a positive definite kernel $k(x, y) = K(x - y)$ is the Fourier transform of a non-negative measure [9]. If we find that measure p, then if ω is an sample from p and $z_{omega}(x) = [\cos(\omega^T x), \sin(\omega^T x)]$ then $E_\omega[z_\omega(x)^T z_\omega(y)] = k(x, y)$, and so $\phi(x)^T \phi(y) \approx z_\omega(x)^T z_\omega(y)$. We then use the hashing $h_\omega(x) = z^T \omega(x)$. This method is less accurate than the first one, so we did not use it. See figure 2 for a comparison of the three nonlinear hashing methods, with and without clustering.

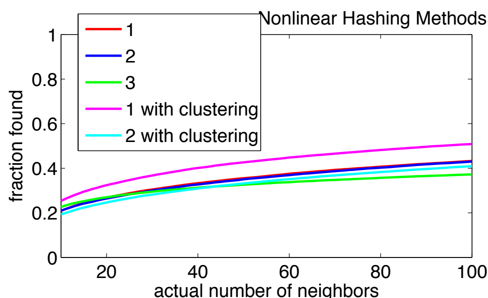

FIGURE 2. Nonlinear hashing

- Clustering (see figure 3):

  The "Anchor graph" method uses a small set of c points called anchors to approximate the data cluster centers. Points are encoded by their nearest anchors. As demonstrated in figure 3 we create a hash table where each row corresponds to a query point and each column corresponds to an anchor. If anchor m is within the first s nearest cluster centers then set bit m to one and to zero otherwise.

  The idea is to use Anchor Graphs to obtain tractable low-rank adjacency matrices in order to make the approach computationally feasible. The memory cost of an Anchor graph is $\mathcal{O}(sn)$ and time cost is $\mathcal{O}(dmnT + dmn)$, where m is the number of cluster centers and T is the number of iteration in K-means clustering. (See [5]).

Our experiments compared the construction time and accuracy of these three hashing methods. Accuracy is calculated by finding the k points with the nearest Hamming distance to a query and finding the fraction of those k points that are actually in the top k true nearest neighbors.

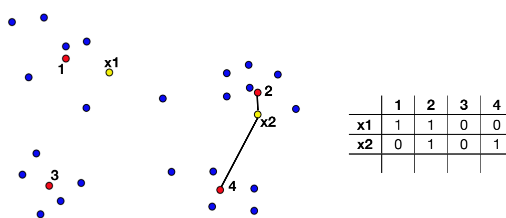

Figure 3. Hashing with clusters

The data we use consist of 100 000 points, each with dimension 128. We hash 64-bits in our experiments. Figure 4 summarizes our experimental results. As expected, the nonlinear hashing method has the greatest accuracy, followed by the linear hashing method, and then the clustering method. Figure 5 summarizes the running time for these algorithms.

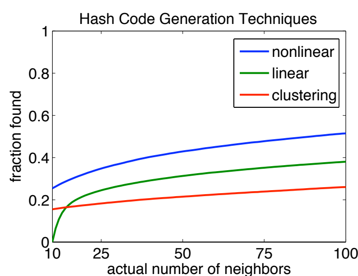

Figure 4. Accuracy Comparison of Hashing Methods

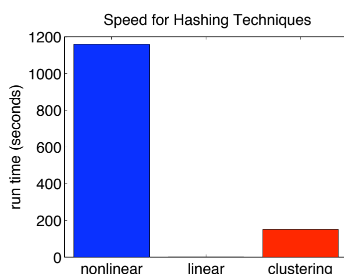

Figure 5. Construction Time Comparison of Hashing Methods

2.2. **Search Methods.** As mentioned above, there are three state-of-the-art methods we implemented to search for the approximate nearest neighbors, after mapping each point to a hash function. The first, brute force pairwise comparison of elements takes $\mathcal{O}(n^2)$ time. We used this method as our baseline for comparison. Secondly, we used lookup tables. That is,

we found the distance between the query, and elements hashed to the same bit-string.

The last search algorithm we implemented requires greater explanation. The steps are now outlined.

(1) We randomly choose a set of p permutations $\{\sigma_i \mid i \in \{1, \ldots, p\}\}$.
(2) For each point in the point set, we permute the bits in its hash value according to σ_i, for each i. Call the set of permuted hash values Ω_i.
(3) Sort each Ω_i in lexicographical order.
(4) For a given query, find the elements above and below it in (the now sorted) Ω_i. Put these elements in the set N_i.
(5) Calculate the true distance between q and each element in $\bigcup_i N_i$.
(6) Return the point with the smallest distance from q, as the approximate nearest neighbor.

Clearly, as the number of permutations taken (p) increases the accuracy of this algorithm also increases. However, in doing so we increase the run time of our search algorithm.

In our experiment we use the same linear hashing in order to compare the search time and accuracy of the three methods. As figures 6 and 7 show, our experiments yielded the expected results. The pairwise search, which has the longest time runtime, is the most accurate, followed by the permutation method, which has the second longest run time. The reasoning behind the superior accuracy of the pairwise search and the relationships between search times is obvious. The reason that the permutation method has better results than the lookup table method is that the permutation method allows for comparison of points both with the same hash code and distinct hash code, whereas the lookup table method only allows for the former.

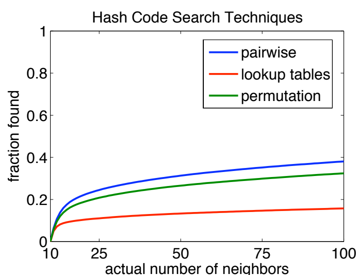

Figure 6. Accuracy Comparison of Search Methods

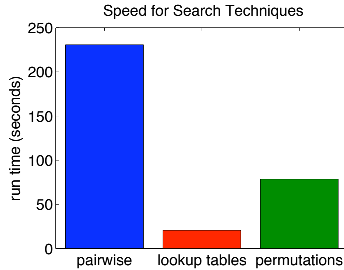

FIGURE 7. Time Comparison of Search Methods

3. PROJECT'S CONTRIBUTIONS

3.1. **Mixture of Gaussians.** It is well-known that large data sets (S), such as the ones we are interested in, can be modeled with a mixture of gaussians:

$$S \sim \sum_{i=1}^{K} \omega_i \mathcal{N}(\mu_i, \Sigma_i)$$

where K is the number of Gaussians (i.e. clusters) in the mixture models; ω_i is the mixing probability (i.e. the weight of each component in the mixture); and μ_i, Σ_i are the mean and variance, respectively, of the ith component $\mathcal{N}(\mu_i, \Sigma_i)$. In other words, x has a probability ω_i to come from a Gaussian with parameters μ_i and Σ_i:

$$Pr[x] = \sum_{i=1}^{K} \omega_i Pr[x \mid \mu_i, \Sigma_i].$$

A question naturally arises when studying our ANN problem and assuming a Mixture of Gaussians distribution of the point set: what is the distribution of distances in the point set? With the knowledge of the distribution of distances, we were hoping to construct a more efficient and accurate hashing method. The distribution of distances is a mixture of non-centralized chi-square distribution. Using the characteristic function, we obtained equations that could not be represented in closed form. We could find the mean and the variance, but we were not able to use this information to create a new hashing scheme. Figure 3.1 below is a histogram that generally represents what the distribution of distances looks like. In producing the figure, we let there be K=7 components and let $Pr[\omega_i] = \frac{1}{K}$ for $i = 1, 2, 3, 4, 5, 6, 7$.

We could not use the distribution to create a new hashing function, but instead use the notion of anchor hashing. Although in our algorithms we calculate the location of anchors or cluster centers manually, if we assume that the distribution of the data is known, e.g. mixture of Gaussians, we can simply place the cluster centers at the Gaussian peaks, eliminating the need to manually find suitable locations.

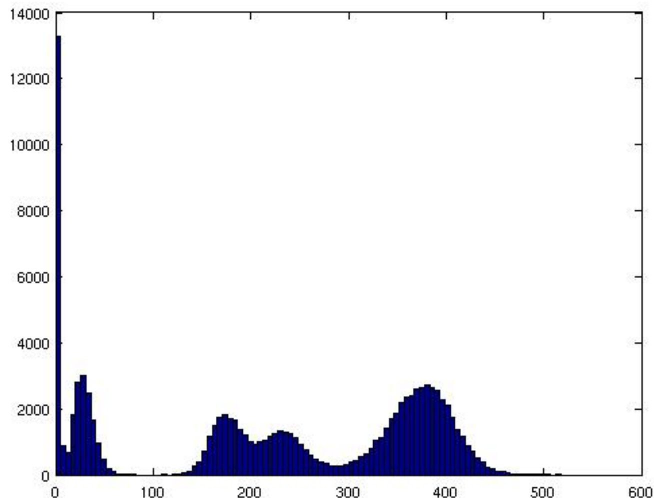

FIGURE 8. Distribution of Distances

3.2. **Our algorithm.** After studying the results of the methods outlined above, we combined several of them to create an algorithm which we believe to be quite good in terms of both speed and accuracy. It uses clustering, linear hashing, and lookup tables to find nearest neighbors; the procedure follows.

First, we find appropriate locations for c clusters using the MATLAB function KMEANS. We find the s nearest clusters for every point in the data set. Then, within each cluster, we re-center the data by subtracting its median. To create hash codes, we first find m_j vectors sampled from $\mathcal{N}(0, \mathbb{I})$, then take their dot product with the re-centered data. The Heaviside of this result gives our binary code. We choose m_j, the number of bits in the hash code, so that the expected value of the number of points with the same code is k. This gives $m_j = \log_2(n_j/k)$, where n_j is the number of points in cluster j. We repeat this process t times in each cluster to create t lookup tables from the hash codes.

To search for the nearest neighbors of a query point q, we must repeat the following procedure for every cluster q belongs to. For each lookup table in the cluster, we match q to its code in the lookup table, and save the first k points with the same code. It would be preferable, in terms of accuracy, to choose randomly from the points with the same code; however, since bins may have up to $\mathcal{O}(n)$ points, this would increase the algorithm complexity excessively. After searching every lookup table in every cluster q belongs to, there will be skt points saved as potential nearest neighbors. We find the $\mathbb{L}^2$ distance from q to each of these points, and save the closest k as the approximate nearest neighbors.

The complexity of finding the nearest clusters is $\mathcal{O}(10cdn)$. The construction of the lookup tables is $\mathcal{O}(dt\sum_{j=1}^{c} m_j n_j) = \mathcal{O}(dt\sum_{j=1}^{c} \log_2(n_j/k)n_j)$. The search complexity is $\mathcal{O}(stn)$. Thus, the computation time is limited by the creation of the lookup tables. It has been noted that if we were to choose a fixed number of bits over all clusters, the logarithmic dependence on n would disappear. However, this decreases our accuracy.

We may improve our accuracy by varying c, s, or t, the number of clusters, the number of nearest clusters, and the number of tables per cluster, respectively. See figures 3.2-3.2 for results.

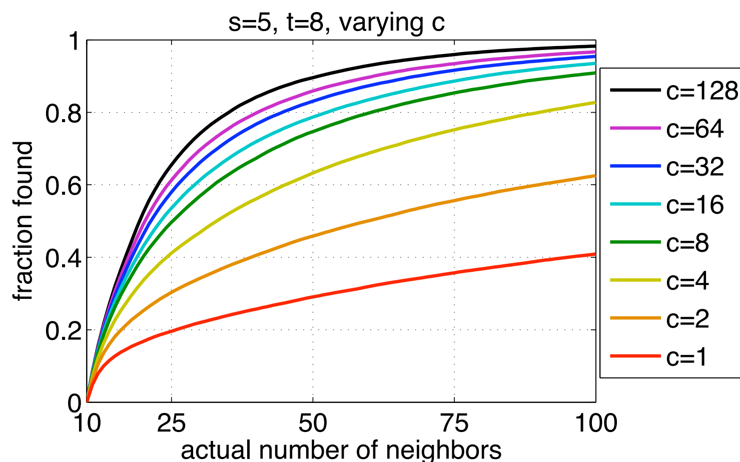

FIGURE 9. Accuracy of algorithm, varying number of clusters c.

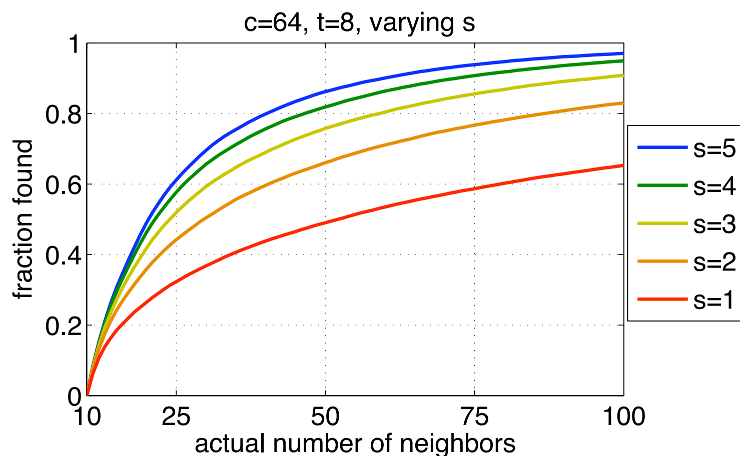

FIGURE 10. Accuracy of algorithm, varying number of nearest clusters s.

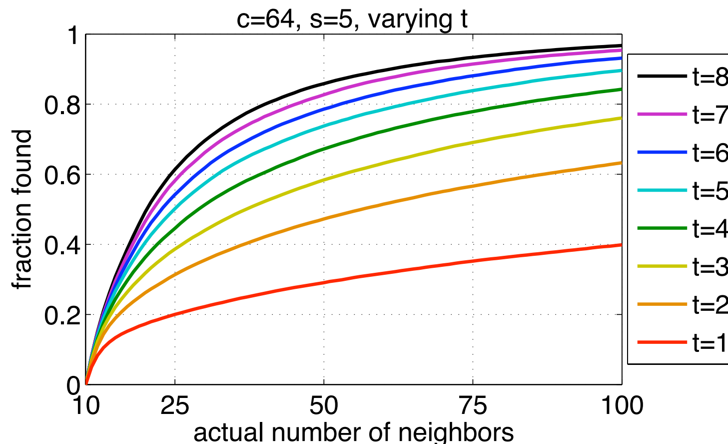

FIGURE 11. Accuracy of algorithm, varying number of lookup tables within each cluster t.

4. CONCLUSION

To conclude, we have not found an $\mathcal{O}(n)$ time algorithm. Nevertheless, we created an algorithm which works better than the state-of-the-art algorithms on our data set. Our algorithm evaluates the real distance between some points, but only for very few of them. As we wisely choose the points to compare, the algorithm is accurate. Furthermore, since we compute so few distances, the algorithm is fast. Moreover, we found some analytical bounds to quantify the accuracy of our algorithm in terms of the probability to find the k nearest neighbors.

4.1. **Future work.**

It would still be interesting to find an $\mathcal{O}(n)$ algorithm for the pairwise search problem. One could use the fact that we search for the nearest neighbors for all our n points to speed up the search.

It is certainly possible to find an $\mathcal{O}(n)$ algorithm, but it might not be accurate. One should show some analytical bound on how accurate the $\mathcal{O}(n)$ algorithm could be.

APPENDIX A. CLUSTERING ERROR BOUND

Suppose $\{x_i\}_{i=1}^n \sim p$ where p is a mixture of c gaussians. Then

$$p(x) = \sum_{j=1}^{c} \omega_j \frac{1}{|2\pi\Sigma|^{1/2}} \exp\left(-\frac{1}{2}||x-\mu_j||_{M_j}\right)$$

Where $||x - \mu_j||_{M_j} = (x - \mu_j)^T \Sigma^{-1}(x - \mu_j)$ indicates the Mahalnobis distance in cluster j. Then let $S(x) = \min_S\{|S| = s, \max_{j \in S}\{||x - \mu_j||_{M_j}\}\}$. $S(x)$ will contain indices of the nearest s clusters to x (in the Mahalnobis distance). Thus, after sampling, the set $S(x_i)$ will contain the indices of s clusters which x_i was most likely to come from. Now, given this mixture of gaussians we can compute a bound on the probability that a sample x and its nearest neighbor, denoted $nn(x)$, will have at least one closest cluster in common.

First we will start with some definitions. Let $A_{j,\rho} = \{x : ||x - \mu_j||_{M_j} < \rho\}$. Then define

$$r = \inf_{\rho > 0}\{\exists J : |J| = 2s, \cap_{j \in J} A_{j,\rho} \neq \emptyset, j \in J \text{ distinct}\}$$

This definition for r will ensure that at most $2s - 1$ of the $A_{j,r}$ intersect in the interior of any $A_{j,r}$. This constant can be computed given the mixture of gaussians.

Theorem A.1.

$$\mathbb{P}[S(q) \cap S(nn(q)) \neq \emptyset] \geq 1 - \prod_{j=1}^{c}(1 - \omega_j \text{erf}(\rho)(1 - e^{-2n}))$$

where

$$\rho = r - \mathcal{O}(n^{1/d})$$

Proof This theorem hinges on a few key observations. First observe that by the definition of r if q and $nn(q)$ are both in the same $A_{j,r}$ then there are only $2s - 1$ neighboring clusters that can possibly chosen for $S(q)$ and $S(nn(q))$. Thus $S(q) \cap S(nn(q)) \neq \emptyset$. We will begin by considering a single set $A_{j,r}$ and assuming we know the location of q inside that set. We will then use a concentration inequality (Hoeffding) to ensure that $nn(q)$ is near enough to also lie inside $A_{j,r}$ with high probability.

The first thing to do is get a lower bound for the probability that a sample x_i lands in a small ball around q. In what follows we will consider $B(m, x)$ to be balls (in Mahalnobis distance) of radius m centered around x. We will also drop any subscripts indicating the Mahalnobis distance.

Lemma A.2. *Assume that for some* $||q - \mu_j|| > 2\varepsilon > 0$, $r - ||q - \mu_j|| \leq 2\varepsilon$.

$$\gamma_\varepsilon := \mathbb{P}[x_i \in B(2\varepsilon, q)] \geq \omega_j \frac{4\varepsilon^{d-1}}{\pi r^d} e^{-r^2}$$

Proof We will use a ball packing method in a ring of distance $||q - \mu_j||$ around μ_j and radius ε to show that each ball (and in particular, the one around q) has at least a fixed amount of mass, dependent upon ε and d. Let R_ε be the ring at distance $||q - \mu_j||$ around μ_j. Formally,

$$R_\varepsilon = B(||q - \mu_j|| + \varepsilon, \mu_j) \setminus B(||q - \mu_j|| - \varepsilon, \mu_j)$$

The size of R_ε in M_j distance is

$$|R_\varepsilon| = \alpha_d((||q-\mu_j||+\varepsilon)^2 - (||q-\mu_j||-\varepsilon)^2)$$

Where α_d is the size of a d dimensional ball of radius 1. Then there can be at most $N = \frac{1}{\varepsilon^d}((||q-\mu_j||+\varepsilon)^2 - (||q-\mu_j||-\varepsilon)^2)$ balls of radius ε packed into R_ε. We can create then a set of disjoint balls contained in R_ε of radius ε, $\mathcal{B}$, with $|B| < N$ such that no more disjoint balls can be added. Call their centers $\{b_l\}_{l\in L}$. Then $\cup_{l\in L} B(2\varepsilon, b_l)$ covers R_ε and contains N balls. All of these balls have the same p_j mass. Thus, the mass of each ball has at least $p_j(R_\varepsilon)/N$ p_j mass. From the definition of R_ε,

$$p_j(R_\varepsilon) = \text{erf}(||q-\mu_j||+\varepsilon) - \text{erf}(||q-\mu_j||-\varepsilon) \geq \frac{4\varepsilon}{\sqrt{\pi}} e^{-r^2}$$

.

Since $N < \left(\frac{r}{\varepsilon}\right)^d$, the lemma is proven. ∎

Fix $q \in A_{j,r}$ for some j. Then we can choose ε as in the above lemma. Now we wish to calculate the probability that $nn(q)$ lies within $r - ||q-\mu_j||$ of q. Then $nn(q)$ will surely also lie inside the successful region $A_{j,r}$. Using Hoeffding's inequality,

$$\begin{aligned} \mathbb{P}[nn(q) \in A_{j,r}] &= \mathbb{P}\left[\sum_{i=1}^{n-1} \mathbf{1}\{x_i \in B(r - ||q-\mu_j||, q\} \geq 1\right] \\ &\geq 1 - \exp(-2n(n\gamma_\varepsilon - 1)^2) \end{aligned}$$

Now we wish to choose ε_j such that $\gamma_{\varepsilon_j} \geq 2/n$. This leads to the choice

$$\varepsilon_j = \left(\frac{\pi r^d e^{r^2}}{2n\omega_j}\right)^{\frac{1}{d+1}}$$

This choice of ε_j will enforce a distance between q and the boundary of $A_{j,r}$. Now choose $\varepsilon = \max_j \varepsilon$. Then let $\rho = r - \varepsilon$. Now putting what we have learned together,

$$\begin{aligned} \mathbb{P}[\{q, nn(q)\} \subset A_{j,r}] &= \mathbb{P}[nn(q) \in A_{j,r} | q \in A_{j,\rho}]\mathbb{P}[q \in A_{j,\rho}] \\ &\geq \omega_j \text{erf}(\rho)(1 - e^{-2n}) \end{aligned}$$

The only thing left to do is compute the probability that at least one $A_{j,r}$ contains q and $nn(q)$. That step is left as an excercise for the reader.

Appendix B. Proof 2

Theoretically, we want to know how similar objects are. Objects will be represented as points in a high dimensional space. Given a query we want to find its s nearest neighbors.

Result 1: Our LSH code is is a (r_1, p_1, r_2, p_2) sensitive family and can solve the (R, c) NN problem under measure $1 - \frac{\theta}{\pi}$.

Here $\theta(x,y) = arccos(\frac{x'y}{\|x\|\|y\|})$. If $\max_{\text{within cluster}} ||x_i - a_j|| < M$, $\min_{\text{within cluster}} ||x_i - a_j|| > \frac{1}{M}$, then $1 - \frac{\theta}{\pi}$ and $|| \cdot ||_{\mathbb{L}^2}$ are equivalent up to constants.

Proof. Let p be a point in $B(q, r_1)$. We want to to find p_1 and p_2 such that $\mathbb{P}r_H[h_a(q) = h_a(v)] \geq p_1$.

Our hashing function is defined by
$H = \{h_a : v \mapsto (sgn(v'a^1), \cdots, sgn(v'a^t)) \in \mathbb{R}^t\}$ where a^i, are Gaussian random variables.

Our similarity function is $sim(p,q) =$Theoretically, we want to know how similar objects are. Objects will be represented as points in a high dimensional space. Given a query we want to find its s nearest neighbors.

Result 1: Our LSH code is is a (r_1, p_1, r_2, p_2) sensitive family and can solve the (R, c) NN problem under measure $1 - \frac{\theta}{\pi}$.

Here $\theta(x,y) = arccos(\frac{x'y}{\|x\|\|y\|})$. If $\max_{\text{within cluster}} ||x_i - a_j|| < M$, $\min_{\text{within cluster}} ||x_i - a_j|| > \frac{1}{M}$, then $1 - \frac{\theta}{\pi}$ and $|| \cdot ||_{\mathbb{L}^2}$ are equivalent up to constants.

Proof. Let p be a point in $B(q, r_1)$. We want to to find p_1 and p_2 such that $\mathbb{P}r_H[h_a(q) = h_a(v)] \geq p_1$.

Our hashing function is defined by
$H = \{h_a : v \mapsto (sgn(v'a^1), \cdots, sgn(v'a^t)) \in \mathbb{R}^t\}$ where a^i, are Gaussian random variables.

Our similarity function is $sim(p,q) = 1 - \theta(p,q)/\pi$.

Note that from the definition of a LSH, $sim(p,q) = \mathbb{P}r_H[h_a(q) = h_a(v)]$, so we must show $sim(p,q) \geq p_1$.

$p \in B(q, r_1) \Rightarrow 1 - \theta(p,q)/\pi \geq p_1$
but the ball $B(q, r_1)$ is defined by $1 - \theta(p,q)/\pi \geq r_1$. This implies
$r_1 \geq p_1$
So if we choose $r_1 \geq p_1$, we'll be set.

In the same manner, $p_2 \in$

□

We place ourselves in $\mathbb{R}^n$.

Given a kernel that is translation invariant i.e $k(x,y) = k(x-y)$, can we find a simpler method?

We want to have $\phi(x)'\phi(y) = k(x-y)$ for an imbedding $\phi \in \mathbb{R}^d$.

This is possible thanks to Bochner's theorem. However, this is a theoretical result that will typically give us a dimension $d >> 1$.

Now, by using the Fourier transform of k, we can compute a probability density p.

Here is my hashing method.

We choose ω randomly from the distribution p to produce m vectors.

For D a small integer to be fixed later: We compute the vector $z_D(x) = (\cos(\omega_1' * x) \cdots \cos(\omega_D' * x) \sin(\omega_1' * x) \cdots \sin(\omega_D' * x)) \in \mathbb{R}^{2D}$

Now, we notice that

$\mathrm{E}_\omega \left[z_D(x)' z_D(y) \right] = \sum_{i=1}^{D} p(\omega_i) \cos(\omega_1' * (x-y)) \simeq \int_0^{\mathbb{R}^d} p(w) k(\omega'(x-y)) dw$

Now the bigger D is, the better we capture the modes of oscillations and approximate the true kernel.

We finish our search using the traditional matrix K and the same algorithm as Kernel-1.

We obtain slightly worse results than in the case of the Kernel-1. $1 - \theta(p,q)/\pi$.

Note that from the definition of a LSH, $sim(p,q) = \mathbb{P}r_H[h_a(q) = h_a(v)]$, so we must show $sim(p,q) \geq p_1$.

$p \in B(q, r_1) \Rightarrow 1 - \theta(p,q)/\pi \geq p_1$
but the ball $B(q, r_1)$ is defined by $1 - \theta(p,q)/\pi \geq r_1$. This implies
$r_1 \geq p_1$
So if we choose $r_1 \geq p_1$, we'll be set.

In the same manner, $p_2 \in$

□

We place ourselves in $\mathbb{R}^n$.

Given a kernel that is translation invariant i.e $k(x,y) = k(x-y)$, can we find a simpler method?

We want to have $\phi(x)'\phi(y) = k(x-y)$ for an imbedding $\phi \in \mathbb{R}^d$.

This is possible thanks to Bochner's theorem. However, this is a theoretical result that will typically give us a dimension $d >> 1$.

Now, by using the Fourier transform of k, we can compute a probability density p.

Here is my hashing method.

We choose ω randomly from the distribution p to produce m vectors.

For D a small integer to be fixed later: We compute the vector $z_D(x) = (\cos(\omega_1' * x) \cdots \cos(\omega_D' * x) \sin(\omega_1' * x) \cdots \sin(\omega_D' * x)) \in \mathbb{R}^{2D}$

Now, we notice that

$\mathrm{E}_\omega \left[z_D(x)' z_D(y) \right] = \sum_{i=1}^{D} p(\omega_i) \cos(\omega_1' * (x-y)) \simeq \int_0^{\mathbb{R}^d} p(w) k(\omega'(x-y)) dw$

Now the bigger D is, the better we capture the modes of oscillations and approximate the true kernel.

We finish our search using the traditional matrix K and the same algorithm as Kernel-1.

We obtain slightly worse results than in the case of the Kernel-1.

References

[1] J. Langford A. Beygelzimer, S. Kakade. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, pages 97–104, New York, NY, USA, 2006. ACM.

[2] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *In Proc. of 34th STOC*, pages 380–388. ACM, 2002.

[3] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[4] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *IEEE International Conference on Computer Vision (ICCV*, 2009.

[5] S. Kumar S.-F. Chang L. Wei, Wang J. Hashing with graphs. *Proceedings of the 28th International Conference on Machine Learning*, 2011.

[6] P. Indyk V. Mirrokni M. Datar, N. Immortica. Locality-sensitive hashing scheme based on p-stable distributions. *Annual Symposium on Computational Geometry*, 2004.

[7] W.B. March A.G. Gray P. Ram, L. Dongryeol. Linear-time algorithms for pairwise statistical problems. *Advances in Neural Information Precessing Systems (NIPS)*, (22), 2009.

[8] M. Raginksy and S. Lazebnik. Locality-sensitive binary codes from shift-invariant kernels. In *Proc. of Advances in nueral information processing systems*, 2009.

[9] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *Neural Infomration Processing Systems*, 2007.