Fast Nearest-neighbor Search

Abstract

This paper evaluates three fast nearest-neighbor search algorithms: locally sensitive hashing (LSH), K-Dimensional tree (K-D tree), and ball tree. This paper performs the theoretical asymptotic time complexity analysis for each algorithm and conducts experimental verifications on performance to the aspect of training size and data point dimensionality. LSH exhibits the highest efficiency in both aspects. The three algorithms have lower efficiency than brute-force for smaller training data sizes, yet both LSH and K-D tree performs significantly better as the size increases. K-D tree performs better on low-dimensional data sets while ball tree gains higher efficiency on high-dimensional data sets. Furthermore, all three algorithms exhibit a high level of correctness. The paper also provides an overview of applications for all of the fast nearest-neighbor search algorithms.

1 Introduction

In this paper, we explore three fast nearest-neighbor search algorithms. To be specific, we explore locally sensitive hashing, K-Dimensional tree, and ball tree algorithms. We explain them in detail respectively, then compare their performance on inputs of different dimensions. Finally, we construct kNN classifier based on the searching algorithms, evaluating the performance, while at the same time comparing the run time to brute force kNN in sklearn.

1.1 Locally sensitive hashing (LSH)

Locality-sensitive hashing (LSH) is a hashing technique that hashes data points that are close to each other in the feature space into the same buckets with high probability, while data points far from each other tend to be hashed into different buckets [1]. The distinction that sets LSH and conventional hashing techniques apart is that hash collisions are maximized instead of minimized in the LSH method. Because of this special property, LSH has applications in nearest neighbor search and data clustering [2].

This technique has several advantages. First, LSH is especially useful for reducing the dimensionality of high-dimension data as it reduces high-dimensional data points to low-dimensional versions while maintaining the relative distance between points [2]; Second, LSH makes it easier to identify observations with various degrees of similarity [1].

The LSH approach is implemented utilizing a hash table, and a family of hash function, often referred to as LSH family. The choice of hashing functions is directly associated with the metric we use. In the experiments, we use Euclidean and Manhattan distance metrics. Thus, we use the hash function $h_{a,b}(\mathbf{v}) : \mathcal{R}^d \to \mathcal{N}$ which maps a d dimensional vector $\mathbf{v}$ on to the set of integers. Each hash function in the family is indexed by a choice of random $\mathbf{a}$ and $\mathbf{b}$ where $\mathbf{a}$ is a d dimensional vector with entries chosen independently from a stable distribution (e.g. Gaussian distribution) and $\mathbf{b}$ is a real number chosen uniformly from the range $[0, r]$, where r represents the distance threshold. For fixed $\mathbf{a}$ and $\mathbf{b}$, $h_{a,b}(\mathbf{v}) = \lfloor \frac{\mathbf{a}\cdot\mathbf{v}+\mathbf{b}}{r} \rfloor$ [3].

Exercise 1 True or False: All LSH families share the same properties, hence each of them can work well with hashing tasks that involve a variety of distance metrics.

1.2 K-Dimensional tree (K-D tree)

K-Dimensional tree is a binary tree where every leaf node is k-dimensional. K-D tree is a special case of binary space partitioning trees, and every non-leaf node can be considered as implicitly generating a splitting hyperplane that divides the space into two halfspaces [4]. Therefore, in nearest neighbours search, the K-D tree can search the k nearest points with higher efficiency as it is capable of eliminating large portion of the search space and find the desired candidates.

In low-dimensional space ($\texttt{dim} \leq 3$), K-D tree algorithm performs lots of pruning, which significantly improves the efficiency and lowers the runtime. Therefore, it has better performance for many low-dimensional datasets. Nonetheless, since the node splitting of K-D tree are only axis-aligned, the distribution may not be accurately mapped in high-dimensional space and thus it has worse performance for high-dimensional datasets [5].

Exercise 2 True or False: In kNN search using K-D tree, we can prune a branch if this branch has less than k points.

1.3 Ball tree

Ball tree is a recursive algorithm that partitions all data into sets with size smaller than a leaf size of k. This algorithm is illustrated by the following example, where the data has dimension two, the leaf size is 3, and distance is defined by Euclidean distance.

First, we randomly fix a centroid C1. Then, centroid 2 is defined to be the farthest data from C1, and centroid 3 is defined to be the farthest data from C2. Then, we partition all data into two sets, S1 and S2. In S1, all data are closer to C2 than to C3. In S2, vice versa. In S1, we only have two points, so S1 itself forms a leaf. In S2, we have five points, so we recursively apply the ball tree algorithm with centroid C3. Then, we fix C4 and C5 to be the next two centroids, as C4 is the farthest data in S2 to C3, and C5 is the farthest one from C4. After assigning all data in S2 to S3 and S4 based on whether they are closer to C4 or C5, we find the size of all sets are no larger than 3. Thus, we stop. The tree we build has three leaves: S1, S3, and S4.

This algorithm is similar to decision tree, where we recursively partitions data in to smaller sets.

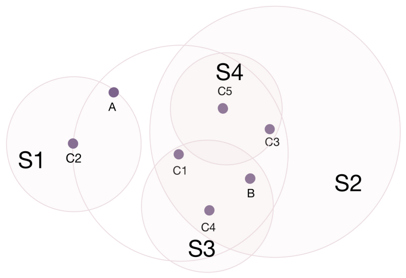

Figure 1: Ball tree algorithm

Exercise 3 How to prevent overfitting when using ball tree?

Exercise 4 Which of the following algorithm always performs the best in kNN classifier for multidimensional datasets considering the accuracy only?
(a) LSH (b) Brute force (c) Ball tree (d) K-D tree

2 Complexity analysis

In this section, we perform theoretical asymptotic time complexity analysis on each of the three algorithms on k-nearest-neighbours.

2.1 LSH

$\mathcal{H}$ is a $(c, cr, p_1, p_2) - LSH$ if for all $p, q \in P$, where P is the input space p, q are data points, c is some constant, r is threshold distance, the following holds 1. if $dist(p, q) \leq r$, then $\mathbf{Pr}[h(p) = h(q)] \geq p_1$, and 2. if $dist(p, q) > cr$, then $\mathbf{Pr}[h(p) = h(q)] \leq p_2$; where $p1 > p2$ are the probabilities over random choices of $h \in \mathcal{H}$. That is, if we choose a random hash function h from $\mathcal{H}$, we have these success probabilities [6].

We boost p_2 by taking k independent hash functions from $\mathcal{H}$ and hashing each point $p \in P$ to a k-dim vector. To improve p_1, we use multiple hash families. Specifically, we choose l independent copies of the above k-dim hash functions:$f_1, ..., f_l$ where each f_i is one of the k-dim hash function. Let n denote the number of data points. We choose k such that $p_2^k = \frac{1}{n}$, and write $p_1 = p_2^\rho$ for some $\rho > 1$. Note that this ρ determines the time/space bounds of LSH algorithm. Then, we choose $l \propto n^\rho \ln n$ [6].

The query time is $O(lk)$ for computing $f_i(q)$, for all $1 \leq i \leq l$. For each candidate close to q, we spend $O(d)$ time to compute the distance $dist(p, q)$. Let x be the size of the output. The total query time is $O(d(x + lk))$. This works out to $O(n^\rho)$ sub-linear time complexity ignoring lower order terms [6].

2.2 K-D tree

The asymptotic time complexity of building a static K-D tree depends on the time complexity of selecting medians for splitting. In this paper, we use linear median of median algorithm that has $O(n)$ complexity [7]. For K-D tree with n points, the adding new nodes and removing nodes both has $O(logn)$ complexity [8]. Therefore, the building of the entire K-D tree with an recursive top down approach has the asymptotic complexity of $O(nlogn)$. Also note that if the n points are presorted in each of k dimensions using an $O(nlogn)$ sort prior to building the k-d tree, the asymptotic complexity of the entire kNN search is $O(knlogn)$.

2.3 Ball tree

We use top down approach when building the ball tree recursively. Like building decision tree, we choose the best dimension and best split in each recursive call. There are n dimensions, and it takes $O(logn)$ to find the best split. So, each recursive call is within $O(nlogn)$. Assuming balanced tree, the depth is $O(logn)$, thus the total asymptotic complexity is $O(n(logn)^2)$.

3 Performance with respect to training size

In this section, we evaluate the performance, especially the runtime, with respect to different sizes of training data.

3.1 Dataset used: Spam E-mail

The dataset we used as the 1-dimensional or vector input dataset is a spam E-mail dataset from `https://www.kaggle.com/somesh24/spambase` In this dataset, there are 4601 samples with 1813 of them labeled as spam E-mail and 2788 of them labeled as none-spam E-mail, and each sample has 57 continuous features.

3.2 Runtime analysis

To evaluate the runtime, we repeatedly select top k training data, and apply all three algorithms. For each algorithm, we set the distance metric to both euclidean and manhattan.

3.2.1 Runtime

In our experiment, we repeatedly find the nearest 20 neighbors for dataset with size 50, 100, 150, ..., 4600. The x-axis represents the size of dataset, and the y-axis represents the runtime. Firstly, according to Figure 2 we have LSH < K-D tree < ball tree with respect to runtime for same distance definition. This matches our analysis that $O(n^\rho) < O(nlogn) < O(n(logn)^2)$ in section 2. Furthermore, we find that the shape of the green line, which represents LSH, is roughly linear; and that the shape of orange line and blue line, which represents K-D tree and ball tree, also matches $O(nlogn)$ and $O(n(logn)^2)$

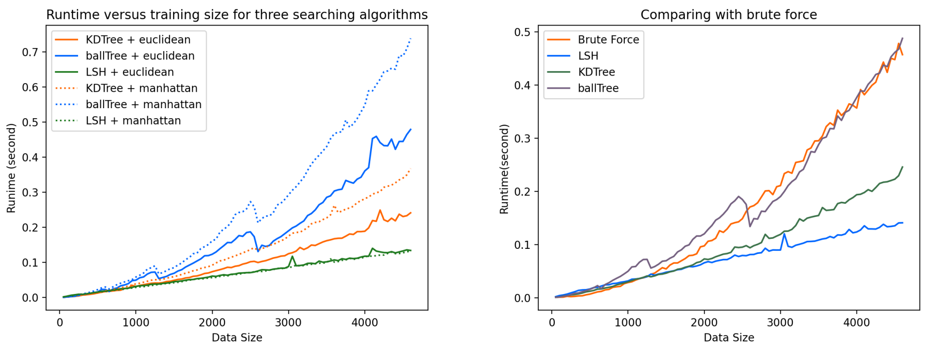

Figure 2: Runtime vs. training size plot of spam E-mail

3.2.2 Comparison with brute-force

According to Figure 2, we can figure out that all three algorithms have lower efficiency than brute-force for smaller training datasizes, yet both LSH and K-D tree performs significantly better as the size increases.

3.3 Correctness analysis

The goal of exploring different searching algorithms is to improve the runtime of kNN. While improving runtime, we notice there might be a tradeoff between runtime and correctness, as none of the three fast search algorithms guarantees correctness.

In this part, we calculate the confusion matrix using all three algorithms, with leaf size of 40 for K-D tree and ball tree. We repeatedly consider each data as the test data, and find that the correctness is roughly the same across three versions of kNN using three different searching algorithms, as in Figure 3.

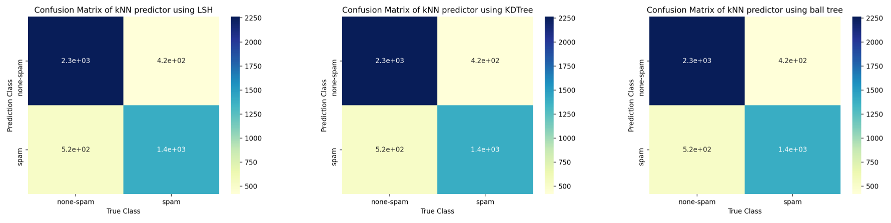

Figure 3: Confusion matrix of kNN predictors using LSH, K-D tree, and ball tree

4 Performance with respect to data dimensionality

In this section, we explore the performance, especially runtime, when given data with different dimension. We employ images, which are naturally of high dimension.

4.1 Dataset used: Fish Images

The dataset we used as the high-dimensional input dataset is part of a fish image dataset from `https://www.kaggle.com/crowww/a-large-scale-fish-dataset` To reduce the runtime to a reasonable range for later analysis, we randomly select 900 images in total from three classes: hrimp, red mullet, and trout. We preprocessed the image by encoding each image into a vector of length 900. Since we have a much larger dimension compared to the last example, we are interested in how this affects the runtime.

4.2 Runtime analysis

We repeatedly select the first k features of the training data, with k = 5, 10, ..., 300. For each selection, we perform the kNN algorithm implemented by the three searching algorithms. From figure 4, we find that LSH outperforms ball tree and K-D tree significantly, especially when dimension is large.

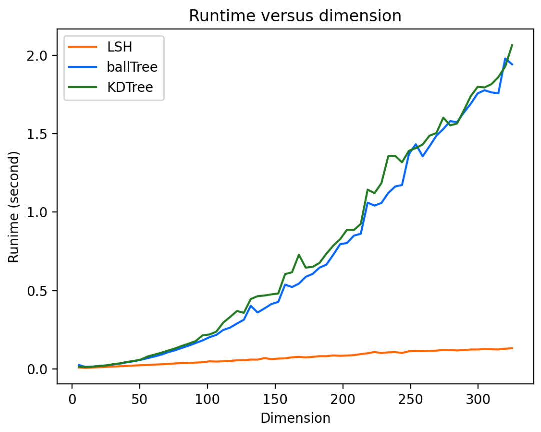

Figure 4: Runtime vs. dimension plot of fish images

Exercise 5 According to the trend shown in previous figures, which algorithm is the better choice for the kNN classification on a 2-dimensional dataset with data size of 5000, the K-D tree or the ball tree?

4.3 Correctness analysis

Similar as in 3.3, we plot the confusion data. We achieve an accuracy of 77%. From the dataset, we found that both shrimp and red mullet are red, while trout is grey. This explains why we have a high accuracy of 97% for test trout image. Furthermore, the most common mistake is to output trout while we have a shrimp. Notice that shrimp is the smallest among the three, so the big blue background is noise that leads to a lower accuracy.

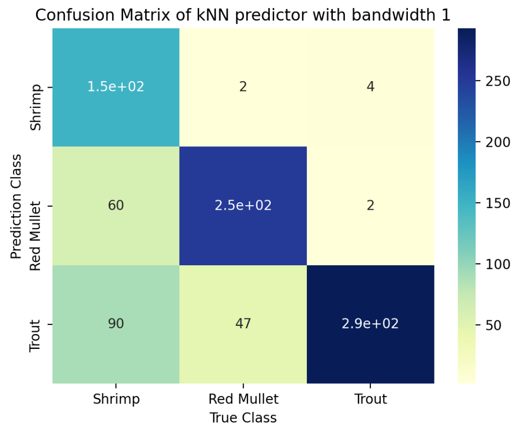

Figure 5: Confusion matrix of kNN predictors of fish images

5 Applications

5.1 LSH

LHS algorithm is an approximate algorithm that can provide faster, scalable solutions to many tasks. In addition to be used in nearest neighbor search and hierarchical clustering, LSH is commonly used to deduplicate large quantities of files. In genome-wide association studies, biologists often use LSH to identify similar gene expressions in genome. Besides, LSH is widely used for audio and image similarities identification. Moreover, in multimedia technologies including digital video and audio, LSH is widely used as a fingerprinting technique.

5.2 K-D tree

The kNN search algorithm using K-D trees have efficient and effective real-time applications in many fields. For instance, this algorithm can be used to solve a near neighbor problem for cross-identification of huge catalogs and realize the classification of astronomical objects with higher efficiency comparing to other classification algorithms [9].

Furthermore, K-D tree earns wide range of applications not limited to kNN search. It can be used in range search and creating point clouds, and it may also be converted to an approximation algorithm to achieve greater efficiency.

5.3 Ball tree

Firstly, The ball tree algorithm itself offers a solution to classification, similar to decision tree. In some scenario, we want to know some near neighbors quickly. Although the complexity required to build the ball tree is more expensive than the brute force searching algorithm, we still prefer ball tree when we need to repeatedly search near neighbors.

6 Answers to exercises

6.1 Exercise 1 answer

False. Though LSH families share certain properties, the choice of a specific hash family that best suits the hashing need is depending on the choice of distance metric.

6.2 Exercise 2 answer

False. We can only prune a branch if we have already found k current nearest points and the branch does not have points nearer than any of the best k points.

6.3 Exercise 3 answer

Similar to decision tree, if we have a larger leaf size, we are reducing the height of the tree, and thus prevents overfitting.

6.4 Exercise 4 answer

(b). Since brute force considers all data points, it will never assign a date point into a wrong class and thus it should the method with highest accuracy. Note that even though brute force has great accuracy, it is highly inefficient in large multidimensional datasets.

6.5 Exercise 5 answer

According to Figure 2 we have the runtime of both K-D tree and ball tree shows an increasing trend when the data size increases for 1-dimensional data. Since from Figure 4 we have for 2-dimensional data the dimensionality does not have significant influence on the rumtime, we have the K-D tree is the better option as it has lower runtime according to the trends shown in Figure 2.

References

[1] Gupta, S. (2018, June 30). Understanding locality sensitive hashing. https://towardsdatascience.com/. https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134

[2] Rajaraman, A.; Ullman, J. (2010). "Mining of Massive Datasets, Ch. 3".

[3] Datar, M.; Immorlica, N.; Indyk, P.; Mirrokni, V.S. (2004). "Locality-Sensitive Hashing Scheme Based on p-Stable Distributions". Proceedings of the Symposium on Computational Geometry.

[4] Bentley, J. L. (1975). "Multidimensional binary search trees used for associative searching". Communications of the ACM. 18 (9): 509–517. doi:10.1145/361002.361007. S2CID 13091446.

[5] Marius, H. (2020). "Tree algorithms explained: Ball Tree Algorithm vs. KD Tree vs. Brute Force". Medium. https://towardsdatascience.com/tree-algorithms-explained-ball-tree-algorithm-vs-kd-tree-vs-brute-force-9746debcd940

[6] Suri, S. (2019, November 19). Locality Sensitive Hashing. Https://Sites.Cs.Ucsb.Edu/. https://sites.cs.ucsb.edu/ suri/cs235/LSH.pdf

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L, Clifford S.: Introduction to Algorithms 3rd edition. MIT Press and McGraw-Hill, (2009).

[8] Munaga H., Jarugumalli V. (2012) Performance Evaluation: Ball-Treeand KD-Tree in the Context of MST. In: Das V.V., Ariwa E., Rahayu S.B. (eds) Signal Processing and Information Technology. SPIT 2011. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 62. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-32573-1

[9] Gao, D., Zhang, Y., Zhao, Y. (2007). The Application of kd-tree in Astronomy. Astronomical Data Analysis Software and Systems ASP Conference Series, Vol. 394. http://adsabs.harvard.edu/full/2008ASPC..394..525G