Learning Space Partitions for Nearest Neighbor Search

Yihe Dong* Microsoft Research
Piotr Indyk MIT
Ilya Razenshteyn Microsoft Research
Tal Wagner MIT

June 19, 2019

Abstract

Space partitions of $\mathbb{R}^d$ underlie a vast and important class of fast nearest neighbor search (NNS) algorithms. Inspired by recent theoretical work on NNS for general metric spaces [ANN+18a, ANN+18b], we develop a new framework for building space partitions reducing the problem to *balanced graph partitioning* followed by *supervised classification*. We instantiate this general approach with the KaHIP graph partitioner [SS13] and neural networks, respectively, to obtain a new partitioning procedure called *Neural Locality-Sensitive Hashing (Neural LSH)*. On several standard benchmarks for NNS [ABF17], our experiments show that the partitions obtained by Neural LSH consistently outperform partitions found by quantization-based and tree-based methods.

1 Introduction

The *Nearest Neighbor Search (NNS)* problem is defined as follows. Given an n-point dataset P in a d-dimensional Euclidean space $\mathbb{R}^d$, we would like to preprocess P to answer k-nearest neighbor queries quickly. That is, given a query point $q \in \mathbb{R}^d$, we want to find the k data points from P that are closest to q. NNS is a cornerstone of the modern data analysis and, at the same time, a fundamental geometric data structure problem that led to many exciting theoretical developments over the past decades. See, e.g., [WLKC16, AIR18] for an overview.

The main two approaches to constructing efficient NNS data structures are *indexing* and *sketching*. The goal of indexing is to construct a data structure that, given a query point, produces a small subset of P (called *candidate set*) that includes the desired neighbors. Such a data structure can be stored on a single machine, or (if the data set is very large) distributed among multiple machines. In contrast, the goal of sketching is to compute compressed representations of points to enable computing approximate distances quickly (e.g., compact binary hash codes with the Hamming distance used as an estimator [WSSJ14, WLKC16]). Indexing and sketching can be (and often are) combined to maximize the overall performance [WGS+17, JDJ17].

Both indexing and sketching have been the topic of a vast amount of theoretical and empirical literature. In this work, we consider the *indexing* problem. In particular, we focus on indexing based on *space partitions*. The overarching idea is to build a partition of the ambient space $\mathbb{R}^d$ and split the dataset P accordingly. Given a query point q, we identify the bin containing q and form the resulting list of candidates from the data points residing in the same bin (or, to boost the accuracy, nearby bins as well). Some of the popular space partitioning methods include locality-sensitive hashing (LSH) [LJW+07, AIL+15, DSN17]; quantization-based approaches, where partitions are

*Author names are ordered alphabetically.

obtained via k-means clustering of the dataset [JDS11, BL12]; and tree-based methods such as random-projection trees or PCA trees [Spr91, BCG05, DS13, KS18].

Compared to other indexing methods, space partitions have multiple benefits. First, they are naturally applicable in *distributed* settings, as different bins can be stored on different machines [BGS12, NCB17, LCY+17, BW18]. Moverover, the computational efficiency of search can be further improved by using any nearest neighbor search algorithm locally on each machine. Second, partition-based indexing is particularly suitable for GPUs due to the simple and predictable memory access pattern [JDJ17]. Finally, partitions can be combined with cryptographic techniques to yield efficient *secure* similarity search algorithms [CCD+19]. Thus, in this paper we focus on designing space partitions that optimize the trade-off between their key metrics: the number of reported candidates, the fraction of the true nearest neighbors among the candidates, the number of bins, and the computational efficiency of the point location.

Recently, there has been a large body of work that studies how modern machine learning techniques (such as neural networks) can help tackle various classic algorithmic problems (a partial list includes [MPB15, BLS+16, BJPD17, DKZ+17, MMB17, KBC+18, BDSV18, LV18, Mit18, PSK18]). Similar methods—under the name "learn to hash"—have been used to improve the *sketching* approach to NNS [WLKC16]. However, when it comes to *indexing*, while some unsupervised techniques such as PCA or k-means have been successfully applied, the full power of modern tools like neural networks has not yet been harnessed. This state of affairs naturally leads to the following general question: **Can we employ modern (supervised) machine learning techniques to find good space partitions for nearest neighbor search?**

1.1 Our contribution

In this paper we address the aforementioned challenge and present a new framework for finding high-quality space partitions of $\mathbb{R}^d$. Our approach consists of three major steps:

1. Build the k-NN graph G of the dataset by connecting each data point to k nearest neighbors;
2. Find a balanced partition $\mathcal{P}$ of the graph G into m parts of nearly-equal size such that the number of edges between different parts is as small as possible;
3. Obtain a partition of $\mathbb{R}^d$ by training a classifier on the data points with labels being the parts of the partition $\mathcal{P}$ found in the second step.

See Figure 1 for illustration. The new algorithm *directly optimizes* the performance of the partition-based nearest neighbor data structure. Indeed, if a query is chosen as a uniformly random *data point,* then the average k-NN accuracy is exactly equal to the fraction of edges of the k-NN graph G whose endpoints are separated by the partition $\mathcal{P}$. This generalizes to out-of-sample queries provided that the query and dataset distributions are close, and the test accuracy of the trained classifier is high.

At the same time, our approach is directly related to and inspired by recent theoretical work [ANN+18a, ANN+18b] on NNS for general metric spaces. The two relevant contributions in these papers are as follows. First, the following structural result is shown for a large class of metric spaces (which includes Euclidean space, and, more generally, all normed spaces). Any graph embeddable into such a space in a way that (a) all edges are short, yet (b) there are no low-radius balls that contain a large fraction of vertices, must contain a sparse cut. It is natural to expect that the k-NN graph of a well-behaved dataset would have these properties, which implies the existence of a desired balanced partition. The second relevant result from [ANN+18a, ANN+18b] shows that, under additional assumptions on a metric space, any such sparse cut in an embedded graph can

be assumed to have a certain nice form, which makes it efficient to store and query. This result has strong parallels with our learning step, where we extend a graph partition to a partition of the ambient $\mathbb{R}^d$ induced by an (algorithmically nice) classifier. Unlike [ANN+18a, ANN+18b], where the whole space is discretized into a graph, we build a graph supported only on the dataset points and learn the extension to the ambient space using supervised learning.

The new framework is very flexible and uses partitioning and learning in a black-box way. This allows us to plug various models (linear models, neural networks, etc.) and explore the trade-off between the quality and the algorithmic efficiency of the resulting partitions. We emphasize the importance of *balanced* partitions for the indexing problem, where all bins contain roughly the same number of data points. This property is crucial in the distributed setting, since we naturally would like to assign a similar number of points to each machine. Furthermore, balanced partitions allow tighter control of the number of candidates simply by varying the number of retrieved parts. Note that a priori, it is unclear how to partition $\mathbb{R}^d$ so as to induce balanced bins of a given dataset. Here the combinatorial portion of our approach is particularly useful, as balanced graph partitioning is a well-studied problem, and our supervised extension to $\mathbb{R}^d$ naturally preserves the balance by virtue of attaining high training accuracy.

We speculate that the new method might be potentially useful for solving the NNS problem for *non-Euclidean* metrics, such as the edit distance [ZZ17] or optimal transport distance [KSKW15]. Indeed, for any metric space, one can compute the k-NN graph and then partition it. The only step that needs to be adjusted to the specific metric at hand is the learning step.

Let us finally put forward the challenge of scaling our method up to billion-sized or even larger datasets. For such scale, one needs to build an *approximate* k-NN graph as well as to use graph partitioning algorithms that are faster than KaHIP. We leave this exciting direction to future work.

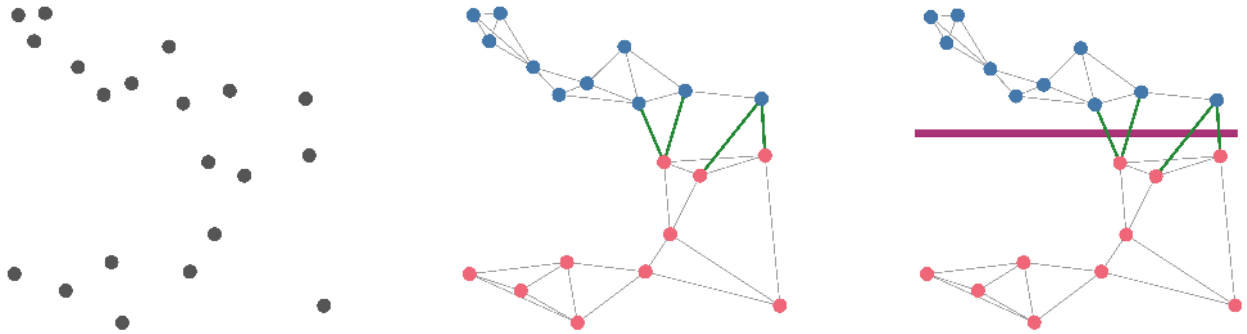
Evaluation We instantiate our framework with the KaHIP algorithm [SS13] for the graph partitioning step, and either linear models or small-size neural networks for the learning step. We evaluate it on several standard benchmarks for NNS [ABF17] and conclude that in terms of quality of the resulting partitions, it consistently outperforms quantization-based and tree-based partitioning procedures, while maintaining comparable algorithmic efficiency. In the high accuracy regime, our framework yields partitions that lead to processing up to 2.3× fewer candidates than alternative approaches.

As a baseline method we use k-means clustering [JDS11]. It produces a partition of the dataset into k bins, in a way that naturally extends to all of $\mathbb{R}^d$, by assigning a query point q to its nearest centroid. (More generally, for multi-probe querying, we can rank the bins by the distance of their centroids to q). This simple scheme produces very high-quality results for indexing.

1.2 Related work

On the empirical side, currently the fastest indexing techniques for the NNS problem are *graph-based* [MY18]. The high-level idea is to construct a graph on the dataset (it can be the k-NN graph, but other constructions are also possible), and then for each query perform a walk, which eventually converges to the nearest neighbor. Although very fast, graph-based approaches have suboptimal "locality of reference", which makes them less suitable for several modern architectures. For instance, this is the case when the algorithm is run on a GPU [JDJ17], or when the data is stored in external memory [SWQ+14] or in a distributed manner [BGS12, NCB17]. Moreover, graph-based indexing requires many rounds of adaptive access to the dataset, whereas partition-based indexing accesses the dataset in one shot. This is crucial, for example, for nearest neighbor search over encrypted data [CCD+19]. These benefits justify further study of partition-based methods.

Machine learning techniques are particularly useful for the *sketching* approach, leading to a vast



(a) Dataset

(b) k-NN graph together with a balanced partition

(c) Learned partition

Figure 1: Stages of our framework

body of research under the label "learning to hash" [WSSJ14, WLKC16]. In particular, several recent works employed neural networks to obtain high-quality sketches [LLW+15, SDSJ19]. The fundamental difference from our work is that sketching is designed to speed up *linear scans* over the dataset, by reducing the *cost* of distance evaluation, while indexing is designed for *sublinear time* searches, by reducing the *number* of distance evaluations. We highlight the work [SDSJ19], which uses neural networks to learn a mapping $f : \mathbb{R}^d \to \mathbb{R}^{d'}$ that improves the geometry of the dataset and the queries to facilitate subsequent sketching. It is natural to apply the same family of maps for *partitions*; however, as our experiments show, in the high accuracy regime the maps learned using the algorithm of [SDSJ19] consistently degrade the quality of partitions.

Prior work [CD07] has used learning to tune the parameters of certain structured classes of partitions, such as KD-trees or rectilinear LSH. This is substantially different from our method, which learns a much more general class of partitions, whose only structural constraint stems from the chosen learning component—say, the class of space partitions that can be learned by SVM, a neural network, and so on.

2 Our method

Training Given a dataset $P \subseteq \mathbb{R}^d$ of n points, and a number of bins $m > 0$, our goal is to find a partition $\mathcal{R}$ of $\mathbb{R}^d$ into m bins with the following properties:

1. *Balanced:* The number of data points in each bin is not much larger than n/m.
2. *Locality sensitive:* For a typical query point $q \in \mathbb{R}^d$, most of its nearest neighbors belong to the same bin of $\mathcal{R}$. We assume that queries and data points come from similar distributions.
3. *Simple:* The partition should admit a compact description and, moreover, the point location process should be computationally efficient. For example, we might look for a space partition induced by hyperplanes.

First, suppose that the query is chosen as a *uniformly random data point*, $q \sim P$. Let G be the k-NN graph of P, whose vertices are the data points, and each vertex is connected to k nearest neighbors. Then the above problem boils down to partitioning vertices of the graph G into m bins such that each bin contains roughly n/m vertices, and the number of edges crossing between different bins is as small as possible (see Figure 1(b)). This *balanced graph partitioning* problem is

extremely well-studied, and there are available combinatorial partitioning solvers that produce very high-quality solutions. In our implementation, we use the open-source solver KaHIP [SS13], which is based on a sophisticated local search.

More generally, we need to handle out-of-sample queries, i.e., which are not contained in P. Let $\widetilde{\mathcal{R}}$ denote the partition of G (equivalently, of the dataset P) found by the graph partitioner. To convert $\widetilde{\mathcal{R}}$ into a solution to our problem, we need to extend it to a partition $\mathcal{R}$ of the whole space $\mathbb{R}^d$ that would work well for query points. In order to accomplish this, we train a model that, given a query point $q \in \mathbb{R}^d$, predicts which of the m bins of $\widetilde{\mathcal{R}}$ the point q belongs to (see Figure 1(c)). We use the dataset P as a training set, and the partition $\widetilde{\mathcal{R}}$ as the labels – i.e., each data point is labeled with the ID of the bin of $\widetilde{\mathcal{R}}$ containing it. The geometric intuition for this learning step is that – even though the partition $\widetilde{\mathcal{R}}$ is obtained by combinatorial means, and in principle might consist of ill-behaved subsets of $\mathbb{R}^d$ – in most practical scenarios, we actually expect it to be close to being induced by a simple partition of the ambient space. For example, if the dataset is fairly well-distributed on the unit sphere, and the number of bins is $m = 2$, a balanced cut of G should be close to a hyperplane.

The choice of model to train depends on the desired properties of the output partition $\mathcal{R}$. For instance, if we are interested in a hyperplane partition, we can train a linear model using SVM or regression. In this paper, we instantiate the learning step with both *linear models* and *small-sized neural networks.* Here, there is natural tension between the size of the model we train and the accuracy of the resulting classifier, and hence the quality of the partition we produce. A larger model yields better NNS accuracy, at the expense of computational efficiency. We discuss this more in Section 3.

Multi-probe querying Given a query point q, the trained model can be used to assign it to a bin of a partition $\mathcal{R}$, and search for nearest neighbors within the data points in that part. In order to achieve high search accuracy, we actually train the model to predict *several* bins for a given query point, which are likely to contain nearest neighbors. For neural networks, this can be done naturally by taking several largest outputs of the last layer. By searching through more bins (in the order of preference predicted by the model) we can achieve better accuracy, allowing for a trade-off between computational resources and accuracy.

Hierarchical partitions When the required number of bins m is large, in order to improve the efficiency of the resulting partition, it pays off to produce it in a hierarchical manner. Namely, we first find a partition of $\mathbb{R}^d$ into m_1 bins, then recursively partition each of the bins into m_2 bins, and so on, repeating the partitioning for L levels. The total number of bins in the overall partition is $m = m_1 \cdot m_2 \cdot \ldots m_L$. See Figure 7 in the appendix for illustration. The advantage of such a hierarchical partition is that it is much simpler to navigate than a one-shot partition with m bins.

2.1 Neural LSH

In one instantiation of the supervised learning component, we use neural networks with a small number of layers and constrained hidden dimensions. The exact parameters depend on the size of the training set, and are specified in the next section.

Soft labels In order to support effective multi-probe querying, we need to infer not just the bin that contains the query point, but rather a *distribution* over bins that are likely to contain this point and its neighbors. A T-probe candidate list is then formed from all data points in the T most likely bins.

In order to accomplish this, we use *soft labels* for data points generated as follows. For $S \geq 1$ and a data point p, the soft label $\mathcal{P} = (p_1, p_2, \ldots, p_m)$ is a distribution over the bin containing a point chosen uniformly at random among S nearest neighbors of p (including p itself). Now,

for a predicted distribution $\mathcal{Q} = (q_1, q_2, \ldots, q_m)$, we seek to minimize the KL divergence between $\mathcal{P}$ and $\mathcal{Q}$: $\sum_{i=1}^{m} p_i \log \frac{p_i}{q_i}$. Intuitively, soft labels help guide the neural network with information about multiple bin ranking. S is a hyperparameter that needs to be tuned. We study its setting in Section 3.4.

3 Experiments

Datasets For the experimental evaluation, we use three standard ANN benchmarks [ABF17]: SIFT (image descriptors, 1M 128-dimensional points), GloVe (word embeddings [PSM14], approximately 1.2M 100-dimensional points, normalized), and MNIST (images of digits, 60K 784-dimensional points). All three datasets come with 10 000 query points, which we use for evaluation. We include the results for SIFT and GloVe in the main text, and MNIST in Appendix A.

Evaluation metrics We mainly investigate the trade-off between the number of candidates generated for a query point, and the k-NN accuracy, defined as the fraction of its k nearest neighbors that are among those candidates. The number of candidates determines the processing time of an individual query. Over the entire query set, we report both the *average* as well as the 0.95-*th quantile* of the number of candidates. The former measures the *throughput*[1] of the data structure, while the latter measures its *latency*.[2] We mostly focus on parameter regimes that lead to k-NN accuracy of at least 0.8. In virtually all of our experiments, $k = 10$.

Methods evaluated We evaluate two variants of our method, corresponding to two different choices of the supervised learning component in our framework.

- **Neural LSH:** In this variant we use small neural networks. Their exact architecture is detailed in the next section. We compare Neural LSH to partitions obtained by k-means clustering. As mentioned in Section 1, this method produces high quality partitions of the dataset that naturally extend to all of $\mathbb{R}^d$, and other existing methods we have tried (such as LSH) did not match its performance. We evaluate partitions into 16 bins and 256 bins. We test both one-level (non-hierarchical) and two-level (hierarchical) partitions. Queries are multi-probe.
- **Regression LSH:** This variant uses logistic regression as the supervised learning component and, as a result, produces very simple partitions induced by *hyperplanes.* We compare this method with PCA trees [Spr91, KZN08, AAKK14], random projection trees [DS13], and recursive bisections using 2-means clustering. We build trees of hierarchical bisections of depth up to 10 (thus, the total number of leaves is up to 1024). The query procedure descends a single root-to-leaf path and returns the candidates in that leaf.

3.1 Implementation details

Neural LSH uses a fixed neural network architecture for the top-level partition, and a fixed architecture for all second-level partitions. Both architectures consist of several blocks, where each block is a fully-connected layer + batch normalization [IS15] + ReLU activations. The final block is followed by a fully-connected layer and a softmax layer. The resulting network predicts a distribution over the bins of the partition. The only difference between the top-level network the second-level network architecture is their number of blocks (b) and the size of their hidden layers (s). In the top-level network we use $b = 3$ and $s = 512$. In the second-level networks we use $b = 2$ and $s = 390$. To reduce overfitting, we use dropout with probability 0.1 during training. The networks are trained

[1] Number of queries per second.

[2] Maximum time per query, modulo a small fraction of outliers.

using the Adam optimizer [KB15] for under 20 epochs on both levels. We reduce the learning rate multiplicatively at regular intervals. We use the Glorot initialization to generate the initial weights. To tune soft labels, we try different values of S between 1 and 120.

We evaluate two settings for the number of bins in each level, $m = 16$ and $m = 256$ (leading to a total number of bins of the total number of bins in the two-level experiments are $16^2 = 256$ and $256^2 = 65\,536$, respectively). In the two-level setting with $m = 256$ the bottom level of Neural LSH uses k-means instead of a neural network, to avoid overfitting when the number of points per bin is tiny. The other configurations (two-levels with $m = 16$ and one-level with either $m = 16$ or $m = 256$) we use Neural LSH at all levels.

3.2 Comparison with k-means

Figure 2 shows the empirical comparison of Neural LSH with k-means. The points listed are those that attained an accuracy ≥ 0.8. We note that two-level partitioning with $m = 256$ is the best performing configuration of k-means, for both SIFT and GloVe.[3] Thus we evaluate the baseline at its optimal performance. However, if one wishes to use partitions to split points across machines to build a distributed NNS data structure, then a single-level settings seems to be more suitable.

In all settings considered, Neural LSH yields consistently better partitions than k-means. Depending on the setting, k-means requires significantly more candidates to achieve the same accuracy:

- Up to 117% more for the average number of candidates for GloVe;
- Up to 130% more for the 0.95-quantiles of candidates for GloVe;
- Up to 18% more for the average number of candidates for SIFT;
- Up to 34% more for the 0.95-quantiles of candidates for SIFT;

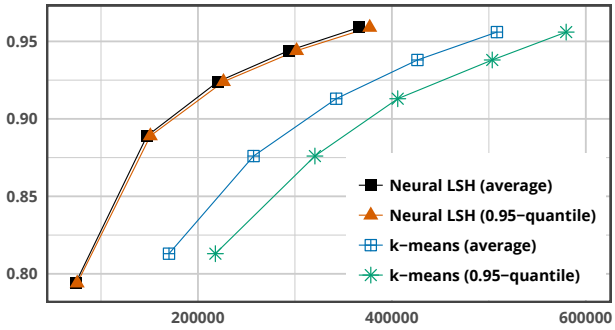
Figure 8 in the appendix lists the largest multiplicative advantage in the number of candidates of Neural LSH compared to k-means, for accuracy values of at least 0.85. Specifically, for every configuration of k-means, we compute the ratio between the number of candidates in that configuration and the number of candidates of Neural LSH in its optimal configuration, among those that attained at least the same accuracy as that k-means configuration.

We also note that in all settings except two-level partitioning with $m = 256$,[4] Neural LSH produces partitions for which the 0.95-quantiles for the number of candidates are very close to the average number of candidates, which indicates very little variance between query times over different query points. In contrast, the respective gap in the partitions produced by k-means is much larger, since unlike Neural LSH, it does not directly favor balanced partitions. This implies that Neural LSH might be particularly suitable for latency-critical NNS applications.

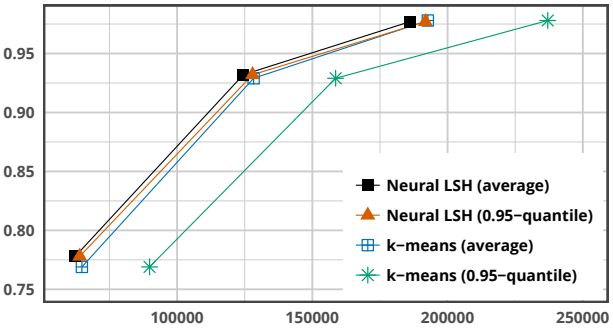
Model sizes. The largest model size learned by Neural LSH is equivalent to storing about ≈ 5700 points for SIFT, or ≈ 7100 points for GloVe.This is considerably larger than k-means with $k \leq 256$, which stores at most 256 points. Nonetheless, we believe the larger model size is acceptable for Neural LSH, for the following reasons. First, in most of the NNS applications, especially for the distributed setting, the bottleneck in the high accuracy regime is the memory accesses needed to retrieve candidates and the further processing (such as distance computations, exact or approximate). The model size is not a hindrance as long as does not exceed certain reasonable limits (e.g., it should

[3] In terms of the minimum number of candidates that attains 0.9 accuracy.

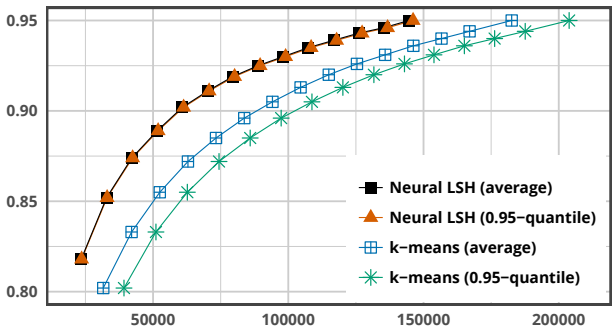
[4] As mentioned earlier, in this setting Neural LSH uses k-means at the second level, due to the large overall number of bins compared to the size of the datasets. This explains why the gap between the average and the 0.95-quantile number of candidates of Neural LSH is larger for this setting.



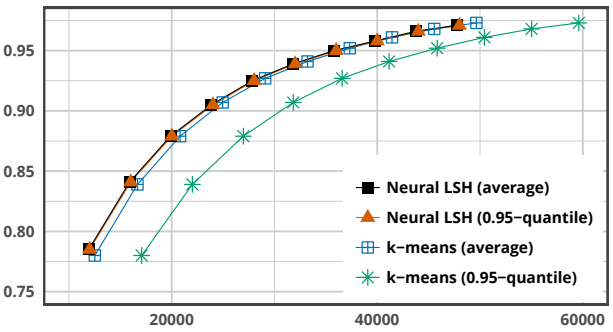
(a) GloVe, one level, 16 bins



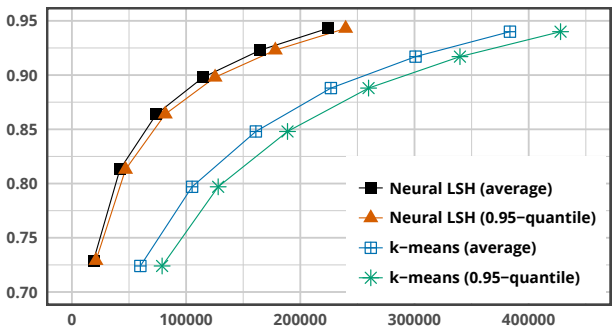
(b) SIFT, one level, 16 bins



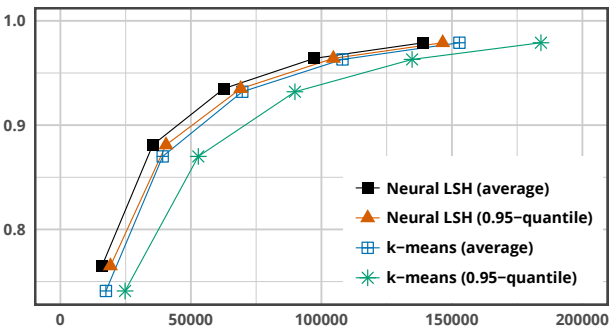
(c) GloVe, one level, 256 bins



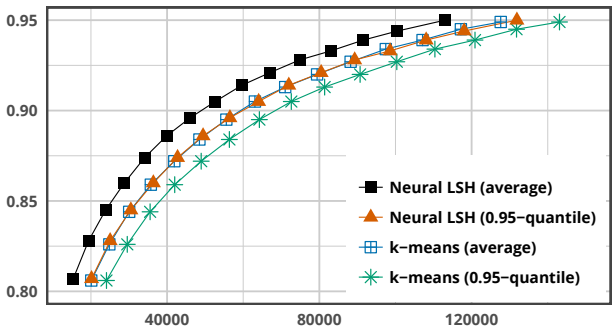
(d) SIFT, one level, 256 bins



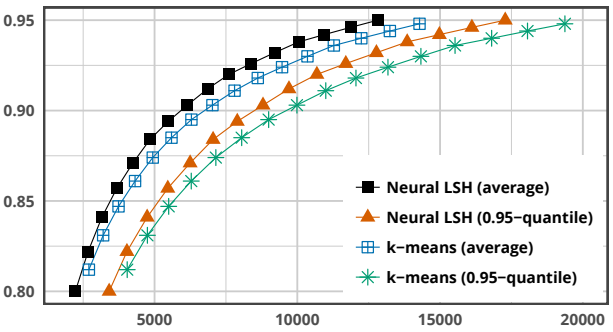
(e) GloVe, two levels, 16 bins



(f) SIFT, two levels, 16 bins



(g) GloVe, two levels, 256 bins, k-means at 2nd level



(h) SIFT, two levels, 256 bins, k-means at 2nd level

Figure 2: Comparison of Neural LSH with k-means; x-axis is the number of candidates, y-axis is the 10-NN accuracy

fit into a CPU cache). Neural LSH significantly reduces the memory access cost, while increasing the model size by an acceptable amount. Second, we have observed that the quality of the Neural LSH partitions is not too sensitive to decreasing the sizes the hidden layers. The model sizes we report are, for the sake of concreteness, the largest ones that still lead to improved performance. Larger models do not increase the accuracy, and sometimes decrease it due to overfitting.

3.3 Comparison with tree-based methods

Next we compare binary decision trees, where in each tree node a *hyperplane* is used to determine which of the two subtrees to descend into. We generate hyperplanes via multiple methods: Regression LSH, cutting the dataset into two equal halves along the top PCA direction [Spr91, KZN08], 2-means clustering, and random projections of the centered dataset [DS13, KS18]. We build trees of depth up to 10, which corresponds to hierarchical partitions with the total number of bins up to $2^{10} = 1024$. We summarize the results for GloVe and SIFT datasets in Figure 9 (see appendix). For random projections, we run each configuration 30 times and average the results.

For GloVe, Regression LSH significantly outperforms 2-means, while for SIFT, Regression LSH essentially matches 2-means in terms of the *average* number of candidates, but shows a noticeable advantage in terms of the 0.95-percentiles. In both instances, Regression LSH significantly outperforms PCA tree, and all of the above methods dramatically improve upon random projections.

Note, however, that random projections have an additional benefit: in order to boost search accuracy, one can simply repeat the sampling process several times and generate an ensemble of decision trees instead of a single tree. This allows making each individual tree relatively deep, which decreases the overall number of candidates, trading space for query time. Other considered approaches (Regression LSH, 2-means, PCA tree) are inherently deterministic, and boosting their accuracy requires more care: for instance, one can use partitioning into blocks as in [JDS11], or alternative approaches like [KS18]. Since we focus on individual partitions and not ensembles, we leave this issue out of the scope.

3.4 Additional experiments

We perform several additional experiments that we describe in a greater detail in the appendix.

1. We evaluate the 50-NN accuracy of Neural LSH when the partitioning step is run on either the 10-NN or the 50-NN graph.[5] Both settings outperform k-means, and the gap between using the 10-NN and 50-NN graphs is negligible, which indicates the robustness of Neural LSH.
2. We show that effect of tuning the size of soft labels S. We show that setting S to be at least 15 is immensely beneficial compared to $S = 1$, but beyond that we start observing diminishing returns.
3. We evaluate the effect of Neural Catalyzer [SDSJ19] on the partitions produced by k-means.

4 Conclusions and future directions

We presented a new technique for finding partitions of $\mathbb{R}^d$ which support high-performance indexing for sublinear-time NNS. It proceeds in two major steps: (1) We perform a combinatorial balanced partitioning of the k-NN graph of the dataset; (2) We extend the resulting partition to the whole ambient space $\mathbb{R}^d$ by using supervised classification (such as logistic regression, neural networks,

[5] Neural LSH can solve k-NNS by partitioning the k'-NN graph, for any k, k'; they do not have to be equal.

etc.). Our experiments show that the new approach consistently outperforms quantization-based and tree-based partitions. There is a number of exciting open problems we would like to highlight:

- Can we use our approach for NNS over *non-Euclidean* geometries, such as the edit distance [ZZ17] or the optimal transport distance [KSKW15]? The graph partitioning step directly carries through, but the learning step may need to be adjusted.
- Can we jointly optimize a graph partition *and* a classifier at the same time? By making the two components aware of each other, we expect the quality of the resulting partition of $\mathbb{R}^d$ to improve.
- Can our approach be extended to learning *several* high-quality partitions that complement each other? Such an ensemble might be useful to trade query time for memory usage [ALRW17].
- Can we use machine learning techniques to improve *graph-based* indexing techniques [MY18] for NNS? (This is in contrast to partition-based indexing, as done in this work).
- Our framework is an example of combinatorial tools aiding "continuous" learning techniques. A more open-ended question is whether other problems can benefit from such symbiosis.

References

[AAKK14] Amirali Abdullah, Alexandr Andoni, Ravindran Kannan, and Robert Krauthgamer, *Spectral approaches to nearest neighbor search*, arXiv preprint arXiv:1408.0751 (2014).

[ABF17] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull, *Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms*, International Conference on Similarity Search and Applications, Springer, 2017, pp. 34–49.

[AIL+15] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt, *Practical and optimal lsh for angular distance*, Advances in Neural Information Processing Systems, 2015, pp. 1225–1233.

[AIR18] Alexandr Andoni, Piotr Indyk, and Ilya Razenshteyn, *Approximate nearest neighbor search in high dimensions*, arXiv preprint arXiv:1806.09823 (2018).

[ALRW17] Alexandr Andoni, Thijs Laarhoven, Ilya Razenshteyn, and Erik Waingarten, *Optimal hashing-based time-space trade-offs for approximate near neighbors*, Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 2017, pp. 47–66.

[ANN+18a] Alexandr Andoni, Assaf Naor, Aleksandar Nikolov, Ilya Razenshteyn, and Erik Waingarten, *Data-dependent hashing via nonlinear spectral gaps*, Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (2018), 787–800.

[ANN+18b] ———, *Hölder homeomorphisms and approximate nearest neighbors*, 2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS), IEEE, 2018, pp. 159–169.

[BCG05] Mayank Bawa, Tyson Condie, and Prasanna Ganesan, *Lsh forest: self-tuning indexes for similarity search*, Proceedings of the 14th international conference on World Wide Web, ACM, 2005, pp. 651–660.

[BDSV18] Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik, *Learning to branch*, International Conference on Machine Learning, 2018.

[BGS12] Bahman Bahmani, Ashish Goel, and Rajendra Shinde, *Efficient distributed locality sensitive hashing*, Proceedings of the 21st ACM international conference on Information and knowledge management, ACM, 2012, pp. 2174–2178.

[BJPD17] Ashish Bora, Ajil Jalal, Eric Price, and Alexandros G Dimakis, *Compressed sensing using generative models*, International Conference on Machine Learning, 2017, pp. 537–546.

[BL12] Artem Babenko and Victor Lempitsky, *The inverted multi-index*, Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on, IEEE, 2012, pp. 3069–3076.

[BLS+16] Luca Baldassarre, Yen-Huan Li, Jonathan Scarlett, Baran Gözcü, Ilija Bogunovic, and Volkan Cevher, *Learning-based compressive subsampling*, IEEE Journal of Selected Topics in Signal Processing **10** (2016), no. 4, 809–822.

[BW18] Aditya Bhaskara and Maheshakya Wijewardena, *Distributed clustering via lsh based data partitioning*, International Conference on Machine Learning, 2018, pp. 569–578.

[CCD+19] Hao Chen, Ilaria Chillotti, Yihe Dong, Oxana Poburinnaya, Ilya Razenshteyn, and M Sadegh Riazi, *Sanns: Scaling up secure approximate k-nearest neighbors search*, arXiv preprint arXiv:1904.02033 (2019).

[CD07] Lawrence Cayton and Sanjoy Dasgupta, *A learning framework for nearest neighbor search*, Advances in Neural Information Processing Systems, 2007, pp. 233–240.

[DKZ+17] Hanjun Dai, Elias Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song, *Learning combinatorial optimization algorithms over graphs*, Advances in Neural Information Processing Systems, 2017, pp. 6351–6361.

[DS13] Sanjoy Dasgupta and Kaushik Sinha, *Randomized partition trees for exact nearest neighbor search*, Conference on Learning Theory, 2013, pp. 317–337.

[DSN17] Sanjoy Dasgupta, Charles F Stevens, and Saket Navlakha, *A neural algorithm for a fundamental computing problem*, Science **358** (2017), no. 6364, 793–796.

[IS15] Sergey Ioffe and Christian Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, arXiv preprint arXiv:1502.03167 (2015).

[JDJ17] Jeff Johnson, Matthijs Douze, and Hervé Jégou, *Billion-scale similarity search with gpus*, arXiv preprint arXiv:1702.08734 (2017).

[JDS11] Herve Jégou, Matthijs Douze, and Cordelia Schmid, *Product quantization for nearest neighbor search*, IEEE transactions on pattern analysis and machine intelligence **33** (2011), no. 1, 117–128.

[KB15] Diederik Kingma and Jimmy Ba, *Adam: A method for stochastic optimization*, International Conference for Learning Representations, 2015.

[KBC+18] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis, *The case for learned index structures*, Proceedings of the 2018 International Conference on Management of Data, ACM, 2018, pp. 489–504.

[KS18] Omid Keivani and Kaushik Sinha, *Improved nearest neighbor search using auxiliary information and priority functions*, International Conference on Machine Learning, 2018, pp. 2578–2586.

[KSKW15] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger, *From word embeddings to document distances*, International Conference on Machine Learning, 2015, pp. 957–966.

[KZN08] Neeraj Kumar, Li Zhang, and Shree Nayar, *What is a good nearest neighbors algorithm for finding similar patches in images?*, European conference on computer vision, Springer, 2008, pp. 364–378.

[LCY+17] Jinfeng Li, James Cheng, Fan Yang, Yuzhen Huang, Yunjian Zhao, Xiao Yan, and Ruihao Zhao, *Losha: A general framework for scalable locality sensitive hashing*, Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, 2017, pp. 635–644.

[LJW+07] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li, *Multi-probe lsh: efficient indexing for high-dimensional similarity search*, Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment, 2007, pp. 950–961.

[LLW+15] Venice Erin Liong, Jiwen Lu, Gang Wang, Pierre Moulin, and Jie Zhou, *Deep hashing for compact binary codes learning*, Proceedings of the IEEE conference on computer vision and pattern recognition, 2015, pp. 2475–2483.

[LV18] Thodoris Lykouris and Sergei Vassilvitskii, *Competitive caching with machine learned advice*, International Conference on Machine Learning, 2018.

[Mit18] Michael Mitzenmacher, *A model for learned bloom filters and optimizing by sandwiching*, Advances in Neural Information Processing Systems, 2018.

[MMB17] Chris Metzler, Ali Mousavi, and Richard Baraniuk, *Learned d-amp: Principled neural network based compressive image recovery*, Advances in Neural Information Processing Systems, 2017, pp. 1772–1783.

[MPB15] Ali Mousavi, Ankit B Patel, and Richard G Baraniuk, *A deep learning approach to structured signal recovery*, Communication, Control, and Computing (Allerton), 2015 53rd Annual Allerton Conference on, IEEE, 2015, pp. 1336–1343.

[MY18] Yury A Malkov and Dmitry A Yashunin, *Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs*, IEEE transactions on pattern analysis and machine intelligence (2018).

[NCB17] Y Ni, K Chu, and J Bradley, *Detecting abuse at scale: Locality sensitive hashing at uber engineering*, 2017.

[PSK18] Manish Purohit, Zoya Svitkina, and Ravi Kumar, *Improving online algorithms via ml predictions*, Advances in Neural Information Processing Systems, 2018, pp. 9661–9670.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher Manning, *Glove: Global vectors for word representation*, Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), 2014, pp. 1532–1543.

[SDSJ19] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Herve Jégou, *Spreading vectors for similarity search*, International Conference on Learning Representations, 2019.

[Spr91] Robert F Sproull, *Refinements to nearest-neighbor searching ink-dimensional trees*, Algorithmica **6** (1991), no. 1-6, 579–589.

[SS13] Peter Sanders and Christian Schulz, *Think Locally, Act Globally: Highly Balanced Graph Partitioning*, Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13), LNCS, vol. 7933, Springer, 2013, pp. 164–175.

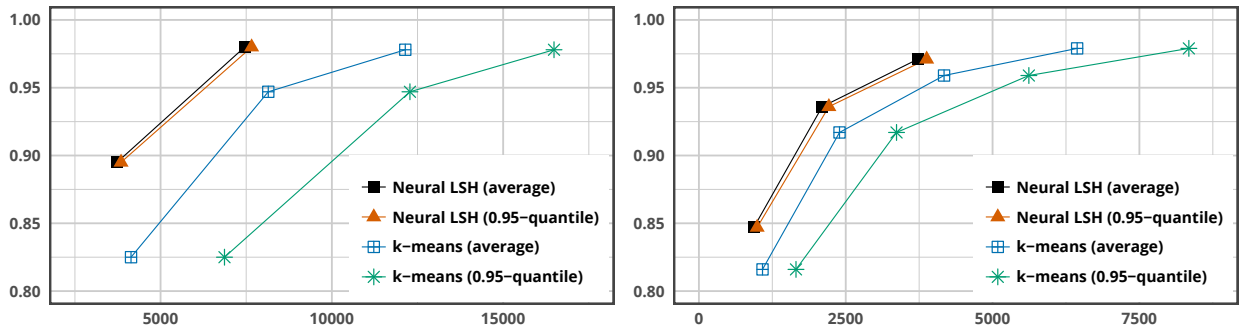
[SWQ+14] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin, *Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index*, Proceedings of the VLDB Endowment **8** (2014), no. 1, 1–12.

[WGS+17] Xiang Wu, Ruiqi Guo, Ananda Theertha Suresh, Sanjiv Kumar, Daniel N Holtmann-Rice, David Simcha, and Felix Yu, *Multiscale quantization for fast similarity search*, Advances in Neural Information Processing Systems, 2017, pp. 5745–5755.

[WLKC16] Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang, *Learning to hash for indexing big data - a survey*, Proceedings of the IEEE **104** (2016), no. 1, 34–57.

[WSSJ14] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji, *Hashing for similarity search: A survey*, arXiv preprint arXiv:1408.2927 (2014).

[ZZ17] Haoyu Zhang and Qin Zhang, *Embedjoin: Efficient edit similarity joins via embeddings*, Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2017, pp. 585–594.



(a) One level, 16 bins

(b) Two levels, 16 bins

Figure 3: MNIST, comparison of Neural LSH with k-means; x-axis is the number of candidates, y-axis is the 10-NN accuracy.

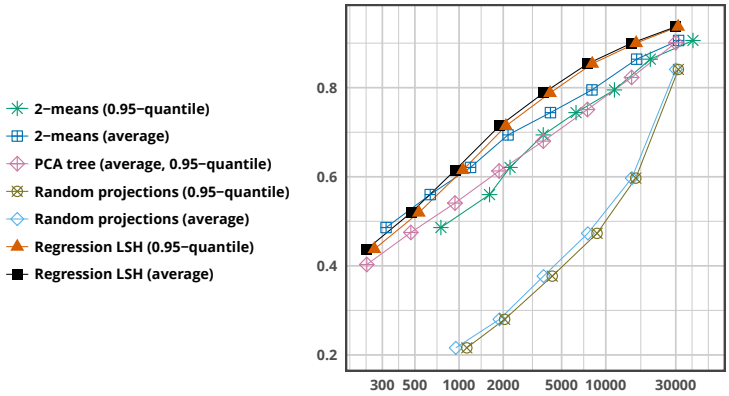


Figure 4: Comparison of decision trees built from hyperplanes; x-axis is the number of candidates, y-axis is the 10-NN accuracy

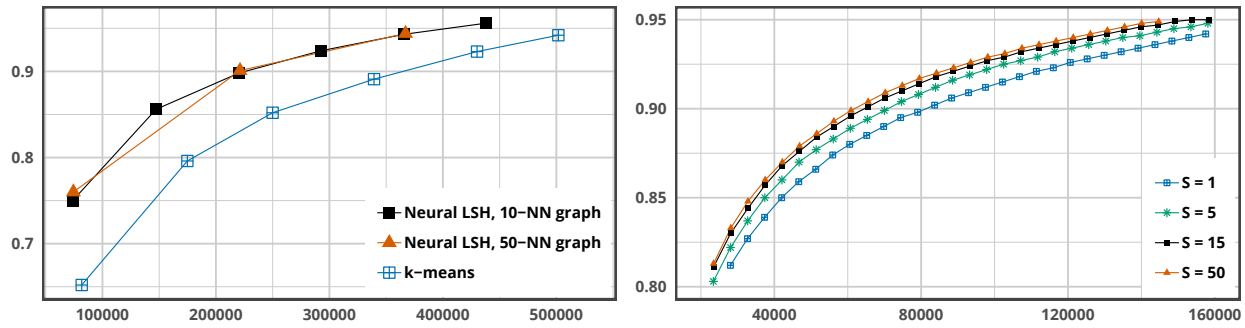
A Results for MNIST

We include experimental results for the MNIST dataset, where all the experiments are performed exactly in the same way as for SIFT and GloVe. Consistent with the trend we observed for SIFT and GloVe, Neural LSH consistently outperforms k-means (see Figure 3) both in terms of average number of candidates and especially in terms of the 0.95-th quantiles. We also compare Regression LSH with recursive 2-means, as well as PCA tree and random projections (see Figure 4), where Regression LSH consistently outperforms the other methods.

B Additional experiments

Here we describe three additional experiments that we referred to in Section 3.4 in a greater detail.

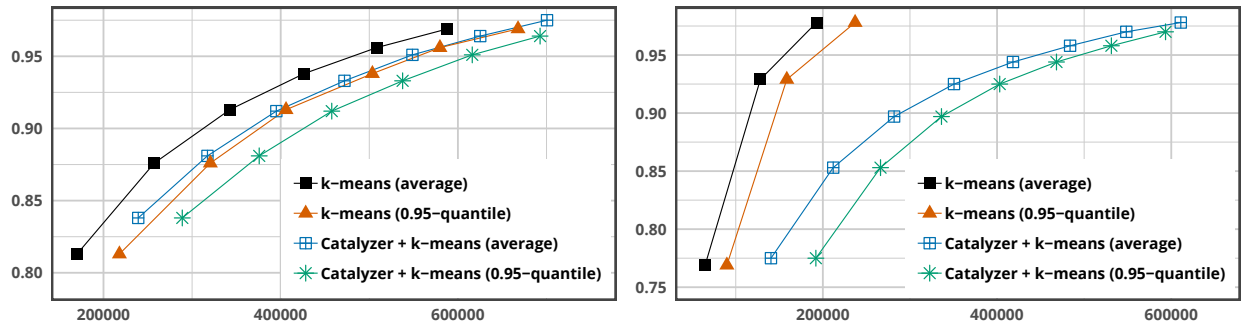
First, we compare Neural LSH and k-means for $k = 50$ (instead of the default setting of $k = 10$). Moreover, we consider two variants of Neural LSH. In one of them, we use the 50-NN graph for partitioning, but for the other variant we use merely the 10-NN graph. Figure 5a compares these three algorithms on GloVe for 16 bins reporting average numbers of candidates. From this plot, we can see that for $k = 50$, Neural LSH convincingly outperforms k-means, and whether we use 10-NN



(a) GloVe, one level, 16 bins, 50-NN accuracy using 10-NN and 50-NN graphs

(b) GloVe, one level, 256 bins, varying S

Figure 5: Effect of various hyperparameters



(a) GloVe, one level, 16 bins

(b) SIFT, one level, 16 bins

Figure 6: Comparison of k-means and Catalyzer + k-means

or 50-NN graph matters very little.

Second, we study the effect of varying S (the soft labels parameter) for Neural LSH on GloVe for 256 bins. See Figure 5b where we report the average number of candidates. As we can see from the plot, the setting $S = 15$ yields much better results compared to the vanilla case of $S = 1$. However, increasing S beyond 15 has little effect on the overall accuracy.

Finally, we compare vanilla k-means with k-means run after applying a Neural Catalyzer map [SDSJ19]. The goal is to check whether the Neural Catalyzer, which has been designed to boost up the performance of sketching methods for NNS by adjusting the input geometry, could also improve the quality of space partitions for NNS. See Figure 6 for the comparison on GloVe and SIFT with 16 bins. On both datasets (especially SIFT) Neural Catalyzer in fact degrades the quality of the partitions. We observed a similar trend for other numbers of bins than the setting reported here. These findings support our observation that while both indexing and sketching for NNS can benefit from learning-based enhancements, they are fundamentally different approaches and require different specialized techniques.

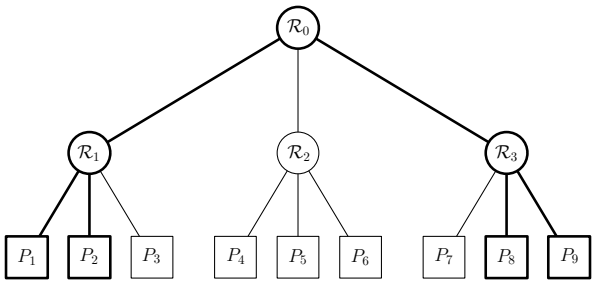


Figure 7: Hierarchical partition into 9 bins with $m_1 = m_2 = 3$. $\mathcal{R}_i$'s are partitions, P_j's are the bins of the dataset. Multi-probe query procedure, which descends into 2 bins, may visit the bins marked in bold.

		GloVe		**SIFT**	
		Averages	0.95-quantiles	Averages	0.95-quantiles
One level	16 bins	1.745	2.125	1.031	1.240
	256 bins	1.491	1.752	1.047	1.348
Two levels	16 bins	2.176	2.308	1.113	1.306
	256 bins	1.241	1.154	1.182	1.192

Figure 8: Largest ratio between the number of candidates for Neural LSH and k-means over the settings where both attain the same target 10-NN accuracy, over accuracies of at least 0.85. See details in Section 3.2.

C Additional implementation details

We slightly modify the KaHIP partitioner to make it more efficient on the k-NN graphs. Namely, we introduce a hard threshold of 2000 on the number of iterations for the local search part of the algorithm, which speeds up the partitioning dramatically, while barely affecting the quality of the resulting partitions.

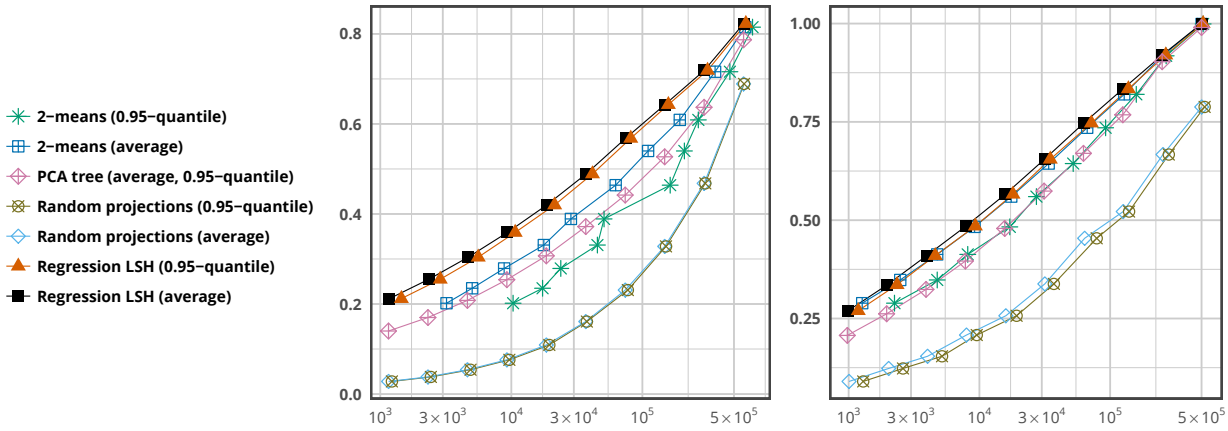


Figure 9: Comparison of decision trees built from hyperplanes: the left plot is GloVe, the right plot corresponds to SIFT; x-axis is the number of candidates, y-axis is the 10-NN accuracy