Learning Sublinear-Time Indexing for Nearest Neighbor Search

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

January 25, 2019

Abstract

Most of the efficient sublinear-time indexing algorithms for the high-dimensional nearest neighbor search problem (NNS) are based on space partitions of the ambient space \(R^d\). Inspired by recent theoretical work on NNS for general metric spaces [ANN+18a, ANN+18b], we develop a new framework for constructing such partitions that reduces 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 found by Neural LSH consistently outperform partitions found by quantization- 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 \(R^d\), we want to preprocess \(P\) to answer \(k\)-nearest neighbor queries quickly. That is, given a query point \(q \in R^d\), 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. In contrast, the goal of sketching is to compute compressed representations of points (e.g., compact binary hash codes with the Hamming distance used as an estimator [WSSJ14, WLKC16]) to enable computing approximate distances quickly. Indexing and sketching can be (and often are) combined to maximize performance [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, and focus on optimizing the trade-off between three metrics: the number of reported candidates, the fraction of the true nearest neighbors among the candidates, and the computational efficiency of the indexing data structure.

Most of the efficient indexing methods are based on space partitions (with some exceptions mentioned below). The overarching idea is to find a partition of the ambient space \(R^d\) and split the dataset \(P\) accordingly. Given a query point \(q\), we identify the part containing \(q\) and form the resulting list of candidates from the data points residing in the same part. To boost the search

*Author names are ordered alphabetically.
accuracy, it is often necessary to add all data points from nearby parts to the candidate list (this is often referred as the multi-probe technique). Some of the popular indexing methods include locality-sensitive hashing (LSH) [LJW+07, AIL+15, DSN17]; quantization-based approaches, where partitions are 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].

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 (for a small sample, see [KBC+18, BDSV18, LV18, Mit18]). Similar methods were used to improve the sketching approach to NNS (more on this below) [WLKC16]. However, when it comes to indexing, only very “rudimentary” unsupervised techniques such as PCA or k-means have been (successfully) used. 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 $\mathbb{R}^d$, by directly optimizing the objective function that quantifies the performance of a partition for the NNS problem. At a high level, our approach consists of two steps. First, we perform balanced partitioning of the k-NN graph built on the data points, where each point is connected to the k nearest neighbors. Then we train a model to solve the supervised classification task, with inputs being the data points and labels given by the partition found during the first step. (See Figure 1 for the illustration.) The resulting classifier induces a partition of the whole space $\mathbb{R}^d$, which is our end result.

The new framework has multiple benefits:

- We directly reduce the question of interest (geometric partitioning) to two well-studied problems, namely graph partitioning and supervised learning.
- Our reduction 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.
- Our framework aims to optimize an objective function that directly controls the quality of a partition (assuming the distribution of queries is similar to the distribution of the data points), whereas many of the previous methods that work well in practice (e.g., k-means) use various proxies instead.

It is important to note that our method is unsupervised; in particular, it does not require any given labeling of the input data points. Instead, we harness supervised learning to extend the solution to a finite unsupervised problem – graph partitioning on a fully observed set of points – to a space partition that generalizes to any unseen point in $\mathbb{R}^d$.

Further, we emphasize the importance of balanced partitions in the indexing problem. In a balanced partition of $\mathbb{R}^d$, all parts contain roughly the same number of data points. Unbalanced partitions lead to large variance in the number of candidates reported for different queries, leading to an unpredictable computational cost. Conversely, balanced partitions allow us to control the number of candidates by parameterizing the total number of parts in the partition as well as the number of parts probed per query. A priori, it is unclear how to partition $\mathbb{R}^d$ so as to respect the balance of a given dataset. This makes the combinatorial portion of our approach 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.

**Evaluation** We instantiate our framework with the KaHIP algorithm [SS13] for the partitioning step, and linear models and small-size neural networks for the learning part. We evaluate our approach 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 require to process up to $2.3\times$ fewer candidates than alternative approaches.

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

1.2 Related work

The new framework is inspired by a recent line of theoretical work that studies the NNS problem for general metric spaces [ANN+18a, ANN+18b]. The two relevant contributions of these works are as follows. First, they prove that graphs embedded into sufficiently “nice” metric spaces (including Euclidean space, but also many others) with short edges but without “dense regions”, must have sparse cuts. Second, for the special case of normed spaces defined on $\mathbb{R}^d$, such sparse cuts can be assumed to be induced by geometrically nice subsets of the ambient $d$-dimensional space. This is directly related to the method developed in the present paper, where the starting point is a sparse (multi-)cut in a graph embedded into $\mathbb{R}^d$, which is then “deformed” to a geometrically nice cut using supervised learning.

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 the data is stored in external memory [SWQ+14].) This justifies further study of the partition-based methods.

Machine learning techniques are particularly useful for the sketching approach, leading to a vast 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. Note that in principle, one could use sketches to generate space partitions, since a $t$-bit sketch induces a partition of $\mathbb{R}^d$ into $2^t$ parts. However, this is a substantially different use of sketches than the one intended in the above mentioned works. Indeed, we observed that partitions induced by high-quality sketching techniques do not perform well compared to, say, quantization-based partitions.

A different application of neural networks related to NNS is to optimize the performance of the nearest neighbor classifier. Given a labeled dataset in a classification setting, the idea is to learn a representation of the dataset – either as sketches [KW17, JZPG17] or as a high-dimensional embedding [ST18] – that would render the nearest neighbor classifier (i.e., labeling each query point with the label of its nearest data point) as accurate as possible. Apart from not producing an
indexing method, these works are also different from ours by being inherently supervised, relying on a fully labeled dataset, whereas our approach is unsupervised.

2 Our method

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

1. Balanced: The number of data points in each part 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 part of $\mathcal{R}$. We assume that queries and data points come from similar distributions.

3. Simple: The partition should admit a compact description. 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 the graph $G$ into $m$ parts such that each part contains roughly $n/m$ vertices, and the number of edges crossing between different parts 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].

More generally, we need to handle out-of-sample queries, i.e., which are not contained in $P$. Let $\overline{\mathcal{R}}$ denote the partition of $G$ (equivalently, of the dataset $P$) found by the graph partitioner. To convert $\overline{\mathcal{R}}$ into a solution to our problem, we need to extend it to a “simple” 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 $m$ parts of $\overline{\mathcal{R}}$ the point $q$ belongs to (see Figure 1(c)). We use the dataset $P$ as a training set, and the partition $\overline{\mathcal{R}}$ as the labels – i.e., each data point is labeled with the ID of the part of $\overline{\mathcal{R}}$ containing it. The geometric intuition for this learning step is that – even though the partition $\mathcal{R}$ is obtained by combinatorial means, and in principle might consist of ill-behaved subsets of $\mathbb{R}^d$ – in most practical scenarios, we actually
Figure 2: Hierarchical partition into 9 parts with $m_1 = m_2 = 3$. $\mathcal{R}_i$’s are partitions, $P_j$’s are the parts of the dataset. Multi-probe query procedure, which descends into 2 parts, may visit the parts marked in bold.

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 parts is $m = 2$, a balanced cut of $G$ should be close to a hyperplane.

The choice of model to train depends on the level of “simplicity” we wish to impose on the final 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 a 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 would yield 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 part of $\mathcal{R}$, 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 parts 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 parts (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 parts $m$ 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 $m_1$ parts, then recursively partition each of the parts into $m_2$ parts, and so on, repeating the partitioning for $L$ levels (see Figure 2 for the illustration). The total number of parts in the overall partition is $m = m_1 \cdot m_2 \cdot \ldots m_L$. The advantage of such a hierarchical partition is that it is much simpler to navigate than a one-shot partition with $m$ parts.

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 part in which the query point resides, but rather a distribution over parts 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 parts.

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 part 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 $\mathcal{Q}$: \[
\sum_{i=1}^{m} p_i \log \frac{p_i}{q_i}.
\]
The purpose of the soft labels is to guide the neural network with information about the ranking of parts for searching nearest neighbors. Optimizing w.r.t. $\mathcal{P}$ allows the model to predict multiple parts more accurately, which is necessary for achieving high accuracy via multi-probe querying.

$S$ is a hyperparameter that needs to be tuned. In practice, accuracy in the objective function increases in the regime when $S$ is noticeably larger than $k$, as more neighbors give the network a more “complete” distribution over parts.

3 Experiments

3.1 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.

3.2 Metrics used for the evaluation

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 of the data structure, while the latter measures its latency. We mostly focus on parameter regimes that lead to $k$-NN accuracy of at least 0.8. In all of our experiments, $k = 10$.

3.3 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 parts and 256 parts. We test both one-level (non-hierarchical) and two-level (hierarchical) partitions. Queries are multi-probe.

\begin{footnotesize}
1\textsuperscript{Number of queries per second.} \\
2\textsuperscript{Maximum time per query, modulo a small fraction of outliers.}
\end{footnotesize}
**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 bisecting using 2-means clustering. We build trees of hierarchical bisecting 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.

### 3.4 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 parts of the partition. The only difference between the top-level network and 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 during training. The networks are trained using the Adam optimizer [KB15] for under 20 epochs on both levels. We reduce the learning rate multiplicatively at regular intervals.

A hierarchical partition produces a tree in which each node corresponds to a partition into $m$ parts. In our experiments, we evaluate $m = 16$ and $m = 256$, thus the total number of parts in the two-level experiments are $16^2 = 256$ and $256^2 = 65,536$ respectively. In the latter case, each part contains fewer than 20 data points, which is too small for supervised learning without overfitting. Therefore, in the two-level experiment with $m = 256$, we use Neural LSH at the top-level and $k$-means clustering at the bottom level. In the other experiments (two-levels with $m = 16$ and one-level with $m \in \{16, 256\}$) we use Neural LSH at all levels.

Note that multiple partitions of $\mathbb{R}^d$ can be combined in ways other than the hierarchical approach we evaluate. For example, a common technique called Product Quantization combines multiple invocations of $k$-means in a Cartesian product fashion, over a decomposition of $\mathbb{R}^d$ into orthogonal subspaces [JDS11, BL12]. There are various techniques to tune and improve this approach [NF13, GHKS14, WGS+17]. Since the focus of our paper is to compare the quality of individual partitions, we use hierarchical partitioning as a baseline approach to combining partitions. Nonetheless, we note that the above Cartesian product approach and related ideas can be readily applied to Neural LSH as well.

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.

### 3.5 Comparison with $k$-means

Figure 3 shows the empirical comparison of Neural LSH with $k$-means. The points listed are those that attained an accuracy of at least 0.8. We note that the reported setting of two-level partitioning with $m = 256$ is the best performing configuration of $k$-means, for both SIFT and GloVe. Thus we evaluate the baseline at its optimal performance.

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.

---

3 In terms of the minimum number of candidates that attains 0.9 accuracy.
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 4 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. The table lists the maximum ratio over all accuracy values of at least 0.85.

We also note that in all settings except two-level partitioning with $m = 256$, 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 $\approx 5700$ points for SIFT, or $\approx 7100$ 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:

- In most of the NNS applications, 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 fit into a CPU cache). Neural LSH significantly reduces the memory access cost, while increasing the model size by an acceptable amount.

- 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.

### 3.6 Comparison with tree-based methods

Here 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 parts up to $2^{10} = 1024$.

### Footnotes

4As mentioned earlier, in this setting Neural LSH uses $k$-means at the second level, due to the large overall number of parts 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.

5The difference accounts for the different network architecture used for them, as well as their different dimensionality.
Figure 3: Comparison of Neural LSH with $k$-means; $x$-axis is the number of candidates, $y$-axis is the 10-NN accuracy
Figure 4: 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 3.5.

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

Figure 5: 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

We summarize the results for GloVe and SIFT datasets in Figure 5. 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: if one is willing to boost the search accuracy, it is enough to simply repeat the sampling process several times and generate an ensemble of decision trees instead of a single tree. This allows us to make 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 the accuracy requires more care: for instance, one can use partitioning into blocks in spirit of [JDS11] or see [KS18] for alternative approaches. Since we focus on individual partitions and not ensembles, we leave this issue out of the scope.

4 Conclusions and future directions

In this paper, 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:

- We start with combinatorial balanced partitioning of the \( k \)-NN graph of the dataset;
• We extend the resulting partition to the whole ambient space $\mathbb{R}^d$ by using supervised classification (such as logistic regression, neural networks, etc.).

Our experiments show that the new approach consistently outperforms quantization-based and tree-based partitions.

We believe that this study is just the first step in exploring the new partitioning approach, and 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 can potentially be used 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 there are other problems that 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. In International Conference on Similarity Search and Applications, pages 34–49. Springer, 2017.

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

[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. In Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 47–66. Society for Industrial and Applied Mathematics, 2017.

[ANN+18a] Alexandr Andoni, Assaf Naor, Aleksandar Nikolov, Ilya Razenshteyn, and Erik Waingarten. Data-dependent hashing via nonlinear spectral gaps. In Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, pages 787–800. ACM, 2018.
[ANN+18b] Alexandr Andoni, Assaf Naor, Aleksandar Nikolov, Ilya Razenshteyn, and Erik Wain- garten. Hölder homeomorphisms and approximate nearest neighbors. In 2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS), pages 159–169. IEEE, 2018.

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

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

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

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

[DSN17] Sanjoy Dasgupta, Charles F Stevens, and Saket Navlakha. A neural algorithm for a fundamental computing problem. Science, 358(6364):793–796, 2017.

[GHKS14] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. IEEE transactions on pattern analysis and machine intelligence, 36(4):744–755, 2014.

[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(1):117–128, 2011.

[JZPG17] Himalaya Jain, Joaquin Zepeda, Patrick Pérez, and Rémi Gribonval. Subic: A supervised, structured binary code for image search. In Proc. Int. Conf. Computer Vision, volume 1, page 3, 2017.

[KB15] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In 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. In Proceedings of the 2018 International Conference on Management of Data, pages 489–504. ACM, 2018.

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

[SKW15] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In International Conference on Machine Learning, pages 957–966, 2015.
[KW17] Benjamin Klein and Lior Wolf. In defense of product quantization. *arXiv preprint arXiv:1711.08589*, 2017.

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

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

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

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

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

[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.

[NF13] Mohammad Norouzi and David J Fleet. Cartesian k-means. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3017–3024, 2013.

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

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

[Spr91] Robert F Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(1-6):579–589, 1991.

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

[ST18] Ludwig Schmidt and Kunal Talwar. Learning representations for faster similarity search. 2018.

[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(1):1–12, 2014.

[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. In *Advances in Neural Information Processing Systems*, pages 5745–5755, 2017.
Figure 6: Comparison of Neural LSH with \( k \)-means; x-axis is the number of candidates, y-axis is the 10-NN accuracy.

Figure 7: Comparison of decision trees built from hyperplanes; x-axis is the number of candidates, y-axis is the 10-NN accuracy

[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(1):34–57, 2016.

[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. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 585–594. ACM, 2017.

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 6) 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 7), where Regression LSH consistently outperforms the other methods.