DEANN: Speeding up Kernel-Density Estimation using Approximate Nearest Neighbor Search

Matti Karppa[1,2] Martin Aumüller[1] Rasmus Pagh[2,3]

March 2, 2022

Abstract

Kernel Density Estimation (KDE) is a nonparametric method for estimating the shape of a density function, given a set of samples from the distribution. Recently, *locality-sensitive hashing*, originally proposed as a tool for nearest neighbor search, has been shown to enable fast KDE data structures. However, these approaches do not take advantage of the many other advances that have been made in algorithms for nearest neighbor algorithms. We present an algorithm called Density Estimation from Approximate Nearest Neighbors (DEANN) where we apply Approximate Nearest Neighbor (ANN) algorithms as a *black box* subroutine to compute an unbiased KDE. The idea is to find points that have a large contribution to the KDE using ANN, compute their contribution exactly, and approximate the remainder with Random Sampling (RS). We present a theoretical argument that supports the idea that an ANN subroutine can speed up the evaluation. Furthermore, we provide a C++ implementation with a Python interface that can make use of an arbitrary ANN implementation as a subroutine for kernel density estimation. We show empirically that our implementation outperforms state of the art implementations in all high dimensional datasets we considered, and matches the performance of RS in cases where the ANN yield no gains in performance.

[1]IT University of Copenhagen
[2]BARC
[3]University of Copenhagen

1 Introduction

Kernel Density Estimation (KDE) is a nonparametric method for estimating the shape of a density function, given a sample from the distribution. For a *dataset* $X \subseteq \mathbb{R}^d$ and a *kernel function* $K_h : \mathbb{R}^d \times \mathbb{R}^d \to [0,1]$, the kernel density estimate of the *query vector* $y \in \mathbb{R}^d$ is given by

$$\mathrm{KDE}_X(y) = \frac{1}{|X|} \sum_{x \in X} K_h(x,y) \,. \tag{1}$$

A common choice for the kernel function is the *Gaussian kernel*

$$K_h(x,y) = \exp\left(-\frac{||x-y||_2^2}{2h^2}\right) , \tag{2}$$

where the constant $h > 0$ is the *bandwidth* parameter. In the one-dimensional case, the KDE has a simple interpretation with this kernel function: given a set of points, plot a Gaussian Probability Density Function (PDF) centered at each point, and the KDE is the density function we get by taking the average of all these PDFs at each point. The bandwidth is thus the variance parameter, controlling the width of each bell curve. The KDE may thus be viewed as a generalization of the histogram with soft bins, and is routinely used for smoothing with libraries such as Seaborn.[1]

The Gaussian kernel is an example of a *radially decreasing kernel*, that is, its value depends only on the *distance* between the two operands x and y, and

[1]`https://seaborn.pydata.org/`, see particularly the function `kdeplot`.

is monotonically decreasing, exponentially so. This family includes, for example, the *exponential kernel* $K_h = \exp\left(-\frac{||x-y||_2}{h}\right)$ and the *Laplacian kernel* $K_h = \exp\left(-\frac{||x-y||_1}{h}\right)$. Other common kernels include the Epanechnikov kernel, the rectangular (or tophat) kernel, or the triangular (or linear) kernel (Silverman, 1986, Chapter 3), see also (scikit-learn developers, 2021, Section 2.8.2). Though our methods will apply to any radial kernel, we focus on the exponentially decreasing radial kernels.

The KDE is easily generalized into the multivariate case. The bandwidth may also be generalized into a cross-dimensional matrix that corresponds to the covariance matrix, but we restrict ourselves to scalar constant bandwidth. For kernels dependent only on the distance between points, the bandwidth parameter can be seen as a scaling parameter for the distances, and in practical applications, the choice of proper bandwidth is important to ensure that the KDE values are meaningful, that they show essential features of the underlying distribution without becoming overly smooth while at the same time avoiding the introduction of sampling artifacts (Jones et al., 1996). It is immediate from Equation (2) that, if we let $h \to \infty$, the contribution of each summand in Equation (1) approaches 1; conversely, if we let $h \to 0$, only the nearest neighbors have significant contribution to the sum.

The KDE has seen use in applications such as estimating gradient lines of densities (Arias-Castro et al., 2016) and outlier detection (Schubert et al., 2014). In machine learning, KDE is used in classification (Gan and Bailis, 2017).

The problem with a naïve application of Equation (1) to compute the KDE value is that the sum depends on *all* points in the dataset; that is, an individual query requires $\Omega(nd)$ operations. If the number of queries is large, this may be prohibitively expensive. An immediate improvement over the naïve summation is to use Random Sampling (RS): it can be shown that computing the KDE on a subset of $m = O(\frac{1}{\varepsilon^2\tau})$ points, sampled uniformly at random with or without repetition, yields an unbiased estimator that provides a relative $(1+\varepsilon)$-approximation guarantee on KDE values in the excess of τ, with constant probability.[2] Despite this simplicity, it has turned out to be difficult to improve on RS asymptotically whilst preserving theoretical guarantees in high dimensions (Charikar and Siminelakis, 2017).

1.1 Our contribution

In this paper:

(i) We introduce an algorithmic approach to speed up kernel density estimation using approximate nearest neighbor algorithms as a black box. We call our approach *DEANN*, for Density Estimation from Approximate Nearest Neighbors,

(ii) We provide a theoretical justification for the unbiasedness, and for the correctness and viability of our approach on real-world data, and

(iii) We report on an extensive experimental study that compares our implementation to previous state-of-the-art approaches.

All of our code is available online[3] under the MIT license, inlcuding the experimental pipeline.[4] The code includes dataset generation and preprocessing as well as post-processing of results, allowing for reproducibility and serving as a starting point for future work.

In more detail, a central idea in the attempt to speed up the evaluation of KDE sums of the form of Equation (1) is to split the sum into near and far components, depending on the distance to the dataset points from the query vector. We then wish to compute the contributions of the near points exactly, and approximate the contribution of the far away points. However, we cannot hope to retrieve the actual nearest neighbors of the query point in a high-dimensional space efficiently, so we resort to ANN and compute the exact contribution of an *approximate nearest neighbors* set. Combining ANN and random sampling naïvely does not result in an unbiased estimator, but we show how to efficiently correct for this bias. In fact, we obtain an estimator that is unbiased *regardless* of the quality of the ANN data

[2] If $\mu \geq \tau$ is the KDE value, the estimate E produced by the algorithm satisfies $\max\{E/\mu, \mu/E\} \leq (1+\varepsilon)$ with constant probability.

[3] `https://github.com/mkarppa/deann`

[4] `https://github.com/mkarppa/deann-experiments`

structure. Only the variance of the resulting estimator is affected by the quality of the nearest neighbors approximation.

In Section 3 we formally define the DEANN algorithm, prove that it is an unbiased estimator of the KDE value, and provide theoretical arguments that support the idea that (and when) nearest neigbors can help in the estimation of KDE values. In Section 4, we discuss our actual C++ implementation with a Python interface that can utilize an arbitrary ANN implementation as a black box, and show in Section 5 that the result performs well in a practical experimental setting. Due to lack of space, we have relegated some of the additional experiments into the appendix.

Limitations. While our work is very general, this generality also manifests itself in that we have so far no theoretically grounded way to choose the parameters except empirical grid search of the parameter space. Also, we are dependent on the ANN subroutine which means we cannot provide a theoretical runtime analysis for the algorithm without knowing the internals of the ANN algorithm.

1.2 Related work

Kernel density estimation. Three independent lines of research can be identified based on space-partitioning trees, data sparsification, and Locality-Sensitive Hashing (LSH). Methods based on creating a tree structure for partitioning the search space include (Gray and Moore, 2000, 2003; Lee et al., 2005; Lee and Gray, 2008; Morariu et al., 2008; Ram et al., 2009), but these methods are prone to suffer from the curse of dimensionality. An interesting development of this line of research is ASKIT (March et al., 2015) that is in some cases able to perform also with high dimensional data if the data exhibits suitable structure; the authors provide an implementation as free software.

In particular, March et al. (2015) also use the idea of splitting up the contributions of near and far points, but compute the contribution of far points in a different way. They prune the KDE computation in a tree-based space partitioning by approximating the contributions to the KDE value during a sub-tree traversal. To apply this pruning, they run a bottom-up phase in the tree construction. For each node in the tree, they look at the nearest neighbor information *among the nodes in the sub-tree* and enrich these results with random samples. From that, they can store a short summary in the node. This allows them to prune the computation at intermediate nodes in the top-down traversal for points that are guaranteed to be far away from the query. In contrast to the approach of March et al. (2015), we use simple, data-independent random sampling, which is not only faster but also has the benefit of providing an unbiased estimator.

A second line of research includes *ε-samples* or *coresets* (Phillips, 2013; Zheng et al., 2013; Phillips and Tai, 2020), subsamples of the data that offer approximation guarantees. Optimal coresets are often constructed as random samples with high-probability guarantees, and thus offer performance similar to RS.

The third line of work was initiated with the Hashing Based Estimators (HBE) of Charikar and Siminelakis (2017). They applied importance sampling to model KDE values through the collision probability of Euclidean Locality Sensitive Hashing (ELSH) (Datar et al., 2004). Follow-up work includes Hashing Based Sketches (HBS) (Siminelakis et al., 2019) that was empirically shown to outperform ASKIT, and the work of Backurs et al. (2019) who presented an improvement on the space usage. Very recently, Charikar et al. (2020) further improved the asymptotic running time and space complexity in this line of research by using data-dependent LSH (Andoni et al., 2017).

A more detailed discussion of the different methods is presented in Appendix B.

Approximate Nearest Neighbor Search. Nearest neighbor search is a key primitive in many data mining and machine learning applications. If vectors are embedded in a high-dimensional space, as is standard in computer vision (Netzer et al., 2011) or natural language processing (Pennington et al., 2014), *exact* nearest neighbor search becomes difficult, an instance of the curse of dimensionality.

A long line of research focused on providing efficient implementations to find *approximate* nearest neighbors. While these approaches often lack theoretical guarantees, they provide a large speed-up over an exact linear scan with only a small loss in accuracy on real-world data; see for example the large-scale evaluation study in Aumüller et al. (2020). Several techniques can be used to build efficient ANN systems: graph-based approaches, such as Iwasaki and Miyazaki (2018) and Malkov and Yashunin (2020), provide fast query times but are expensive in preprocessing; cluster-based techniques like Johnson et al. (2017) and Guo et al. (2020) feature faster index building times with a small loss in throughput. LSH-based approaches such as Andoni et al. (2015) and Aumüller et al. (2019) give theoretical, probabilistic guarantees on the result quality, but are often slower than the aforementioned approaches in practice.

2 Preliminaries

We write $[n] = \{0, 1, \ldots, n-1\}$. We say that a bijection $\pi\colon [n] \to [n]$ is a *permutation*.

We define the KDE problem formally as follows.

Definition 1 (Kernel Density Estimate)**.** Given a dataset $X = \{x_0, x_1, \ldots, x_{n-1}\} \subseteq \mathbb{R}^d$ of d-dimensional vectors, a constant bandwidth $h > 0$, a kernel function $K_h : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, and a query vector $y \in \mathbb{R}^d$, we say that the Kernel Density Estimate (KDE) of y is

$$\mathrm{KDE}_X(y) = \frac{1}{n} \sum_{i=0}^{n-1} K_h(x_i, y)\,.$$

We often write $\mu = \mathrm{KDE}_X(y)$ when y, X, h, and K_h are clear from the context.

We call kernels that are monotonically decreasing functions of the distance between a pair of points radially decreasing. If the kernel K_h is a function of the Euclidean distance of the pair of points, such as the Gaussian or exponential kernels, we say it is *Euclidean*.

Given the dataset $X \subseteq \mathbb{R}^d$ and a query vector $y \in \mathbb{R}^d$, we denote with $(x'_0, \ldots, x'_{n-1})$ the sequence of dataset vectors sorted by distance to y.

We say that a random variable Z is an unbiased estimator of μ if $E[Z] = \mu$. We present the following well-known result that the KDE can be efficiently approximated with random sampling.

Lemma 2 (Random Sampling)**.** *Let $X \subseteq \mathbb{R}^d, y \in \mathbb{R}^d$. Let $\tau \in (0,1)$ such that $\mathrm{KDE}_X(y) \geq \tau$. Drawing a uniform random sample $X' \subseteq X$ (with repetition) of size $m = O(\frac{1}{\varepsilon^2 \tau})$ and computing $\mathrm{KDE}_{X'}(y)$ yields an unbiased $(1+\varepsilon)$-approximation of $\mathrm{KDE}_X(y)$, with constant probability.*

Proof. See Appendix C. ☐

The bound on m in Lemma 2 is tight up to a constant for worst-case input (see Appendix C).

3 Algorithmic approach and theoretical foundations

3.1 Decomposing the KDE

We start by proving the following lemma that states that the KDE of a query y can be estimated from individual estimates on a partition of the dataset.

Lemma 3. *Let the n-vector dataset $X \subseteq \mathbb{R}^d$ be partitioned into two non-empty parts $A, B \subseteq \mathbb{R}^d$, that is, $X = A \cup B$ and $A \cap B = \emptyset$. Let $y \in \mathbb{R}^d$ be an arbitrary query vector, and let Z_A and Z_B be unbiased estimators of $\mathrm{KDE}_A(y)$ and $\mathrm{KDE}_B(y)$, respectively. Then,*

$$Z' = \frac{|A|}{n} Z_A + \frac{|B|}{n} Z_B$$

is an unbiased estimator for $\mathrm{KDE}_X(y)$.

Proof. By linearity of expectation and the definition of unbiased estimators, we have

$$\begin{aligned}
\mathrm{E}[Z'] &= \mathrm{E}\left[\frac{|A|}{n} Z_A + \frac{|B|}{n} Z_B\right] = \frac{|A|}{n}\,\mathrm{E}[Z_A] + \frac{|B|}{n}\,\mathrm{E}[Z_B] \\
&= \frac{|A|}{n} \frac{1}{|A|} \sum_{a \in A} K_h(a, y) + \frac{|B|}{n} \frac{1}{|B|} \sum_{b \in B} K_h(b, y) \\
&= \frac{1}{n} \sum_{x \in A \cup B} K_h(x, y) = \mathrm{KDE}_X(y) \ .
\end{aligned}$$

☐

3.2 Algorithmic approach

Given a query $y \in \mathbb{R}^d$ and a dataset $X = \{x_0, x_1, \ldots, x_{n-1}\} \subseteq \mathbb{R}^d$ of n points, assume we have access to a black box subroutine $\mathrm{ANN}_X(y)$ that returns (the indices of) k approximate nearest neighbors $X_1 \subseteq X$ of $y \in \mathbb{R}^d$. We can apply Algorithm 1 to compute an unbiased estimate $\widetilde{\mathrm{KDE}}_X(y)$ of the KDE.

The algorithm works by partitioning the dataset into two parts: one where all data points are close to the query vector, and the remainder. The contribution of the near vectors is computed exactly, and the remainder is approximated by random sampling. This idea bears resemblance to that of the hierarchical tree methods, but is expressed very concisely, and the nearest neighbors algorithm is treated as black box. Indeed, the algorithm is very general: it admits arbitrary kernels, metrics, and ANN algorithms, assuming they are compatible with one another.

The algorithm has two parameters: the number of neighbors to query k and the number of random samples m. At the extremes, when either k or m is zero, the algorithm either falls back to simple random sampling, or simply discards all far points. Both cases may be appropriate for certain datasets at very small or very large bandwidth values. This also guarantees that the algorithm performs asymptotically at least as well as simple random sampling.

Since $\mathrm{KDE}_{X_1}(y)$ is the exact contribution of k data points to the KDE of y, and a random sample on $X \setminus X_1$ results in an unbiased estimator of $\mathrm{KDE}_{X \setminus X_1}(y)$, we may conclude by Lemma 3 that Algorithm 1 returns an unbiased estimator.

Corollary 4. *The value* $\widetilde{\mathrm{KDE}}_X(y)$ *in* DEANN *(Algorithm 1) is an unbiased estimator of* $\mathrm{KDE}_X(y)$.

The estimate is unbiased no matter the quality of the near neighbors returned by $\mathrm{ANN}_X(y)$. This property is crucial: it allows us to use fast ANN implementations in practice that have no theoretical guarantees on the quality of their answers.

Algorithm 1 DEANN.

Input: Dataset $X = \{x_0, x_1, \ldots, x_{n-1}\} \subseteq \mathbb{R}^d$, query vector $y \in \mathbb{R}^d$, kernel function $K_h \colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, approximate nearest neighbor function $\mathrm{ANN}_X \colon \mathbb{R}^d \to [n]^k$.

Output: Unbiased est. $\widetilde{\mathrm{KDE}}_X(y)$ of $\mathrm{KDE}_X(y)$.

function DEANN(X, K_h, ANN_X, y)
- $X_1 \leftarrow \{x_i : i \in \mathrm{ANN}_X(y)\}$. ▷ Find k ANN
- $X_2 \leftarrow X \setminus X_1$. ▷ $\{X_1, X_2\}$ is a partition of X.
- $Z_1 \leftarrow \mathrm{KDE}_{X_1}(y) = \frac{1}{k} \sum_{x \in X_1} K_h(x, y)$.
- $S \leftarrow$ size-m uniform random sample from X_2.
- $Z_2 \leftarrow \mathrm{KDE}_S(y) = \frac{1}{m} \sum_{x \in S} K_h(x, y)$.
- $\widetilde{\mathrm{KDE}}_X(y) \leftarrow \frac{k}{n} Z_1 + \frac{n-k}{n} Z_2$.
- **return** $\widetilde{\mathrm{KDE}}_X(y)$.

end function

3.3 Contribution of nearest neighbors in real-world datasets

According to Dong et al. (2008), the distance distribution of distances from query points follows a Gamma distribution in many real-world datasets. While the shape and scale parameters of the distribution may differ widely between various datasets, they can be estimated efficiently from a small sample. As Dong et al. (2008) observe, the same is true for the distance distribution of the k-th nearest neighbors. In particular, Pagel et al. (2000) propose that the average distance of the k-th nearest neighbor under squared Euclidean distance can be modeled as a power-law function $\alpha(k/n)^\beta$, where $\alpha > 0$ is a constant depending on d, and $1/\beta > 1$ is the *intrinsic dimensionality* of X.

A rule of thumb for the selection of the bandwidth is to pick the median distance to the nearest neighbor as a bandwidth parameter (Jaakkola et al., 1999). The following lemma shows that, given a distance distribution that follows a power-law distribution, this bandwidth selection rule results in KDE values dominated by the contribution of a poly-logarithmic number of nearest neighbors. Deviating from this rule by much results in KDE values that are *meaningless*: too close to 0 or 1.

Lemma 5. *Given $\alpha, 1/\beta > 0$, $X \subseteq \mathbb{R}^d$ with $|X| = n$, and $y \in \mathbb{R}^d$, assume that $||x'_i - y||_2^2 = \alpha((i+1)/n)^\beta$ for $i \in [n]$. For the Gaussian kernel $K_h(x, y) = \exp\left(-||x - y||_2^2/(2h^2)\right)$, it holds that*

(a) If $h^2 = (\alpha/2)n^{-\beta}$, the contribution of the first $k = \Theta(\log^{1/\beta} n)$ nearest neighbors is a $(1+o(1))$-approximation of the KDE value.

(b) Let $\tau \in (0, 1)$. If $h^2 \le (\alpha/2)n^{-\beta}/\ln(1/\tau)$, $\mathrm{KDE}_X(y) \le \tau$.

(c) If $h^2 \ge \ln(1/(1-\delta))\alpha/(2\beta)$, $\mathrm{KDE}_X(y) \ge 1 - \delta$.

Proof. See Appendix D. □

3.4 How nearest neighbors help random sampling

While the previous subsection gave a theoretical reason why the rule-of-thumb for bandwidth selection is useful in practice, it assumed exact distances and ignored the fact that, in practice, $\log^{1/\beta} n$ might be a large number. In general, every partition of the dataset X into S and $X \setminus S$ in Algorithm 1 results in an unbiased estimator. However, it is unclear how the random sampling approach improves the estimate when the contribution of the k-nearest neighbors is known. This is because the number of samples m in Algorithm 1 is independent of the size $n - |S|$ of $X \setminus S$ (see Lemma 2). The following definition and the resulting lemma show that the larger the contribution of the nearest neighbors, the fewer samples suffice to obtain a $(1 + \varepsilon)$-approximation of the KDE value.

Definition 6. Given $n \ge 1$, $\delta \in (0, 1)$, and $k \in [n]$, let $X \subseteq \mathbb{R}^d$ with $|X| = n$. Given $y \in \mathbb{R}^d$, we say that the pair (k, δ) *dominates* $\mathrm{KDE}_X(y)$ if $\sum_{i=0}^{k-1} K_h(x'_i, y) = (1 - \delta) \sum_{i=0}^{n-1} K_h(x'_i, y)$.

The following lemma says that if the KDE value is (k, δ)-dominated, a δ-fraction of random samples is sufficient to obtain a $(1 + \varepsilon)$-approximation.

Lemma 7. *Let $\varepsilon > 0$, and $\mathrm{KDE}_X(y) \ge \tau$. If (k, δ) dominates $\mathrm{KDE}_X(y)$, then using $m = \Theta\left(\frac{\delta}{\varepsilon^2 \tau}\right)$ samples guarantees that with constant probability, $\widetilde{\mathrm{KDE}}_X(y)$ is a $(1 + \varepsilon)$-approximation.*

Proof. See Appendix E. □

4 Implementation and engineering choices

Implementation. We have implemented our algorithm in C++, using Intel MKL as backend for linear algebra and vectorized array computations. The implementation can be used as a Python module, and accepts arbitrary ANN libraries as a black box through a Python interface. We provide example interfaces for using scikit-learn `NearestNeighbors` as a baseline, and FAISS (Johnson et al., 2017) as a practical ANN implementation. The code is available online[5] under the MIT license and includes the naïve algorithm, random sampling, and DEANN.

Optimizations for Euclidean kernels. While Algorithm 1 is agnostic to the choice of the kernel, some further optimizations are possible if we restrict ourselves to Euclidean kernels. We make the following observation regarding the Euclidean norm. Using of the identity $||x - y||_2^2 = ||x||_2^2 + ||y||_2^2 - 2\langle x, y\rangle$ enables the use of the matrix-matrix multiplication primitive `GEMM` to speed-up batch evaluation of Euclidean distances, described in more detail in Appendix F.

Optimizing random sampling. A practical limitation of the random sampling routine is that a direct implementation would mandate random access to memory. To make effective use of a CPU's prefetching ability, data must be accessed in a linear or otherwise well-predictable fashion. We speed up our random sampling scheme by permuting the dataset vectors during preprocessing. We can then take a contiguous subset of the permuted vectors as the sample which can also be combined with the matrix multiplication optimization described above, using the matrix-vector multiplication primitive `GEMV`. For completeness, pseudocode is given in Appendix G. For a single query, this *permuted random sampling*

[5] `https://github.com/mkarppa/deann`

amounts to random sampling without replacement; however, we lose independence when considering multiple queries. Although problematic when facing an adversary, the results are equally good in practice, as shown in the next section.

5 Experiments

Implementations. All implementations considered in our experiments are listed in Table 1. We disambiguate implementations from abstract algorithms by writing the name of the implementation in typewriter typeface. For example, we distinguish between the naive and permuted random sampling implementations by writing `RS` and `RSP`, respectively. We refer to the variants of DEANN that use naive and permuted random sampling as a subroutine by `DEANN` and `DEANNP`, respectively. We evaluate our implementation against the `HBE` implementation of Siminelakis et al. (2019), and the standard implementation provided by scikit-learn (Pedregosa et al., 2011).

The variant of `HBE` considered is called `AdaptiveHBE` in the code of Siminelakis et al. (2019), and uses the HBS procedure (Siminelakis et al., 2019, Algorithm 4) for subsampling the data and the Adaptive Mean Relaxation (AMR) procedure (Siminelakis et al., 2019, Algorithm 2) for early termination of queries. For completeness, we also evaluate the `AdaptiveRS` variant of random sampling provided by Siminelakis et al. (2019) that uses AMR with the RS estimator, and denote it by `RSA`. To our understanding, these are the particular varieties evaluated in Siminelakis et al. (2019). We instrumented their code to produce the output necessary in post-processing; the full version of their code used for this paper is accessible through the `deann-experiments` repository.

We include the `KernelDensity` from scikit-learn (Pedregosa et al., 2011) as a baseline since scikit-learn is widely used in practical data science applications. This implementation uses k-d trees or ball trees with an optional error tolerance parameter for accelerating KDE evaluations. We denote the two different choices for data structure by `SKKD` and `SKBT`, respectively.

We use FAISS (Johnson et al., 2017) as the ANN implementation with our estimator algorithms. In particular, we use their *inverted file* index which runs k-means on the dataset. From the centroids of k-means, it builds a linear-space data structure in which each dataset point is assigned to its closest centroid. When answering a query, it inspects all points associated with the n_q closest centroids to the query. Both k and n_q are user-defined parameters that are provided to the implementation. Although FAISS supports extensive parallelism with GPUs, we limit ourselves to the single-threaded CPU version. This is because our implementation is entirely single-threaded to make it comparable with pre-existing single-threaded implementations; we also disabled multithreading in MKL.

In the appendices, we provide additional evaluation results that include (i) further considerations on the robustness of parameter choices in Appendices H and I, and (ii) experiments using the Gaussian kernel (including ASKIT by March et al. (2015) as a competitor) in Appendix J. The trends observed in the main text translate well into these settings.

Datasets. The datasets considered are presented in Table 2. The names of datasets are written in small caps. The choice of datasets includes the ones that were used in previous works (Siminelakis et al., 2019; Backurs et al., 2019) for the sake of reproducibility of results, and also present variation in the quality of data, the size of the dataset, and the number of dimensions. In all cases, we split the datasets in three disjoint subsets: a validation set of 500 vectors, a test set of 500 vectors, and a training set consisting of the remainder of the data. The training set is used as the set X against which the KDE values are computed. The validation and the test set are used as queries.

Bandwidth selection. Following the approach in Backurs et al. (2019), we chose four *target KDE values* 10^{-2}, 10^{-3}, 10^{-4}, and 10^{-5} and applied binary search on the validation set to find a bandwidth parameter h such that the *median* exact KDE value of the validation set vectors is within a *relative error* [6] of 0.01 from the target value. The reason for

[6] For an individual query vector y, let the estimated KDE be Z and the correct KDE be μ. We then say that the relative

Table 1: Implementations Used in the Experiments.

Name	Description	Reference
`Naive`	Exact using `GEMM`	Section 4
`RS`	Naive RS	Lemma 2
`RSP`	Permuted RS	Section 4
`DEANN`	DEANN with `RS`	Section 4
`DEANNP`	DEANN with `RSP`	Section 4
`HBE`	HBE estimator	Siminelakis et al. (2019)
`RSA`	Adaptive RS	Siminelakis et al. (2019)
`SKKD`	sklearn k-d-tree	Pedregosa et al. (2011)
`SKBT`	sklearn balltree	Pedregosa et al. (2011)

Table 2: Description of the Datasets.

Dataset	n	d	Reference
ALOI	108,000	128	Geusebroek et al. (2005)
Census	2,458,285	68	US Census Bureau
Covtype	581,012	54	Blackard and Dean (1999)
GloVe	1,193,514	100	Pennington et al. (2014)
last.fm	292,385	65	Celma (2010)
MNIST	60,000	784	Lecun et al. (1998)
MSD	515,345	90	Bertin-Mahieux et al. (2011)
Shuttle	58,000	9	NASA
SVHN	531,131	3072	Netzer et al. (2011)

this choice of multiple bandwidth values is that the KDE values are very sensitive to a right choice of bandwidth; as the bandwidth serves as a scaling factor to distances, a very large bandwidth will make the distances meaningless and it does not matter which points we look at, whereas a very small bandwidth together with the exponential decay of the kernel as a function of distance means that the nearest neighbors completely determine the KDE values. By trying different bandwidths, we explore the intermediate region where both far-away points and nearby points contribute to the typical density values. For brevity, we will sometimes refer to the target value by the letter μ in the remainder of this section.

Experimental pipeline. We evaluate the validation set using the exponential kernel on different algorithms and with different parameter values. The supplementary material includes additional experiments with the Gaussian kernel. The parameters were chosen by a grid search over pre-selected parameter ranges; see the supplemental code for detailed hyperparameter ranges. We exclude the parameter choices that exceed relative error 0.1, and then choose the fastest set of parameters with respect to average query time.

The best choice of parameters is used to evaluate the test set, on which we report the relative error, average query time, and the number of samples looked at, as an average of five independent repetitions. For `HBE`, we treat the relative approximation error ε and the minimum KDE value τ as free parameters to be optimized. For the scikit-learn-based implementations `SKKD` and `SKBT`, the parameters are relative tolerance t_r which controls which subtrees the implementation disregards, and the leaf size ℓ of the evaluation tree, where the implementation falls back to brute force. For DEANN, the parameters are the number of nearest neighbors k, the number of random samples to consider m, the number of clusters FAISS constructs n_ℓ, and the number of clusters FAISS queries n_q.

Machine details. The experiments were run on a shared computer with two 14-core Intel Xeon E5-2690 v4 CPUs, amounting to 28 physical CPU cores, running at 2.6 GHz, 512 GiB RAM, and using Ubuntu 16.04 LTS. The code was compiled with CLang 8.0.0, against Intel MKL version 2020.2, and the experiments were run using CPython 3.8.5, NumPy 1.19.2, scikit-learn 0.23.2, and FAISS version 1.7.0. The Python environment, inlcuding MKL and FAISS, were managed through Anaconda 2020.11. A small amount of other load was present on the computer.

Results on validation set. Computing the KDE value with different methods on the validation set provided the following insights: For target KDE values of 10^{-2} and 10^{-3}, DEANN will usually fall back to random sampling which provides faster query times. For smaller KDE values, the best query times were achieved by combining the contribution of the nearest neighbors and random sampling. Notable exceptions were last.fm where using k nearest neighbors pays off even for large KDE values, and GloVe

error is $|Z-\mu|/\mu$. For a query set $Q=\{q_1,q_2,\dots,q_m\}$ such that the estimated KDE for the query vector q_j is Z_j and the correct KDE is μ_j, we say that the average relative error is $\frac{1}{m}\sum_{j=1}^{m}|Z_j-\mu_j|/\mu_j$.

and SVHN, where random sampling was the best choice for all μ.

Table 3 lists the parameters that achieved the best query time with respect to the validation set at relative error below 0.1 for a subset of datasets. For lack of space, only the parameters for `RSP`, `DEANNP`, `HBE`, and `SKKD` are reported; the parameters for other algorithms are very similar. The subset was chosen to represent three different cases: a mixed case (ALOI) where `DEANNP` performs the best for some bandwidth choices and is on par with `RSP` for others, a case that favors `DEANNP` (LAST.FM), and a case where `RSP` performs the best (SVHN) and `DEANNP` essentially falls back to random sampling. The full set of parameters is reported in Appendix H.

Table 4 shows the average recall rates for FAISS at the choice of parameter that provided the best results. The subset of results is different from Table 3 to highlight the extrema. The average fraction of true neighbors returned ranged from 0.23 (ALOI, $k = 400$, $\mu = 10^{-3}$) to 0.98 (SHUTTLE, $k = 50$, $\mu = 10^{-5}$) with a wide range of different values attained between these extrema. The full set of results together with an extended discussion is presented in Appendix H.

Results on test set. A subset of the main results are reported in Table 5, the same subset as in Table 3. The full set of results is presented in Appendix H. The table lists the average query time per query vector in milliseconds, ordered by the dataset and the target μ.

Performance discussion. In almost all cases, either `DEANNP` or `RSP` was the fastest implementation, as indicated by bold typeface (with the exception of `Covtype` at $\mu = 10^{-5}$). In cases where `RSP` was the fastest algorithm, `DEANNP` does not lose significantly because it falls back to random sampling; the runtimes are very similar in those cases, apart from the slight overhead of the more complex implementation. `RSP` provides speedups of a factor of 2–10 for most workloads compared to `RS`. In the small bandwidth regime where the ANN contribution helps most, `RSP` is often slower by a factor of 10 or more than DEANN. Contrasting our implementations to competitors, we can compare to `HBE` consistently only for target KDE value of 0.01 and, usually, 0.001. In this setting, performance is closest on COVTYPE with target KDE value 0.001 (`HBE` is roughly 2.5 times slower), but we observe a speedup of 1-2 orders of magnitudes in many other settings, while being robust even for very small target values. The tree-based methods of scikit-learn did not perform very well in our experiments. This is largely due to the fact that the datasets are high-dimensional and the space-partitioning methods tend to scale exponentially with dimension. Indeed, scikit-learn performed adequately in comparison to our `Naive` implementation only on SHUTTLE, the dataset with smallest d, and—surprisingly—COVTYPE with smallest target KDE.

Task difficulty. Some results are missing: for SHUTTLE at target value of 0.00001, `RS` would have required more samples than there are datapoints to achieve the desired relative error. Several `HBE` and `RSA` results are missing due to our experimental setup, as a very small value of τ ought to have been used to achieve a sufficiently small relative error, as we included *all* query vectors in our experiments, even those with extremely small KDE values. However, the implementation did not permit use of sufficiently small τ values because either the runtimes grew excessively large or the size of the data structure grew so large that we ran out of RAM on our computer. For finished runs, our results are in line with the results in Siminelakis et al. (2019).

Preprocessing times. Our algorithm has no intrinsic data structure to construct; the preprocessing time is determined by the ANN algorithm, and the time it takes to create a permuted copy of the data for permuted sampling. Table 6 shows a subset of preprocessing times that have been collected when evaluating a similar set of experiments against the Gaussian kernel. As such, this table also includes `ASKIT` for comparison. The data points have been cherry-picked to reflect various extreme cases, including the extreme case of over 7 hours for scikit-learn when constructing the tree for the CENSUS dataset. For `DEANN` and `DEANNP`, the wide variation in the construction times is determined by the choice of the FAISS parameters which provide a tradeoff between construction and query time. Full results and discussion are presented in Appendix K.

Robustness considerations. Table 11 shows the relative errors achieved when evaluating the query set against the test set with the best parameters, show-

Table 3: The Best Choice Of Parameters Achieving Less Than 0.1 Relative Error For A Subset Of Dataset/Target μ Choices.

			RSP	DEANNP				HBE		SKKD	
Dataset	Target μ	h	m	k	m	n_ℓ	n_q	ϵ	τ	ℓ	t_r
ALOI	0.01	3.3366	230	0	170	512	1	1.1	0.001	40	0.2
ALOI	0.001	2.0346	1800	0	2100	512	1	0.6	0.0001	90	0.2
ALOI	0.0001	1.3300	29000	170	500	1024	5	*n/a*	*n/a*	80	0.2
ALOI	0.00001	0.8648	78000	120	430	1024	5	*n/a*	*n/a*	90	0.2
LAST.FM	0.01	0.0041	75000	60	350	1024	1	*n/a*	*n/a*	10	0.2
LAST.FM	0.001	0.0026	85000	70	800	512	1	*n/a*	*n/a*	10	0.15
LAST.FM	0.0001	0.0019	160000	50	350	2048	5	*n/a*	*n/a*	20	0.1
LAST.FM	0.00001	0.0015	200000	80	450	2048	5	*n/a*	*n/a*	100	0.15
SVHN	0.01	632.7492	150	0	120	512	1	1.2	0.0001	70	0.2
SVHN	0.001	391.3900	400	0	350	512	1	*n/a*	*n/a*	60	0.2
SVHN	0.0001	277.1836	900	0	800	512	1	*n/a*	*n/a*	60	0.2
SVHN	0.00001	211.4066	1900	0	2000	512	1	*n/a*	*n/a*	60	0.2

Table 4: Average Recall Rates For The Approximate Nearest Neighbors Returned By FAISS.

		DEANN					DEANNP				
Dataset	Target μ	k	m	n_ℓ	n_q	R	k	m	n_ℓ	n_q	R
ALOI	0.001	170	400	512	1	0.23	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*
ALOI	0.0001	200	430	1024	5	0.72	170	500	1024	5	0.74
LAST.FM	0.01	50	400	2048	1	0.24	60	350	1024	1	0.88
LAST.FM	0.001	70	200	2048	5	0.86	70	800	512	1	0.97
MSD	0.00001	210	1800	4096	10	0.45	210	2100	2048	5	0.43
SHUTTLE	0.00001	50	0	512	5	0.98	50	0	512	5	0.98

Table 5: Results of Evaluating the Different Algorithms Against the Test Set in Milliseconds / Query.

Dataset	Target μ	Naive	RS	RSP	DEANN	DEANNP	HBE	RSA	SKKD	SKBT
ALOI	0.01	1.051	0.050	0.022	0.025	**0.016**	0.623	0.808	58.498	48.353
ALOI	0.001	1.058	0.326	**0.105**	0.211	0.148	12.192	41.411	59.353	47.644
ALOI	0.0001	1.055	6.477	1.698	0.270	**0.197**	*n/a*	*n/a*	55.786	47.916
ALOI	0.00001	1.057	21.781	4.548	0.219	**0.182**	*n/a*	*n/a*	47.930	49.698
LAST.FM	0.01	2.593	12.704	2.145	0.227	**0.181**	*n/a*	*n/a*	104.039	94.147
LAST.FM	0.001	2.621	17.183	2.455	0.277	**0.222**	*n/a*	*n/a*	99.893	86.006
LAST.FM	0.0001	2.753	48.630	4.699	0.294	**0.247**	*n/a*	*n/a*	98.582	83.999
LAST.FM	0.00001	2.923	40.249	5.993	0.330	**0.263**	*n/a*	*n/a*	85.621	83.367
SVHN	0.01	42.094	0.290	**0.189**	0.255	0.448	11.830	56.613	3447.218	2521.555
SVHN	0.001	42.172	0.747	**0.500**	0.698	0.938	*n/a*	56.270	3471.669	2509.883
SVHN	0.0001	42.260	2.207	**1.096**	1.503	1.459	*n/a*	83210.996	3455.433	2495.796
SVHN	0.00001	41.748	3.743	**2.262**	3.758	2.852	*n/a*	*n/a*	3496.380	2445.718

Table 6: A Subset Of Preprocessing Times In Seconds.

Dataset	Target μ	`Naive`	`RS`	`RSP`	`DEANN`	`DEANNP`	`HBE`	`RSA`	`SKKD`	`SKBT`	`ASKIT`
ALOI	0.01	0.006	**0.000**	0.055	0.377	8.775	22.285	0.000	4.929	5.155	21.455
ALOI	0.00001	**0.006**	*n/a*	*n/a*	0.154	0.146	*n/a*	*n/a*	5.782	4.949	6.372
Census	0.01	0.081	**0.000**	0.945	3.568	14.269	101.727	0.000	25573.250	22917.678	*n/a*
Covtype	0.01	0.017	**0.000**	0.179	26.056	0.336	11.008	0.000	5.644	4.098	572.824
Covtype	0.00001	**0.016**	*n/a*	*n/a*	0.593	0.621	*n/a*	*n/a*	5.026	4.010	75.267
MNIST	0.01	0.017	**0.000**	0.159	1.700	0.813	100.323	0.000	12.369	11.154	14.053
MNIST	0.00001	0.016	**0.000**	0.155	0.447	0.443	*n/a*	*n/a*	12.461	11.022	4.397
MSD	0.01	0.020	**0.000**	0.223	9.319	9.395	*n/a*	*n/a*	12.378	10.359	144.805
MSD	0.00001	0.019	**0.000**	0.224	0.446	0.460	*n/a*	*n/a*	11.614	10.028	144.940
Shuttle	0.01	0.001	**0.000**	0.007	0.238	0.070	2.006	0.000	0.687	0.658	0.593
SVHN	0.01	0.583	**0.000**	5.590	262.613	1651.727	*n/a*	*n/a*	454.764	473.117	*n/a*
SVHN	0.00001	0.772	**0.000**	5.592	16.252	16.640	*n/a*	*n/a*	431.374	452.096	*n/a*

ing that DEANN generalizes nicely: our experiments show that this choice translated to a low average relative error also in the test set, as the greatest individual observed value was on LAST.FM at $\mu = 0.01$ where the relative error reached 0.114. The full set of results is presented in Appendix H.

In Appendix I, we discuss robust parameter selection for DEANN. Instead of an expensive grid search, we report on experiments using one fixed set of parameters for different datasets and different target values. This single fixed parameter setting provided low relative error and good performance in most cases.

Acknowledgements

We thank Kexin Rong and Paris Siminelakis for helpful discussion regarding their code. Matti Karppa and Rasmus Pagh are part of BARC, supported by VILLUM Foundation grant 16582.

References

Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack J. Dongarra. Performance, design, and autotuning of batched GEMM for gpus. In Julian M. Kunkel, Pavan Balaji, and Jack J. Dongarra, editors, *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, volume 9697 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2016. doi: 10.1007/978-3-319-41321-1_2. URL `https://doi.org/10.1007/978-3-319-41321-1_2`.

Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya P. Razenshteyn, and Ludwig Schmidt. Practical and optimal LSH for angular distance. In *NIPS*, pages 1225–1233, 2015.

Alexandr Andoni, Thijs Laarhoven, Ilya P. Razenshteyn, and Erik Waingarten. Optimal hashing-based time-space trade-offs for approximate near neighbors. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 47–66. SIAM, 2017. doi: 10.1137/1.9781611974782.4. URL `https://doi.org/10.1137/1.9781611974782.4`.

Ery Arias-Castro, David Mason, and Bruno Pelletier. On the estimation of the gradient lines of a density and the consistency of the mean-shift algorithm. *J. Mach. Learn. Res.*, 17:43:1–43:28, 2016. URL `http://jmlr.org/papers/v17/ariascastro16a.html`.

Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. PUFFINN: parameterless and universally fast finding of nearest neighbors. In *ESA*, volume 144 of *LIPIcs*, pages 10:1–10:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.*, 87, 2020.

Arturs Backurs, Piotr Indyk, and Tal Wagner. Space and time efficient kernel density estimation in high dimensions. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 15773–15782, 2019. URL `https://proceedings.neurips.cc/paper/2019/hash/a2ce8f1706e52936dfad516c23904e3e-Abstract.html`.

Thierry Bertin-Mahieux, Daniel P. W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In Anssi Klapuri and Colby Leider, editors, *Proceedings of the 12th International Society for Music Information Retrieval Conference, ISMIR 2011, Miami, Florida, USA, October 24-28, 2011*, pages 591–596. University of Miami, 2011. URL `http://ismir2011.ismir.net/papers/OS6-1.pdf`.

Jock A. Blackard and Denis J. Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in Agriculture*, 24 (3):131–151, 1999. ISSN 0168-1699. doi: https://doi.org/10.1016/S0168-1699(99)00046-0. URL `https://www.sciencedirect.com/science/article/pii/S0168169999000460`.

L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

Òscar Celma. *Music Recommendation and Discovery in the Long Tail*. Springer, 2010.

Moses Charikar and Paris Siminelakis. Hashing-based-estimators for kernel density in high dimensions. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 1032–1043. IEEE Computer Society, 2017. doi: 10.1109/FOCS.2017.99. URL `https://doi.org/10.1109/FOCS.2017.99`.

Moses Charikar, Michael Kapralov, Navid Nouri, and Paris Siminelakis. Kernel density estimation through density constrained near neighbor search. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 172–183. IEEE, 2020. doi: 10.1109/FOCS46700.2020.00025. URL `https://doi.org/10.1109/FOCS46700.2020.00025`.

Yutian Chen, Max Welling, and Alexander J. Smola. Super-samples from kernel herding. In Peter Grünwald and Peter Spirtes, editors, *UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, July 8-11, 2010*, pages 109–116. AUAI Press, 2010. URL `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2148&proceeding_id=26`.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In Jack Snoeyink and Jean-Daniel Boissonnat, editors, *Proceedings of the 20th ACM Symposium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004*, pages 253–262. ACM, 2004. doi: 10.1145/997817.997857. URL `https://doi.org/10.1145/997817.997857`.

Wei Dong, Zhe Wang, William Josephson, Moses Charikar, and Kai Li. Modeling LSH for performance tuning. In *CIKM*, pages 669–678. ACM, 2008.

Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 2009. ISBN 978-0-521-88427-3.

François Le Gall. Powers of tensors and fast matrix multiplication. In Katsusuke Nabeshima, Kosaku Nagasaka, Franz Winkler, and Ágnes Szántó, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303. ACM, 2014. doi: 10.1145/2608628.2608664. URL `https://doi.org/10.1145/2608628.2608664`.

Francois Le Gall and Florent Urrutia. Improved rectangular matrix multiplication using powers of the coppersmith-winograd tensor. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1029–1046. SIAM, 2018. doi: 10.1137/1.9781611975031.67. URL `https://doi.org/10.1137/1.9781611975031.67`.

Edward Gan and Peter Bailis. Scalable kernel density classification via threshold-based pruning. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 945–959. ACM, 2017. doi: 10.1145/3035918.3064035. URL `https://doi.org/10.1145/3035918.3064035`.

Jan-Mark Geusebroek, Gertjan J. Burghouts, and Arnold W. M. Smeulders. The amsterdam library of object images. *Int. J. Comput. Vis.*, 61(1): 103–112, 2005. doi: 10.1023/B:VISI.0000042993.50813.60. URL `https://doi.org/10.1023/B:VISI.0000042993.50813.60`.

Alexander G. Gray and Andrew W. Moore. 'n-body' problems in statistical learning. In Todd K. Leen, Thomas G. Dietterich, and Volker Tresp, editors, *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA*, pages 521–527. MIT Press, 2000. URL `https://proceedings.neurips.cc/paper/2000/hash/7385db9a3f11415bc0e9e2625fae3734-Abstract.html`.

Alexander G. Gray and Andrew W. Moore. Nonparametric density estimation: Toward computational tractability. In Daniel Barbará and Chandrika Kamath, editors, *Proceedings of the Third SIAM International Conference on Data Mining, San Francisco, CA, USA, May 1-3, 2003*, pages 203–211. SIAM, 2003. doi: 10.1137/1.9781611972733.19. URL `https://doi.org/10.1137/1.9781611972733.19`.

L Greengard and V Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987. ISSN 0021-9991. doi: 10.1016/0021-9991(87)90140-9.

Leslie Greengard and John Strain. The fast gauss transform. *SIAM J. Sci. Comput.*, 12(1):79–94, 1991. doi: 10.1137/0912004.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pages 3887–3896. PMLR, 2020.

Masajiro Iwasaki and Daisuke Miyazaki. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data. *ArXiv e-prints*, October 2018.

Tommi S. Jaakkola, Mark Diekhans, and David Haussler. Using the fisher kernel method to detect remote protein homologies. In *ISMB*, pages 149–158. AAAI, 1999.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *CoRR*, abs/1702.08734, 2017. URL `http://arxiv.org/abs/1702.08734`.

M. C. Jones and H. W. Lotwick. On the errors involved in computing the empirical characteristic function. *Journal of Statistical Computation and Simulation*, 17(2):133–149, 1983. doi: 10.1080/00949658308810650.

M. C. Jones and H. W. Lotwick. Remark as r50: A remark on algorithm as 176. kernal density estimation using the fast fourier transform. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 33(1):120–122, 1984. ISSN 00359254, 14679876. doi: 10.2307/2347674.

M. C. Jones, J. S. Marron, and S. J. Sheather. A brief survey of bandwidth selection for density estimation. *Journal of the American Statistical Association*, 91(433):401–407, 1996. doi: 10.1080/01621459.1996.10476701. URL `https://www.tandfonline.com/doi/abs/10.1080/01621459.1996.10476701`.

Bo Kågström, Per Ling, and Charles Van Loan. Gemm-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.*, 24(3):268–302, 1998. URL `http://portal.acm.org/citation.cfm?id=292395.292412`.

Raehyun Kim, Jaeyoung Choi, and Myungho Lee. Optimizing parallel GEMM routines using autotuning with intel AVX-512. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2019, Guangzhou, China, January 14-16, 2019*, pages 101–110. ACM, 2019. doi: 10.1145/3293320.3293334. URL `https://doi.org/10.1145/3293320.3293334`.

Donald Knuth. *The Art of Computer Programming. Volume 1. Fundamental Algorithms.* Addison-Wesley, Boston, MA, USA, 1997. ISBN 978-0-201-89683-1. 3rd Edition.

Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

Dongryeol Lee and Alexander G. Gray. Fast high-dimensional kernel summations using the monte carlo multipole method. In Daphne Koller, Dale Schuurmans, Yoshua Bengio, and Léon Bottou, editors, *Advances in Neural Information Processing Systems 21, Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 8-11, 2008*, pages 929–936. Curran Associates, Inc., 2008. URL `https://proceedings.neurips.cc/paper/2008/hash/39059724f73a9969845dfe4146c5660e-Abstract.html`.

Dongryeol Lee, Alexander G. Gray, and Andrew W. Moore. Dual-tree fast gauss transforms. In *Advances in Neural Information Processing Systems 18 [Neural Information Processing Systems, NIPS 2005, December 5-8, 2005, Vancouver, British Columbia, Canada]*, pages 747–754, 2005. URL `https://proceedings.neurips.cc/paper/2005/hash/9087b0efc7c7acd1ef7e153678809c77-Abstract.html`.

Yinan Li, Jack J. Dongarra, and Stanimire Tomov. A note on auto-tuning GEMM for gpus. In Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, G. Dick van Albada, Jack J. Dongarra, and Peter M. A. Sloot, editors, *Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I*, volume 5544 of *Lecture Notes in Computer Science*, pages 884–892. Springer, 2009. doi: 10.1007/978-3-642-01970-8_89. URL `https://doi.org/10.1007/978-3-642-01970-8_89`.

Yury A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, 2020.

William B. March, Bo Xiao, and George Biros. ASKIT: approximate skeletonization kernel-independent treecode in high dimensions. *SIAM J. Sci. Comput.*, 37

(2), 2015. doi: 10.1137/140989546. URL https://doi.org/10.1137/140989546.

Vlad I. Morariu, Balaji Vasan Srinivasan, Vikas C. Raykar, Ramani Duraiswami, and Larry S. Davis. Automatic online tuning for fast gaussian summation. In Daphne Koller, Dale Schuurmans, Yoshua Bengio, and Léon Bottou, editors, *Advances in Neural Information Processing Systems 21, Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 8-11, 2008*, pages 1113–1120. Curran Associates, Inc., 2008. URL https://proceedings.neurips.cc/paper/2008/hash/d96409bf894217686ba124d7356686c9-Abstract.html.

Nima Mousavi, 2012. URL https://ece.uwaterloo.ca/~nmousavi/Papers/Chernoff-Tightness.pdf. Note.

NASA. Statlog (shuttle) data set. Donated by Jason Catlett to the UCI Machine Learning Repository. URL https://archive.ics.uci.edu/ml/datasets/Statlog+(Shuttle).

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011. URL http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf.

Bernd-Uwe Pagel, Flip Korn, and Christos Faloutsos. Deflating the dimensionality curse using multiple fractal dimensions. In *ICDE*, pages 589–598. IEEE Computer Society, 2000.

Jagdish K. Patel and Campbell B. Read. *Handbook of the Normal Distribution.* Marcel Dekker, Inc., New York, NY, USA, 1982. ISBN 0-8247-1541-1.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL, 2014. doi: 10.3115/v1/d14-1162. URL https://doi.org/10.3115/v1/d14-1162.

Jeff M. Phillips. ϵ-samples for kernels. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1622–1632. SIAM, 2013. doi: 10.1137/1.9781611973105.116. URL https://doi.org/10.1137/1.9781611973105.116.

Jeff M. Phillips and Wai Ming Tai. Near-optimal coresets of kernel density estimates. *Discret. Comput. Geom.*, 63(4):867–887, 2020. doi: 10.1007/s00454-019-00134-6. URL https://doi.org/10.1007/s00454-019-00134-6.

Parikshit Ram, Dongryeol Lee, William B. March, and Alexander G. Gray. Linear-time algorithms for pairwise statistical problems. In Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. I. Williams, and Aron Culotta, editors, *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada*, pages 1527–1535. Curran Associates, Inc., 2009. URL https://proceedings.neurips.cc/paper/2009/hash/2421fcb1263b9530df88f7f002e78ea5-Abstract.html.

Erich Schubert, Arthur Zimek, and Hans-Peter Kriegel. Generalized outlier detection with flexible kernel density estimates. In Mohammed Javeed

Zaki, Zoran Obradovic, Pang-Ning Tan, Arindam Banerjee, Chandrika Kamath, and Srinivasan Parthasarathy, editors, *Proceedings of the 2014 SIAM International Conference on Data Mining, Philadelphia, Pennsylvania, USA, April 24-26, 2014*, pages 542–550. SIAM, 2014. doi: 10.1137/1.9781611973440.63. URL https://doi.org/10.1137/1.9781611973440.63.

scikit-learn developers. scikit-learn user guide, 2021. URL https://scikit-learn.org/stable/user_guide.html. Version 0.24.2.

Bernard W. Silverman. Algorithm as 176: Kernel density estimation using the fast fourier transform. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 31(1):93–99, 1982. ISSN 00359254, 14679876. doi: 10.2307/2347084.

Bernard W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, London, 1986. ISBN 0-412-24620-1.

Paris Siminelakis, Kexin Rong, Peter Bailis, Moses Charikar, and Philip Levis. Rehashing kernel evaluation in high dimensions. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 5789–5798. PMLR, 2019. URL http://proceedings.mlr.press/v97/siminelakis19a.html.

Eric V Slud. Distribution inequalities for the binomial law. *The Annals of Probability*, 5(3):404–412, 1977.

US Census Bureau. Us census data (1990) data set. Donated by Chris Meek, Bo Thiesson, and David Heckerman to the UCI Machine Learning Repository. URL https://archive.ics.uci.edu/ml/datasets/US+Census+Data+(1990).

Da Yan, Wei Wang, and Xiaowen Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*, pages 634–643. IEEE, 2020. doi: 10.1109/IPDPS47924.2020.00071. URL https://doi.org/10.1109/IPDPS47924.2020.00071.

Xianyi Zhang, Qian Wang, and Yunquan Zhang. Model-driven level 3 BLAS performance optimization on loongson 3a processor. In *18th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2012, Singapore, December 17-19, 2012*, pages 684–691. IEEE Computer Society, 2012. doi: 10.1109/ICPADS.2012.97. URL https://doi.org/10.1109/ICPADS.2012.97.

Yan Zheng, Jeffrey Jestes, Jeff M. Phillips, and Feifei Li. Quality and efficiency for kernel density estimates in large data. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 433–444. ACM, 2013. doi: 10.1145/2463676.2465319. URL https://doi.org/10.1145/2463676.2465319.

A Asymptotic notation

We use the asymptotic notation as defined by Knuth (1997, Section 1.2.11). For $f, g : \mathbb{N} \to \mathbb{N}$, we write $f(n) = O(g(n))$ if there exist positive constants n_0 and M such that $f(n) \leq Mg(n)$ for all $n \geq n_0$. We also write $f(n) = \Omega(g(n))$ if there exist positive constants n_0 and L such that $f(n) \geq Lg(n)$ for all $n \geq n_0$. We write $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Finally, for real-valued functions $f : \mathbb{N} \to \mathbb{R}$, we write $f(n) = o(1)$ if $\lim_{n\to\infty} |f(n)| = 0$.

B Related work and historical perspectives on KDE

This section provides an extended discussion on the related work, and especially the historical discussion on earlier work.

Early developments in nontrivial computation of the KDE in low dimensions include methods based on the Fast Fourier Transform, such as Silverman (1982) and Jones and Lotwick (1983, 1984) for the univariate KDE, the Fast Multipole Method (Greengard and Rokhlin, 1987), and the Fast Gauss Transform (Greengard and Strain, 1991). This line of work has been followed by a line of *dual-tree* data structures (Gray and Moore, 2000, 2003; Lee et al., 2005; Ram et al., 2009). However, these methods suffer from the curse of dimensionality. An attempt to mitigate this effect in higher dimensions with *subspace trees*, applying dimension reduction technologies such as Principal Component Analysis (PCA) together with random sampling, was presented by Lee and Gray (2008), but even this method requires $\Theta(\frac{1}{\epsilon^2})$ samples.

Morariu et al. (2008) presented an algorithm based on tree data structures and *Improved Fast Gauss Transform* along with an implementation called FigTree. March et al. (2015) presented *ASKIT*, a tree-based space-partitioning method based on *treecodes* that can make efficient use of the low-rank block structure of the matrix of pairwise kernel evaluations of the data points even in high dimensions when such structure exists. They also provided an implementation of ASKIT as free software.[7]

Another line of research is focused on finding subsamples of the data set that preserve the KDE values with arbitrary queries up to an approximation factor, called *ϵ-samples* or *coresets* (Phillips, 2013; Zheng et al., 2013; Phillips and Tai, 2020). However, despite offering better approximation guarantees, asymptotically coresets require a similar $\Theta(\frac{1}{\epsilon^2})$ number of samples as simple Random Sampling.

There are also other approaches to subsampling the dataset, such as Kernel Herding (Chen et al., 2010), and also HBS (Siminelakis et al., 2019) and the independent subsampling of hash tables in (Backurs et al., 2019).

Charikar and Siminelakis (2017) applied importance sampling to model the KDE values through the collision probability of the Euclidean Locality Sensitive Hashing (ELSH) scheme of Datar et al. (2004) to create a data structure called *Hashing Based Estimators (HBE)*. This data structure presented first asymptotical improvement with theoretical guarantees over simple RS in high dimensions. In particular, HBE improves upon RS in the regime where a large amount of the contribution comes from a small number of dataset points close to the query point.

The theoretical nature of the results of Charikar and Siminelakis (2017) were made more practical by Siminelakis et al. (2019) who presented a data structure using *Hashing Based Sketches (HBS)*. Roughly, the idea of their KDE estimation algorithm is to first subsample the dataset into a number of sketches using ELSH and weighted sampling, and then construct the HBE estimators from these subsampled datasets by reapplying ELSH, thus "rehashing" the dataset. They also presented an adaptive variant of the algorithm whereby the ELSH data structures are constructed at a number of levels, each containing an increasing number of hash tables, corresponding to a lower bound of the estimated KDE value. Assuming a sufficiently large KDE estimate can be made, the query terminates early, but otherwise continues to a larger number of hash tables. They also provide an implementation of their algo-

[7] Available at `https://padas.oden.utexas.edu/libaskit/`.

rithm as free software[8] that can be used for comparison. They showed empirically in (Siminelakis et al., 2019) that their HBE implementation is competitive with ASKIT and in some performs an order of magnitude better than ASKIT.

Another improvement on the HBE scheme was presented by Backurs et al. (2019) who improved on the space usage of the algorithm by observing that HBE tends to store the same points in several hash tables. They showed that, for each hash table, it suffices to include each point hashed to the table with a certain probability to guarantee that the point is stored in approximately one hash table, and the approximation guarantees of HBE are still sufficiently preserved. They provided a Python implementation[9] and used the number of kernel function evaluations as a proxy for the runtime in their experiments.

In recent work, Charikar et al. (2020) provided asymptotic improvements in running time and space complexity by using data-dependent LSH.

C Proof of Lemma 2

In this appendix, we present the proof of Lemma 2. The proof is presented for completeness only without any claim to originality. While the result is well known, it seems to be difficult to find a useful version of the proof in the literature.

We need the following form of the Chernoff bound in the proof.

Lemma 8 (Chernoff (Dubhashi and Panconesi, 2009, Theorem 1.1, pp. 6–7))**.** *Let $X = \sum_{i=1}^{n} X_i$ where $X_i \in [0,1]$ are independently distributed random variables. Then, for $\epsilon > 0$,*

$$\Pr[X > (1+\epsilon)\,\mathrm{E}[X]] \leq \exp\left(-\frac{\epsilon^2}{3}\,\mathrm{E}[X]\right) , \tag{3}$$

$$\Pr[X < (1-\epsilon)\,\mathrm{E}[X]] \leq \exp\left(-\frac{\epsilon^2}{2}\,\mathrm{E}[X]\right) . \tag{4}$$

[8] Available at `https://github.com/kexinrong/rehashing`.

[9] Available at `https://github.com/talwagner/efficient_kde/`.

We recall Lemma 2. We bound the number of random samples required using the Chernoff bound with respect to an arbitrary constant probability δ.

Lemma 2 (Random Sampling)**.** *Let $X \subseteq \mathbb{R}^d, y \in \mathbb{R}^d$. Let $\tau \in (0,1)$ such that $\mathrm{KDE}_X(y) \geq \tau$. Drawing a uniform random sample $X' \subseteq X$ (with repetition) of size $m = O(\frac{1}{\varepsilon^2\tau})$ and computing $\mathrm{KDE}_{X'}(y)$ yields an unbiased $(1+\varepsilon)$-approximation of $\mathrm{KDE}_X(y)$, with constant probability.*

Proof. Fix constant $0 < \delta < 1$. Let $X' = (x'_1, x'_2, \ldots, x'_m)$ be the random sample such that each x'_i is drawn from X independently and uniformly distributed at random with repetition. We treat each x'_i as a random variable taking values from the set X and hold the query vector y arbitrary but fixed.

For all $i = 1, 2, \ldots, m$, define $Z_i = K_h(x'_i, y)$ where $K_h : \mathbb{R}^d \times \mathbb{R}^d \to [0,1]$ is the kernel function; without loss of generality, we may assume all Z_i satisfy $0 \leq Z_i \leq 1$ by dividing the value of the kernel function with an appropriate constant. Clearly, $\mathrm{E}[Z_i] = \frac{1}{n}\sum_{j=1}^{n} K_h(x_i, y) = \mu$, so each Z_i is an unbiased estimator for $\mu = \mathrm{KDE}_X(y)$.

Letting $Z = \sum_{i=1}^{m} Z_i$, we get by linearity of expectation that $\mathrm{E}[Z] = m\,\mathrm{E}[Z_i] = m\mu \geq m\tau$. From Equation (3), we get

$$\Pr[Z > (1+\epsilon)\mu] \leq \exp\left(-\frac{\epsilon^2}{3}m\mu\right) \leq \exp\left(-\frac{\epsilon^2}{3}m\tau\right) . \tag{5}$$

If we let the probability on the right hand side of Equation (5) be less than or equal to the constant δ, we get

$$-\frac{\epsilon^2}{3}m\tau \leq \ln\delta\,,$$

and solving for m,

$$m \geq \frac{3\ln\frac{1}{\delta}}{\epsilon^2\tau}\,, \tag{6}$$

and by the same argument, Equation (4) yields the same bound on m up to constant, so we can thus conclude that $m = O(\frac{1}{\varepsilon^2\tau})$ samples suffice to bound the error to the desired range. □

It should be noted that, although not present in the statement of Lemma 2, the number of random samples m depends on the constant δ by a factor of $\ln \frac{1}{\delta}$.

Furthermore, Lemma 2 is tight up to a constant. To see why, we must consider a worst-case input that consists of vectors such that a τ-fraction of the dataset has kernel value of 1 and the remainder are (essentially) 0. The random sample can be modelled as a sum of Bernoulli variables such that the kernel values are either 0 or 1 with probability τ, which yields the correct KDE in expectation.

This input has a geometric interpretation, where the query is situated such with respect to the dataset that a significant fraction (a τ-fraction) of the dataset essentially coincides with the query vector (possibly up to a negligible amount of additive noise), and the remainder of the dataset resides infinitely far (with respect to the exponential decay of the kernel). This is precisely the regime where we are looking for a needle in the haystack and nearest neighbors essentially determine the KDE value, but we need to look at a large fraction of the dataset at random to be able to find the needle.

We will show that, with such input, the Chernoff bound is tight up to a constant, which implies that also the required size of the sample is tight up to a constant. To show this, we need the following lemma that we have restated in the notation presented here.

Lemma 9 (Slud (1977, Theorem 2.1))**.** *Let $0 \leq \tau \leq \frac{1}{4}$ and $\varepsilon > 0$. Let $X = \sum_{i=1}^{m} X_i$ with $X_i \sim \mathrm{Bernoulli}(\tau)$. Then*

$$\Pr[X \geq (1+\varepsilon)m\tau] \geq 1-\Phi\left(\frac{\varepsilon\sqrt{m\tau}}{\sqrt{1-\tau}}\right) > 1-\Phi(2\varepsilon\sqrt{m\tau})\,,$$

where Φ is the standard normal cumulative distribution function.

Lemma 10. *Lemma 2 is tight up to a constant for worst-case input.*

Proof. This proof is almost the same as given by Mousavi (2012) and is presented here for completeness without claim to originality.

Let us denote random variables X_i for $i = 1, 2, \ldots, m$ such that each $X_i \sim \mathrm{Bernoulli}(\tau)$, yielding the worst-case input, drawn independently and identically distributed. As before, $X = \sum_{i=1}^{m} X_i$, so $E[X] = m\tau$. Let us approximate X with the normal distribution using Lemma 9. It is known (Patel and Read, 1982, Equation 3.7.2) that, for $x > 0$,

$$1 - \Phi \geq \frac{1-\sqrt{1-\exp(-x^2)}}{2}\,.$$

Furthermore, by the fact that $1-\sqrt{x} \geq \frac{1-x}{2}$, we can approximate

$$\begin{aligned}
&\Pr[X \geq (1+\varepsilon)E[X]] \\
\geq &1-\Phi(2\varepsilon\sqrt{m\tau}) \\
\geq &\frac{1-\sqrt{1-\exp(-\varepsilon^2 m\tau)}}{2} \\
\geq &\frac{\exp(-\varepsilon^2 m\tau)}{4}\,,
\end{aligned} \tag{7}$$

and since Equation (7) is of the same form as the Chernoff bounds of Lemma 8, we can conclude by the same argument as in the proof of Lemma 2 that the bound is tight up to a constant for the worst-case input. □

D Proof of Lemma 5

We recall Lemma 5.

Lemma 5. *Given $\alpha, 1/\beta > 0$, $X \subseteq \mathbb{R}^d$ with $|X| = n$, and $y \in \mathbb{R}^d$, assume that $||x_i' - y||_2^2 = \alpha((i+1)/n)^\beta$ for $i \in [n]$. For the Gaussian kernel $K_h(x, y) = \exp\left(-||x-y||_2^2/(2h^2)\right)$, it holds that*

(a) If $h^2 = (\alpha/2)n^{-\beta}$, the contribution of the first $k = \Theta(\log^{1/\beta} n)$ nearest neighbors is a $(1+o(1))$-approximation of the KDE value.

(b) Let $\tau \in (0, 1)$. If $h^2 \leq (\alpha/2)n^{-\beta}/\ln(1/\tau)$, $\mathrm{KDE}_X(y) \leq \tau$.

(c) If $h^2 \geq \ln(1/(1-\delta))\alpha/(2\beta)$, $\mathrm{KDE}_X(y) \geq 1-\delta$.

Proof. With $h^2 = (\alpha/2)n^{-\beta}$ the kernel evaluates to $K_h(x_i', y) = \exp(-(i+1)^\beta)$. With $k = \Theta(\log^{1/\beta} n)$, we get that $K_h(x_i', y) = \exp(-(i+1)^\beta) = o(1/n)$ for all $i \geq k$. Thus $\mathrm{KDE}_{(x_k', \ldots, x_{n-1}')}(y) = n\,o(1/n) = o(1)$, which proves the first statement.

For the second statement, observe that with $h^2 \geq (\alpha/2)n^{-\beta}/\ln(1/\tau)$, already the nearest neighbor evaluates to $K_h(x_0', y) = \exp(-1/\ln(1/\tau)) = \tau$. Since all other data points contribute at most τ, $\mathrm{KDE}_X(y) \leq \tau$.

Finally, by the inequality of arithmetic and geometric means we can lower bound the KDE value as follows:

$$\begin{aligned}
&1/n \sum_{i=0}^{n-1} \exp(-\alpha((i+1)/n)^{\beta}(1/h^2)) \\
\geq &\prod_{i=0}^{n-1} \exp\left(-\alpha(i+1)^{\beta} n^{-\beta-1}(1/h^2)\right) \\
= &\exp\left(-(\alpha/(h^2 n^{\beta+1})) \sum_{i=1}^{n} i^{\beta}\right) \\
\geq &\exp(-(\alpha/(h^2\beta))) \geq 1-\delta\,.
\end{aligned}$$

Here, we used that $\sum_{i=1}^{n} i^{\beta} = \frac{n^{\beta+1}}{\beta+1} + O(n^{\beta})$ and thus, asymptotically for large enough n,

$$\sum_{i=1}^{n} i^{\beta} < n^{\beta+1}/\beta\,.$$

□

E Proof of Lemma 7

We recall Lemma 7.

Lemma 7. *Let $\varepsilon > 0$, and $\mathrm{KDE}_X(y) \geq \tau$. If (k,δ) dominates $\mathrm{KDE}_X(y)$, then using $m = \Theta\left(\frac{\delta}{\varepsilon^2\tau}\right)$ samples guarantees that with constant probability, $\widetilde{\mathrm{KDE}}_X(y)$ is a $(1+\varepsilon)$-approximation.*

Proof. Given y, let $X = (x'_0, \ldots, x'_{n-1})$ be ordered in increasing order by distance to y. Given $\varepsilon' > 0$ to be set later, let $(n-k)\mathrm{RS}_{(x'_k,\ldots,x'_{n-1})}(y)$ be the value of an $(1+\varepsilon')$ approximation of $(n-k)\mathrm{KDE}_{(x'_k,\ldots,x'_{n-1})}(y)$. We compute:

$$\begin{aligned}
&\sum_{i=0}^{k-1} K_h(x'_i, y) + (n-k)\mathrm{RS}_{(x'_k,\ldots,x'_{n-1})}(y) \\
\leq &\sum_{i=0}^{k-1} K_h(x'_i, y) + (1+\varepsilon') \sum_{i=k}^{n-1} K_h(x'_i, y) \\
= &n\mathrm{KDE}(y) + \varepsilon' \sum_{i=k}^{n-1} K_h(x'_i, y) \\
= &n(\mathrm{KDE}(y) + \varepsilon'\delta\mathrm{KDE}(y))\,.
\end{aligned}$$

This means that to compute a $(1+\varepsilon)$ approximation, it suffices to compute a $(1+\varepsilon') = (1+\varepsilon/\delta)$ approximation on $(x'_k, \ldots, x'_{n-1})$. Since $\mathrm{KDE}_{(x'_k,\ldots,x'_{n-1})}(y) \geq \delta\tau$, a sample of $\Theta\left(\frac{\delta}{\varepsilon^2\tau}\right)$ elements suffices to guarantee a $(1+\varepsilon')$ approximation with constant probability. □

F Naïve algorithm

In this section, we describe how matrix multiplication can be used to speed up the evaluation of the naïve KDE sum when the kernel is Euclidean. We make no claims of originality, but simply present the material here for completeness. In this section, we treat the dataset X as a row-major $n \times d$ matrix.

Suppose we are working in a batch processing case with a set of N queries $Q = \{q_0, q_1, \ldots, q_{N-1}\}$ which we similarly treat as a row-major $N \times d$ matrix. We want to evaluate the N-element result vector z whose elements are given by

$$z_j = \frac{1}{n} \sum_{i=0}^{n-1} K_h(q_j, x_i)\,. \tag{8}$$

Assuming K_h is Euclidean, the evaluation of Equation (8) for all $j = 0, 1, \ldots, N-1$ can be considered to consist of (i) evaluating the $N \times n$ matrix D whose elements are given by

$$D_{j,i} = ||q_j - x_i||_2\,, \tag{9}$$

(ii) applying the (vectorized) functions, the composition of which equals K_h, and (iii) computing the row-wise mean of the resulting matrix.

Matrix multiplication helps in step (i) through the following observation:

$$||x-y||_2^2 = ||x||_2^2 + ||y||_2^2 - 2\langle x, y\rangle \; . \tag{10}$$

Let us write auxiliary matrices X_{sq} and Q_{sq} such that for all $i = 0, 1, \ldots, n-1$ and $j = 0, 1, \ldots, N-1$, we have

$$(X_{\mathrm{sq}})_{j,i} = ||x_i||_2^2 \, , \tag{11}$$

and

$$(Q_{\mathrm{sq}})_{j,i} = ||q_j||_2^2 \, . \tag{12}$$

Importantly, from Equations (11) and 12, we have that

$$(X_{\mathrm{sq}} + Q_{\mathrm{sq}})_{j,i} = ||q_j||_2^2 + ||x_i||_2^2 \, . \tag{13}$$

Now consider the matrix product $QX^\top$. From the definition of the matrix product, it is immediate that

$$(QX^\top)_{j,i} = \langle q_j, x_i\rangle \; . \tag{14}$$

If we then let $D^2 = X_{\mathrm{sq}} + Q_{\mathrm{sq}} - 2QX^\top$, we get from Equations (10), (13), and (14) that

$$D^2_{j,i} = ||q_j||_2^2 + ||x_i||_2^j - 2\langle q_j, x_i\rangle = ||x_i - q_j||_2^2 \, . \tag{15}$$

The key observation is that it is possible to use matrix multiplication as a primitive for evaluating the inner product matrix in Equation (15). Evaluating the values of the matrix D directly from the definition of Equation (9) one element at a time requires $\Theta(nNd)$ operations. However, matrix multiplication is asymptotically faster. For $n = N = d$, the evaluation goes down to $O(n^\omega)$ operations for $\omega < 2.3728639$ (Gall, 2014). Assuming $n = N$ and $d < n^\alpha$ for $\alpha > 0.31389$, the evaluation can be performed in $n^{2+o(1)}$ operations (Gall and Urrutia, 2018). Although these theoretical developments are impractical, significant gains can be made over implementing the evaluation naively even with the elementary matrix multiplication algorithm by using, for example, the BLAS Level 3 subroutine `GEMM`[10] that is an aggressively optimized primitive (Kågström et al., 1998; Li et al., 2009; Zhang et al., 2012; Abdelfattah et al., 2016; Kim et al., 2019; Yan et al., 2020). Highly tuned implementations of `GEMM`, such as the one provided by the Intel MKL, make efficient use of the CPU features, such as vectorization and cache hierarchy, and provide a considerable performance boost over simple implementations.

[10] Generalized Matrix Multiply, a BLAS (Blackford et al., 2002) Level 3 subroutine for computing the matrix multiplication operation $C \leftarrow \alpha A^\top B + \beta C$. The Intel MKL provides a highly optimized implementation of this routine.

G Permuted Random Sampling

We present here for completeness the subroutine we use for taking the optimized random sample in case of Euclidean kernels. Preprocessing and sampling are presented in Algorithm 2. We make no claim to originality, and simply present the algorithm here for completeness.

Algorithm 2 Permuted random sampling.

Input: Dataset $X = \{x_0, x_1, \ldots, x_{n-1}\} \subseteq \mathbb{R}^d$

1: **procedure** PREPROCESS(X)
2: Draw permutation π on n elements at random.
3: $X' \leftarrow \{x'_0, x'_1, \ldots, x'_{n-1}\}$ such that $x'_i = x_{\pi(i)}$.
4: $\ell \leftarrow 0$. ▷ Running index.
5: **end procedure**

Input: Query vector $y \in \mathbb{R}^d$, integer number of samples $1 \le m \le n$

Output: A random sample estimate of $\mathrm{KDE}_X(y)$.

1: **function** RANDOMSAMPLEPERMUTED(y, m)
2: $Z \leftarrow \sum_{i=\ell}^{\ell+m-1} K_h(x'_{i \bmod n}, y)$.
3: $\ell \leftarrow \ell + m \mod n$.
4: **return** $\frac{1}{m} Z$.
5: **end function**

Importantly, if the kernel K_h is Euclidean, the evaluation of the sample on line 2 can be treated as follows. First, we have either one or two contiguous, rectangular submatrices of the permuted data matrix; the latter case occures when the row index i overflows. We can then consider the evaluation to take place such that we evaluate the Euclidean distance to all points in the sample, evaluate the kernel individually on each distance, possibly using vectorized operations, and finally compute the mean.

Assume now that $\ell + m < n$. Let $x_{\mathrm{sq}} \in \mathbb{R}^m$ be a vector of the squared norms of the vectors in the sample, that is, $(x_{\mathrm{sq}})_j = ||x'_{\ell+j \bmod n}||_2^2$ for $j = 0, 1, \ldots, m-1$. The elements of this vector can be precomputed during preprocessing. Then, let X'' be the $m \times d$ matrix consisting of the rows $x'_\ell, x'_{\ell+1}, \ldots, x'_{\ell+m-1}$. The vector of squared Euclidean norms can then be computed in terms of matrix-vector multiplication as follows:

$$z = x_{\mathrm{sq}} + X''y + ||y||_2^2\,,$$

where the last scalar addition is considered to be broadcast to all elements in the output vector. The matrix-vector product $X''y$ can be evaluated efficiently using the `GEMV`[11] subroutine. Generalization to arbitrary cases follows by performing the operation in two steps whenever the running index i overflows the size of the data matrix, and in all cases by applying the relevant vectorized operations for evaluating the kernel value.

H Detailed discussion of experimental evaluation with the exponential kernel

Full results. Table 7 shows the full set of results, average query time as milliseconds / query, when evaluating the query set against the test set, with different algorithms. The best values are indicated with a bold typeface.

Results on validation set. Results of the validation step of the experiments are presented in Table 8. The table lists the instances by dataset and target median KDE value μ, the bandwidth h selected for the particular instance by binary search with respect to the validation set, and the best performing parameters for different algorithms. The parameters include the number of random samples m for Permuted Random Sampling (`RSP`), the number of nearest neighbors k, the number of random samples m, the number of clusters n_ℓ, and the number of clusters queried n_q by our ANN estimator `DEANNP` when using FAISS, the relative approximation ϵ and minimum KDE value τ of the `HBE` implementation, and the tree leaf size ℓ and relative error tolerance t_r for one the scikit-learn algorithms `SKKD`. Due to lack of space, the parameters for other variants are not shown but they are very similar to the ones shown here. In some cases, particularly for `HBE`, no suitable choice of parameters was found, which is indicated in the table by the text *n/a*.

The bandwidth values are very small in cases where the nearest neighbors help a lot with the performance. Indeed, in some cases, such as LAST.FM, the bandwidth is below 1, meaning that it actually expands the distances between the vectors. In some cases, such as SHUTTLE at target μ of 0.00001, the random samples provide such a small contribution to the overall KDE value that the best performing parameters for the DEANN use no random samples at all. Conversely, in several cases, such as all instances of SVHN, the best choice of parameters for the DEANN was to fall back to random sampling.

ANN recall. In most cases, the number of clusters in the FAISS data structure was rather large in comparison to the size of the dataset, but only very few clusters were queried. This means that only a small fraction of the dataset was inspected to find nearest neighbors. While this is good for the throughput of the ANN estimator, it might result in far-away points being included as nearest neighbors, or some true neighbors being missed. Let $\mathrm{NN}_k(q)$ and $\widetilde{\mathrm{NN}}_k(q)$ be the correct set of k nearest neighbors for the query vector q and the set returned by FAISS, respectively, and let the query set Q be the validation set. The average recall

$$R = \frac{1}{|Q|}\sum_{q\in Q}\frac{|\mathrm{NN}_k(q)\cap\widetilde{\mathrm{NN}}_k(q)|}{|\mathrm{NN}_k(q)|}$$

[11] Generalized Matrix Vector multiply, a BLAS (Blackford et al., 2002) Level 2 subroutine for computing the matrix vector multiplication and addition operation of $y \leftarrow \alpha Ax + \beta y$. The Intel MKL provides a highly optimized implementation of this routine.

is reported per dataset and target KDE value in Table 9 for both `DEANN` and `DEANNP`. The table only includes instances where a non-zero number of nearest neighbors was queried, that is, cases where DEANN fell back to random sampling are excluded. The table shows that a surprisingly small recall is sometimes sufficient to achieve a small relative error. This is particularly true for datasets where the majority of the contribution came from the random samples. The extreme cases are ALOI at $\mu = 0.001$ with $k = 170$ and $m = 400$ where a measly $R = 0.23$ was sufficient to achieve the desired relative error, and, at the other end, SHUTTLE at $\mu = 0.00001$ with $k = 50$ and $m = 0$ where we got $R = 0.98$.

Robustness considerations. Table 10 shows empirically that DEANN generalizes nicely. The parameters were chosen such that the average relative error did not exceed 0.1 in the validation set; the table shows that this translates to low average relative error also in the test set. The greatest individual observed value was on LAST.FM at a target value of 0.01 where the average relative error reached 0.114.

Figure 1 shows the dependence between different parameter choices from the validation step. Different parameter choices are plotted and the corresponding average relative error is shown on the x-axis and the effect on runtime—the number of queries processed per second—on the y-axis. Each individual parameter choice is presented with a marker, and to help visualize the dependence, a lineplot is drawn between the markers. Each subplot corresponds to a single dataset, and the different target KDE values are shown in the same plot with different colors and markers. Only meaningful parameter choices are shown here; parameter choices that would yield a worse relative error without gain in query speed are excluded. The figure shows that the parameter choices form a clear tradeoff between approximation quality and runtime, meaning it is possible to tune DEANN to various use cases, depending on the requirements on approximation quality and query times.

I Fixed-parameter experiments

In this section, we report on experiments that we carried out using a *fixed set of parameters*, that is, we made an educated guess for the constants k and m, and evaluated each dataset / target μ combination against this choice of parameters using the test set as the query set. The point of this exercise is to show that the expensive grid search is not necessary for a practical application; that it is, in fact, possible to find good enough parameters by evaluating a the algorithm against a small sample with a good guess of parameters. This shows the robustness of our algorithm: that it is not sensitive to the exact correct choice of parameters.

Table 12 shows the results of evaluating `DEANNP` against the test set with the exponential kernel using the fixed parameters $k = 100$, $m = 1000$, $n_\ell = 512$, and $n_q = 1$. The table lists the time per query, the average relative error, and the corresponding runtime for the best parameters obtained from the grid search for comparison at relative error below 0.1. As expected, a fixed choice of parameters favors some dataset/bandwidth choices more than others, but overall, the results are encouraging. In terms of error, the worst behavior is observed in the case of CENSUS with small bandwidths, and the reason is clear: too few neighbors are looked at; this is also reflected in the runtime which is more than a factor of 2 faster than with the parameters that achieve the error below 0.1. To the other extreme, in the case of GLOVE with the large bandwidth, we get a relative error of 0.014, suggesting that we could have done with a lot fewer samples.

The practical implication of this exercise is that it suggests the following procedure for a practical application of the algorithm: Choose a smallish query set, make a guess of parameters, evaluate against ground truth, and if the results are not good enough (too high error or too high runtime), refine the parameters by taking a new guess; since the algorithm behaves in a very predictable manner, only very few guesses should suffice in a practical setting to find "good enough" parameters, meaning that an expensive hyperparameter tuning may not always be necessary.

J Experiments with the Gaussian kernel

This section details supplementary experiments that were evaluated with respect to the Gaussian kernel. The experiments also included `ASKIT` (March et al., 2015) as a competitor.

The experiments were performed on the same physical hardware as those with the exponential kernel, but with an improved experimental framework where all implementations ran inside Docker containers to isolate them from the rest of the environment; in particular, this enabled us to run `ASKIT` which depends on the Intel toolchain, including the Intel MPI libraries, for compilation. The scripts for creating the Docker images are included in the code[12]. In the experiments, all implementations were limited to 32 GiB of memory, and any runs that exceeded the memory limitation were terminated. The total runtime of a run (including preprocessing) was limited to 6 hours, excluding `SKKD` and `SKBT`; this means that if the implementation could not build its datastructures and evaluate the queries with a certain set of parameters within the time limit, the run was terminated, and any subsequent runs whose parametrization would imply a longer runtime were not allowed to run either.

For `DEANN` and `DEANNP`, we fixed $n_q = 1$ and let $n_\ell \in \{32, 64, 128, 256, 512, 1024, 2048, 4096\}$. The perception here is that, for example, the choice $n_q = 2, n_\ell = 64$ is essentially the same as $n_q = 1, n_\ell = 32$ since we would expect to look at a similar number of near neighbors, assuming the points in the training set are somewhat well-behaved in their distribution among the different clusters in the k-means that FAISS does in its index building. The parameters k and m were selected to be from multiple scales using the formula

$$k, m \in \{10 \cdot (\sqrt{2})^i : i = 0, 1, \ldots\}, \qquad (16)$$

such that k and m satisfy $k + m < n$. A cartesian product of the parameters (k, m, n_ℓ) was then probed against the validation set.

For `RS` and `RSP`, the candidates for the parameter m were chosen by the same formula of Equation (16) such that $m < n$.

For `SKKD` and `SKBT`, the parameter ℓ was set to $\mathrm{round}(10 \cdot (\sqrt{2})^i)$ for $i \in \{0, 1, \ldots, 9\}$. The parameter t_r was set to $\{0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5\}$. A cartesian product of the values (ℓ, t_r) was probed against the validation set.

For `HBE` and `RS`, the parameter ϵ was set to $\{0.1, 0.15, 0.2, \ldots, 1.5\}$ and the parameter τ was set to $0.01/(\sqrt{2})^i$ for $i \in \{0, 1, \ldots, 19\}$ and rounded to 5 decimal digits, so τ ranged from 0.01 to 0.00001. A cartesian product of the values (ϵ, τ) was probed against the validation set.

For `ASKIT`, we followed the suggestions provided by March et al. (2015). For the initial pilot experiments, we set $\mathrm{id}_{\mathrm{tol}}$ to 10^i with $-10 \le i \le 2$, provide $\kappa = 100$ nearest neighbors for each dataset and query set point, set the max number of points m to 2^i with $6 \le i \le 12$, and set the oversampling factor to $f \in \{2, 5, 10\}$. After noticing that we could not obtain low relative error for small bandwidth choices by exploring these parameter choices, we further set skeleton targets $t_{\mathrm{skel}} \in \{2, 5, 10\}$ and and set the minimum skeleton level to $\ell_{\mathrm{skel}} \in \{2, 5, 10, 20\}$. After these initial experiments, we pruned the parameter space and details can be seen in the script `generate_askit.py` in the GitHub repository[13]. We remark that `ASKIT` assumes that the nearest neighbors for the whole dataset and query set are given as input during preprocessing, which is extremely costly. On our setup, it took 16 hours using FAISS with multi-threading using 48 threads to precompute this information for the 9 datasets in question. In contrast, our variants of `DEANN` compute these neighbors during query time.

When computing the relative error (especially in the validation step), we excluded points whose exact KDE value was below 10^{-16}, since the single-precision floating-point arithmetic turned out to be numerically too unstable to be useful at that point.

The main results are shown in Table 13. The results agree with the those obtained with the expo-

[12] `https://github.com/mkarppa/deann-experiments`

[13] `https://github.com/mkarppa/deann-experiments`

Table 7: The Full Set of Results of Evaluating the Different Algorithms Against the Test Set in Milliseconds / Query.

Dataset	Target μ	`Naive`	`RS`	`RSP`	`DEANN`	`DEANNP`	`HBE`	`RSA`	`SKKD`	`SKBT`
ALOI	0.01	1.051	0.050	0.022	0.025	**0.016**	0.623	0.808	58.498	48.353
ALOI	0.001	1.058	0.326	**0.105**	0.211	0.148	12.192	41.411	59.353	47.644
ALOI	0.0001	1.055	6.477	1.698	0.270	**0.197**	*n/a*	*n/a*	55.786	47.916
ALOI	0.00001	1.057	21.781	4.548	0.219	**0.182**	*n/a*	*n/a*	47.930	49.698
CENSUS	0.01	21.201	0.257	**0.045**	0.185	0.082	0.705	19.493	420.866	542.229
CENSUS	0.001	21.821	1.268	**0.192**	0.902	0.215	*n/a*	803.509	350.470	606.949
CENSUS	0.0001	51.656	8.648	1.723	1.237	**0.757**	*n/a*	*n/a*	253.440	462.727
CENSUS	0.00001	22.282	51.162	9.037	1.312	**0.736**	*n/a*	*n/a*	207.266	366.852
COVTYPE	0.01	4.921	1.036	0.128	0.269	**0.055**	0.314	20.534	46.734	50.446
COVTYPE	0.001	4.913	1.797	**0.222**	0.678	0.279	0.629	433.858	26.425	28.755
COVTYPE	0.0001	5.992	8.182	1.824	0.596	**0.473**	*n/a*	*n/a*	11.348	13.923
COVTYPE	0.00001	7.818	94.322	10.177	**0.223**	0.265	*n/a*	*n/a*	3.953	6.098
GLOVE	0.01	11.302	0.011	**0.001**	0.005	0.003	0.347	0.207	674.429	582.650
GLOVE	0.001	11.054	0.019	**0.003**	0.012	0.007	6.617	0.225	699.529	586.988
GLOVE	0.0001	11.050	0.030	**0.005**	0.019	0.014	*n/a*	0.410	704.741	581.489
GLOVE	0.00001	11.101	0.048	**0.015**	0.041	0.022	*n/a*	1.804	709.414	621.037
LAST.FM	0.01	2.593	12.704	2.145	0.227	**0.181**	*n/a*	*n/a*	104.039	94.147
LAST.FM	0.001	2.621	17.183	2.455	0.277	**0.222**	*n/a*	*n/a*	99.893	86.006
LAST.FM	0.0001	2.753	48.630	4.699	0.294	**0.247**	*n/a*	*n/a*	98.582	83.999
LAST.FM	0.00001	2.923	40.249	5.993	0.330	**0.263**	*n/a*	*n/a*	85.621	83.367
MNIST	0.01	1.495	0.029	**0.024**	0.024	0.029	1.577	0.884	94.960	63.640
MNIST	0.001	1.507	0.090	**0.062**	0.091	0.065	12.073	6.886	94.545	61.830
MNIST	0.0001	1.504	0.422	0.213	0.345	**0.202**	*n/a*	8.915	89.835	59.892
MNIST	0.00001	1.524	1.172	0.773	0.609	**0.536**	*n/a*	*n/a*	94.857	64.299
MSD	0.01	4.725	0.053	**0.016**	0.033	0.028	*n/a*	1.196	181.871	209.109
MSD	0.001	4.720	0.196	**0.065**	0.248	0.066	*n/a*	88.375	165.613	197.519
MSD	0.0001	4.729	1.301	**0.234**	0.461	0.266	*n/a*	*n/a*	171.721	203.407
MSD	0.00001	4.754	9.898	1.482	0.754	**0.405**	*n/a*	*n/a*	127.574	169.668
SHUTTLE	0.01	0.407	0.145	**0.017**	0.138	0.024	0.308	8.207	3.671	4.097
SHUTTLE	0.001	0.402	0.864	**0.062**	0.141	0.113	1.595	398.961	2.525	3.873
SHUTTLE	0.0001	0.569	3.088	0.358	0.113	**0.097**	545.129	*n/a*	1.917	3.437
SHUTTLE	0.00001	0.672	*n/a*	0.527	0.070	**0.065**	*n/a*	*n/a*	1.064	2.436
SVHN	0.01	42.094	0.290	**0.189**	0.255	0.448	11.830	56.613	3447.218	2521.555
SVHN	0.001	42.172	0.747	**0.500**	0.698	0.938	*n/a*	56.270	3471.669	2509.883
SVHN	0.0001	42.260	2.207	**1.096**	1.503	1.459	*n/a*	83210.996	3455.433	2495.796
SVHN	0.00001	41.748	3.743	**2.262**	3.758	2.852	*n/a*	*n/a*	3496.380	2445.718

nential kernel; in almost all cases, either `DEANNP` or `RSP` is the fastest algorithm, and `DEANNP` is seldom very far behind, as it can fall back to essentially the same random sampling algorithm. Surprisingly, in the cases of LAST.FM and MNIST at target KDE value of 0.00001, the `Naive` algorithm turned out to be unbeatable. Low error requires too many points to be looked at for approximate algorithms to be very

Table 8: Results Of The Validation Step Of The Experiments Including The Best Performing Parameters For Some Algorithms.

			RSP	DEANNP				HBE		SKKD	
Dataset	Target μ	h	m	k	m	n_ℓ	n_q	ϵ	τ	ℓ	t_r
ALOI	0.01	3.3366	230	0	170	512	1	1.1	0.001	40	0.2
ALOI	0.001	2.0346	1800	0	2100	512	1	0.6	0.0001	90	0.2
ALOI	0.0001	1.3300	29000	170	500	1024	5	n/a	n/a	80	0.2
ALOI	0.00001	0.8648	78000	120	430	1024	5	n/a	n/a	90	0.2
Census	0.01	3.6228	1000	0	800	512	1	0.95	0.0005	80	0.4
Census	0.001	1.9416	6000	0	5000	512	1	n/a	n/a	100	0.25
Census	0.0001	1.1907	40000	700	5500	1024	1	n/a	n/a	10	0.2
Census	0.00001	0.7826	300000	800	5000	4096	5	n/a	n/a	60	0.2
Covtype	0.01	245.8858	5000	0	1300	512	1	1.3	0.0001	90	0.3
Covtype	0.001	119.2450	9000	0	8500	512	1	1.5	0.0001	100	0.2
Covtype	0.0001	63.4887	70000	1300	1400	2048	5	n/a	n/a	30	0.2
Covtype	0.00001	33.1331	350000	300	500	2048	5	n/a	n/a	100	0.2
GloVe	0.01	1.5782	20	0	20	512	1	1.2	0.001	90	0.15
GloVe	0.001	1.0372	50	0	50	512	1	0.75	0.0001	50	0.2
GloVe	0.0001	0.7674	90	0	90	512	1	n/a	n/a	50	0.1
GloVe	0.00001	0.6028	160	0	160	512	1	n/a	n/a	90	0.2
last.fm	0.01	0.0041	75000	60	350	1024	1	n/a	n/a	10	0.2
last.fm	0.001	0.0026	85000	70	800	512	1	n/a	n/a	10	0.15
last.fm	0.0001	0.0019	160000	50	350	2048	5	n/a	n/a	20	0.1
last.fm	0.00001	0.0015	200000	80	450	2048	5	n/a	n/a	100	0.15
MNIST	0.01	532.9814	40	0	40	512	1	1.2	0.001	50	0.2
MNIST	0.001	348.4158	150	0	150	512	1	1.05	0.0001	50	0.0
MNIST	0.0001	255.3234	600	0	600	512	1	n/a	n/a	100	0.5
MNIST	0.00001	198.7733	2200	140	450	512	5	n/a	n/a	50	0.0
MSD	0.01	498.4585	230	0	230	512	1	n/a	n/a	90	0.2
MSD	0.001	312.7048	1200	0	1000	512	1	n/a	n/a	90	0.2
MSD	0.0001	222.0082	5500	0	5300	512	1	n/a	n/a	90	0.1
MSD	0.00001	168.9344	36000	210	2100	2048	5	n/a	n/a	20	0.2
Shuttle	0.01	4.9727	1900	0	1900	512	1	1.1	0.0001	20	0.2
Shuttle	0.001	2.3504	11000	200	500	512	5	1.0	0.00001	60	0.2
Shuttle	0.0001	1.1605	45000	100	500	512	5	0.1	0.000005	100	0.2
Shuttle	0.00001	0.5648	52000	50	0	512	5	n/a	n/a	10	0.2
SVHN	0.01	632.7492	150	0	120	512	1	1.2	0.0001	70	0.2
SVHN	0.001	391.3900	400	0	350	512	1	n/a	n/a	60	0.2
SVHN	0.0001	277.1836	900	0	800	512	1	n/a	n/a	60	0.2
SVHN	0.00001	211.4066	1900	0	2000	512	1	n/a	n/a	60	0.2

effective in this low bandwidth regime.

Table 14 shows complementary results where we have limited ourselves to the case that $k = m$ when tuning the parameters of DEANN and DEANNP. To make the effect even clearer, Table 15 shows the ratio of the runtime for DEANN and DEANNP from Table 14 divided

Table 9: Recall Rates For Approximate Nearest Neighbors Returned By FAISS At Different Parameter Values.

		DEANN					DEANNP				
Dataset	Target μ	k	m	n_ℓ	n_q	R	k	m	n_ℓ	n_q	R
ALOI	0.001	170	400	512	1	0.23	n/a	n/a	n/a	n/a	n/a
ALOI	0.0001	200	430	1024	5	0.72	170	500	1024	5	0.74
ALOI	0.00001	200	270	1024	5	0.72	120	430	1024	5	0.78
Census	0.001	500	3000	4096	1	0.31	n/a	n/a	n/a	n/a	n/a
Census	0.0001	1300	3000	4096	5	0.50	700	5500	1024	1	0.69
Census	0.00001	1400	3000	4096	10	0.77	800	5000	4096	5	0.58
Covtype	0.001	1200	1100	1024	5	0.97	n/a	n/a	n/a	n/a	n/a
Covtype	0.0001	900	1000	1024	5	0.99	1300	1400	2048	5	0.72
Covtype	0.00001	350	0	2048	5	0.85	300	500	2048	5	0.86
last.fm	0.01	50	400	2048	1	0.24	60	350	1024	1	0.88
last.fm	0.001	70	200	2048	5	0.86	70	800	512	1	0.97
last.fm	0.0001	70	300	2048	5	0.86	50	350	2048	5	0.90
last.fm	0.00001	80	400	2048	5	0.86	80	450	2048	5	0.86
MNIST	0.00001	400	300	512	5	0.77	140	450	512	5	0.95
MSD	0.0001	140	1000	2048	5	0.46	n/a	n/a	n/a	n/a	n/a
MSD	0.00001	210	1800	4096	10	0.45	210	2100	2048	5	0.43
Shuttle	0.001	300	350	512	5	0.84	200	500	512	5	0.87
Shuttle	0.0001	200	200	512	5	0.87	100	500	512	5	0.89
Shuttle	0.00001	50	0	512	5	0.98	50	0	512	5	0.98

by the runtime with the best parameters from Table 13. While it is clear that sometimes the number of nearest neighbors and random samples that one ought to look at are unbalanced, the situation is not desperate, and in many cases useful results can be obtained even while restricting the search space.

K Preprocessing Times

Table 16 lists the preprocessing times for the various algorithms. The reported times are in seconds and were obtained when evaluating the test using the Gaussian kernel.

Generally, `RS` has the smallest construction time, which is almost zero, since there is no data structure to construct; the code only stores a pointer to the data. The larger (but still insignificant) construction time for `Naive` is explained by the fact that, upon construction, the data structure stores the Euclidean norm of each vector in the training set. The construction time for `RSP` consists of copying the training set into memory in random order.

For `DEANN` and `DEANNP`, there is a huge variance among the construction times. This is explained by the fact that it is dominated by the construction time for FAISS, and is very sensitive to the parameter n_ℓ, the number of clusters. Thus the construction time can range from almost-insignificant to rather long, depending on the construction time of the underlying NN object. Like `RS`, `DEANN` has no construction time of its own, whereas `DEANNP` performs the same initialization of random sampling as `RSP`.

In all instances where a comparison could be made, `DEANNP` and `DEANN` can be constructed considerably faster than `HBE` or `ASKIT`. `SKKD` and `SKBT` are in a class of their own with respect to construction times, reaching over 7 hours for the Census dataset.

Table 10: Average Relative Error Against The Test Set With Best Parameters.

Dataset	Target μ	Naive	RS	RSP	DEANN	DEANNP	HBE	RSA	SKKD	SKBT
ALOI	0.01	0.000	0.095	0.090	0.100	0.102	0.110	0.099	0.076	0.091
ALOI	0.001	0.000	0.106	0.113	0.104	0.101	0.096	0.097	0.092	0.097
ALOI	0.0001	0.000	0.102	0.099	0.100	0.100	*n/a*	*n/a*	0.098	0.098
ALOI	0.00001	0.000	0.072	0.102	0.092	0.094	*n/a*	*n/a*	0.099	0.098
Census	0.01	0.001	0.081	0.087	0.087	0.094	0.090	0.079	0.092	0.087
Census	0.001	0.002	0.087	0.082	0.094	0.091	*n/a*	0.064	0.091	0.095
Census	0.0001	0.002	0.084	0.088	0.103	0.105	*n/a*	*n/a*	0.088	0.099
Census	0.00001	0.001	0.077	0.079	0.095	0.103	*n/a*	*n/a*	0.094	0.098
Covtype	0.01	0.001	0.047	0.045	0.094	0.094	0.095	0.086	0.098	0.099
Covtype	0.001	0.000	0.093	0.094	0.098	0.097	0.099	0.065	0.081	0.088
Covtype	0.0001	0.000	0.142	0.097	0.096	0.092	*n/a*	*n/a*	0.090	0.090
Covtype	0.00001	0.000	0.074	0.098	0.098	0.093	*n/a*	*n/a*	0.092	0.087
GloVe	0.01	0.000	0.095	0.096	0.095	0.097	0.124	0.089	0.069	0.096
GloVe	0.001	0.000	0.093	0.092	0.093	0.093	0.091	0.090	0.097	0.070
GloVe	0.0001	0.000	0.095	0.095	0.102	0.098	*n/a*	0.108	0.047	0.080
GloVe	0.00001	0.000	0.097	0.098	0.096	0.098	*n/a*	0.060	0.090	0.020
last.fm	0.01	0.001	0.061	0.052	0.111	0.114	*n/a*	*n/a*	0.094	0.091
last.fm	0.001	0.001	0.095	0.092	0.111	0.089	*n/a*	*n/a*	0.086	0.056
last.fm	0.0001	0.002	0.056	0.086	0.109	0.108	*n/a*	*n/a*	0.051	0.073
last.fm	0.00001	0.004	0.093	0.088	0.092	0.096	*n/a*	*n/a*	0.105	0.161
MNIST	0.01	0.000	0.090	0.094	0.091	0.092	0.103	0.093	0.082	0.093
MNIST	0.001	0.000	0.098	0.097	0.094	0.096	0.093	0.083	0.000	0.000
MNIST	0.0001	0.000	0.088	0.095	0.092	0.093	*n/a*	0.104	0.006	0.000
MNIST	0.00001	0.000	0.102	0.100	0.098	0.094	*n/a*	*n/a*	0.000	0.000
MSD	0.01	0.000	0.103	0.097	0.097	0.100	*n/a*	0.068	0.080	0.087
MSD	0.001	0.000	0.101	0.148	0.091	0.107	*n/a*	0.097	0.091	0.095
MSD	0.0001	0.000	0.148	0.096	0.107	0.098	*n/a*	*n/a*	0.047	0.098
MSD	0.00001	0.000	0.096	0.091	0.103	0.100	*n/a*	*n/a*	0.096	0.099
Shuttle	0.01	0.000	0.094	0.095	0.096	0.098	0.105	0.091	0.080	0.093
Shuttle	0.001	0.000	0.119	0.102	0.099	0.101	0.090	0.069	0.091	0.095
Shuttle	0.0001	0.002	0.120	0.065	0.096	0.102	0.097	*n/a*	0.095	0.094
Shuttle	0.00001	0.002	*n/a*	0.084	0.073	0.073	*n/a*	*n/a*	0.094	0.090
SVHN	0.01	0.000	0.081	0.081	0.092	0.093	0.109	0.048	0.098	0.098
SVHN	0.001	0.000	0.084	0.084	0.088	0.090	*n/a*	0.080	0.099	0.099
SVHN	0.0001	0.000	0.076	0.087	0.090	0.091	*n/a*	0.053	0.099	0.099
SVHN	0.00001	0.000	0.090	0.098	0.089	0.091	*n/a*	*n/a*	0.099	0.099

Table 11: Average Relative Error Against The Test Set With Best Parameters.

Dataset	Target μ	`Naive`	`RS`	`RSP`	`DEANN`	`DEANNP`	`HBE`	`RSA`	`SKKD`	`SKBT`
ALOI	0.01	0.000	0.095	0.090	0.100	0.102	0.110	0.099	0.076	0.091
ALOI	0.001	0.000	0.106	0.113	0.104	0.101	0.096	0.097	0.092	0.097
ALOI	0.0001	0.000	0.102	0.099	0.100	0.100	*n/a*	*n/a*	0.098	0.098
ALOI	0.00001	0.000	0.072	0.102	0.092	0.094	*n/a*	*n/a*	0.099	0.098
LAST.FM	0.01	0.001	0.061	0.052	0.111	0.114	*n/a*	*n/a*	0.094	0.091
LAST.FM	0.001	0.001	0.095	0.092	0.111	0.089	*n/a*	*n/a*	0.086	0.056
LAST.FM	0.0001	0.002	0.056	0.086	0.109	0.108	*n/a*	*n/a*	0.051	0.073
LAST.FM	0.00001	0.004	0.093	0.088	0.092	0.096	*n/a*	*n/a*	0.105	0.161
SVHN	0.01	0.000	0.081	0.081	0.092	0.093	0.109	0.048	0.098	0.098
SVHN	0.001	0.000	0.084	0.084	0.088	0.090	*n/a*	0.080	0.099	0.099
SVHN	0.0001	0.000	0.076	0.087	0.090	0.091	*n/a*	0.053	0.099	0.099
SVHN	0.00001	0.000	0.090	0.098	0.089	0.091	*n/a*	*n/a*	0.099	0.099

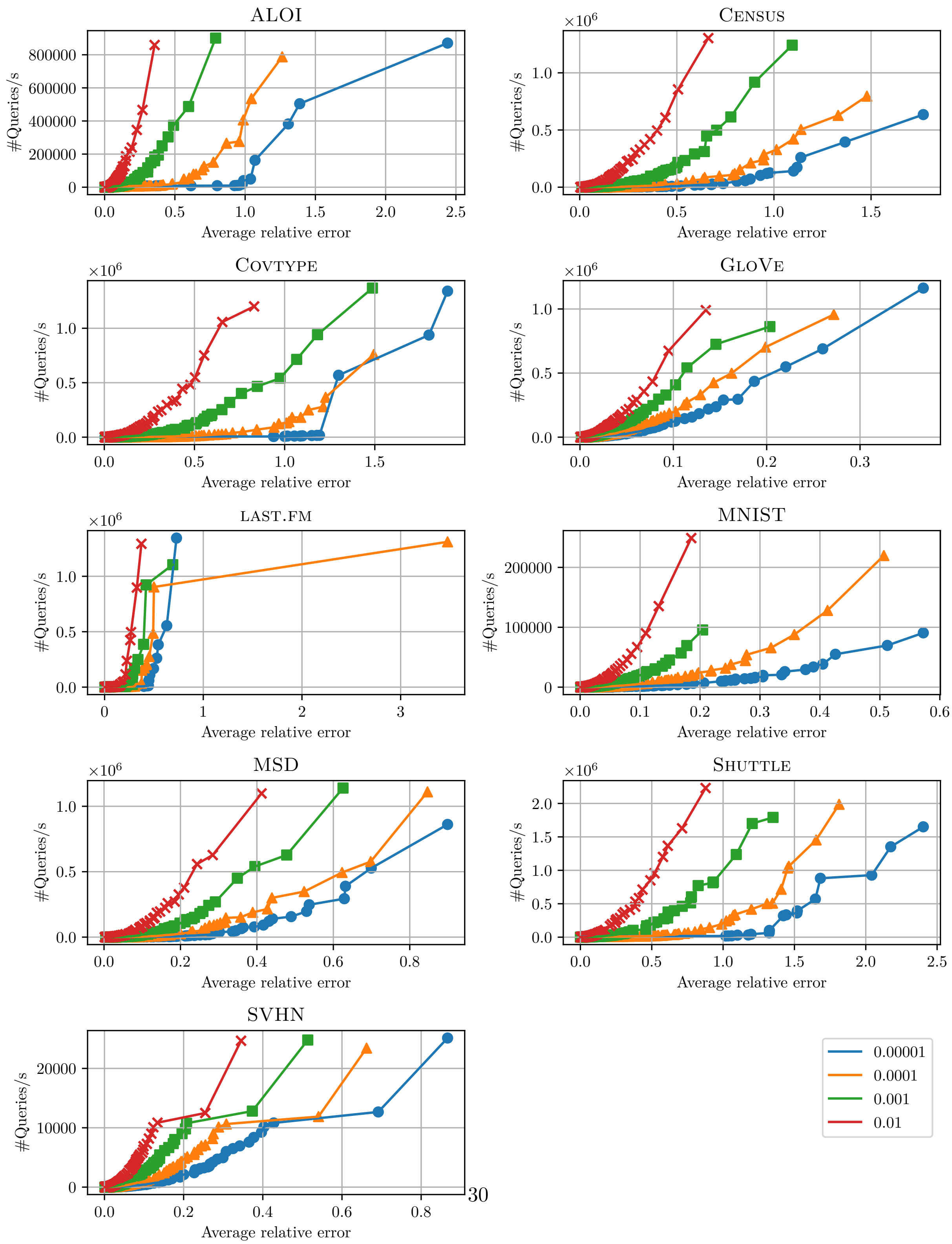

Figure 1: Average Relative Error Vs. Query Time Tradeoff At Different Parameter Choices, Reported For `DEANNP`.

Table 12: Query Times in Milliseconds / Query and Relative Errors Using Fixed Parameters ($k = 100$, $m = 1000$, $n_\ell = 512$, $n_q = 1$) vs. Parameters from Grid Search.

		Grid Search	Fixed parameters	
Dataset	Target μ	Query Time	Query Time	Relative Error
ALOI	0.01	0.014	0.188	0.036
ALOI	0.001	0.156	0.169	0.070
ALOI	0.0001	0.229	0.167	0.117
ALOI	0.00001	0.209	0.167	0.174
Census	0.01	0.031	0.329	0.082
Census	0.001	0.189	0.334	0.179
Census	0.0001	0.876	0.345	0.317
Census	0.00001	0.889	0.331	0.452
Covtype	0.01	0.046	0.177	0.099
Covtype	0.001	0.272	0.176	0.206
Covtype	0.0001	0.536	0.222	0.358
Covtype	0.00001	0.357	0.184	0.359
GloVe	0.01	0.001	0.316	0.014
GloVe	0.001	0.003	0.321	0.020
GloVe	0.0001	0.005	0.316	0.029
GloVe	0.00001	0.008	0.321	0.039
last.fm	0.01	0.204	0.253	0.073
last.fm	0.001	0.240	0.230	0.140
last.fm	0.0001	0.269	0.226	0.109
last.fm	0.00001	0.286	0.230	0.214
MNIST	0.01	0.015	0.492	0.018
MNIST	0.001	0.052	0.494	0.034
MNIST	0.0001	0.212	0.490	0.059
MNIST	0.00001	0.724	0.493	0.104
MSD	0.01	0.012	0.202	0.047
MSD	0.001	0.054	0.201	0.083
MSD	0.0001	0.266	0.199	0.126
MSD	0.00001	0.426	0.208	0.175
Shuttle	0.01	0.025	0.091	0.090
Shuttle	0.001	0.133	0.120	0.143
Shuttle	0.0001	0.111	0.109	0.224
Shuttle	0.00001	0.078	0.101	0.274
SVHN	0.01	0.145	3.133	0.032
SVHN	0.001	0.423	2.925	0.052
SVHN	0.0001	1.063	2.929	0.079
SVHN	0.00001	2.404	2.894	0.117

Table 13: Results of Evaluating the Different Algorithms Against the Test Set in Milliseconds / Query with the Gaussian Kernel.

Dataset	Target μ	Naive	RS	RSP	DEANN	DEANNP	HBE	RSA	SKKD	SKBT	ASKIT
ALOI	0.01	1.036	0.329	**0.084**	0.286	0.089	4.748	15.502	51.882	43.581	0.491
ALOI	0.001	1.002	7.858	1.676	0.768	**0.296**	*n/a*	971.296	48.001	44.614	1.242
ALOI	0.0001	1.079	16.646	5.128	1.552	**0.519**	*n/a*	*n/a*	36.452	41.628	1.252
ALOI	0.00001	1.894	*n/a*	*n/a*	0.515	**0.446**	*n/a*	*n/a*	24.127	31.434	1.236
Census	0.01	21.167	0.718	**0.117**	0.749	0.207	2.056	69.093	286.878	375.744	*n/a*
Census	0.001	22.291	6.963	1.239	1.550	**0.891**	*n/a*	*n/a*	160.692	353.577	*n/a*
Census	0.0001	23.465	65.938	15.354	2.350	**1.400**	*n/a*	*n/a*	116.222	295.493	*n/a*
Census	0.00001	28.710	278.683	62.955	2.520	**1.582**	*n/a*	*n/a*	88.905	249.631	*n/a*
Covtype	0.01	6.305	0.612	**0.091**	0.637	0.128	0.708	43.133	23.449	24.650	3.334
Covtype	0.001	6.467	4.844	0.808	1.294	**0.672**	6.032	*n/a*	8.982	10.943	4.970
Covtype	0.0001	5.984	47.721	6.112	1.423	**1.079**	*n/a*	*n/a*	2.900	4.348	5.289
Covtype	0.00001	5.027	*n/a*	*n/a*	0.242	**0.225**	*n/a*	*n/a*	0.628	1.171	5.495
GloVe	0.01	10.650	0.021	**0.004**	0.021	0.005	8.908	0.867	577.850	549.691	*n/a*
GloVe	0.001	10.675	0.071	0.021	0.065	**0.013**	*n/a*	1.769	574.288	551.816	1.423
GloVe	0.0001	10.447	0.163	**0.042**	0.162	0.069	*n/a*	38.040	578.709	551.685	3.548
GloVe	0.00001	10.320	0.880	0.189	0.429	**0.167**	*n/a*	1862.553	630.915	499.688	*n/a*
lastfm	0.01	2.989	32.091	5.142	0.583	**0.321**	*n/a*	*n/a*	90.750	78.711	*n/a*
lastfm	0.001	3.022	51.482	7.119	1.018	**0.496**	*n/a*	*n/a*	83.927	86.719	*n/a*
lastfm	0.0001	3.046	43.680	7.196	2.683	**0.848**	*n/a*	*n/a*	89.954	87.373	*n/a*
lastfm	0.00001	**2.888**	*n/a*	7.133	7.260	3.841	*n/a*	*n/a*	88.133	*n/a*	*n/a*
MNIST	0.01	1.477	0.132	0.087	0.105	**0.067**	16.628	8.517	95.594	63.673	0.684
MNIST	0.001	1.466	0.939	0.602	0.679	**0.382**	*n/a*	146.002	90.962	59.170	3.058
MNIST	0.0001	1.594	5.409	3.385	1.655	**1.085**	*n/a*	19838.656	96.430	63.810	2.984
MNIST	0.00001	**1.457**	32.994	13.151	3.743	2.602	*n/a*	*n/a*	95.879	63.010	3.226
MSD	0.01	4.891	3.546	0.434	0.574	**0.256**	*n/a*	*n/a*	161.008	188.234	5.108
MSD	0.001	5.224	36.903	6.724	1.603	**0.663**	*n/a*	*n/a*	131.380	162.818	5.251
MSD	0.0001	5.585	70.699	13.940	4.316	**1.693**	*n/a*	*n/a*	128.709	180.890	5.126
MSD	0.00001	5.982	101.900	13.784	10.268	**3.783**	*n/a*	*n/a*	115.804	175.152	5.262
Shuttle	0.01	0.519	0.368	**0.045**	0.270	0.086	1.153	16.054	2.291	2.919	0.329
Shuttle	0.001	0.555	2.993	0.233	0.316	**0.175**	36.222	*n/a*	1.297	2.631	0.331
Shuttle	0.0001	0.497	*n/a*	*n/a*	**0.084**	0.087	*n/a*	*n/a*	0.622	2.230	0.363
Shuttle	0.00001	0.411	*n/a*	*n/a*	0.049	**0.047**	*n/a*	*n/a*	0.284	1.972	0.407
SVHN	0.01	40.710	1.129	**0.836**	0.856	1.145	*n/a*	*n/a*	2797.206	1925.226	*n/a*
SVHN	0.001	40.653	4.440	4.545	4.377	**2.841**	*n/a*	*n/a*	2570.958	1920.286	*n/a*
SVHN	0.0001	40.241	55.776	23.587	12.627	**11.466**	*n/a*	*n/a*	2548.359	1930.954	*n/a*
SVHN	0.00001	41.309	279.773	140.986	52.181	**36.585**	*n/a*	*n/a*	2529.778	1869.640	*n/a*

Table 14: Results of Evaluating the Different Algorithms Against the Test Set in Milliseconds / Query with the Gaussian Kernel with the Restriction that $k = m$.

Dataset	Target μ	`Naive`	`RS`	`RSP`	`DEANN`	`DEANNP`	`HBE`	`RSA`	`SKKD`	`SKBT`	`ASKIT`
ALOI	0.01	1.036	0.329	**0.084**	0.270	0.193	4.748	15.502	51.882	43.581	0.491
ALOI	0.001	1.002	7.858	1.676	0.768	**0.338**	*n/a*	971.296	48.001	44.614	1.242
ALOI	0.0001	**1.079**	16.646	5.128	2.586	1.245	*n/a*	*n/a*	36.452	41.628	1.252
ALOI	0.00001	1.894	*n/a*	*n/a*	7.061	2.393	*n/a*	*n/a*	24.127	31.434	**1.236**
Census	0.01	21.167	0.718	**0.117**	0.778	0.561	2.056	69.093	286.878	375.744	*n/a*
Census	0.001	22.291	6.963	1.239	2.098	**1.192**	*n/a*	*n/a*	160.692	353.577	*n/a*
Census	0.0001	23.465	65.938	15.354	3.503	**2.570**	*n/a*	*n/a*	116.222	295.493	*n/a*
Census	0.00001	28.710	278.683	62.955	3.496	**2.625**	*n/a*	*n/a*	88.905	249.631	*n/a*
Covtype	0.01	6.305	0.612	**0.091**	0.654	0.399	0.708	43.133	23.449	24.650	3.334
Covtype	0.001	6.467	4.844	**0.808**	1.900	0.983	6.032	*n/a*	8.982	10.943	4.970
Covtype	0.0001	5.984	47.721	6.112	2.443	**1.924**	*n/a*	*n/a*	2.900	4.348	5.289
Covtype	0.00001	5.027	*n/a*	*n/a*	0.594	**0.477**	*n/a*	*n/a*	0.628	1.171	5.495
GloVe	0.01	10.650	0.021	**0.004**	0.172	0.168	8.908	0.867	577.850	549.691	*n/a*
GloVe	0.001	10.675	0.071	**0.021**	0.283	0.234	*n/a*	1.769	574.288	551.816	1.423
GloVe	0.0001	10.447	0.163	**0.042**	0.393	0.359	*n/a*	38.040	578.709	551.685	3.548
GloVe	0.00001	10.320	0.880	**0.189**	0.693	0.471	*n/a*	1862.553	630.915	499.688	*n/a*
lastfm	0.01	2.989	32.091	5.142	1.077	**1.029**	*n/a*	*n/a*	90.750	78.711	*n/a*
lastfm	0.001	**3.022**	51.482	7.119	7.126	3.992	*n/a*	*n/a*	83.927	86.719	*n/a*
lastfm	0.0001	**3.046**	43.680	7.196	14.758	6.060	*n/a*	*n/a*	89.954	87.373	*n/a*
lastfm	0.00001	**2.888**	*n/a*	7.133	30.810	10.142	*n/a*	*n/a*	88.133	*n/a*	*n/a*
MNIST	0.01	1.477	0.132	**0.087**	0.300	0.310	16.628	8.517	95.594	63.673	0.684
MNIST	0.001	1.466	0.939	0.602	0.679	**0.572**	*n/a*	146.002	90.962	59.170	3.058
MNIST	0.0001	1.594	5.409	3.385	1.655	**1.392**	*n/a*	19838.656	96.430	63.810	2.984
MNIST	0.00001	**1.457**	32.994	13.151	4.944	3.987	*n/a*	*n/a*	95.879	63.010	3.226
MSD	0.01	4.891	3.546	**0.434**	1.118	0.671	*n/a*	*n/a*	161.008	188.234	5.108
MSD	0.001	5.224	36.903	6.724	2.635	**2.143**	*n/a*	*n/a*	131.380	162.818	5.251
MSD	0.0001	5.585	70.699	13.940	13.175	6.986	*n/a*	*n/a*	128.709	180.890	**5.126**
MSD	0.00001	5.982	101.900	13.784	18.616	15.204	*n/a*	*n/a*	115.804	175.152	**5.262**
Shuttle	0.01	0.519	0.368	**0.045**	0.270	0.194	1.153	16.054	2.291	2.919	0.329
Shuttle	0.001	0.555	2.993	**0.233**	0.316	0.272	36.222	*n/a*	1.297	2.631	0.331
Shuttle	0.0001	0.497	*n/a*	*n/a*	**0.133**	0.156	*n/a*	*n/a*	0.622	2.230	0.363
Shuttle	0.00001	0.411	*n/a*	*n/a*	**0.051**	0.052	*n/a*	*n/a*	0.284	1.972	0.407
SVHN	0.01	40.710	1.129	**0.836**	4.370	3.141	*n/a*	*n/a*	2797.206	1925.226	*n/a*
SVHN	0.001	40.653	**4.440**	4.545	6.220	6.680	*n/a*	*n/a*	2570.958	1920.286	*n/a*
SVHN	0.0001	40.241	55.776	23.587	**13.814**	14.836	*n/a*	*n/a*	2548.359	1930.954	*n/a*
SVHN	0.00001	**41.309**	279.773	140.986	52.181	43.192	*n/a*	*n/a*	2529.778	1869.640	*n/a*

Table 15: Runtime Ratio Between Best Parameters and $k = m$ Parameters.

Dataset	Target μ	DEANN	DEANNP
ALOI	0.01	0.943	2.178
ALOI	0.001	1.000	1.141
ALOI	0.0001	1.666	2.397
ALOI	0.00001	13.707	5.366
Census	0.01	1.040	2.707
Census	0.001	1.353	1.339
Census	0.0001	1.491	1.836
Census	0.00001	1.387	1.659
Covtype	0.01	1.028	3.122
Covtype	0.001	1.468	1.463
Covtype	0.0001	1.716	1.784
Covtype	0.00001	2.454	2.115
GloVe	0.01	8.295	33.587
GloVe	0.001	4.378	18.506
GloVe	0.0001	2.430	5.221
GloVe	0.00001	1.616	2.811
last.fm	0.01	1.848	3.209
last.fm	0.001	7.000	8.051
last.fm	0.0001	5.500	7.148
last.fm	0.00001	4.244	2.640
MNIST	0.01	2.861	4.647
MNIST	0.001	1.000	1.500
MNIST	0.0001	1.000	1.282
MNIST	0.00001	1.321	1.532
MSD	0.01	1.949	2.618
MSD	0.001	1.644	3.234
MSD	0.0001	3.052	4.126
MSD	0.00001	1.813	4.019
Shuttle	0.01	1.000	2.245
Shuttle	0.001	1.000	1.550
Shuttle	0.0001	1.576	1.779
Shuttle	0.00001	1.044	1.106
SVHN	0.01	5.103	2.743
SVHN	0.001	1.421	2.351
SVHN	0.0001	1.094	1.294
SVHN	0.00001	1.000	1.181

Table 16: Preprocessing Times When Evaluating the Different Algorithms Against the Test Set with the Gaussian Kernel in Seconds.

Dataset	Target μ	Naive	RS	RSP	DEANN	DEANNP	HBE	RSA	SKKD	SKBT	ASKIT
ALOI	0.01	0.006	**0.000**	0.055	0.377	8.775	22.285	0.000	4.929	5.155	21.455
ALOI	0.001	0.006	**0.000**	0.059	1.078	0.975	*n/a*	0.000	5.349	5.680	6.626
ALOI	0.0001	0.006	**0.000**	0.056	0.146	0.153	*n/a*	*n/a*	5.588	5.756	6.425
ALOI	0.00001	**0.006**	*n/a*	*n/a*	0.154	0.146	*n/a*	*n/a*	5.782	4.949	6.372
Census	0.01	0.081	**0.000**	0.945	3.568	14.269	101.727	0.000	25573.250	22917.678	*n/a*
Census	0.001	0.077	**0.000**	0.922	14.238	39.615	*n/a*	*n/a*	25405.120	25516.643	*n/a*
Census	0.0001	0.075	**0.000**	0.971	3.346	6.440	*n/a*	*n/a*	25486.341	25455.323	*n/a*
Census	0.00001	0.068	**0.000**	0.938	2.498	3.345	*n/a*	*n/a*	23033.750	23078.722	*n/a*
Covtype	0.01	0.017	**0.000**	0.179	26.056	0.336	11.008	0.000	5.644	4.098	572.824
Covtype	0.001	0.017	**0.000**	0.189	0.439	7.590	40.260	*n/a*	5.413	4.360	339.547
Covtype	0.0001	0.016	**0.000**	0.186	0.336	0.340	*n/a*	*n/a*	5.443	4.576	76.198
Covtype	0.00001	**0.016**	*n/a*	*n/a*	0.593	0.621	*n/a*	*n/a*	5.026	4.010	75.267
GloVe	0.01	0.052	**0.000**	0.553	1.067	2.521	135.074	0.000	29.145	30.301	*n/a*
GloVe	0.001	0.049	**0.000**	0.553	13.145	2.561	*n/a*	0.000	28.012	30.064	175.622
GloVe	0.0001	0.044	**0.000**	0.553	1.240	1.267	*n/a*	0.000	28.417	30.106	578.841
GloVe	0.00001	0.044	**0.000**	0.542	144.898	12.779	*n/a*	0.000	33.952	26.619	*n/a*
last.fm	0.01	0.010	**0.000**	0.106	2.320	7.022	*n/a*	*n/a*	4.327	3.140	*n/a*
last.fm	0.001	0.010	**0.000**	0.105	2.317	2.331	*n/a*	*n/a*	4.168	3.226	*n/a*
last.fm	0.0001	0.009	**0.000**	0.108	0.249	2.312	*n/a*	*n/a*	4.148	3.193	*n/a*
last.fm	0.00001	**0.009**	*n/a*	0.107	0.794	0.850	*n/a*	*n/a*	4.125	*n/a*	*n/a*
MNIST	0.01	0.017	**0.000**	0.159	1.700	0.813	100.323	0.000	12.369	11.154	14.053
MNIST	0.001	0.017	**0.000**	0.156	1.739	1.650	*n/a*	0.000	12.051	10.671	4.424
MNIST	0.0001	0.016	**0.000**	0.163	0.838	1.768	*n/a*	0.000	12.097	11.201	4.423
MNIST	0.00001	0.016	**0.000**	0.155	0.447	0.443	*n/a*	*n/a*	12.461	11.022	4.397
MSD	0.01	0.020	**0.000**	0.223	9.319	9.395	*n/a*	*n/a*	12.378	10.359	144.805
MSD	0.001	0.021	**0.000**	0.227	0.443	0.785	*n/a*	*n/a*	10.183	9.093	143.934
MSD	0.0001	0.019	**0.000**	0.223	0.432	0.435	*n/a*	*n/a*	11.974	10.106	145.066
MSD	0.00001	0.019	**0.000**	0.224	0.446	0.460	*n/a*	*n/a*	11.614	10.028	144.940
Shuttle	0.01	0.001	**0.000**	0.007	0.238	0.070	2.006	0.000	0.687	0.658	0.593
Shuttle	0.001	0.001	**0.000**	0.007	0.049	0.067	63.746	*n/a*	0.659	0.621	1.634
Shuttle	0.0001	**0.001**	*n/a*	*n/a*	0.046	0.048	*n/a*	*n/a*	0.670	0.639	1.679
Shuttle	0.00001	**0.001**	*n/a*	*n/a*	0.262	0.257	*n/a*	*n/a*	0.636	0.619	1.542
SVHN	0.01	0.583	**0.000**	5.590	262.613	1651.727	*n/a*	*n/a*	454.764	473.117	*n/a*
SVHN	0.001	0.582	**0.000**	5.624	35.396	13.996	*n/a*	*n/a*	446.738	462.035	*n/a*
SVHN	0.0001	0.657	**0.000**	5.662	36.748	35.219	*n/a*	*n/a*	445.377	463.516	*n/a*
SVHN	0.00001	0.772	**0.000**	5.592	16.252	16.640	*n/a*	*n/a*	431.374	452.096	*n/a*