Faster Linear Algebra for Distance Matrices

Piotr Indyk
MIT
indyk@mit.edu

Sandeep Silwal
MIT
silwal@mit.edu

Abstract

The distance matrix of a dataset X of n points with respect to a distance function f represents all pairwise distances between points in X induced by f. Due to their wide applicability, distance matrices and related families of matrices have been the focus of many recent algorithmic works. We continue this line of research and take a broad view of algorithm design for distance matrices with the goal of designing fast algorithms, which are specifically tailored for distance matrices, for fundamental linear algebraic primitives. Our results include efficient algorithms for computing matrix-vector products for a wide class of distance matrices, such as the ℓ_1 metric for which we get a linear runtime, as well as an $\Omega(n^2)$ lower bound for any algorithm which computes a matrix-vector product for the ℓ_∞ case, showing a separation between the ℓ_1 and the ℓ_∞ metrics. Our upper bound results, in conjunction with recent works on the matrix-vector query model, have many further downstream applications, including the fastest algorithm for computing a relative error low-rank approximation for the distance matrix induced by ℓ_1 and ℓ_2^2 functions and the fastest algorithm for computing an additive error low-rank approximation for the ℓ_2 metric, in addition to applications for fast matrix multiplication among others. We also give algorithms for constructing distance matrices and show that one can construct an approximate ℓ_2 distance matrix in time faster than the bound implied by the Johnson-Lindenstrauss lemma.

1 Introduction

Given a set of n points $X = \{x_1, \ldots, x_n\}$, the distance matrix of X with respect to a distance function f is defined as the $n \times n$ matrix A satisfying $A_{i,j} = f(x_i, x_j)$. Distances matrices are ubiquitous objects arising in various applications ranging from learning image manifolds [TSL00, WS06], signal processing [SY07], biological analysis [HS93], and non-linear dimensionality reduction [Kru64, Kru78, TSL00, CC08], to name a few[1]. Unfortunately, explicitly computing and storing A requires at least $\Omega(n^2)$ time and space. Such complexities are prohibitive for scaling to large datasets.

A silver lining is that in many settings, the matrix A is not explicitly required. Indeed in many applications, it suffices to compute some underlying function or property of A, such as the eigenvalues and eigenvectors of A or a low-rank approximation of A. Thus an algorithm designer can hope to use the special geometric structure encoded by A to design faster algorithms tailored for such tasks.

Therefore, it is not surprising that many recent works explicitly take advantage of the underlying geometric structure of distance matrices, and other related families of matrices, to design fast algorithms (see Section 1.2 for a thorough discussion of prior works). In this work, we continue this line of research and take a broad view of algorithm design for distance matrices. Our main motivating question is the following:

[1] We refer the reader to the survey [DPRV15] for a more thorough discussion of applications of distance matrices.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

Can we design algorithms for fundamental linear algebraic primitives which are specifically tailored for distance matrices and related families of matrices?

We make progress towards the motivating question by studying three of the most fundamental primitives in algorithmic linear algebra. Specifically:

1. We study upper and lower bounds for computing matrix-vector products for a wide array of distance matrices,
2. We give algorithms for multiplying distance matrices faster than general matrices, and,
3. We give fast algorithms for constructing distance matrices.

1.1 Our Results

We now describe our contributions in more detail.

1. We study upper and lower bounds for constructing matrix-vector queries for a wide array of distance matrices.

A matrix-vector query algorithm accepts a vector z as input and outputs the vector Az. There is substantial motivation for studying such queries. Indeed, there is now a rich literature for fundamental linear algebra algorithms which are in the "matrix free" or "implicit" model. These algorithms only assume access to the underlying matrix via matrix-vector queries. Some well known algorithms in this model include the power method for computing eigenvalues and the conjugate gradient descent method for solving a system of linear equations. For many fundamental functions of A, nearly optimal bounds in terms of the number of queries have been achieved [MM15, BHSW20, BCW22]. Furthermore, having access to matrix-vector queries also allows the simulation of any randomized sketching algorithm, a well studied algorithmic paradigm in its own right [Woo14]. This is because randomized sketching algorithms operate on the matrix ΠA or $A\Pi$ where Π is a suitably chosen random matrix, such as a Gaussian matrix. Typically, Π is chosen so that the sketches ΠA or $A\Pi$ have significantly smaller row or column dimension compared to A. If A is symmetric, we can easily acquire both types of matrix sketches via a small number of matrix-vector queries.

Therefore, creating efficient versions of matrix-vector queries for distance matrices automatically lends itself to many further downstream applications. We remark that our algorithms can access to the set of input points but *do not* explicitly create the distance matrix. A canonical example of our upper bound results is the construction of matrix-vector queries for the function $f(x, y) = \|x - y\|_p^p$.

Theorem 1.1. *Let $p \geq 1$ be an integer. Suppose we are given a dataset of n points $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$. X implicitly defines the matrix $A_{i,j} = \|x_i - x_j\|_p^p$. Given a query $z \in \mathbb{R}^n$, we can compute Az exactly in time $O(ndp)$. If p is odd, we also require $O(nd \log n)$ preprocessing time.*

We give similar guarantees for a wide array of functions f and we refer the reader to Table 1 which summarizes our matrix-vector query upper bound results. Note that some of the functions f we study in Table 1 do not necessarily induce a metric in the strict mathematical sense (for example the function $f(x, y) = \|x - y\|_2^2$ does not satisfy the triangle inequality). Nevertheless, we still refer to such functions under the broad umbrella term of "distance functions" for ease of notation. We always explicitly state the function f we are referring to.

Crucially, most of our bounds have a linear dependency on n which allows for scalable computation as the size of the dataset X grows. Our upper bounds are optimal in many cases, see Theorem A.13.

Combining our upper bound results with optimized matrix-free methods, immediate corollaries of our results include faster algorithms for eigenvalue and singular value computations and low-rank approximations. Low-rank approximation is of special interest as it has been widely studied for distance matrices; for low-rank approximation, our bounds outperform prior results for specific distance functions. For example, for the ℓ_1 and ℓ_2^2 case (and in general PSD matrices), [BCW20] showed that a rank-k approximation can be found in time $O(ndk/\varepsilon + nk^{w-1}/\varepsilon^{w-1})$. This bound has extra poly$(1/\varepsilon)$ overhead compared to our bound stated in Table 2. The work of [IVWW19] has a worse poly$(k, 1/\varepsilon)$ overhead for an additive error approximation for the ℓ_2 case. See Section 1.2 for further discussion of prior works. The downstream applications of matrix-vector queries are summarized in Table 2.

Function	$f(x, y)$	Preprocessing	Query Time	Reference
ℓ_p^p for p even	$\|x - y\|_p^p$	–	$O(ndp)$	Thms. A.1 / A.3
ℓ_p^p for p odd	$\|x - y\|_p^p$	$O(nd \log n)$	$O(ndp)$	Thms. 2.2 / A.4
Mixed ℓ_∞	$\max_{i,j} \|x_i - y_j\|$	$O(nd \log n)$	$O(n^2)$	Thm. A.5
Mahalanobis Distance^2	$x^T M y$	$O(nd^2)$	$O(nd)$	Thm. A.6
Polynomial Kernel	$\langle x, y \rangle^p$	–	$O(nd^p)$	Thm. A.7
Total Variation Distance	$\mathrm{TV}(x, y)$	$O(nd \log n)$	$O(nd)$	Thm. A.8
KL Divergence	$\mathrm{D}_{\mathrm{KL}}(x \,\|\, y)$	–	$O(nd)$	Thm. A.2
Symmetric Divergence	$\mathrm{D}_{\mathrm{KL}}(x \,\|\, y) + \mathrm{D}_{\mathrm{KL}}(y \,\|\, x)$	–	$O(nd)$	Thm. A.9
Cross Entropy	$H(x, y)$	–	$O(nd)$	Thm. A.9
Hellinger Distance^2	$\sum_{i=1}^d \sqrt{x(i)y(i)}$	–	$O(nd)$	Thm. A.10

Table 1: A summary of our results for exact matrix-vector queries.

We also study fundamental limits for any upper bound algorithms. In particular, we show that *no algorithm* can compute a matrix-vector query for general inputs for the ℓ_∞ metric in subquadratic time, assuming a standard complexity-theory assumption called the *Strong Exponential Time Hypothesis (SETH)* [IP01, IPZ01].

Theorem 1.2. *For any $\alpha > 0$ and $d = \omega(\log n)$, any algorithm for exactly computing Az for any input z, where A is the ℓ_∞ distance matrix, requires $\Omega(n^{2-\alpha})$ time (assuming SETH).*

This shows a separation between the functions listed in Table 1 and the ℓ_∞ metric. Surprisingly, we can create queries for the *approximate* matrix-vector query in substantially faster time.

Theorem 1.3. *Suppose $X \subseteq \{0, 1, \ldots, O(1)\}^d$. We can compute By in time $O(n \cdot d^{O(\sqrt{d} \log(d/\varepsilon))})$ where $\|A - B\|_\infty \leq \varepsilon$.*

To put the above result into context, the lower bound of Theorem 1.2 holds for points sets in $\{0, 1, 2\}^d$ in $d \approx \log n$ dimensions. In contrast, if we relax to an approximation guarantee, we can obtain a subquadratic-time algorithm for d up to $\Theta(\log^2(n)/\log\log(n))$.

Finally, we provide a general understanding of the limits of our upper bound techniques. In Theorem B.1, we show that essentially the only f for which our upper bound techniques apply have a "linear structure" after a suitable transformation. We refer to Appendix Section B for details.

2. We give algorithms for multiplying distance matrices faster than general matrices.

Fast matrix-vector queries also automatically imply fast matrix multiplication, which can be reduced to a series of matrix-vector queries. For concreteness, if f is the ℓ_p^p function which induces A, and B is any $n \times n$ matrix, we can compute AB in time $O(n^2 dp)$. This is substantially faster than the general matrix multiplication bound of $n^w \approx n^{2.37}$. We also give an improvement of this result for the case where we are multiplying two distance matrices arising from ℓ_2^2. See Table 2 for summary.

3. We give fast algorithms for constructing distance matrices.

Finally, we give fast algorithms for constructing approximate distance matrices. To establish some context, recall the classical Johnson-Lindenstrauss (JL) lemma which (roughly) states that a random projection of a dataset $X \subset \mathbb{R}^d$ of size n onto a dimension of size $O(\log n)$ approximately preserves all pairwise distances [JL84]. A common applications of this lemma is to *instantiate* the ℓ_2 distance matrix. A naive algorithm which computes the distance matrix after performing the JL projection requires approximately $O(n^2 \log n)$ time. Surprisingly, we show that the JL lemma is not tight with respect to creating an approximate ℓ_2 distance matrix; we show that one can initialize the ℓ_2 distance in an asymptotically better runtime.

Theorem 1.4 (Informal; See Theorem D.5). *We can calculate a $n \times n$ matrix B such that each (i, j) entry B_{ij} of B satisfies $(1 - \varepsilon)\|x_i - x_j\|_2 \leq B_{ij} \leq (1 + \varepsilon)\|x_i - x_j\|_2$ in time $O(\varepsilon^{-2} n^2 \log^2(\varepsilon^{-1} \log n))$.*

Problem	$f(x,y)$	Runtime	Prior Work
$(1+\varepsilon)$ Relative error rank k low-rank approximation	ℓ_1, ℓ_2^2	$\tilde{O}\left(\frac{ndk}{\varepsilon^{1/3}} + \frac{nk^{w-1}}{\varepsilon^{(w-1)/3}}\right)$ Theorem C.4	$O\left(\frac{ndk}{\varepsilon} + \frac{nk^{w-1}}{\varepsilon^{w-1}}\right)$ [BCW20]
Additive error $\varepsilon\|A\|_F$ rank k low-rank approximation	ℓ_2	$\tilde{O}\left(\frac{ndk}{\varepsilon^{1/3}} + \frac{nk^{w-1}}{\varepsilon^{(w-1)/3}}\right)$ Theorem C.6	$\tilde{O}(nd \cdot \text{poly}(k, 1/\varepsilon))$ [IVWW19]
$(1+\varepsilon)$ Relative error rank k low-rank approximation	Any in Table 1	$\tilde{O}\left(\frac{Tk}{\varepsilon^{1/3}} + \frac{nk^{w-1}}{\varepsilon^{(w-1)/3}}\right)$ Theorem C.7	$\tilde{O}\left(\frac{n^2dk}{\varepsilon^{1/3}} + \frac{nk^{w-1}}{\varepsilon^{(w-1)/3}}\right)$ [BCW22]
$(1\pm\varepsilon)$ Approximation to top k singular values	Any in Table 1	$\tilde{O}\left(\frac{Tk}{\varepsilon^{1/2}} + \frac{nk^2}{\varepsilon} + \frac{k^3}{\varepsilon^{3/2}}\right)$ Theorem C.8	$\tilde{O}\left(\frac{n^2dk}{\varepsilon^{1/2}} + \frac{nk^2}{\varepsilon} + \frac{k^3}{\varepsilon}^{3/2}\right)$ [MM15]
Multiply distance matrix A with any $B \in \mathbb{R}^{n\times n}$	Any in Table 1	$O(Tn)$ Lemma C.9	$O(n^w)$
Multiply two distance matrices A and B	ℓ_2^2	$O(n^2d^{w-2})$ Lemma C.11	$O(n^w)$

Table 2: Applications of our matrix-vector query results. T denotes the matrix-vector query time, given in Table 1. $w \approx 2.37$ is the matrix multiplication constant [AW21].

Our result can be viewed as the natural runtime bound which would follow if the JL lemma implied an embedding dimension bound of $O(\text{poly}(\log\log n))$. While this is impossible, as it would imply an exponential improvement over the JL bound which is tight [LN17], we achieve our speedup by carefully reusing distance calculations via tools from metric compression [IRW17]. Our results also extend to the ℓ_1 distance matrix; see Theorem D.5 for details.

Notation. Our dataset will be the n points $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$. For points in X, we denote $x_i(j)$ to be the jth coordinate of point x_i for clarity. For all other vectors v, v_i denotes the ith coordinate. We are interested in matrices of the form $A_{i,j} = f(x_i, x_j)$ for $f : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ which measures the similarity between any pair of points. f might not necessarily be a distance function but we use the terminology "distance function" for ease of notation. We will always explicitly state the function f as needed. $w \approx 2.37$ denotes the matrix multiplication constant, i.e., the exponent of n in the time required to compute the product of two $n \times n$ matrix [AW21].

1.2 Related Works

Matrix-Vector Products Queries. Our work can be understood as being part of a long line of classical works on the matrix free or implicit model as well as the active recent line of works on the matrix-vector query model. Many widely used linear algebraic algorithms such as the power method, the Lanczos algorithm [Lan50], conjugate gradient descent [S$^+$94], and Wiedemann's coordinate recurrence algorithm [Wie86], to name a few, all fall into this paradigm. Recent works such as [MM15, BHSW20, BCW22] have succeeded in precisely nailing down the query complexity of these classical algorithms in addition to various other algorithmic tasks such as low-rank approximation [BCW22], trace estimation [MMMW21], and other linear-algebraic functions [SWYZ21b, RWZ20]. There is also a rich literature on query based algorithms in other contexts with the goal of minimizing the number of queries used. Examples include graph queries [Gol17], distribution queries [Can20], and constraint based queries [ES20] in property testing, inner product queries in compressed sensing [EK12], and quantum queries [LSZ21, CHL21].

Most prior works on query based models assume black-box access to matrix-vector queries. While this is a natural model which allows for the design non-trivial algorithms and lower bounds, it is not always clear how such queries can be initialized. In contrast, the focus of our work is not on obtaining query complexity bounds, but rather complementing prior works by creating an efficient matrix-vector query for a natural class of matrices.

Subquadratic Algorithms for Distance Matrices. Most work on subquadratic algorithms for distance matrices have focused on the problem of computing a low-rank approximation. [BW18, IVWW19] both obtain an additive error low-rank approximation applicable for all distance matrices. These works only assume access to the entries of the distance matrix whereas we assume we also have access to the underlying dataset. [BCW20] study the problem of computing the low-rank approximation of PSD matrices with also sample access to the entries of the matrix. Their results extend to low-rank approximation for the ℓ_1 and ℓ_2^2 distance matrices in addition to other more specialized metrics such as spherical metrics. Table 2 lists the runtime comparisons between their results and ours.

Practically, the algorithm of [IVWW19] is the easiest to implement and has outstanding empirical performance. We note that we can easily simulate their algorithm with no overall asymptotic runtime overhead using $O(\log n)$ vector queries. Indeed, their algorithm proceeds by sampling rows of the matrix according to their ℓ_2^2 value and then post-processing these rows. The sampling probabilities only need to be accurate up to a factor of two. We can acquire these sampling probabilities by performing $O(\log n)$ matrix-vector queries which sketches the rows onto dimension $O(\log n)$ and preserves all row-norms up to a factor of two with high probability due to the Johnson-Lindenstrauss lemma [JL84]. This procedure only incurs an additional runtime of $O(T \log n)$ where T is the time required to perform a matrix-vector query.

The paper [ILLP04] shows that the exact L_1 distance matrix can be created in time $O(n^{(w+3)/2}) \approx n^{2.69}$ in the case of $d = n$, which is asymptotically faster than the naive bound of $O(n^2 d) = O(n^3)$. In contrast, we focus on creating an (entry-wise) approximate distance matrices for all values of d.

We also compare to the paper of [ACSS20]. In summary, their main upper bounds are approximation algorithms while we mainly focus on exact algorithms. Concretely, they study matrix vector products for matrices of the form $A_{i,j} = f(\|x_i - x_j\|_2^2)$ for some function $f : \mathbb{R} \to \mathbb{R}$. They present results on approximating the matrix vector product of A where the approximation error is additive. They also consider a wide range of f, including polynomials and other kernels, but the input to is always the ℓ_2 distance squared. In contrast, we also present exact algorithms, i.e., with no approximation errors. For example one of our main upper bounds is an exact algorithm when $A_{i,j} = \|x_i - x_j\|_1$ (see Table 1 for the full list). Since it is possible to approximately embed the ℓ_1 distance into ℓ_2^2, their methods could be used to derive approximate algorithms for ℓ_1, but not the exact ones. Furthermore, we also study a wide variety of other distance functions such as ℓ_∞ and ℓ_p^p (and others listed in Table 1) which are not studied in Alman et al. In terms of technique, the main upper bound technique of Alman et al. is to expand $f(\|x_i - x_j\|_2^2)$ and approximate the resulting quantity via a polynomial. This is related to our upper bound results for ℓ_p^p for even p where we also use polynomials. However, our results are exact, while theirs are approximate. Our ℓ_1 upper bound technique is orthogonal to the polynomial approximation techniques used in Alman et al. We also employ polynomial techniques to give upper bounds for the approximate ℓ_∞ distance function which is not studied in Alman et al. Lastly, Alman et al. also focus on the Laplacian matrix of the weighted graph represented by the distance matrix, such as spectral sparsification and Laplacian system solving. In contrast, we study different problems including low-rank approximations, eigenvalue estimation, and the task of initializing an approximate distance matrix. We do not consider the distance matrix as a graph or consider the associated Laplacian matrix.

It is also easy to verify the "folklore" fact that for a gram matrix AA^T, we can compute $AA^T v$ in time $O(nd)$ if $A \in \mathbb{R}^{n \times d}$ by computing $A^T v$ first and then $A(A^T v)$. Our upper bound for the ℓ_2^2 function can be reduced to this folklore fact by noting that $\|x - y\|_2^2 = \|x\|_2^2 + \|y\|_2^2 - 2\langle x, y\rangle$. Thus the ℓ_2^2 matrix can be decomposed into two rank one components due to the terms $\|x\|_2^2$ and $\|y\|_2^2$, and a gram matrix due to the term $\langle x, y\rangle$. This decomposition of the ℓ_2^2 matrix is well-known (see Section 2 in [DPRV15]). Hence, a matrix-vector query for the ℓ_2^2 matrix easily reduces to the gram matrix case. Nevertheless, we explicitly state the ℓ_2^2 upper bound for completeness since we also consider all ℓ_p^p functions for any integer $p \geq 1$.

Polynomial Kernels. There have also been works on faster algorithms for approximating a kernel matrix K defined as the $n \times n$ matrix with entries $K_{i,j} = k(x_i, x_j)$ for a kernel function k. Specifically for the polynomial kernel $k(x_i, x_j) = \langle x_i, x_j\rangle^p$, recent works such as [ANW14, AKK$^+$20, WZ20, SWYZ21a] have shown how to find a sketch K' of K which approximately satisfies $\|K'z\|_2 \approx \|Kz\|_2$ for all z. In contrast, we can exactly simulate the matrix-vector product Kz. Our runtime is $O(nd^p)$ which has a linear dependence on n but an exponential dependence on p while the

aforementioned works have at least a quadratic dependence on n but a polynomial dependence on p. Thus our results are mostly applicable to the setting where our dataset is large, i.e. $n \gg d$ and p is a small constant. For example, $p = 2$ is a common choice in practice [CHC+10]. Algorithms with polynomial dependence in d and p but quadratic dependence in n are suited for smaller datasets which have very large d and large p. Note that a large p might arise if approximates a non-polynomial kernel using a polynomial kernel via a taylor expansion. We refer to the references within [ANW14, AKK+20, WZ20, SWYZ21a] for additional related work. There is also work on kernel density estimation (KDE) data structures which upon query y, allow for estimation of the sum $\sum_{x \in X} k(x, y)$ in time sublinear in $|X|$ after some preprocessing on the dataset X. For widely used kernels such as the Gaussian and Laplacian kernels, KDE data structures were used in [BIMW21] to create a matrix-vector query algorithm for kernel matrices in time subquadratic in $|X|$ for input vectors which are entry wise non-negative. We refer the reader to [CS17, BCIS18, SRB+19, BIW19, CKNS20] and references within for prior works on KDE data structures.

2 Faster Matrix-Vector Product Queries for ℓ_1

We derive faster matrix-vector queries for distance matrices for a wide array of distance metrics. First we consider the case of the ℓ_1 metric such that $A_{i,j} = f(x_i, x_j)$ where $f(x, y) = \|x - y\|_1 = \sum_{i=1}^{d} |x_i - y_i|$.

Algorithm 1 Preprocessing

Input: Dataset $X \subset \mathbb{R}^d$
procedure PREPROCESSING
  for $i \in [d]$ **do**
    $T_i \leftarrow$ sorted array of the ith coordinates of all $x \in X$.
  end for
end procedure

Algorithm 2 matrix-vector Query for $p = 1$

Input: Dataset $X \subset \mathbb{R}^d$
Output: $z = Ay$
procedure QUERY($\{T_i\}_{i \in [d]}, y$)
  $y_1, \cdots, y_n \leftarrow$ coordinates of y.
  Associate every $x_i \in X$ with the scalar y_i
  for $i \in [d]$ **do**
    Compute two arrays B_i, C_i as follows.
    B_i contains the partial sums of $y_j x_j(i)$ computed in the order induced by T_i
    C_i contains the partial sums of y_j computed in the order induced by T_i
  end for
  $z \leftarrow 0^n$
  for $k \in [n]$ **do**
    for $i \in [d]$ **do**
      $q \leftarrow$ position of $x_k(i)$ in the order of T_i
      $S_1 \leftarrow B_i[q]$
      $S_2 \leftarrow B_i[n] - B_i[q]$
      $S_3 \leftarrow C_i[q]$
      $S_4 \leftarrow C_i[n] - C_i[q]$
      $z(k) += x_k(i) \cdot (S_3 - S_4) + S_2 - S_1$
    end for
  end for
end procedure

We first analyze the correctness of Algorithm 2.

Theorem 2.1. *Let $A_{i,j} = \|x_i - x_j\|_1$. Algorithm 2 computes Ay exactly.*

Proof. Consider any coordinate $k \in [n]$. We show that $(Ay)_k$ is computed exactly. We have

$$(Ay)(k) = \sum_{j=1}^{n} y_j \|x_k - x_j\|_1 = \sum_{j=1}^{n}\sum_{i=1}^{d} y_j |x_k(i) - x_j(i)| = \sum_{i=1}^{d}\sum_{j=1}^{n} y_j |x_k(i) - x_j(i)|.$$

Let π^i denote the order of $[n]$ induced by T_i. We have

$$\sum_{i=1}^{d}\sum_{j=1}^{n} y_j |x_k(i) - x_j(i)| = \sum_{i=1}^{d}\left(\sum_{j:\pi^i(k)\le\pi^i(j)} y_j(x_j(i) - x_k(i)) + \sum_{j:\pi^i(k)>\pi^i(j)} y_j(x_k(i) - x_j(i))\right).$$

We now consider the inner sum. It rearranges to the following:

$$\begin{aligned} &x_k(i)\left(\sum_{j:\pi^i(k)>\pi^i(j)} y_j - \sum_{j:\pi^i(k)\le\pi^i(j)} y_j\right) + \sum_{j:\pi^i(k)\le\pi^i(j)} y_j x_j(i) - \sum_{j:\pi^i(k)>\pi^i(j)} y_j x_j(i) \\ &= x_k(i)\cdot(S_3 - S_4) + S_2 - S_1, \end{aligned}$$

where S_1, S_2, S_3, and S_4 are defined in lines $15-18$ of Algorithm 2 and the last equality follows from the definition of the arrays B_i and C_i. Summing over all $i \in [d]$ gives us the desired result. □

The following theorem readily follows.

Theorem 2.2. *Suppose we are given a dataset $\{x_1, \ldots, x_n\}$ which implicitly defines the distance matrix $A_{i,j} = \|x_i - x_j\|_1$. Given a query $y \in \mathbb{R}^d$, we can compute Ay exactly in $O(nd)$ query time. We also require a one time $O(nd \log n)$ preprocessing time which can be reused for all queries.*

3 Lower and Upper bounds for ℓ_∞

In this section we give a proof of Theorem 1.2. Specifically, we give a reduction from a so-called *Orthogonal Vector Problem* (OVP) [Wil05] to the problem of computing matrix-vector product Az, where $A_{i,j} = \|x_i - x_j\|_\infty$, for a given set of points $X = \{x_1, \ldots, x_n\}$. The orthogonal vector problem is defined as follows: given two sets of vectors $A = \{a^1, \ldots, a^n\}$ and $B = \{b^1, \ldots, b^n\}$, $A, B \subset \{0,1\}^d$, $|A| = |B| = n$, determine whether there exist $x \in A$ and $y \in B$ such that the dot product $x \cdot y = \sum_{j=1}^{d} x_j y_j$ (taken over reals) is equal to 0. It is known that if OVP can be solved in strongly subquadratic time $O(n^{2-\alpha})$ for any constant $\alpha > 0$ and $d = \omega(\log n)$, then SETH is false. Thus, an efficient reduction from OVP to the matrix-vector product problem yields Theorem 1.2.

Lemma 3.1. *If the matrix-vector product problem for ℓ_∞ distance matrices induced by n vectors of dimension d can be solved in time $T(n, d)$, then OVP (with the same parameters) can be solved in time $O(T(n, d))$.*

Proof. Define two functions, $f, g : \{0,1\}^d \to [0,1]$, such that $f(0) = g(0) = 1/2$, $f(1) = 0$, $g(1) = 1$. We extend both functions to vectors by applying f and g coordinate wise and to sets by letting $f(\{a^1, \ldots, a^n\}) = \{f(a^1), \ldots, f(a^n)\}$; the function g is extended in the same way for B. Observe that, for any pair of non-zero vectors $a, b \in \{0,1\}^d$, we have $\|f(a) - g(b)\|_\infty = 1$ if and only if $a \cdot b > 0$, and $\|f(a) - g(b)\|_\infty = 1/2$ otherwise.

Consider two sets of binary vectors A and B. Without loss of generality we can assume that the vectors are non-zero, since otherwise the problem is trivial. Define three distance matrices: matrix M_A defined by the set $f(A)$, matrix M_B defined by the set $g(B)$ and M_{AB} defined by the set $f(A) \cup f(B)$. Furthermore, let M be the "cross-distance" matrix, such that such that $M_{i,j} = \|f(a^i) - g(b^j)\|_\infty$. Observe that the matrix M_{AB} contains blocks M_A and M_B on its diagonal, and blocks M and M^T off-diagonal. Thus, $M_{AB} \cdot 1 = M_A \cdot 1 + M_B \cdot 1 + 2M \cdot 1$, where 1 denotes an all-ones vector of the appropriate dimension. Since $M \cdot 1 = (M_{AB} \cdot 1 - M_A \cdot 1 - M_B \cdot 1)/2$, we can calculate $M \cdot 1$ in time $O(T(n, d))$. Since all entries of M are either 1 or $1/2$, we have that $M \cdot 1 < n^2$ if and only if there is an entry $M_{i,j} = 1/2$. However, this only occurs if $a^i \cdot b^j = 0$. □

3.1 Approximate ℓ_∞ Matrix-Vector Queries

In light of the lower bounds given above, we consider initializing *approximate* matrix-vector queries for the ℓ_∞ function. Note that the lower bound holds for points in $\{0,1,2\}^d$ and thus it is natural to consider approximate upper bounds for the case of limited alphabet.

Binary Case. We first consider the case that all points $x \in X$ are from $\{0,1\}^d$. We first claim the existence of a polynomial T with the following properties. Indeed, the standard Chebyshev polynomials satisfy the following lemma, see e.g., see Chapter 2 in [SV+14].

Lemma 3.2. *There exists a polynomial $T : \mathbb{R} \to \mathbb{R}$ of degree $O(\sqrt{d}\log(1/\varepsilon))$ such that $T(0) = 0$ and $|T(x) - 1| \le \varepsilon$ for all $x \in [1/d, 1]$.*

Now note that $\|x-y\|_\infty$ can only take on two values, 0 or 1. Furthermore, $\|x-y\|_\infty = 0$ if and only if $\|x-y\|_2^2 = 0$ and $\|x-y\|_\infty = 1$ if and only if $\|x-y\|_2^2 \ge 1$. Therefore, $\|x-y\|_\infty = 0$ if and only if $T(\|x-y\|_2^2/d) = 0$ and $\|x-y\|_\infty = 1$ if and only if $|T(\|x-y\|_2^2/d) - 1| \le \varepsilon$. Thus, we have that

$$|A_{i,j} - T(\|x_i - x_j\|_2^2/d)| = |\|x_i - x_j\|_\infty - T(\|x_i - x_j\|_2^2/d)| \le \varepsilon$$

for all entries $A_{i,j}$ of A. Note that $T(\|x-y\|_2^2/d)$ is a polynomial with $O((2d)^t)$ monomials in the variables $x(1), \ldots, x(d)$. Consider the matrix B satisfying $B_{i,j} = T(\|x_i - x_j\|_2^2/d)$. Using the same ideas as our upper bound results for $f(x,y) = \langle x, y\rangle^p$, it is straightforward to calculate the matrix vector product By (see Section A.2). To summarize, for each $k \in [n]$, we write the kth coordinate of By as a polynomial in the d coordinates of x_k. This polynomial has $O((2d)^t)$ monomials and can be constructed in $O(n(2d)^t)$ time. Once constructed, we can evaluate the polynomial at $x_1, \ldots, x_n$ to obtain all the n coordinates of By. Each evaluation requires $O((2d)^t)$ resulting in an overall time bound of $O(n(2d)^t)$.

Theorem 3.3. *Let $A_{i,j} = \|x_i - x_j\|_\infty$. We can compute By in time $O(n(2d)^{\sqrt{d}\log(1/\varepsilon)})$ where $\|A - B\|_\infty \le \varepsilon$.*

Entries in $\{0, \ldots, M\}$. We now consider the case that all points $x \in X$ are from $\{0, \ldots, M\}^d$. Our argument will be a generalization of the previous section. At a high level, our goal is to detect which of the $M+1$ possible values in $\{0, \ldots, M\}$ is equal to the ℓ_∞ norm. To do so, we appeal to the prior section and design estimators which approximate the indicator function "$\|x-y\|_\infty \ge i$". By summing up these indicators, we can approximate $\|x-y\|_\infty$.

Our estimators will again be designed using the Chebyshev polynomials. To motivate them, suppose that we want to detect if $\|x-y\|_\infty \ge i$ or if $\|x-y\|_\infty < i$. In the first case, some entry in $x-y$ will have absolute value value at least i where as in the other case, all entries of $x-y$ will be bounded by $i-1$ in absolute value. Thus if we can boost this 'signal', we can apply a polynomial which performs thresholding to distinguish the two cases. This motivates considering the functions of $\|x-y\|_k^k$ for a larger power k. In particular, in the case that $\|x-y\|_\infty \ge i$, we have $\|x-y\|_k^k \ge i^k$ and otherwise, $\|x-y\|_k^k \le di^{k-1}$. By setting $k \approx \log(d)$, the first value is much larger than the latter, which we can detect using the 'threshold' polynomials of the previous section.

We now formalize our intuition. It is known that appropriately scaled Chebyshev polynomials satisfy the following guarantees, see e.g., see Chapter 2 in [SV+14].

Lemma 3.4. *There exists a polynomial $T : \mathbb{R} \to \mathbb{R}$ of degree $O(\sqrt{t}\log(t/\varepsilon))$ such that $|T(x)| \le \varepsilon/t$ for all $x \in [0, 1/(10t)]$ and $|T(x) - 1| \le \varepsilon/t^2$ for all $x \in [1/t, 1]$.*

Given $x, y \in \mathbb{R}^d$, our estimator will first try to detect if $\|x-y\|_\infty \ge i$. Let T_1 be a polynomial from Lemma 3.4 with $t = O(M^k)$ for $k = O(M\log(Md))$ and assuming k is even. Let T_2 be a polynomial from Lemma 3.4 with $t = O(\sqrt{d}\log(M/\varepsilon))$. Our estimator will be

$$T_2\left(\frac{1}{d}\sum_{j=1}^{d} T_1\left(\frac{(x(j)-y(j))^k}{i^k \cdot M^k}\right)\right).$$

If coordinate j is such that $|x(j) - y(j)| \ge i$, then

$$\frac{(x(j)-y(j))^k}{i^k \cdot M^k} \ge \frac{1}{M^k}$$

and so T_1 will evaluate to a value very close to 1. Otherwise, we know that

$$\frac{(x(j)-y(j))^k}{i^k \cdot M^k} \leq \frac{(i-1)^k}{i^k M^k} = \frac{1}{M^k}(1-1/i)^k \ll \frac{1}{M^k} \cdot \frac{1}{\text{poly}(M,d)}$$

by our choice of k, which means that T_1 will evaluate to a value close to 0. Formally,

$$\frac{1}{d}\sum_{j=1}^{d} T_1\left(\frac{(x(j)-y(j))^k}{i^k \cdot M^k}\right)$$

will be at least $1/d$ if there is a $j \in [d]$ with $|x(j)-y(j)| \geq i$ and otherwise, will be at most $1/(10d)$. By our choice of T_2, the overall estimate output at least $1-\varepsilon$ in the first case and a value at most ε in the second case.

The polynomial which is the concatenation of T_2 and T_1 has $O\left((dk \cdot \deg(T_1))^{\deg(T_2)}\right) = (dM)^{O(M\sqrt{d}\log(Md))}$ monomials, if we consider the expression as a polynomial in the variables $x(1), \ldots, x(d)$. Our final estimator will be the sum across all $i \geq 1$. Following our upper bound techniques for matrix-vector products for polynomial, e.g. in Section A.2, and as outlined in the prior section, we get the following overall query time:

Theorem 3.5. *Suppose we are given $X = \{x_1, \ldots, x_n\} \subseteq \{0, \ldots, M\}^d$ which implicitly defines the matrix $A_{i,j} = \|x_i - x_j\|_\infty$. For any query y, we can compute By in time $n \cdot (dM)^{O(M\sqrt{d}\log(Md/\varepsilon))}$ where $\|A-B\|_\infty \leq \varepsilon$.*

4 Empirical Evaluation

We perform an empirical evaluation of our matrix-vector query for the ℓ_1 distance function. We chose to implement our ℓ_1 upper bound since it's a clean algorithm which possesses many of the same underlying algorithmic ideas as some of our other upper bound results. We envision that similar empirical results hold for most of our upper bounds in Table 1. Furthermore, matrix-vector queries are the dominating subroutine in many key practical linear algebra algorithms such as the power method for eigenvalue estimation or iterative methods for linear regression: a fast matrix-vector query runtime automatically translates to faster algorithms for downstream applications.

Dataset	(n,d)	Algo.	Preprocessing Time	Avg. Query Time
Gaussian Mixture	$(5 \cdot 10^4, 50)$	Naive	453.7 s	43.3 s
		Ours	0.55 s	0.09 s
MNIST	$(5 \cdot 10^4, 784)$	Naive	2672.5 s	38.6 s
		Ours	5.5 s	1.9 s
Glove	$(1.2 \cdot 10^6, 50)$	Naive	-	$\approx$ 2.6 days (estimated)
		Ours	16.8 s	3.4 s

Table 3: Dataset description and empirical results. (n,d) denotes the number of points and dimension of the dataset, respectively. Query times are averaged over 10 trials with Gaussian vectors as queries.

Experimental Design. We chose two real and one synthetic dataset for our experiments. We have two "small" datasets and one "large" dataset. The two small datasets have $5 \cdot 10^4$ points whereas the large dataset has approximately 10^6 points. The first dataset is points drawn from a mixture of three spherical Gaussians in $\mathbb{R}^{50}$. The second dataset is the standard MNIST dataset [LeC98] and finally, our large dataset is Glove word embeddings[2] in $\mathbb{R}^{50}$ [PSM14].

The two small datasets are small enough that one can feasibly initialize the full $n \times n$ distance matrix in memory in reasonable time. A $5 \cdot 10^4 \times 5 \cdot 10^4$ matrix with each entry stored using 32 bits requires 10 gigabytes of space. This is simply impossible for the Glove dataset as approximately 5.8 terabytes of space is required to initialize the distance matrix (in contrast, the dataset itself only requires < 0.3 gigabytes to store).

[2]Can be accessed here: `http://github.com/erikbern/ann-benchmarks/`

The naive algorithm for the small datasets is the following: we initialize the full distance matrix (which will count towards preprocessing), and then we use the full distance matrix to perform a matrix-vector query. Note that having the full matrix to perform a matrix-vector product only helps the naive algorithm since it can now take advantage of optimized linear algebra subroutines for matrix multiplication and does not need to explicitly calculate the matrix entries. Since we cannot initialize the full distance matrix for the large dataset, the naive algorithm in this case will compute the matrix-vector product in a standalone fashion by generating the entries of the distance matrix on the fly. We compare the naive algorithm to our Algorithms 1 and 2.

Our experiments are done in a 2021 M1 Macbook Pro with 32 gigabytes of RAM. We implement all algorithms in Python 3.9 using Numpy with Numba acceleration to speed up all algorithms whenever possible.

Results. Results are shown in Table 3. We show preprocessing and query time for both the naive and our algorithm in seconds. The query time is averaged over 10 trials using Gaussian vectors as queries. For the Glove dataset, it was infeasible to calculate even a single matrix-vector product, even using fast Numba accelerated code. We thus estimated the full query time by calculating the time on a subset of $5 \cdot 10^4$ points of the Glove dataset and extrapolating to the full dataset by multiplying the query time by $(n/(5 \cdot 10^4))^2$ where n is the total number of points. We see that in all cases, our algorithm outperforms the naive algorithm in both preprocessing time and query time and the gains become increasingly substantial as the dataset size increases, as predicted by our theoretical results.

Acknowledgements. This research was supported by the NSF TRIPODS program (award DMS-2022448), Simons Investigator Award, MIT-IBM Watson AI Lab, GIST- MIT Research Collaboration grant, and NSF Graduate Research Fellowship under Grant No. 1745302.

References

[ACSS20] Josh Alman, Timothy Chu, Aaron Schild, and Zhao Song. Algorithms and hardness for linear algebra on geometric graphs. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 541–552. IEEE, 2020.

[AG11] Uri M Ascher and Chen Greif. *A first course on numerical methods*. SIAM, 2011.

[AKK+20] Thomas D Ahle, Michael Kapralov, Jakob BT Knudsen, Rasmus Pagh, Ameya Velingker, David P Woodruff, and Amir Zandieh. Oblivious sketching of high-degree polynomial kernels. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 141–160. SIAM, 2020.

[ANW14] Haim Avron, Huy Nguyen, and David Woodruff. Subspace embeddings for the polynomial kernel. *Advances in neural information processing systems*, 27, 2014.

[AW21] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539. SIAM, 2021.

[BCIS18] Arturs Backurs, Moses Charikar, Piotr Indyk, and Paris Siminelakis. Efficient density evaluation for smooth kernels. *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 615–626, 2018.

[BCW20] Ainesh Bakshi, Nadiia Chepurko, and David P Woodruff. Robust and sample optimal algorithms for psd low rank approximation. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 506–516. IEEE, 2020.

[BCW22] Ainesh Bakshi, Kenneth L Clarkson, and David P Woodruff. Low-rank approximation with $1/\varepsilon^{1/3}$ matrix-vector products. *arXiv preprint arXiv:2202.05120*, 2022.

[BHSW20] Mark Braverman, Elad Hazan, Max Simchowitz, and Blake Woodworth. The gradient complexity of linear regression. In *Conference on Learning Theory*, pages 627–647. PMLR, 2020.

[BIMW21] Arturs Backurs, Piotr Indyk, Cameron Musco, and Tal Wagner. Faster kernel matrix algebra via density estimation. In *Proceedings of the 38th International Conference on Machine Learning*, pages 500–510, 2021.

[BIW19] Arturs Backurs, Piotr Indyk, and Tal Wagner. Space and time efficient kernel density estimation in high dimensions. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS*, pages 15773–15782, 2019.

[BW18] Ainesh Bakshi and David Woodruff. Sublinear time low-rank approximation of distance matrices. *Advances in Neural Information Processing Systems*, 31, 2018.

[Can20] Clément L Canonne. A survey on distribution testing: Your data is big. but is it blue? *Theory of Computing*, pages 1–100, 2020.

[CC08] Michael AA Cox and Trevor F Cox. Multidimensional scaling. In *Handbook of data visualization*, pages 315–347. Springer, 2008.

[CHC+10] Yin-Wen Chang, Cho-Jui Hsieh, Kai-Wei Chang, Michael Ringgaard, and Chih-Jen Lin. Training and testing low-degree polynomial data mappings via linear svm. *Journal of Machine Learning Research*, 11(4), 2010.

[CHL21] Andrew M Childs, Shih-Han Hung, and Tongyang Li. Quantum query complexity with matrix-vector products. *arXiv preprint arXiv:2102.11349*, 2021.

[CKNS20] Moses Charikar, Michael Kapralov, Navid Nouri, and Paris Siminelakis. Kernel density estimation through density constrained near neighbor search. *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 172–183, 2020.

[CS17] 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*, pages 1032–1043, 2017.

[DPRV15] Ivan Dokmanic, Reza Parhizkar, Juri Ranieri, and Martin Vetterli. Euclidean distance matrices: essential theory, algorithms, and applications. *IEEE Signal Processing Magazine*, 32(6):12–30, 2015.

[EK12] Yonina C Eldar and Gitta Kutyniok. *Compressed sensing: theory and applications*. Cambridge university press, 2012.

[ES20] Rogers Epstein and Sandeep Silwal. Property testing of lp-type problems. In *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[Gol17] Oded Goldreich. *Introduction to property testing*. Cambridge University Press, 2017.

[HS93] Liisa Holm and Chris Sander. Protein structure comparison by alignment of distance matrices. *Journal of molecular biology*, 233(1):123–138, 1993.

[ILLP04] Piotr Indyk, Moshe Lewenstein, Ohad Lipsky, and Ely Porat. Closest pair problems in very high dimensions. In *International Colloquium on Automata, Languages, and Programming*, pages 782–792. Springer, 2004.

[IP01] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

[IPZ01] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

[IRW17] Piotr Indyk, Ilya Razenshteyn, and Tal Wagner. Practical data-dependent metric compression with provable guarantees. *Advances in Neural Information Processing Systems*, 30, 2017.

[IVWW19] Pitor Indyk, Ali Vakilian, Tal Wagner, and David P Woodruff. Sample-optimal low-rank approximation of distance matrices. In *Conference on Learning Theory*, pages 1723–1751. PMLR, 2019.

[JL84] W. Johnson and J. Lindenstrauss. Extensions of lipschitz maps into a hilbert space. *Contemporary Mathematics*, 26:189–206, 01 1984.

[Kru64] Joseph B Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[Kru78] Joseph B Kruskal. *Multidimensional scaling*. Number 11. Sage, 1978.

[Kuc09] Marek Kuczma. *An introduction to the theory of functional equations and inequalities: Cauchy's equation and Jensen's inequality*. Springer Science & Business Media, 2009.

[Lan50] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. 1950.

[LeC98] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[LN17] Kasper Green Larsen and Jelani Nelson. Optimality of the johnson-lindenstrauss lemma. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 633–638. IEEE, 2017.

[LSZ21] Troy Lee, Miklos Santha, and Shengyu Zhang. Quantum algorithms for graph problems with cut queries. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 939–958. SIAM, 2021.

[MM15] Cameron Musco and Christopher Musco. Randomized block krylov methods for stronger and faster approximate singular value decomposition. *Advances in neural information processing systems*, 28, 2015.

[MMMW21] Raphael A Meyer, Cameron Musco, Christopher Musco, and David P Woodruff. Hutch++: Optimal stochastic trace estimation. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 142–155. SIAM, 2021.

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

[RWZ20] Cyrus Rashtchian, David P Woodruff, and Hanlin Zhu. Vector-matrix-vector queries for solving linear algebra, statistics, and graph problems. *arXiv preprint arXiv:2006.14015*, 2020.

[S+94] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[SRB+19] Paris Siminelakis, Kexin Rong, Peter Bailis, Moses Charikar, and Philip Alexander Levis. Rehashing kernel evaluation in high dimensions. In *Proceedings of the 36th International Conference on Machine Learning, ICML*, pages 5789–5798, 2019.

[SV+14] Sushant Sachdeva, Nisheeth K Vishnoi, et al. Faster algorithms via approximation theory. *Foundations and Trends® in Theoretical Computer Science*, 9(2):125–210, 2014.

[SWYZ21a] Zhao Song, David Woodruff, Zheng Yu, and Lichen Zhang. Fast sketching of polynomial kernels of polynomial degree. In *International Conference on Machine Learning*, pages 9812–9823. PMLR, 2021.

[SWYZ21b] Xiaoming Sun, David P Woodruff, Guang Yang, and Jialin Zhang. Querying a matrix through matrix-vector products. *ACM Transactions on Algorithms (TALG)*, 17(4):1–19, 2021.

[SY07] Anthony Man-Cho So and Yinyu Ye. Theory of semidefinite programming for sensor network localization. *Mathematical Programming*, 109(2):367–384, 2007.

[TSL00] Joshua B Tenenbaum, Vin de Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

[Wie86] Douglas Wiedemann. Solving sparse linear equations over finite fields. *IEEE transactions on information theory*, 32(1):54–62, 1986.

[Wil05] Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2-3):357–365, 2005.

[Woo14] David P Woodruff. Sketching as a tool for numerical linear algebra. *arXiv preprint arXiv:1411.4357*, 2014.

[WS06] Kilian Q Weinberger and Lawrence K Saul. Unsupervised learning of image manifolds by semidefinite programming. *International journal of computer vision*, 70(1):77–90, 2006.

[WZ20] David Woodruff and Amir Zandieh. Near input sparsity time kernel embeddings via adaptive sampling. In *International Conference on Machine Learning*, pages 10324–10333. PMLR, 2020.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] We include formal upper and lower bounds which natural state the applicability and limitations of our theorems.
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes] All proofs are included in the main body or in the Appendix.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] The code is included in the supplementary.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...

   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Omitted Upper Bounds for Faster Matrix-Vector Queries

We now consider the case of ℓ_p^p for $p = 2$. Generalizing the results of $p = 1$ and $p = 2$ allows us to handle general ℓ_p^p functions.

Algorithm 3 matrix-vector Query for $p = 2$

Input: Dataset $X \subset \mathbb{R}^d$
Output: $z = Ay$
procedure QUERY(y)
    $v \leftarrow \sum_{i=j}^{n} y_j x_j$
    $S_1 \leftarrow \sum_{i=j}^{n} y_j^2$
    $S_2 \leftarrow \sum_{i=j}^{n} y_j^2 \|x\|_2^2$
    $z \leftarrow 0^n$
    for $k \in [n]$ **do**
        $z(k) \leftarrow S_1 \|x_k\|_2^2 + S_2 - 2\langle x_k, v\rangle$
    end for
end procedure

Theorem A.1. *We can compute Ay in $O(nd)$ query time.*

Proof. The proof follows from the following calculation of the kth coordinate of Ay:

$$(Ay)(k) = \sum_{j=1}^{n} y_j \|x_k - x_j\|_2^2 = \|x_k\|_2^2 \left(\sum_{j=1}^{n} y_j^2\right) + \sum_{j=1}^{n} y_j^2 \|x_j\|_2^2 - 2\left\langle x_k, \sum_{j=1}^{n} y_j x_j \right\rangle. \qquad \square$$

We can extend our results to general ℓ_p^p functions as well as a wide array of commonly used functions to measure (dis)similarity between vectors. For example, suppose the points x_i represent a probability distribution over the domain $[n] := \{1, \ldots, n\}$. A widely used "distance" function over distributions is the KL-divergence defined as

$$f(x_i, x_j) = \mathrm{D}_{\mathrm{KL}}(x_i \,\|\, x_j) = \sum_{k \in [d]} x_i(k) \log(x_i(k)) - x_i(k) \log(x_j(k)) = -H(x_i) - \sum_{k \in [d]} x_i(k) \log(x_j(k)),$$

where H is the entropy function. Our techniques extend to the KL-divergence as well.

Algorithm 4 matrix-vector Query for KL Divergence

Input: Dataset $X \subset \mathbb{R}^d$
Output: $z = Ay$
procedure QUERY(y)
    $S_i \leftarrow \sum_{j=1}^{n} y_j \log(x_j(i))$ for all $i \in [d]$
    $H_i \leftarrow H(x_i)$ for all $i \in [n]$
    $Y \leftarrow \sum_{j=1}^{n} y_j$
    $z \leftarrow 0^n$
    for $k \in [n]$ **do**
        $z(k) \leftarrow -H_k \cdot Y - \sum_{i=1}^{d} x_k(i) S_i$
    end for
end procedure

Theorem A.2. *We can compute Ay in $O(nd)$ query time.*

Proof. Note that computed all of S_i and H_i takes $O(nd)$ time. Now

$$\begin{aligned}(Ay)(k) &= \sum_{j=1}^{n} y_j \mathrm{D}_{\mathrm{KL}}(x_k \,\|\, x_j) \\ &= \sum_{j=1}^{n} -y_j H(x_k) - \sum_{j=1}^{n} y_j \sum_{k=1}^{d} x_i(k) \log(x_j(k)) \\ &= -H(x_k) \left(\sum_{j=1}^{n} y_j \right) - \sum_{k=1}^{d} \sum_{j=1}^{n} y_j x_i(k) \log(x_j(k)) \\ &= -H_k \cdot Y - \sum_{k=1}^{d} x_i(k) S_k,\end{aligned}$$

as desired. □

A.1 General p

We now consider the case of a general non-negative even integer p.

Algorithm 5 matrix-vector Query for even p

Input: Dataset $X \subset \mathbb{R}^d$
Output: $z = Ay$
procedure QUERY(y)
    Compute all the values $S_{i,t} := \sum_{j=1}^{n} x_j(i)^{p-t}(-1)^{p-t}$ for all $i \in [d]$ and $t \in \{0, \ldots, p\}$
    $z \leftarrow 0^n$
    for $k \in [n]$ **do**
        $z(k) \leftarrow \sum_{i=1}^{d} \sum_{t=1}^{p} x_k(i)^t S_{i,t}$
    end for
end procedure

Theorem A.3. *We can compute Ay in $O(ndp)$ query time.*

Proof. Consider the following calculation of the kth coordinate of Ay:

$$\begin{aligned}(Ay)(k) &= \sum_{j=1}^{n} y_j \|x_k - x_j\|_p^p \\ &= \sum_{j=1}^{n} y_j \sum_{i=1}^{d} (x_k(i) - x_j(i))^p \\ &= \sum_{j=1}^{n} \sum_{i=1}^{d} \sum_{t=1}^{p} x_k(i)^t x_j(i)^{p-t} (-1)^{p-t} \\ &= \sum_{i=1}^{d} \sum_{t=1}^{p} x_k(i)^t \sum_{j=1}^{n} x_j(i)^{p-t} (-1)^{p-t} \\ &= \sum_{i=1}^{d} \sum_{t=1}^{p} x_k(i)^t S_{i,t}.\end{aligned}$$

Note that computing $S_{i,t}$ for all i and t takes $O(ndp)$ time. Then returning the value of $(Ay)_k$ takes $O(dp)$ time resulting in the claimed runtime. □

The case of a general non-negative odd integer p follows in a straightforward manner by combining the above techniques with those of the $p = 1$ case of Theorem 2.2 so we omit the proof.

Theorem A.4. *For odd integer p, we can compute Ay in $O(nd \log n)$ preprocessing time and $O(ndp)$ query time.*

A.2 Other Distance Functions

In this section we initialize matrix-vector queries for a wide variety of "distance" functions.

'Mixed' ℓ_∞. We consider the case of a 'permutation invariant' version of the ℓ_∞ norm defined as follows:

$$f(x, y) = \max_{i \in [d], j \in [d]} |x_i - y_j|.$$

f is not a norm but we will refer to it as 'mixed' ℓ_∞.

Algorithm 6 Preprocessing

1: **Input:** Dataset $X \subset \mathbb{R}^d$
2: **procedure** PREPROCESSING
3: **for** $j \in [n]$ **do**
4: $\min_j, \max_j \leftarrow$ minimum and maximum values of the entries of x_j, respectively.
5: **end for**
6: **end procedure**

Algorithm 7 matrix-vector Query for mixed ℓ_∞

1: **Input:** Dataset $X \subset \mathbb{R}^d$
2: **Output:** $z = Ay$
3: **procedure** QUERY($\{\min_j, \max_j\}_{j=1}^n, y$)
4: $z \leftarrow 0^n$
5: **for** $k \in [n]$ **do**
6: $z(k) \leftarrow \sum_{j=1}^n y_j \cdot \max\left(|\min_k - \min_j|, |\min_k - \max_j|, |\max_k - \min_j|, |\max_k - \max_j|\right)$
7: **end for**
8: **end procedure**

Theorem A.5. *We can compute Ay in $O(nd)$ preprocessing time and $O(n^2)$ query time.*

Proof. The preprocessing time holds because we calculate the maximum and minimum of a list of d numbers a total of n times. For the query time, note that each $z(k)$ takes $O(n)$ time to compute since we do a $O(1)$ operation is each index of the sum in Line 6 of Algorithm 7.

To prove correctness, note that for any two vectors $x, y \in \mathbb{R}^d$, the maximum value of $|x_i - y_j|$ is attained if x_i and y_j are among the minimum / maximum values of the coordinates of x and y respectively. To see this, fix a value of x_i. We can always increase $|x_i - y_j|$ by setting y_j to be the maximum or minimum over all j. □

Mahalanobis Distance Squared. We consider the function

$$f(x, y) = x^T M y$$

for some $d \times d$ matrix M. This is the squared version of the well-known Mahalanobis distance.

Algorithm 8 Preprocessing

1: **Input:** Dataset $X \subset \mathbb{R}^d$
2: **procedure** PREPROCESSING
3: $S \leftarrow d \times n$ matrix where the jth column is Mx_j for all $j \in [n]$.
4: **end procedure**

Algorithm 9 matrix-vector Query for Mahalanobis distance squared

Input: Dataset $X \subset \mathbb{R}^d$
Output: $z = Ay$
procedure QUERY(S, y)
    $v \leftarrow Sy$
    $z \leftarrow 0^n$
    for $k \in [n]$ **do**
        $z(k) \leftarrow \langle x_k, v \rangle$
    end for
end procedure

Theorem A.6. *We can compute Ay in $O(nd^2)$ preprocessing time and $O(nd)$ query time.*

Proof. Note that the kth coordinate of Ay is given by

$$(Ay)(k) = \sum_{j=1}^{n} y_j x_k^T M x_j = \left\langle x_k, \sum_{j=1}^{n} y_j M x_k \right\rangle = \langle x_k, Sy \rangle$$

which proves correctness. It takes $O(nd^2)$ time to compute S, $O(nd)$ time to compute Sy, and then $O(d)$ time to compute the kth coordinate of Ay for all $k \in [n]$. □

Polynomial Kernels. We now consider polynomial kernels of the form

$$f(x, y) = \langle x, y \rangle^p.$$

Theorem A.7. *We can compute Ay in $O(nd^p)$ query time.*

Proof Sketch. Consider the following expression

$$g(z) = \sum_{j=1}^{n} y_j \langle z, x_j \rangle^p$$

as a polynomial $g : \mathbb{R}^d \to \mathbb{R}$ in the d coordinates of z. By rearranging, the above sum can be written as a sum over $O(d^p)$ terms, corresponding to each monomial $z_1^{a_1} \dots z_d^{a_d}$ where $a_1 + \dots + a_d = p$. The coefficient of each term takes $O(nd)$ time to compute given x_i and y. Once computed, we can evaluate the polynomial at $z = x_j$ for all j which form the coordinates of Ay. Again, this can be viewed as "linearizing" the kernel given by $\langle x, y \rangle^p$. □

We note that a proof similar to that of Theorem A.7 was given in Section 5.3 of [ACSS20] by expanding the relevant quantity as a polynomial; see Section 1.2 for detailed comparison between [ACSS20] and our work.

A.3 Distances for Distributions

We now consider the case that each x_i specifies a discrete distribution over a domain of d elements. Matrices A where $A_{i,j}$ is a function computing distances between distributions x_i and x_j have recently been studied in machine learning.

We consider how to construct matrix-vector queries for such matrices for a range of widely used distance measures on distributions. First note that a result on the TV distance follows immediately from our ℓ_1 result.

Theorem A.8. *Suppose $A_{i,j} = \mathrm{TV}(x_i, x_j)$. We can compute Ay in $O(nd \log n)$ preprocessing time and $O(nd)$ query time.*

We now consider some other distance functions on distributions.

Divergences. Through a similar calculation as the KL divergence case, we can also achieve $O(nd)$ query times if f is the Jensen-Shannon divergence, defined as

$$f(x, y) = \frac{\mathrm{D}_{\mathrm{KL}}(x \,\|\, y) + \mathrm{D}_{\mathrm{KL}}(y \,\|\, x)}{2},$$

as well as the cross entropy function.

Theorem A.9. *Let f be the Jensen-Shannon divergence or cross entropy function. Then Ay can be computed in $O(nd)$ time.*

Through a similar calculation as done in Section A.2 (for the case of $p = 1$), we can also perform matrix-vector multiplication queries in the case that

$$f(x, y) = \sum_{i=1}^{d} \sqrt{x(i)y(i)}.$$

This is the squared Hellinger distance.

Theorem A.10. *Let f be the squared Hellinger distance. Then Ay can be computed in $O(nd)$ time.*

A.4 Approximate Matrix-Vector Query for ℓ_2

While our techniques do not extend to the ℓ_2 case for *exact* matrix-vector queries, we can nonetheless instantiate *approximate* matrix-vector queries for the ℓ_2 function. We first recall the following well known embedding result.

Theorem A.11. *Let $\varepsilon \in (0, 1)$ and define $T : \mathbb{R}^d \to \mathbb{R}^k$ by*

$$T(x)_i = \frac{1}{\beta k} \sum_{j=1}^{d} Z_{ij} x_j, \quad i = 1, \ldots, k$$

where $\beta = \sqrt{2/\pi}$. Then for every vector $x \in \mathbb{R}^d$, we have

$$\Pr[(1 - \varepsilon)\|x\|_2 \leq \|T(x)\|_1 \leq (1 + \varepsilon)\|x\|_2] \geq 1 - e^{c\varepsilon^2 k},$$

where $c > 0$ is a constant.

We can instantiate approximate matrix-vector queries for $f(x, y) = \|x - y\|_2$ via the following algorithm.

Algorithm 10 Preprocessing

1: **Input:** Dataset $X \subset \mathbb{R}^d$
2: **procedure** PREPROCESSING(T)
3: $\quad X' \leftarrow TX$ where T is the linear map from Theorem A.11
4: $\quad$ Run Algorithm 1 on X'
5: **end procedure**

For queries, we just run Algorithm 2 on X'. We have the following guarantee:

Theorem A.12. *Let $A_{i,j} = \|x_i - x_j\|_2$. There exists a matrix B such that we can compute By in $O(nd^2 + nd \log n)$ preprocessing time and $O(n \log(n)/\varepsilon^2)$ query time where*

$$\|A - B\|_F \leq \varepsilon \|A\|_F$$

with probability $1 - 1/\mathrm{poly}(n)$.

Proof. The preprocessing and query time follow from the time required to apply the transformation T from Theorem A.11 to our set of points X as well as the time needed for the ℓ_1 matrix-vector query result of Theorem 2.1. The Frobenius norm guarantee follows from the fact that every entry of A will be approximated with relative error in B using Theorem A.11. □

A.5 Matrix-Vector Query Lower Bounds

Table 1 shows that we can initialize matrix-vector queries for a variety of distance functions in $O(nd)$ time. It is straightforward to see that this bound is optimal for a large class of distance matrices.

Theorem A.13. *Consider the case that $A_{i,j} = f(x_i, x_j)$ satisfying $f(x, x) = 0$ for all x. Further assume that for all x, there exists an input y such that $f(x, y) = 1$. An algorithm which outputs an entry-wise approximation of Az to any constant factor for input z requires $\Omega(nd)$ time in the worst case.*

Proof. We consider two cases for input points of A. In the first case, all points in our dataset X are identical. In the second case, a randomly chosen point is distance 1 away from the $n-1$ identical points. Computing the product of A times the all ones vector allows us to distinguish the two cases as $A1$ has entries summing to 0 in the first case whereas $A1$ has entries summing to $n-1$ in the second case. Thus to approximate $A1$ entry-wise to any constant factor, we must distinguish the two cases. If we read $o(n)$ points, then with good probability we will see no duplicates. Thus, we must read $\Omega(n)$ points, require $\Omega(nd)$ time. □

B When Do Our Upper Bound Techniques Work?

By this point, we have seen many examples of matrix-vector queries which can be initialized as well as a lower bound for a natural distance function which prohibits any subquadratic time algorithm. Naturally, we are thus interested in the limits of our upper bound techniques for instantiating fast matrix-vector product. An understanding of such limits sheds light on families of structured matrices which may admit fast matrix-vector queries in general. In this section we fully characterize the capabilities of our upper bound methods and show that essentially our techniques can only work "linear" functions (in a possibly different basis).

First we set some notation. Let A be a $n \times n$ matrix we wish to compute where each (i, j) entry is given by $f(x_i, x_j)$. Given a query vector $z \in \mathbb{R}^n$, the kth coordinate of Az is given by

$$(Az)(k) = \sum_{i=1}^{n} z_i f(x_k, x_i).$$

An example choice of f is given by $f(x, y) = \sum_{j=1}^{d} x(j) \log(y(j))$ (assuming all the coordinates of x and y are entry wise non-negative. Note this is related to the cross entropy function in Table 1).

We first highlight the major steps which are common to all of our upper bound algorithms using f as an example. Our upper bound technique proceeds as follows:

- Break up $f(x, y)$ into a sum over d terms: $\sum_{j=1}^{d} x(j) \log(y(j))$.
- Switch the order of summation:

$$(Az)(k) = \sum_{i=1}^{n} z_i f(x_k, x_i) = \sum_{j=1}^{d} \sum_{i=1}^{n} z_i x_k(j) \log(x_i(j)).$$

- Evaluate each of the inner d summations with 1 evaluation each (after some preprocessing). In other words, for a fixed j, each of the sums $\sum_{i=1}^{n} z_i x_k(j) \log(x_i(j))$ can be computed as one evaluation, namely $x_k(j) \cdot \left(\sum_{i=1}^{n} z_i \log(x_i(j)) \right)$ and in preprocessing, we can compute $\sum_{i=1}^{n} z_i \log(x_i(j))$ as it does not depend on the coordinate k.

The key steps of the above outline, namely switching the order of summation and precomputation of repeated terms, can be encapsulated in the following framework.

Theorem B.1. *Suppose there exist mappings $T_1, T_2 : \mathbb{R}^d \to \mathbb{R}^{d'}$ (possibly non-linear) and a continuous $g : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ such that for every k,*

$$\begin{aligned}(Az)(k) &= \sum_{i=1}^{n} z_i f(x_k, x_i) \\ &= \sum_{i=1}^{n} z_i \sum_{j=1}^{d'} g\left(T_1(x_k)(j), T_2(x_i)(j)\right) \quad \text{(breaking } f \text{ into sum over } d' \text{ terms)} \\ &= \sum_{j=1}^{d'} \sum_{i=1}^{n} z_i \cdot g\left(T_1(x_k)(j), T_2(x_i)(j)\right) \quad \text{(switching order of summation).}\end{aligned}$$

Further suppose that each of the terms $\sum_{i=1}^{n} z_i \cdot g\left(T_1(x_k)(j), T_2(x_i)(j)\right)$ can be evaluated as

$$\sum_{i=1}^{n} z_i \cdot g\left(T_1(x_k)(j), T_2(x_i)(j)\right) = g\left(T_1(x_k)(j), \sum_{i=1}^{n} z_i T_2(x_i)(j)\right)$$

for any choice of the vector z. Then $g(a, b)$ must be a linear function in b.

Theorem B.1 is stated in quite general terms. We are stipulating the following statements: the functions T_1, T_2 represent possibly non-linear transformations to $\mathbb{R}^{d'}$ on x, y respectively such that $f(x, y)$ can be decomposed as a sum over d' function evaluations. Each function evaluation takes in the same coordinate, say the jth coordinate, of both $T_1(x)$ and $T_2(y)$ and computes the function $g(T_1(x)(j), T_2(y)(j))$. Finally the resulting sum $\sum_{i=1}^{n} z_i \cdot g\left(T_1(x_k)(j), T_2(x_i)(j)\right)$ can be computed as $g\Big(T_1(x_k)(j), \sum_{i=1}^{n} h(z_i) T_2(x_i)(j)\Big)$.

If these conditions hold (which is precisely the case in the proof of *all* our upper bound results), then it *must* be the case that g has a very special form, in particular, g must be a linear function in its second variable. To make the setting more concrete, we map the terminology of Theorem B.1 into some examples from our upper bound results.

First consider the case that $f(x, y) = \langle x, y \rangle$. In this case, both T_1 and T_2 are the identity maps and $g(a, b) = ab$. It is indeed the case that $g(a, b)$ is linear in b. Now consider a slightly more complicated choice $f(x, y) = \sum_{j=1}^{d} x(j) \log(y(j))$. Here, we first have the mappings T_1 = identity but T_2 is a coordinate wise map such that $T_2(y) = [\log(y_1), \ldots, \log(y_n)]$. The function g again satisfies $g(a, b) = ab$. Finally we consider the example $f(x, y) = \|x - y\|_2^2$ which sets $d' \gg d$. In particular, the mapppings T_1, T_2 expand x, y into a $O(d^2)$-dimensional vector, whose coordinates represent all possible combinations products of two coordinates of x and y respectively. (The reader may realize that this particular case is an example of "linearizing" the kernel given by f). Again g is the same function as before.

The proof of Theorem B.1 relies on the following classical result on the solutions of Cauchy's functional equation.

Theorem B.2. *Let $t : \mathbb{R} \to \mathbb{R}$ be a continuous function which satisfies $t(x + y) = t(x) + t(y)$ for all inputs x, y in its domain. Then t must be a linear function.*

For us the hypothesis that t is continuous suffices but it is know that the above result follows from much weaker hypothesis. We refer to the reader to [Kuc09] for reference related to Cauchy's functional equation.

Proof of Theorem B.1. Our goal is to show that if

$$\sum_{i=1}^{n} z_i \cdot g\left(T_1(x_k)(j), T_2(x_i)(j)\right) = g\left(T_1(x_k)(j), \sum_{i=1}^{n} z_i T_2(x_i)(j)\right)$$

for all z and choices of input points x_i then g must be linear in the second variable. First set $z_j = 0$ for all $j \geq 2$ and $z_1 = z_2 = 1$. For ease of notation, denote $q := T_1(x_k)(j)$. As we vary the coordinate of the points x_1 and x_2, the values $T_2(x_1)(j)$ and $T_2(x_2)(j)$ also vary over the range of T_2. Thus,

$$g(q, a) + g(q, b) = g(q, a + b)$$

for all possible inputs a, b. However, this is exactly the hypothesis of Theorem B.2 so it follows that g must be a linear function in its second coordinate, as desired. $\square$

While the proof of Theorem B.1 is straightforward, it precisely captures the scenarios where our upper bound techniques apply. In short, it implies that f must have a linear structure, under a suitable change of basis, for our techniques to hold. If its not the case, then our techniques do not apply and new ideas are needed. Nevertheless, as displayed by the versatility of examples in Table 1, such a structure is quite common in many applications where matrices of distance or similarity functions arise.

The observant reader might wonder how our result for the ℓ_1 function fits into the above framework as it is not obviously linear. However, we note that the function $h_j(x) = \sum_{i=1}^n |x(j) - x_i(j)|$ (which appears in the theorem statement of Theorem B.1 as the sum $\sum_{i=1}^n z_i \cdot g\Big(T_1(x_k)(j), T_2(x_i)(j)\Big)$ is actually a *piece-wise* linear function in $x(j)$. The sorting preprocessing we performed for Theorem 2.1 can be thought of as creating a data structure which allows us to efficiently index into the correct linear piece.

C Applications of Matrix-Vector Products

C.1 Preliminary Tools

We highlight specific prior results which we use in conjunction with our matrix-vector query upper bounds to obtain improved algorithmic results. First we recall a result of [BCW22] which gives a nearly optimal low-rank approximation result in terms of the number of matrix-vector queries required.

Theorem C.1 (Theorem 5.1 in [BCW22])**.** *Given matrix-vector query access to a matrix $A \in \mathbb{R}^{n\times n}$, accuracy parameter $\varepsilon \in (0,1), k \in [n]$ and any $p \geq 1$, there exists an algorithm which uses $\tilde{O}(k/\varepsilon^{1/3})$ matrix-vector queries and outputs a $n \times k$ matrix Z with orthonormal columns such that with probability at least $9/10$,*

$$\|A(I - ZZ^T)\|_p \leq (1+\varepsilon) \min_{U:U^TU=I_k} \|A(I - UU^T)\|_p$$

where $\|M\|_p = (\sum_{i=1}^n \sigma_i(M)^p)^{1/p}$ is the p-th Schatten norm where $\sigma_1, \ldots, \sigma(M)$ are the singular values of M. The runtime of the algorithm is $\tilde{O}(Tk/\varepsilon^{1/3} + nk^{w-1}/\varepsilon^{(w-1)/3})$ where T is the time for computing a matrix-vector query.

The second result is of [MM15] which give an optimized analysis of a variant of power method for computing the top k singular values.

Theorem C.2 (Theorem 1 and 7 in [MM15])**.** *Given matrix-vector query access to a matrix $A \in \mathbb{R}^{n\times n}$, accuracy parameter $\varepsilon \in (0,1), k \in [n]$, there exists an algorithm which uses $\tilde{O}(k/\varepsilon^{1/2})$ matrix-vector queries and outputs a $1 \pm \varepsilon$ approximation to the top k singular values of A. The runtime of the algorithm is $\tilde{O}(Tk/\varepsilon^{1/2} + nk^2/\varepsilon + k^3/\varepsilon^{3/2})$.*

Lastly, we recall the gaurantees of the classical conjugate-gradient descent method.

Theorem C.3. *Let A be a symmetric PSD matrix and consider the linear system $Ax = b$ and let $x^* = \text{argmin}_x \|Ax - b\|_2$. Let κ denote the condition number of A. Given a starting vector x_0, the conjugate gradient descent algorithm uses $O(\sqrt{\kappa}\log(1/\varepsilon))$ matrix-vector queries and returns x such that*

$$\|x - x^*\|_A \leq \varepsilon \|x_0 - x^*\|_A$$

where $\|x\|_A = (x^TAx)^{1/2}$ denotes the A-norm.

Note that matrices in our setting are also PSD, for example if $A_{i,j} = \langle x_i, x_j\rangle$. For non PSD matrices A, one can also use the conjugate gradient descent method on the matrix A^TA which squares the condition number. Therefore, there are more complicated algorithms which work directly on the matrix-vector queries of A for non PSD matrices, for example see references in Chapters 6 and 7 of of [AG11]. We omit their discussion for clarity and just note that in practice, iterative methods which directly use matrix-vector queries are preferred for linear system solving.

C.2 Applications

We now derive specific applications using prior results from "matrix-free" methods. First we cover low-rank approximation.

For the ℓ_1 and ℓ_2^2 distance matrices, we improve upon prior works for computing a relative error low-rank approximation. While we can obtain such an approximation for a wide variety of Schatten norms, we state the bound in terms of the Frobenius norm since it has been studied in prior works.

Theorem C.4. *Let $p \geq 1$ and consider the case that $A_{i,j} = \|x_i - x_j\|_p^p$ for all i, j. We can compute a matrix B such that*

$$\|A - B\|_F \leq (1 + \varepsilon)\|A - A_k\|_F$$

where A_k denotes the optimal rank-k approximation to A in Frobenius norm. The runtime is $\tilde{O}(ndpk/\varepsilon^{1/3} + nk^{w-1}/\varepsilon^{(w-1)/3})$.

Proof. The theorem follows from combining the matrix-vector query runtime of Table 1 and Theorem C.1. □

Note that the best prior result for the special case of ℓ_1 and ℓ_2^2 from [BCW20] where they obtained a runtime bound of $O(ndk/\varepsilon + nk^{w-1}/\varepsilon^{w-1})$. Thus our bound improves upon this by a multiplicative factor of $\text{poly}(1/\varepsilon)$. We point out that the bound of $O(ndk/\varepsilon + nk^{w-1}/\varepsilon^{w-1})$ is actually *optimal* for the class of algorithms which sample the entries of A. Thus our results show that if we know the set of points beforehand, which is a natural assumption, one can overcome such lower bounds.

For the case of $A_{i,j} = \|x_i - x_j\|_2$, we cannot hope to achieve a relative error approximation for low-rank approximation since we only have fast matrix-vector queries to the matrix B where $B_{i,j} = (1 \pm \varepsilon)\|x_i - x_j\|_2$ via Theorem A.12. Nevertheless, we can still obtain an additive error low-rank approximation guarantee which outperforms prior works. First we show that our approximation matrix-vector queries are sufficient to obtain such a guarantee.

Lemma C.5. *Let A, B satisfy $\|A - B\|_F \leq \varepsilon\|A\|_F$ and suppose A' and B' are the best rank-r approximation of A and B respectively in the Frobenius norm. Then*

$$\|A - B'\|_F \leq \|A - A'\|_F + 2\varepsilon\|A\|_F.$$

Proof. We have

$$\begin{aligned}
\|A - B'\|_F &\leq \|A - B\|_F + \|B - B'\|_F \\
&\leq \varepsilon\|A\|_F + \|B - A'\|_F \\
&\leq \varepsilon\|A\|_F + \|B - A\|_F + \|A - A'\|_F \\
&\leq \|A - A'\|_F + 2\varepsilon\|A\|_F.
\end{aligned}$$

□

Theorem C.6. *Let $A_{i,j} = \|x_i - x_j\|_2$ for all i, j. We can compute a matrix B such that*

$$\|A - B\|_F \leq \|A - A_k\|_F + \varepsilon\|A\|_F$$

with probability $1 - 1/\text{poly}(n)$ where A_k denotes the optimal rank-k approximation to A in Frobenius norm. The runtime is $\tilde{O}(ndk/\varepsilon^{1/3} + nk^{w-1}/\varepsilon^{(w-1)/3})$.

Proof. The runtime follows from applying Theorem C.1 on the matrix created after applying Theorems A.11 and A.12. The approximation guarantee follows from Lemma C.5. □

The best prior work for additive error low-rank approximation for this case is due to [IVWW19] which obtained such a guarantee with runtime $\tilde{O}(nd \cdot \text{poly}(k, 1/\varepsilon))$ for a large unspecified polynomial in k and $1/\varepsilon$. Lastly we note that our relative error low-rank approximation guarantee holds for any f in Table 1, as summarized in Table 2.

Theorem C.7. *Suppose we have exact matrix-vector query access to a matrix A with each query taking time T. Then we can output a matrix B such that*

$$\|A - B\|_F \leq (1 + \varepsilon)\|A - A_k\|_F$$

where A_k denotes the optimal rank-k approximation to A in Frobenius norm. The runtime is $\tilde{O}(Tk/\varepsilon^{1/3} + nk^{w-1}/\varepsilon^{(w-1)/3})$.

Directly appealing to Theorems C.2 and C.3 in conjunction with our matrix-vector queries achieves the fastest runtime for computing the top k singular values and solving linear systems for a wide variety of distance matrices that we are aware of.

Theorem C.8. *Suppose we have exact matrix-vector query access to a matrix A with each query taking time T. We can compute a $1 \pm \varepsilon$ approximation to the k singular values of A in time $\tilde{O}(T/\varepsilon^{1/2} + nk^2/\varepsilon + k^3/\varepsilon^{3/2})$. Furthermore, we can solve linear systems for A with the same guarantees as any iterative method which only uses matrix-vector queries with an multiplicative overhead of T.*

Finally, we can also perform matrix multiplication faster with distance matrices compared to the general runtime of $n^w \approx n^{2.37}$. This follows from the following lemma.

Lemma C.9. *Suppose $A \in \mathbb{R}^{n\times n}$ admits an exact matrix-vector query algorithm in time T. Then for any $B \in \mathbb{R}^{n\times n}$, we can compute AB in time $O(Tm)$.*

Proof. We can compute AB by computing the product of A with the n columns of B separately. □

As a corollary, we obtain faster matrix multiplication for all the family of matrices which we have obtained a fast matrix-vector query for. We state one such corollary for the ℓ_p^p case.

Corollary C.10. *Let $p \geq 1$ and consider the case that $A_{i,j} = \|x_i - x_j\|_p^p$ for all i, j. For any other matrix B, we can compute AB in time $O(n^2dp)$.*

We can improve upon the above result slightly if we are multiplying two distance matrices for the $p = 2$ case.

Lemma C.11. *Consider the case that $A_{i,j} = \|x_i - x_j\|_2^2$ for all i, j and $B_{i,j} = \|y_i - y_j\|_2^2$, i.e., both A and B are $n \times n$ matrices with $f = \ell_2^2$. We can compute AB in time $O(n^2d^{w-2})$.*

Proof. By decomposing both A and B, it suffices to compute the product XX^TYY^T where $X, Y \in \mathbb{R}^{n\times d}$ are the matrices with the points x_i and y_i in the rows respectively. $Z_1 := X^TY \in \mathbb{R}^{d\times d}$ can be computed in $O(nd^2)$ time. Then $Z_2 := XZ_1 \in \mathbb{R}^{n\times d}$ can also be computed in $O(nd^2)$ time. Finally, we need to compute $Z_2 \times Y^T$. This can be done in $O(n^2d^{w-2})$ time by decomposing both Z_2 and Y^T into n/d many $d \times d$ square matrices and using the standard matrix multiplication bound on each pair of square matrices. This results in the claimed runtime of $O((n/d)^2 \cdot d^w) = O(n^2d^{w-2})$. □

D A Fast Algorithm for Creating ℓ_1 and ℓ_2 Distance Matrices

We now present a fast algorithm for creating distance matrices which addresses our contribution (3) stated in the introduction. Given a set of n points $x_1, \ldots, x_n$ in $\mathbb{R}^d$, our goal is to initialize an approximate $n \times n$ distance matrix B for the ℓ_1 distance which satisfies

$$B_{ij} = (1 \pm \varepsilon)\|x_i - x_j\|_1 \tag{1}$$

for all entries of B where $0 < \varepsilon < 1$ is a precision parameter. The straightforward way to create the *exact* distance matrix takes take $O(n^2d)$ time and by using the stability of Cauchy random variables, we can create B which satisfies (1) in $O(n^2 \log n)$ time for any constant ε. (Note the Johnson-Lindenstrauss lemma implies a similar guarantee for the ℓ_2 distance matrix). The goal of this section is to improve upon this 'baseline' runtime of $O(n^2 \log n)$. The final runtime guarantees of this section will be of the form $O(n^2 \cdot \text{poly}(\log \log n))$.

Our improvement holds in the **Word-RAM** model of computation. Formally, we assume each memory cell (i.e. word) can hold $O(\log n)$ bits and certain computations on words take $O(1)$ time. The only assumptions we require are the arithmetic operations of adding or subtracting words as well as performing left or right bit shifts on words takes constant time.

We first present prior work on metric compression of [IRW17] in Section D.1. Our algorithm description starts from Section D.2 which describes our preprocesing step. Section D.3 then presents our key algorithm ideas whose runtime and accuracy are analyzed in Sections D.4 and D.5.

D.1 Metric Compression Tree of [IRW17]

The starting point of our result is the metric compression tree construction of [IRW17], whose properties we summarize below. First we introduce some useful definitions. The aspect ration Φ of X is defined as

$$\Phi = \frac{\max_{i,j} \|x_i - x_j\|_1}{\min_{i \neq j} \|x_i - x_j\|_1}.$$

Let $\Delta' = \max_{i \in [n]} \|x_1 - x_i\|_1$ and $\Delta = 2^{\lceil \log \Delta' \rceil}$.

Theorem 1 of [IRW17] implies the following result. Given a dataset $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$ with aspect ration Φ, there exists a tree data structure T which allows for the computation of a compressed representation X for the purposes of distance computations. At a high level, T is created by enclosing X in a large enough and appropriately shifted axis-parallel square and then recursively dividing into smaller squares (also called cells) with half the side-length until all points of X are contained in their own cell. The edges of T encode the cell containment relationships. Formally, T has the following properties:

- The leaf nodes of T correspond to the points of X.
- The edges of T are of two types: short edges and long edges which are defined as follows. Short edges have a length d bit vector associated with them whereas long edges have an integer $\leq O(\log \Phi)$ associated with them.
- Each long edge with associated integer k represents a non-branching path of length k of short edges, all of whose associated length d bit vectors are the 0 string.
- Each node of T (including the nodes that are on paths which are compressed as long edges) have an associated level $-\infty < \ell \leq \log(4\Delta)$. A level ℓ of a node v corresponds to an axis-parallel square G_ℓ of side length 2^ℓ which contains all axis-parallel squares of child nodes of v.

The notion of a padded point is important for the metric compression properties of T.

Definition D.1 (Padded Point)**.** *A point x_i is $(\varepsilon, \Lambda, \ell)$-padded, if the grid cell G_ℓ of side length 2^ℓ that contains x_i also contains the ball of radius $\rho(\ell)$ centered at x_i, where*

$$\rho(\ell) = 8\varepsilon^{-1} 2^{\ell - \Lambda} \sqrt{d}.$$

We say that x_i is (ε, Λ)-padded in T, if it is $(\varepsilon, \Lambda, \ell)$-padded for every level ℓ.

The following lemma is proven in [IRW17]. First define

$$\Lambda = \log(16 d^{1.5} \log \Phi / (\varepsilon \delta)). \tag{2}$$

Lemma D.1 (Lemma 1 in [IRW17])**.** *Consider the construction of T defined formally in Section 3 of [IRW17]. Every point x_i is (ε, Λ)-padded in T with probability $1 - \delta$.*

Now let x be any point in our dataset. We can construct $\widetilde{x} \in \mathbb{R}^d$, called the decompression of x, from T with the following procedure: We follow the downward path from the root of T to the leaf associated with x and collect a bit string for every coordinate d of $\widetilde{x}$. When going down a short edge with an associated bit vector b, we concatenate the ith bit of b to the end of the bit string that we are keeping track of for the ith coordinate of $\widetilde{x}$. When going down a long edge, we concatenate with a number of zeros equalling the integer associated with the long edge. Finally, a binary floating point is placed in the resulting bit strings of each coordinate after the bit corresponding to level 0. The collected bits then correspond to the binary expansion of the coordinates of $\widetilde{x}$. For a more thorough description of the decompression scheme, see Section 3 of [IRW17].

The decompression scheme is useful because it allows approximate distance computations.

Lemma D.2 (Lemma 2 in [IRW17])**.** *If a point x_i is (ε, Λ)-padded in T, then for every $j \in [n]$,*

$$(1 - \varepsilon)\|\widetilde{x}_i - \widetilde{x}_j\|_1 \leq \|x_i - x_j\|_1 \leq (1 + \varepsilon)\|\widetilde{x}_i - \widetilde{x}_j\|_1.$$

We now cite the runtime and space required for T. The following theorem follows from the results of [IRW17].

Theorem D.3. *Let $L = \log \Phi + \Lambda$. T has $O(n\Lambda)$ edges, height L, its total size is $O(nd\Lambda + n \log n)$ bits, and its construction time is $O(ndL)$.*

We contrast the above gauranttes with the naive representation of X which stores $O(\log n)$ bits of precision for each coordinate and occupies $O(nd \log n)$ bits of space, whereas T occupies roughly $O(nd \log \log n + n \log n)$ bits.

Finally, Theorem 2 in [IRW17] implies we can create a collection of $O(\log n)$ trees T (by setting δ to be a small constant in (2)) such that every point in X is padded in at least one tree in the collection.

D.2 Step 1: Preprocessing Metric Compression Trees

We now describe the preprocessing steps needed for our faster distance matrix compression. Let

$$w = \frac{4d\Lambda}{\log n} \tag{3}$$

and recall our setting of Λ in (2). Note that we assume w is an integer which implicitly assumes $4d\Lambda \geq \log n$.

First we describe the preprocessing of T. Consider a short edge of T with an associated d length bit string b. We break up b into w equal chunks, each containing d/w bits. Consider a single chunk c. We pad (an equal number of) 2Λ many 0's after each bit in c so that the total number of bits is $\log n/2$. We then store each padded chunk in one word. We do this for every chunk resulting in w words for each short edge and we do this for all short edges in all the trees.

The second preprocessing step we perform is creating a $O(\sqrt{n}) \times O(\sqrt{n})$ table A. The rows and columns of A are indexed by all possible bit strings with $\frac{\log n}{2}$ bits. The entries of A record evaluations of the function $f(x, y)$ defined as follow: given x, y where $x, y \in \{0, 1\}^{\frac{\log n}{2}}$, consider the partition of each of them into d/w blocks, each with an equal number of bits (2Λ bits per block. Note that $2\Lambda \cdot d/w = (\log n)/2$). Each block defines an integer. Doing so results in d/w integers $x^1, \ldots, x^{d/w}$ derived from x and w integers $y^1, \ldots, y^{d/w}$ derived from y. Finally,

$$f(x, y) = \sum_{i=1}^{d/w} |x^i - y^i|.$$

D.3 Step 2: Depth-First Search

We now calculate one row of the distance matrix from point a padded point x to all other points in our dataset. Our main subroutine is a tree search procedure. Its input is a node v in a tree T and it performs a depth-first search on the subtree rooted at v as described in Algorithm 11. Given an internal vertex v, it calculates all the distances between the padded point x to all data points in our dataset which are leaf nodes in the subtree rooted at v. A summary of the algorithm follows.

We perform a depth-first search starting at v. As we traverse the tree, we keep track of the current embedding of the internal nodes via collecting bits along the edges of T: we append bits when we descend the tree and remove as we move up edges. However, we only keep track of this embedding up to 2Λ levels below v. After that, we continue traversing the tree but don't update the embedding. The reason for this is after 2Λ levels, the embedding is precise enough for all nodes below with respect to computing the distance to x. Towards this end, we also track how many levels below v the tree search is currently at and update this value appropriately.

The current embedding is tracked using w words. Recall that the bit string of every short edge has been repackaged into w words, each 'responsible' for d/w coordinates. Furthermore, in each word on the edge, we have padded 0's between the bits of each d/w coordinates. When we need to update the current embedding by incorporating the bits along a short edge e, we simply perform a bit shift on each of the w words on e and add it to the w words we are keeping track of. We need to make sure we place the bits 'in order.' That is for our tracked embedding, for every d coordinates, the bits on an edge e should precede the bits on the edge directly following e in the tree search. Due to the padding from the preprocessing step, the bit shift implies the bits on the short edges after e will be placed in their appropriate corresponding places in order in the embedding representation.

Algorithm 11 DFS in Subtree

Input: Metric Compression Tree T, node v
procedure SEARCH(T, v)
    Initialize a global counter p for the number of levels which have been processed. Initially set to 0 and will be at most 2Λ
    Initialize w words $t_1, \ldots, t_w$, all initially 0
    Initialize a global counter r for the current level which is initially set to the level of v in T
    Perform a depth-first search in the subtree rooted at v. Run `Process-Short-Edge` if a short edge is encountered, `Process-Long-Edge` if a long edge is encountered, and `Process-Leaf` when we arrive at a leaf.
end procedure

While performing the depth-first search, we will encounter both short and long edges. When encountering a short edge, we run the function `Process-Short-Edge` and similarly, we run `Process-Long-Edge` when a long edge is encountered. Finally if we arrive at a leaf node, we run `Process-Leaf`.

Algorithm 12 Process Short Edge

Input: Short edge e, number of processed nodes p, $t_1, \ldots, t_w$
procedure PROCESS-SHORT-EDGE($e, p, t_1, \ldots, t_w$)
    Let $e_1, \ldots, e_w$ be the w words associated with edge e
    If search is traversing down e and $p < 2\Lambda$, add $2^{-p}e_i$ to t_i for all $1 \leq i \leq w$ and increment p
    If search is traversing up e and $p \leq 2\Lambda$, subtract $2^{-p}e_i$ from t_i for all $1 \leq i \leq w$ and decrement p
    Update r to the level of the current node
end procedure

Algorithm 13 Process Long Edge

Input: Long edge e, number of processed nodes p
procedure PROCESS-LONG-EDGE(e, p)
    If search is traversing down e and $p < 2\Lambda$, increment p
    If search is traversing up e and $p \leq 2\Lambda$, decrement p
    Update r to the level of the current node
end procedure

When we arrive at a leaf node y, we currently have the decompression of y computed from the tree. Note that we only have kept track of the bits after node v (up to limited precision) since all prior bits are the same for y and x since they are in the same subtree. More specifically, we have w words $t_1, \ldots, t_w$. The first word t_1 has 2Λ bits of each of the first d/w coordinates of y. For every coordinate, the 2Λ bits respect the order described in the decompression step in Section D.2. A similar fact is true for the rest of the words t_i. Now to calculate the distance between x and y, we just have to consider the 2Λ bits of all d coordinates of x which come after descending down vertex v. We then repackage these $2d\Lambda$ total bits into w words in the same format as y. Note this preprocessing for x only happens once (at the subtreee level) and can be used for all leaves in the subtree rooted at v.

Algorithm 14 Process Leaf

Input: $t_1, \ldots, t_w$
procedure PROCESS-LEAF($y, t_1, \ldots, t_w$)
    Let the point y correspond to the current leaf node
    Let $s_1, \ldots, s_w$ denote the embedding of x after node v, preprocessed to be in the same format as $t_1, \ldots, t_w$
    Report $\sum_{i=1}^{w} A[t_i, s_i]$ as the distance between x and y
end procedure

Finally, the complete algorithm just calls Algorithm 11 on successive parent nodes of x. We mark each subtree that has already been processed (at the root node) so that the subtree is only ever visited once. The number of calls to Algorithm 11 is at most the height of the tree, which is bounded by $O(\log \Phi + \Lambda)$. We then repeat this for all points x in our dataset (using the tree which x is padded in) to create the full distance matrix.

D.4 Runtime Analysis

We consider the runtime required to compute the row corresponding to a padded point x in the distance matrix. Multiplying by n results in the total runtime. Consider the tree T in which x is padded in and which we use for the algorithm described in the previous section and recall the properties of T outlined in Theorem D.3. T has $O(n\Lambda)$ edges, each of which is only visited at most twice in the tree search (going up and down). Thus the time to traverse the tree is $O(n\Lambda)$. There are also at most $O(n\Lambda)$ short edges in T. Updating the embedding given by $t_1, \ldots, t_w$ takes $O(w)$ time per edge since it can be done in $O(w)$ total word operations. Long edges don't require this time since they represent 0 bits; for long edges, we just increment the counter for the current level. Altogether, the total runtime for updating $t_1, \ldots, t_w$ across all calls to Algorithm 11 for the padded point x is $O(n\Lambda w)$. Finally, calculating the distance from x to a fixed point y requires $O(w)$ time since we just index into the array A w times. Thus the total runtime is dominated by $O(n\Lambda w)$. Finally, the total runtime for computing all rows of the distance matrix is

$$O(n^2 \Lambda w) = O\left(\frac{n^2 d \Lambda^2}{\log n}\right) = O\left(\frac{n^2 d}{\log n} \log^2\left(\frac{d \log \Phi}{\varepsilon}\right)\right)$$

by setting δ to be a small constant in (2).

D.5 Accuracy Analysis

We now show that the distances we calculate are accurate within a $1 \pm \varepsilon$ multiplicative factor. The lemma below shows that if a padded point x and another point y have a sufficiently far away least-common ancestor in T, then we can disregard many lower order bits in the decompression computed from T while still guaranteeing accurate distance measurements. The lemma crucially relies on x being padded.

Lemma D.4. *Suppose x is (ε, Λ)-padded in T. For another point y, suppose the least common ancestor of x and y is at level ℓ. Let $\widetilde{x}$ and $\widetilde{y}$ denote the sketches of x and y produced by T. Let $\widetilde{x}'$ be a modified version of $\widetilde{x}$ where for each of the d coordinates, we remove all the bits acquired after level $\ell - 2\Lambda$. Similarly define $\widetilde{y}'$. Then*

$$\|\widetilde{x}' - \widetilde{y}'\|_1 = (1 \pm \varepsilon)\|x - y\|_1.$$

Proof. Since x is padded, we know that $\|x - y\|_1 \geq p(\ell - 1)$ by Definition D.1. On the other hand, if we ignore the bits after level $\ell - 2\Lambda$ for every coordinate of $\widetilde{x}$ and $\widetilde{y}$, the additive approximation error in the distance is bounded by a constant factor times

$$d \cdot \sum_{i=-\infty}^{\ell - 2\Lambda - 1} 2^i = d \cdot 2^{\ell - 2\Lambda}.$$

From our choice of Λ, we can easily verify that $d \cdot 2^{\ell - 2\Lambda} \leq \varepsilon p(\ell - 1)/2$. Putting everything together and adjusting the value of ε, we have

$$\|\widetilde{x}' - \widetilde{y}'\|_1 = \|\widetilde{x} - \widetilde{y}\|_1 \pm \varepsilon p(\ell - 1)/2 = (1 \pm \varepsilon/2)\|x - y\|_1 \pm \varepsilon p(\ell - 1)/2 = (1 \pm \varepsilon)\|x - y\|_1$$

where we have used the fact that $\|\widetilde{x} - \widetilde{y}\|_1 = (1 \pm \varepsilon/2)\|x - y\|_1$ from the guarantees of the compression tree of [IRW17]. □

Putting together our results above along with the Johnson-Lindenstrauss Lemma and Theorem A.11 proves the following theorem.

Theorem D.5. *Let $X = \{x_1, \ldots, x_n\}$ be a dataset of n points in d dimensions with aspect ration Φ. We can calculate a $n \times n$ matrix B such that each (i, j) entry B_{ij} of B satisfies*

$$(1 - \varepsilon)\|x_i - x_j\|_1 \leq B_{ij} \leq (1 + \varepsilon)\|x_i - x_j\|_1$$

in time

$$O\left(\frac{n^2 d}{\log n}\log^2\left(\frac{d\log\Phi}{\varepsilon}\right)\right).$$

Assuming the aspect ratio is polynomially bounded, we can compute $n \times n$ matrix B such that each (i,j) entry B_{ij} of B satisfies

$$(1-\varepsilon)\|x_i - x_j\|_2 \le B_{ij} \le (1+\varepsilon)\|x_i - x_j\|_2$$

with probability $1 - 1/poly(n)$. The construction time is

$$O\left(\frac{n^2}{\varepsilon^2}\log^2\left(\frac{\log n}{\varepsilon}\right)\right).$$

D.6 A Faster Algorithm for ℓ_∞ Distance Matrix Construction Over Bounded Alphabet

In this section, we show how to create the ℓ_∞ distance matrix. Recall from Section 3 that there exists no $o(n^2)$ time algorithm to compute a matrix-vector query for the ℓ_∞ distance matrix, assuming SETH, even for n points in $\{0,1,2\}^d$. This suggests that any algorithm for computing a matrix-vector query needs to initialize the distance matrix. However, there is still a gap between a $\Omega(n^2)$ lower bound for matrix-vector queries and the naive $O(n^2 d)$ time needed to compute the ℓ_∞ distance matrix. We make progress towards showing that this gap is not necessary. Our main result is that surprisingly, we can *initialize* the ℓ_∞ distance matrix in time much faster than the naive $O(n^2 d)$ time.

Theorem D.6. *Given n points over $\{0,1,\ldots,M\}^d$, we can initialize the exact ℓ_∞ distance matrix in time $O(M^{w-1}n^2(d\log M)^{w-2})$ where w is the matrix multiplication constant. We can also initialize a $n \times n$ matrix B such that each (i,j) entry B_{ij} of B satisfies*

$$(1-\varepsilon)\|x_i - x_j\|_\infty \le B_{ij} \le (1+\varepsilon)\|x_i - x_j\|_\infty$$

in time $\tilde{O}(\varepsilon^{-1}n^2(dM)^{w-2})$.

Thus for $M = O(1)$, which is the setting of the lower bound, we can initialize the distance matrix in time $O(n^2 d^{w-2})$ and thus compute a matrix-vector query in that time as well.

Proof. The starting point of the proof is to first design an algorithm which constructs a matrix with (i,j) entry an indicator vector for $\|x-y\|_\infty \le i$ or $\|x-y\|_\infty > i$. Given this, we can then sum across all M choices and construct the full distance matrix. Thus it suffices to solve this intermediate task.

Pick a sufficiently large p such that $di^p \le (i+1)^p$. A choice of $p = O(M\log d)$ suffices. Now in the case that $\|x-y\|_\infty \le i$, we have $\|x-y\|_p^p \le di^p$ and otherwise, $\|x-y\|_p^p \ge (i+1)^p$. Thus, the matrix C with the (i,j) entry being $\|x_i - x_j\|_p^p$ is able to distinguish the two cases so it suffices to create such a matrix. Now we can write $\|x-y\|_p^p$ as an inner product in $O(pd)$ variables, i.e., it is a gram matrix. Thus computing C can be done by computing a product of $n \times O(pd)$ matrix by a $O(pd) \times n$ matrix, which can be done in

$$O\left(\left(\frac{n}{pd}\right)^2 (pd)^{w-2}\right) = O(n^2(pd)^{w-2}).$$

time by partitioning each matrix into square submatrices of dimension $O(pd)$. Plugging in the bound for p and considering all possible choices of i results in the final runtime bound of $O(M^{w-1}n^2(d\log M)^{w-2})$, as desired.

Now if we only want to approximate each entry up to a multiplicative $1 \pm \varepsilon$ factor, it suffices to only loop over i's which are increasing by powers of $1 + c\varepsilon$ for a small constant c. This replaces an $O(M)$ factor by an $O(\varepsilon^{-1}\log M)$ factor. □