Note, this work is currently under review.

BitBooster: Effective Approximation of Distance Metrics via Binary Operations

Y. Spenrath, M. Hassani, B.F. van Dongen

September 7, 2021

Abstract

The Euclidean distance is one of the most commonly used distance metrics. Several approximations have been proposed in the literature to reduce the complexity of this metric for high-dimensional or large datasets. In this paper, we propose BitBooster, an approximation to the Euclidean distance that can be efficiently computed using binary operations and which can also be applied to the Manhattan distance. The introduced approximation error is shown to be negligible when BitBooster is used for both convex- and density-based clustering. While obtaining clusters of almost the same quality as those obtained with the exact computation, we require only a fraction of the computation time. We demonstrate the superiority of our method to alternative approximations on both synthetic and real-world datasets.

1 Introduction

The Euclidean distance metric is used on a plethora of applications. Amongst these are information retrieval and nearest-neighbour search (kNN) [1–3], and clustering methods [4–8]. Especially in the latter, most optimizations are focused towards the clustering algorithm itself. Improvements are proposed to either reduce the number of used dimensions, or restrict the number of distance computations. In this paper we present BitBooster, which can be considered orthogonal to these improvements in the sense that we reduce the precision of each value by discretization without reducing the number of dimensions or distance metric computations. These discretizations allow BitBooster to replace floating-point arithmetic used by the Euclidean distance with binary operations that can effectively be computed by modern CPUs. This provides an efficient approximation of the Euclidean distance metric that runs in $O(\lceil |F|/64 \rceil)$ complexity on a metric space of size $|F|$, reducing computational complexity and improving clustering algorithms.

The contributions we make are the following. 1) We propose a framework that efficiently represents multidimensional data as one or more integers. 2) We show that we can efficiently estimate the Euclidean distance between two

of the real-valued points, based on the integer representations, with a trade-off between precision and computational complexity, and which can also be applied to the Manhattan distance. 3) We apply our methods to convex- and density-based clustering, demonstrating the efficiency and effectiveness of BitBooster on synthetic and real-world datasets, and show the superiority of our approach over alternative approximations.

The remainder of the paper is structured as follows. In Section 2 we discuss the related work. We then propose our BitBooster solution in Section 3 and perform an experimental evaluation in Section 4. We conclude the paper and provide directions for future research in Section 5.

2 Related Work

In this section we discuss related literature. Existing literature focuses on approximating the Euclidean distance and not the Manhattan distance, as the latter is can be regarded as an approximation of the former [9].

One area of optimizations is based on approximations involving the inner product between two real-valued vectors. The relation between the Euclidean distance and the inner product in a metric space F on vectors $\mathbf{x} = (x_1, x_2, \dots x_{|F|})$ and $\mathbf{y} = (y_1, y_2, \dots y_{|F|})$ is given by

$$\begin{aligned} d(\mathbf{x}, \mathbf{y}) &= \sqrt{\|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\langle \mathbf{x}, \mathbf{y} \rangle} \\ &= \sqrt{\|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\,\|\mathbf{x}\|\,\|\mathbf{y}\| \cos\theta_{\mathbf{xy}}} \end{aligned} \tag{1}$$

where $\|\mathbf{x}\|$ is the norm of $\mathbf{x}$, $\theta_{\mathbf{xy}}$ is the angle between vectors $\mathbf{x}, \mathbf{y}$ and $\langle \mathbf{x}, \mathbf{y} \rangle$ is the inner product. The main bottleneck is that the computation of the angle θ between $\mathbf{x}$ and $\mathbf{y}$ can be expensive. One way to overcome this is to use the Cauchy-Schwarz inequality [10], which states that the inner product is upper-bounded by $\|\mathbf{x}\| \cdot \|\mathbf{y}\|$ (as $|\cos\theta| < 1$). Using this upper-bound, the Euclidean distance can be approximated without the use of θ. However, in [1], the authors show that this approximation is too imprecise and propose an alternative approximation. Given a reference vector $\mathbf{r}$, a dataset D is preprocessed by computing $\|\mathbf{x}\|$ and the angle $\theta_{\mathbf{xr}}$ between $\mathbf{r}$ and $\mathbf{x}$ up front, for each $\mathbf{x} \in D$. For the reference vector, the authors recommend the first principle component learned over the dataset. When the distance between $\mathbf{x}$ and $\mathbf{y}$ is required, the angle $\theta_{\mathbf{xy}}$ can be estimated with precomputed values: $|\theta_{\mathbf{xr}} - \theta_{\mathbf{yr}}|$. This substantially reduces the computation cost of the Euclidean distance metric. The authors further show that this approximation is a lower-bound for the Euclidean distance and present its effectiveness in a kNN information retrieval setting, showing the superiority over the approximation using the Cauchy-Schwarz inequality. In the remainder of the paper; we refer to this approximation as `JKKC`, the initials of the authors of [1].

In [2], the authors describe an alternative technique called Angular Quantization, also based on Equation 1. The authors convert $\mathbf{x}$ to binary vectors

in $\{0,1\}^{|F|}$ by selecting the binary vector with the smallest angle to $\mathbf{x}$. The authors explain that this is only possible if the original points are non-negative vectors. They apply additional optimizations to mitigate some introduced approximation errors. If the points are normalized (lie on the unit hyper-sphere), the first line of Equation 1 can be done efficiently by approximating the inner product with the binary vectors, using an AND operator and the Hamming weight [11].

An alternative approximation from the electrical circuit field is given in [9]. The authors use a linear approximation of the Euclidean distance involving the maximum and minimum difference in each dimension. In particular, for two-dimensions the distance can be approximated by $d(\Delta_1, \Delta_2) \approx \alpha \cdot \max(\Delta_1, \Delta_2) + \beta \cdot \min(\Delta_1, \Delta_2)$, where $\Delta_1 = |x_1 - y_1|$, $\Delta_2 = |x_2 - y_2|$, and α, β are chosen such that the error is minimized. The authors proceed by extending this to more than two dimensions, by considering the difference in each dimension, and iterating the approximation; for three dimensions the result would be given by $d(d(\Delta_1, \Delta_2), |x_3 - y_3|)$. This formula is especially useful in electrical circuits, as the optimal values of α and β can be easily approximated by fractions of multiples of two, and the computations do not require squares or square roots. The authors show that their method is more accurate than the Manhattan distance, which is also used to approximate the Euclidean distance without using of the square or square root. In the remainder of this paper, we refer to this approximation as α`Max+`β`Min`.

An alternative in NN search is to use Locality Sensitive Hashing (LSH) [12]. While many variants exist; the concept of LSH is to map points in large feature spaces onto hashing tables, such that similar points map to the same hash value (preserving locality). The advantage to this is that 'near' points can be determined by only comparing hash values; removing the need for expensive (Euclidean) distance computations all together. This is especially useful for image [13,14] and text data [15], where the high dimensionality makes the mapping to the lower dimensionality hashes easier to preserve locality. For the purpose of clustering the drawback of this is that it is easy to retrieve near neighbours (by finding the same hash), but by design not possible to approximate the distance, as required by clustering algorithms. The authors in [14] do provide a way to approximate the *weighted Jaccard* distance using their MinHashing scheme. Roughly speaking, their method projects datapoints on a number of sample lines; remembering the approximate location on the samples with a discrete value. The weighted Jaccard distance is then approximated by the fraction of different projection values between two datapoints. More samples increase the precision of the approximation, but also the computation time of the projection and the comparison. We use this method as one of our competitors in the experimental evaluation, indicated by `WJMH`x, where x is the number of samples.

3 Methodology

In this section we discuss the methodology of our work. We start with a small example. The framework consists of two rationales. The first is that we can approximate the distance between two real-valued points by the distances between two discrete-valued points, the second is that we can efficiently compute the approximation. For the first rationale, consider Figure 1. The figure contains four points in a two-dimensional real-valued space, $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \in \mathbb{R}^2$. It further contains four points in a two-dimensional discrete-valued space, $\mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{h} \in \{0, 1\}^2$.

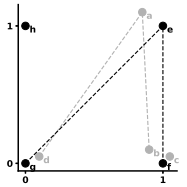

Figure 1: Distances between points in a real-valued space can be approximated by distance between points in a discrete space.

We can graphically see that the distance between points $\mathbf{a}$ and $\mathbf{b}$ is approximately equal to the distance between points $\mathbf{e}$ and $\mathbf{f}$, and that the distance between $\mathbf{a}$ and $\mathbf{d}$ is approximately equal to the distance between $\mathbf{e}$ and $\mathbf{g}$; and similar approximations can be noted between other pairwise combinations of $\mathbf{a}, \mathbf{b}, \mathbf{c}$ and $\mathbf{d}$.

The *Euclidean* distance between two points $\mathbf{x} = (x_1, x_2)$ and $\mathbf{y} = (y_1, y_2)$ in a two-dimensional space is defined as:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \tag{2}$$

The computation can be split in three steps:

- computing squared differences: $d_1(\mathbf{x}, \mathbf{y}) = (x_1 - y_1)^2$ and $d_2(\mathbf{x}, \mathbf{y}) = (x_2 - y_2)^2$
- summing the squared differences: $d_\Sigma(\mathbf{x}, \mathbf{y}) = d_1(\mathbf{x}, \mathbf{y}) + d_2(\mathbf{x}, \mathbf{y})$
- taking the square root of the sum: $d(\mathbf{x}, \mathbf{y}) = \sqrt{d_\Sigma(\mathbf{x}, \mathbf{y})}$

The second rationale behind our approach is that we can efficiently compute the first two steps as long as we restrict the input to discrete values. More specifically, if $x_1, x_2, y_1, y_2 \in \{0, 1\}$ we know that $d_1(\mathbf{x}, \mathbf{y})$ is 1 if $x_1 \neq y_1$ and 0 otherwise. We can alternatively formulate this as

$$d_1(\mathbf{x}, \mathbf{y}) = x_1 \oplus y_1 \tag{3}$$

where $\oplus$ is the binary *XOR* operator, which returns a 1 if its inputs are different, and a 0 otherwise. Because we do the exact same operations to compute $d_2(\mathbf{x}, \mathbf{y})$, we can compute both of them at the same time. To this end; we

Table 1: Conversion of real-valued points into a single integer representative.

x_j	.45	3.97	3.40	.86	2.60	.95	.17	4.47
$f_j(x_j)$	0	1	1	0	1	0	0	1
$x_{\mathbb{B}}$	$01101001_2 = 53$							

combine both x_1, x_2 into a single integer value $x_{\mathbb{B}} = 2 \cdot x_1 + x_2$ and similarly $y_{\mathbb{B}} = 2 \cdot y_1 + y_2$, the values of $x_{\mathbb{B}}, y_{\mathbb{B}}$ are one of $0, 1, 2$ and 3. The $\oplus$ operator can be extended to the integer domain, where it operates on each bit individually, a task efficiently executed by CPUs. For example we have that $3 \oplus 2 = 1$ as $11_2 \oplus 10_2 = 01_2$. This allows us to compute both of d_1 and d_2 with a single $\oplus$ operation. For the two-dimensional example this may seem a bit over the top, but since most modern CPUs allow 64-bit words, we can do the same operation in the same time if we have 64 dimensions, or any number of dimensions in between.

The second step of the Euclidean distance metric is summing all these values. After our $\oplus$ operation, the resulting integer is a sequence of bits, where high (1) bits represent dimensions along which the difference between two points is 1 and a low bit (0) otherwise. As such, it is a matter of summing these bit values; or alternatively, counting the number of high bits. The latter operation is known as the efficiently computable *Hamming weight* [11] of an integer. We first formalize the above in the next section, and then show how we increase the number of values per dimension from 2 to 4 in Section 3.2 and higher in Section 3.3. After that we formalize the conversion of the real-valued, n-dimensional points to the discrete values we perform our operations on in Section 3.4. We conclude with a discussion of BitBooster and how the same techniques also work with the Manhattan distance in Section 3.5.

3.1 Formalization for 1 bit

In this section we formalize BitBooster based on the example of the previous section. We do so in two steps.

Conversion to integer representatives The starting point is converting the real-valued vectors to integer representations. This is done in two steps: converting the value in each dimension to 0 or 1, and then converting the resulting values to a single integer. These steps are schematically presented in Table 1.

Let F be a real-valued, $|F|$-dimensional space, i.e. $F_j \subseteq \mathbb{R}$ for $j = 1 \dots |F|$. We define a function $f_j : F_j \to \{0, 1\}$. We discuss possible realizations of f_j in Section 3.4, but for now we just require that it exists. Starting from point $x = (x_1, x_2, \dots x_{|F|}) \in F$ we apply f_j to each of the values x_j and convert this to an integer $x_{\mathbb{B}} \in \mathbb{N}$. Formally:

$$x_{\mathbb{B}} = \sum_{j=1}^{|F|} 2^{|F|-j} \cdot f_j(x_j) \tag{4}$$

Table 2: Values of $(x-y)^2$, for $0 \leq x, y \leq 3$, and their binary representations.

	$0 = 00_2$	$1 = 01_2$	$2 = 10_2$	$3 = 11_2$
$0 = 00_2$	$0 = 0000_2$	$1 = 0001_2$	$4 = 0100_2$	$9 = 1001_2$
$1 = 01_2$	$1 = 0001_2$	$0 = 0000_2$	$1 = 0001_2$	$4 = 0100_2$
$2 = 10_2$	$4 = 0100_2$	$1 = 0001_2$	$0 = 0000_2$	$1 = 0001_2$
$3 = 11_2$	$9 = 1001_2$	$4 = 0100_2$	$1 = 0001_2$	$0 = 0000_2$

Put differently, the value of the first dimension, x_1, is be represented by the most significant bit in $x_{\mathbb{B}}$, while the value of the last dimension, $x_{|F|}$, is be represented by the least significant bit in $x_{\mathbb{B}}$.

Estimating distance The next step is to estimate the distance between two points $\mathbf{x}$ and $\mathbf{y}$ using their integer representatives $x_{\mathbb{B}}$ and $y_{\mathbb{B}}$. The steps taken here were already explained above:

$$d(\mathbf{x}, \mathbf{y}) \approx d_{\mathbb{B}}(x_{\mathbb{B}}, y_{\mathbb{B}}) = \sqrt{\mathcal{H}\left(x_{\mathbb{B}} \oplus y_{\mathbb{B}}\right)} \tag{5}$$

$d_{\mathbb{B}} : \mathbb{N} \times \mathbb{N} \to \mathbb{R}_{\geq 0}$ is the Euclidean distance metric in our integer representative domain. It is an approximation for $d(\mathbf{x}, \mathbf{y}) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_{\geq 0}$, the Euclidean distance metric in F. $\mathcal{H}$ denotes the Hamming weight [11], which counts the number of 'high' bits in an integer. The use of $\oplus$ comes from the fact that the per-dimension squared distance is the same as the truth table of $\oplus$.

3.2 Formalization for 2 bits

In this section we discuss how the methods of the previous section can be extended to more discrete values per dimension. BitBooster has a trade-off between accuracy and computational complexity, with more values per dimension being more precise but slower. By the design of our method, local optima in this trade-off exist when the number of discrete values per dimension is a power of 2, or in other words, makes full use of the expressiveness of a specific number of bits. We discuss the extension for 2 bits per dimension in detail; a higher number of bits works in a similar way, we formalize this in Section 3.3.

To explain the concept we first look at the logic of a single dimension. Compared to the single bit scenario, we have two main differences. First, the input (i.e. the discrete values of the dimension) now contains two bits, let these be $x[1]$ and $x[0]$ for $x \in \{0, 1, 2, 3\}$ where $x[1]$ is the most significant bit; we have for example $x = 2 \iff x[1] = 1 \wedge x[0] = 0$. Second, the output of the first step (i.e. the squared difference between the discrete dimension values) can now take several values: $0, 1, 4, 9$. More specifically, we need up to 4 bits to represent this output as $9 = 1001_2$. As such, instead of doing a single operation on two input bits, we need to do four operations on two times two input bits. In other words we have four different truth tables, one for each output bit (opposed to only $\oplus$ for 1 bit). Let $z = (x-y)^2$. Table 2 shows all truth tables together; each position in the bit representation of each cell corresponds to a single truth table.

Table 3: Conversion of real-valued point into two integer representatives.

x_j	.45	3.97	3.40	.86	2.60	.95	.17	4.47
$f_j^2(x_j)$	0	3	2	1	2	1	0	3
	00_2	11_2	10_2	01_2	10_2	01_2	00_2	11_2
$\mathbf{x_\mathbb{B}^2}$	$x_\mathbb{B}^{2,1} = 01101001_2 = 105,$				$x_\mathbb{B}^{2,0} = 01010101_2 = 85$			

From this table we can derive the following expressions:

$$\begin{aligned} z[0] &= x[0] \oplus y[0] \\ z[1] &= 0 \\ z[2] &= (x[1] = y[1]) \wedge (x[0] \neq y[0]) \\ &= \overline{(x[1] \oplus y[1])} \wedge (x[0] \oplus y[0]) \\ z[3] &= (x[1] \neq y[1]) \wedge (x[0] \neq y[0]) \wedge (x[0] = x[1]) \\ &= (x[1] \oplus y[1]) \wedge (x[0] \oplus y[0]) \wedge \overline{(x[0] \oplus x[1])} \end{aligned} \tag{6}$$

The second difference with 1 bit is that the conversion from F now maps each point $\mathbf{x}$ to two integers instead of one, each representing a significant bit for every dimension. First, we change our definition of f_j to $f_j^2 : F_j \to \{0, 1, 2, 3\}$, the superscript index indicating that we have a two-bit co-domain. We then combine each bit position of all dimensions:

$$x_\mathbb{B}^{2,i} = \sum_{j=1}^{|F|} 2^{|F|-j} \cdot f_j^2(x_j)[i], \quad i = 0, 1 \tag{7}$$

We now have two integers to represent $\mathbf{x}$. We further define $\mathbf{x_\mathbb{B}^2} = (x_\mathbb{B}^{2,1}, x_\mathbb{B}^{2,0})$. The above is schematically presented in Table 3.

Let $d_\mathbb{B}^{2,i}(\mathbf{x_\mathbb{B}^2}, \mathbf{y_\mathbb{B}^2})$ for $i = 0 \ldots 3$ be the functions that compute each index of Equation 6, for example $d_\mathbb{B}^{2,0}(\mathbf{x_\mathbb{B}^2}, \mathbf{y_\mathbb{B}^2}) = x_\mathbb{B}^{2,0} \oplus y_\mathbb{B}^{2,0}$. Similar to before, we apply the Hamming weight to the result; but we also need to combine the values of $d_\mathbb{B}^{2,i}$. Remember that each bit in $d_\mathbb{B}^{2,3}$ represents the most significant bit of the squared difference of a dimension. As such, the value of this bit needs to be multiplied by $2^3 = 8$. Similarly, we need to multiply $d_\mathbb{B}^{2,2}$ by $2^2 = 4$ and $d_\mathbb{B}^{2,1}$ by $2^1 = 2$. As such we have:

$$d_\mathbb{B}^2(\mathbf{x_\mathbb{B}^2}, \mathbf{y_\mathbb{B}^2}) = \sqrt{\sum_{i=0}^{3} 2^i \cdot \mathcal{H}\left(d_\mathbb{B}^{2,i}(\mathbf{x_\mathbb{B}^2}, \mathbf{y_\mathbb{B}^2})\right)} \tag{8}$$

3.3 Generalization to more than 2 bits

BitBooster can be further generalized to using more than 2 bits. More specifically, let n be the number of bits used to encode each dimension, such that we have 2^n values per dimension. We take two steps: preprocessing, formalized by Algorithm 1, and the metric computation, formalized by Algorithm 2.

Algorithm 1: Conversion of an $|F|$-dimensional point $\mathbf{x}$ to its integer representation $\mathbf{x}_{\mathbb{B}}^{n,i}$ given discretization functions f_j^n

input : Point $\mathbf{x} \in F$, and functions $f_j^n : F_j \to \{0, 1, \dots 2^n - 1\}$ for $F_j \in F$
output: n integer representations: $\mathbf{x}_{\mathbb{B}}^{n,i} = (x_{\mathbb{B}}^{n-1}, x_{\mathbb{B}}^{n-2} \dots x_{\mathbb{B}}^{0})$

1 **for** $j = 1 \dots |F|$ **do**
2 | Compute $f_j^n(x_j)$
3 **for** $i = 0 \dots n-1$ **do**
4 | $x_{\mathbb{B}}^{n,i} \leftarrow 0$
5 | **for** $j = 1 \dots |F|$ **do**
6 | | $x_{\mathbb{B}}^{n,i} \leftarrow x_{\mathbb{B}}^{n,i} + 2^{|F|-j} \cdot f_j^n(x_j)[i]$
7 **return** $\mathbf{x}_{\mathbb{B}}^n = (x_{\mathbb{B}}^{n,n-1}, x_{\mathbb{B}}^{n,n-2} \dots x_{\mathbb{B}}^{n,0})$

The discretization functions f_j^n produce 2^n different values. To convert $\mathbf{x} \in D$ to $x_{\mathbb{B}}^n$, we apply Algorithm 1. This algorithm consists of the discretization first in Lines 1 and 2. The resulting values are combined for every significant bit i in Lines 3 through 6. Lines 4 through 6 are the generalization of Equation 7. A time complexity analysis of Algorithm 1 is difficult, as we use both floating point operations and binary operations. We can say that Lines 1-2 run in $O(|F|)$ flops. Some operations in Line 6 can be done bitwise: the multiplication with $2^{|F|-j}$ can be substituted with a left-shift and the retrieval of index i of $f_j^n(x_j)$ with a $\oplus$, and the addition is limited to the integer domain. Lines 3-6 as such require $O(|F| \cdot n)$ operations, though the individual operations are cheaper than those of Lines 1-2. As such the total duration is dominated by the $O(|F|)$ flops, but small differences may arise for different values of n when $|F|$ is constant.

Algorithm 2: Computation of the approximation of $d(\mathbf{x}, \mathbf{y})$ using their integer representatives as converted by Algorithm 1, and distance functions $d_{\mathbb{B}}^{n,i}$

input : Binary representatives $\mathbf{x}_{\mathbb{B}}^n$ and $\mathbf{y}_{\mathbb{B}}^n$
output: $d_{\mathbb{B}}^n(\mathbf{x}_{\mathbb{B}}^n, \mathbf{y}_{\mathbb{B}}^n)$, an approximation of $d(\mathbf{x}, \mathbf{y})$.

1 $d_{\mathbb{B}}^n \leftarrow 0$
2 **for** $i = 0 \dots m-1$ **do**
3 | Compute $d_{\mathbb{B}}^{n,i}$
4 | $d_{\mathbb{B}}^n \leftarrow d_{\mathbb{B}}^n + 2^i \cdot \mathcal{H}\left(d_{\mathbb{B}}^{n,i}\right)$
5 **return** $\sqrt{d_{\mathbb{B}}^n}$

To compute the distance approximation, we apply Algorithm 2. The squared difference of two numbers from $\{0, 1, \dots 2^n - 1\}$ is at most $(2^n - 1)^2$. The total number of bits needed to express this is $m := \lfloor \log_2((2^n - 1)^2) \rfloor + 1$. This means we require m distance functions $d_{\mathbb{B}}^{n,0}, d_{\mathbb{B}}^{n,1} \dots d_{\mathbb{B}}^{n,m-1}$. For each of these functions

we compute the value in Line 3. We then take the Hamming weight of the result and multiply it by 2^i, for the same reason we multiplied $d_{\mathbb{B}}^{2,3}$ by $2^3 = 8$ in Section 3.3. Lines 1 through 5 are the generalization of Equation 8. The time complexity of Algorithm 2 is dependent on the implementations of $d_{\mathbb{B}}^{n,i}$. In the very least we require a total of $O(m)$ bit operations over all iterations of Line 4 and $O(1)$ flops in total for Line 5. The difficulty now is to get efficient implementations for $d_{\mathbb{B}}^{n,i}$, which influences the remainder of the time complexity, added by Line 3. For $n = 1$ we only require the $\oplus$ operation. For $n = 2$ we already need 11 operations (Equation 6), and as we see in Appendix 6, we require 65 operations for $n = 3$.

3.4 Discretization

In the previous sections we have not made any assumptions about f_j^n. In principle, any non-decreasing function would work. The implementation influences the accuracy of the approximation and the application where it is used. For the scope of this paper, where we are applying BitBooster to clustering algorithms, we partition F_j in 2^n evenly sized ranges and assign higher values of f_j^n to higher ranges. Formally, we have:

$$f_j^n(x_j) = \left\lceil \frac{x_j - min(F_j)}{max(F_j) - min(F_j)} \cdot 2^n \right\rceil - 1 \tag{9}$$

and, $f_j^n(min(F_j)) := 0$. An advantage is that this choice effectively scales each dimension between 0 and $2^n - 1$; comparable to normalizing each dimension individually. Alternative implementations of f_j^n are beyond the scope of this paper.

3.5 Discussion

Based on the above there are some limitations one needs to consider. While theoretically valid for any number of dimensions, the practical implementation limits this approach to a number of dimensions that is at most equal to the word-size of the hardware used to compute the distance. In practice, this effectively limits the number of dimensions to 64 on modern CPUs. Extending this to more than 64 dimensions would then require an extension using more than one 64-bit integer to represent $x_{\mathbb{B}}^{n,i}$. Such an extension would scale the time complexities of Lines 2-4 of Algorithm 2 by $O(\lceil |F|/64 \rceil)$, on the account of adding an addition loop for each used word for each of the n bits. This extension is beyond the scope of the current paper.

The presented work effectively *discretizes* the data, and as such may lead to a relatively large distance when two points are close, but on the other side of, a split in a given F_j. Similarly, points that are on opposite ends of two adjacent ranges, and hence far away in F_j, are approximated with the same distance. This is visually depicted in Figure 2. As we show in Section 4.1, this error is mitigated as $|F|$ gets larger, as the net effect is averaged out over more

Figure 2: Approximation errors caused by discretization. $d(a, b)$ and $d(c, d)$ are approximated by the same value.

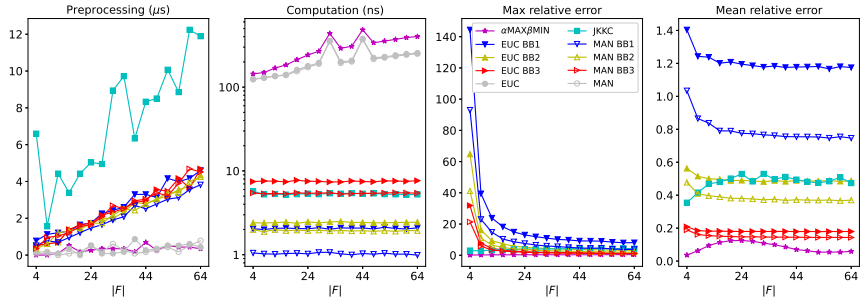

Figure 3: Results for the pure distance metric computations. Preprocessing is per point, computation is per pairwise computation. The error graphs are relative to the respective base metric, i.e. `EBB1`, `EBB2`, `EBB3`, `JKKC` and α`Max+`β`Min` to Euclidean, and `MBB1`, `MBB2`, `MBB3` to Manhattan.

dimensions. This error is also reduced if n is increased; as the individual ranges are smaller.

The work presented above easily extendeds to the *Manhattan* distance, as well as any other distance metric for which the functions $d_{\mathbb{B}}^{n,i}$ in Algorithm 2 can be defined. For two points $\mathbf{x} = (x_1, x_2)$ and $\mathbf{y} = (y_1, y_2)$, the Manhattan distance is defined as:

$$d(\mathbf{x}, \mathbf{y}) = |x_1 - y_1| + |x_2 - y_2| \tag{10}$$

We apply the same reasoning as we we did for the Euclidean distance metric above. In fact, Algorithm 1 does not depend on the metric, and for Algorithm 2, we only need to alter the definitions of $d_{\mathbb{B}}^{m,i}$ of Line 3 and return $d_{\mathbb{B}}^{n}$ instead of $\sqrt{d_{\mathbb{B}}^{n}}$ on Line 5. The derivations of $d_{\mathbb{B}}^{n,i}$ for the Manhattan distance for $n = 1, 2, 3$ can be found in the open source implementation, as discussed in Appendix 6. The advantage of the Manhattan distance with BitBooster is that the expressions require fewer operations which means a further speed-up in computation time.

4 Experiments

In the experiments we use both synthetic and real-world datasets. For the synthetic data we randomly generate $|F|$-dimensional hyper-spheres; varying the number of spheres between 5 and 30 with increments of 5 and the number of

dimensions between 4 and 64 with increments of 4. Each generated datasets contains 10000 datapoints, and is generated 10 times with a different seed every time resulting in a total of 960 datasets. For the real-world experiments we use several datasets available from the UCI repository [16]. We selected several annotated datasets with up to 64 dimensions and at least 10000 datapoints. The characteristics of these datasets are detailed in Table 5 in Section 4.3 together with the results. To show the effectiveness of BitBooster, we compare it with several other approximations and the original metrics. In particular, we compare the Euclidean distance with three BitBooster implementations, `EBB1`, `EBB2`, `EBB3`, each with a different value of n, and with the work of [1], indicated by `JKKC`, and with the work of [9], indicated by α`Max+`β`Min`. We compare the Manhattan distance with `MBB1`, `MBB2`, `MBB3`, again indicating BitBooster versions with different values for n. All these implementations are done in Python and Numba [17]. The competitor approximations have been implemented to the best of our knowledge; partly due to the unavailability of open-source implementations, but also to ensure a fair comparison. Angular Quantization [2] is excluded; the added optimization the authors of that paper implement are too involved to implement within this study, and the open-source code is not available anymore. We use *implementation* to indicate any metric or any of its approximations. We use 'the Euclidean distance' and 'the Manhattan distance' to refer to non-approximated implementations.

We conduct a total of three experiments, all of which have been conducted on a Windows[1] machine with an Intel Core i7 CPU and 32 GB RAM.

The first experiment, Section 4.1, only compares the distances, outside of any clustering setting. To this end we use all synthetic datasets, by averaging over the different number of spheres and the first three seeds and comparing $|F|$ with the duration of preprocessing, the duration of the computation of the distance, and the mean and maximum relative error with the respective base metric (Euclidean or Manhattan). The preprocessing includes normalization for all but the BitBooster implementations, and the conversion for all BitBooster implementations and `JKKC` to their respective operating spaces. In the second experiment, Section 4.2, we cluster all synthetic datasets using Lloyd's algorithm for k-medoids [7], with k the number of generated hyper-spheres. In the third experiment, Section 4.3, we cluster the real-world datasets using DBSCAN as presented in [18], setting $minPts$ to 4 and estimating eps for each implementation-dataset combination using the method described by [4]. The second and third experiments also include a comparison with the Weighted Jaccard distance metric and its MinHash versions, as discussed in Section 2. The latter are denoted by `WJMHx`, where x is the number of samples used for the hashing.

For the clustering algorithms, we use two different algorithms for three reasons. The first is to show the applicability of BitBooster in more than one specific algorithm. The second is that some of the real-world datasets do not

[1] We also ran this experiment on a Linux machine with an Intel Xeon CPU, where we encountered an inexplicable issue where only `JKKC` was negatively affected; being several magnitudes slower than `EBB3` in both preprocessing and computation.

have convex-shaped labelled subsets, which causes Lloyd's algorithm to be less effective. Third, Lloyd's algorithm is more practical for the generated data, as its required parameter k is known, while *eps* in DBSCAN would need to be estimated for each generated dataset-implementation combination.

As metric for measuring the quality of the found clusters we use the *ARI* for the synthetic experiments and the *purity* [19] for the real experiments.

The use of purity for the real experiment comes from the use of DBSCAN. The algorithm may find a different number of clusters from the number of classes in the data; which gives the ARI a skewed representation. We do note that we are not comparing whether an approximation can properly cluster the data, but whether an approximation can do *as good as* a full computation, and whether one approximation is *better than* another.

The repository with the source code of all implementations, as well as the scripts used to conduct the experiments and more details on the datasets can be found at `https://github.com/AnonymousDataResearcher/ICDM2021.git`.

4.1 Distance computations

The results of the distance computations are shown in Figure 3. We discuss each of the four characteristics separately. The distances computed by BitBooster are scaled by $\frac{1}{2^n-1}$. This is due to the fact that the dimensions are scaled to $0 \dots 2^n - 1$ after applying f_j^n, whereas all dimensions in the other implementations are normalized to the range $[0, 1]$. We first discuss the results on the Euclidean metric and its approximations (the filled markers of Figure 3), and discuss the main difference with the Manhattan metric and its approximations afterwards.

Preprocessing We see that the preprocessing of each implementation increases with the number of dimensions; although the rate differs per implementation. For the Euclidean distance and α`Max+`β`Min` we need to normalize each dimension. The normalization is constant in the number of datapoints; so a linear increase may be expected along the number of dimensions. For `JKKC` and BitBooster, we require the same steps first, and then additional preprocessing operations afterwards. For `JKKC` these are computationally heavy and increase with $|F|$: computing the norms of each point, computing the reference vector (first principle component) and computing the angle between this reference vector and the vector for each point in D. For BitBooster we add two steps to the preprocessing, the discretization step, which involves multiplying the normalized values by 2^n and converting them to integers (Algorithm 1, Lines 1-2), and the binarization step, which combines all dimensions into n integers (Lines 3-6). As explained in Section 3.3, this step can be done mostly with binary and integer operations (instead of floating point operations); making the contribution to the total preprocessing relatively small in comparison to the other preprocessing steps. This is reflected in the Figure 3, where one can see a difference between `EBB1`, `EBB2` and `EBB3`, but the difference is small in comparison to the differences with the other implementations.

Computation time All BitBooster implementations, as well as `JKKC` have constant computation time in the number of dimensions. For BitBooster this

is explained before, and for JKKC this also makes sense: after preprocessing the data is no longer dependent on the number of dimensions. The Euclidean distance and αMax+βMin are linear in the number of dimensions. This is expected since each dimension adds a term to the computation. αMax+βMin is slightly slower for higher dimensions, this is due to it actually requiring more floating point operations per additional dimension (1 maximum, 1 minimum, two multiplications and an addition, compared to one multiplication, one subtraction, and one addition for the Euclidean distance). We further see that EBB1 is slightly faster than EBB2 and both are significantly faster than EBB3. This difference can be explained by the number of operations used by Lines 3 and 4 of Algorithm 2: EBB1 only uses the $\oplus$ operation and one Hamming weight call (Equation 5), EBB2 uses a total of 11 operations and 3 Hamming weight calls (Equation 6), while BB3 uses 65 operations and 5 Hamming weight calls (Appendix 6). The computation time of JKKC is between EBB2 and EBB3, on account of additional floating-point multiplications and a cosine.

Mean and maximum error For the relative errors we mainly see that for BitBooster these values decrease with the number of dimensions, which is especially true for the the maximum relative error for EBB1. This is in line with the mitigation effect that was discussed in Section 3.5. Similarly, we see that the errors decrease for more bits per dimension. This, together with the increase in computation time for more bits, is in line with the trade-off explained in Section 3.3. BitBooster with 2 bits is a clear winner over the other approximations: EBB2 is significantly more accurate than EBB1 at little increase in computational complexity and significantly faster than EBB3 at little increased mean relative error. EBB2 also performs significantly faster than JKKC with similar mean approximation error. The mean and maximum relative errors found for αMax+βMin are comparable to the results of its original paper [9]. For JKKC, no results on the error were reported in [1].

Manhattan BitBooster The discussions for each result regarding the Euclidean BitBooster variants also hold for the Manhattan BitBooster. We see similar trends in all four measurements. Between Manhattan and Euclidean BitBooster, the main difference is that the Manhattan variants are both faster and more accurate for the same n. The increase in speed comes from the fact that the Manhattan implementations require fewer operations (Section 3.5), the lower error comes from the fact the Manhattan metric is less penalizing to outliers than the Euclidean metric, which also affects the severity of the approximation error discussed in Section 3.5.

4.2 Synthetic datasets and Lloyd's Algorithm

Figure 4 presents the results from all synthetic experiments. These plots reflect the performance of each implementation in general, independent of the dataset characteristics. Each plot contains a single approximation, scaled to the Euclidean distance. The Euclidean distance is indicated by the black lines and every point in each figure represents one of the synthetic datasets, with its results in terms of ARI and clustering duration relative to the Euclidean

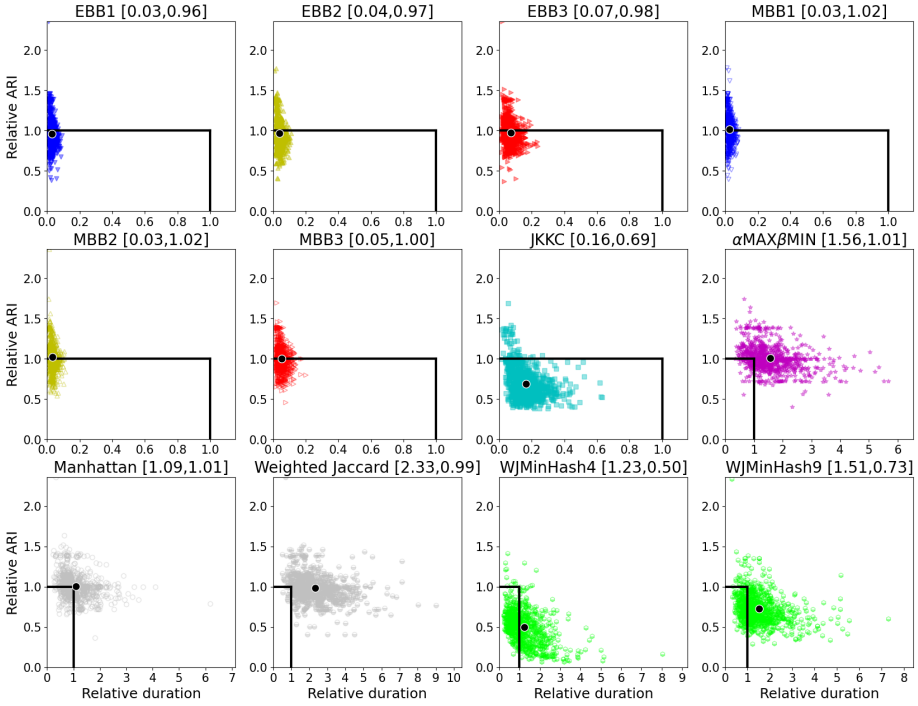

Figure 4: Results for the synthetic datasets, plotting the relative clustering duration against the relative purity. Each point represents one synthetic dataset. Reported points and values are relative to the `Euclidean` metric, represented by the black lines.

distance. As such, any point above the horizontal line has a higher ARI than the Euclidean distance, and any point to the left of the vertical line has a lower computation time than the Euclidean distance. In general, the closer a point to the top-left, the better. Each graph also lists the average performance; reporting the average relative computation time and average relative ARI. These values are also indicated by a black dot in each plot.

From these plots we see that, on average, all Euclidean BitBooster variants deliver nearly the same ARI as the Euclidean distance in only a fraction of the computation time. The effect of the trade-off is visible between `EBB1`, `EBB2` and `EBB3`: the clustering time and ARI increase with n. We further see that `JKKC` is dominated by all BitBooster variants. While this is in line with the results of Section 4.1 for `EBB1` and `EBB2`, it contradicts those results by seeming slower than `EBB3`. The cause of this lies not in the computation of the distance metrics, but in the execution of Lloyd's algorithm. The run-time of Lloyd's algorithm increases with the number of iterations needed to reach stability. As such, any approximation that needs more iterations to converge results in a longer clustering duration. This is in fact what happens for `JKKC`, as seen in Table 4. If we were to divide the clustering duration by the number of iterations, the results would be similar to Section 4.1. Finally, we see that α`Max+`β`Min` is purity-wise the same as the baseline but slower. This is in line with the results of the previous section. For the Manhattan distance and its variants, we see that it is performing even better than the Euclidean distance. This is mainly the result of the difference in the distance metric: the non-BitBooster Manhattan is also better than the non-BitBooster Euclidean. In line with the results from the previous section, the Manhattan BitBooster variants are slightly faster than the Euclidean BitBooster variants, and also the duration of the clustering increases with n. For $n = 3$ the Manhattan BitBooster, `MBB3`, the duration-ARI trade-off does not apply, as it is both slower and has a lower ARI than $n = 1$ and $n = 2$; though the difference is small. For Weighted Jaccard, we see a longer duration, at a similar ARI. The MinHash variant plots summarize a trend from increasing the number of samples from 1 to 9: more samples increases the quality

Table 4: The number of iterations used by Lloyd's algorithm to converge. Values are averaged all synthetic datasets.

Implementation	Iterations	Implementation	Iterations
`EBB1`	3.07	Euclidean	3.55
`EBB2`	3.66	`JKKC`	14.20
`EBB3`	3.60	α`MAX`β`MIN`	3.60
`MBB1`	3.18	Manhattan	3.56
`MBB2`	3.63	Weighted Jaccard	3.47
`MBB3`	3.46	`WJMH4`	1.51
		`WJMH9`	2.73

Table 5: Results on the real datasets. The approximation with the highest ratio between purity (top) and clustering duration (bottom) is highlighted in bold. Reported durations are in seconds, unless stated otherwise. Dataset properties are presented below the dataset name. EGS=Electrical Grid Stability, MGT=Magic Gamma Telescope, Loc. = Localization.

Dataset	Euclidean						Manhattan			
$\|D\|/\|F\|/k/s_E$	EBB1	EBB2	EBB3	Euclidean	JKKC	αMAXβMIN	MBB1	MBB2	MBB3	M
EGS	**.82**	.64	.64	.64	.65	.64	**.82**	.64	.64	
10K/11/2/.17	**.36**	.44	.97	15.16	.84	18.56	**.30**	.41	.81	
Pendigits	.10	.12	.17	.66	**.23**	.48	.10	**.11**	.11	
11K/16/10/.08	.70	.53	1.14	17.98	**.86**	23.17	.56	**.53**	.94	
EEG Eye State	.58	**.58**	.58	.55	.55	.55	.58	**.58**	.58	
15K/14/2/.01	10.31	**9.34**	10.25	42.20	10.67	52.89	9.84	**9.19**	9.89	
MGT	.77	.66	.70	.74	**.67**	.75	.77	.66	**.69**	
19K/10/2/.04	3.59	3.17	3.80	49.86	**2.45**	1m	3.42	3.05	**2.98**	
Polish Bankruptcy	.98	.98	.98	.98	**.98**	.98	.98	.98	**.98**	
20K/64/2/.01	19.30	14.80	6.02	2m	**5.25**	3m	18.44	14.52	**11.86**	
Letter Recognition	.04	**.09**	.06	.15	.05	.09	.04	**.05**	.06	
20K/16/26/.06	2.14	**1.38**	3.38	58.92	2.59	1m	1.77	**1.30**	2.59	
Avila	.41	.41	**.59**	.43	.41	.43	.41	**.46**	.42	
21K/10/12/.03	17.36	16.36	**4.39**	1m	7.66	1m	16.89	**6.88**	6.22	
Credit Card Default	.78	**.78**	.79	.78	.78	.78	.78	**.78**	.78	
30K/28/2/.05	7.75	**5.02**	8.12	3m	7.19	4m	6.94	**4.55**	6.58	
Loc. Chest	.45	.54	**.70**	.34	.34	.34	.45	.54	**.70**	
36K/3/11/.02	18.86	14.12	**13.03**	3m	8.56	3m	17.56	13.27	**10.42**	
Loc. Ankle R	.39	.40	.46	.34	**.34**	.34	.39	.40	**.46**	
43K/3/11/.01	33.41	31.75	21.08	4m	**14.38**	4m	31.78	30.19	**17.28**	
Loc. Belt	.46	.51	**.62**	.34	.33	.34	.46	.51	**.62**	
43K/3/11/.01	26.83	21.42	**19.91**	4m	11.52	5m	24.56	20.75	**16.53**	
Loc. Ankle L	.41	.42	.48	.34	**.33**	.34	.41	.42	**.48**	
44K/3/11/.01	30.81	28.84	21.06	4m	**12.78**	5m	29.89	27.62	**17.31**	

as well as the duration. As discussed in Section 2, MinHash is particularly effective for texts and images, but appears to lose its benefit in this range of dimensions, which results in a slower and worse clustering than any of the BitBooster implementations.

4.3 Real datasets and DBSCAN

Table 5 contains the results on the real datasets. We use *ratio* to refer to the quotient of purity and computation time. A higher ratio can be interpreted as a higher *purity per unit time*. For each dataset and metric, the approximation with the highest ratio is highlighted in bold. Each dataset further contains the value of $s_E := eps_{EUC}/\sqrt{|F|}$, where eps_{EUC} is the *eps* value used in the Euclidean distance as DBSCAN parameter. We use this number as representative of the *sparsity* of the dataset. This number comes from the estimation of the average Euclidean distance between two points in a hyper-cube, which scales with the square root of $|F|$ [20]. By dividing eps_{EUC} by $\sqrt{|F|}$, we compare the distance of the *eps* neighbourhood with the average expected distance between two uniformly sampled points. The lower this number, the smaller the distance to the 4-th nearest neighbour for core points and hence, the more the points are concentrated in a smaller part of the hyper-cube. Conversely, the higher this value, the more the points are spread out over the hyper-cube. We return to how this affects the approximations below.

We first compare the Euclidean approximations. The relatively long duration of the Euclidean distance and α`Max+`β`Min` and the relatively high purity for the (other) approximations, cause the Euclidean distance and α`Max+`β`Min` to never have the best ratio, comparable to Section 4.2. In fact both have a lower ratio than any BitBooster approximation or `JKKC`, on every dataset. This shows the value of approximating the distance metric (either by BitBooster or by `JKKC`). As such we only discuss Euclidean BitBooster and `JKKC`.

Comparing `EBB1` and `EBB2` on the one hand, and `EBB3` and `JKKC` on the other hand; we note that for lower sparsity values (at most 0.04), either `EBB3` or `JKKC` has the best ratio, whereas for higher sparsity either `EBB1` or `EBB2` wins. For low sparsity datasets, many points will be close to each other. As a result, discretizing the feature space in the way discussed in Section 3.4 may cause many of these points to have the same value of f_j^n for many j. As a result, the *eps* neighbourhood of a point increases, since many of the points will have a relatively small distance to their 4-th nearest neighbour. DBSCAN performs linear in the size of this *eps* neighbourhood, the total duration of the algorithm increases to the point where the added advantage of faster computations is irrelevant. This disadvantage for datasets with low sparsity does not exist for a higher number of bits per dimension. This is because the concentrated part of the dataset can still have different values of f_j^n as each dimension is divided over more values. The total number points that BitBooster can distinguish, $(2^{|F|})^n$, increases rapidly with a higher n. As such, one can make a more fine-tuned estimation of *eps*, reducing the size of the *eps* neighbourhood and decreasing the duration of DBSCAN. This allows `EBB3` to overcome the sparsity disadvantage.

The inverse happens with datasets of higher sparsity: `EBB1` and `EBB2` do not suffer from the low sparsity disadvantage. The total duration of DBSCAN is then mostly dependent on the speed of the computation of the distance metric, which benefits `EBB1` and `EBB2` over `EBB3` and `JKKC`. For $0.04 < s_E \leq 0.09$, `EBB2` has the best ratio, and for higher values of s_E, `EBB1` wins.

Comparing the different versions of BitBooster in terms of purity; we see that in almost all datasets, the purity increases with the number of bits, as result of the increase in precision of the distance metric. In general all versions of BitBooster perform similar and often better than `JKKC`.

For the Manhattan distance, no similar relation between the sparsity measure (adapted for Manhattan) and the difference between values of n was found. BitBooster variants always perform nearly as good or better than the Manhattan metric, at a significantly reduced cost. For the Weighted Jaccard metric; a comparison with MinHash is beyond the scope of this paper. We do see that for most datasets, all BitBooster methods reach at least the same and often a better purity, similar to the results from Section 4.2. Two notable exceptions are Pendigits and Letter Recognition. The features of Pendigits are are sampled positions of pen writing; i.e. a form of image data for which MinHashing is useful, as discussed in Section 2. Even though the Letter Recognition data is not graphical data, the features are directly extracted from images, which can be an explanation for MinHashing's effectiveness with respect to the Euclidean and Manhattan distances and their approximations.

Nonetheless, we can say that the purpose of BitBooster has been shown: we generally reach the same or better clustering performances, in only a fraction of the time.

5 Conclusion

In this paper we have presented BitBooster, an efficient approach to approximate the Euclidean and Manhattan distance metrics, which is particularly effective on high dimensional data. BitBooster converts real-valued multidimensional points to a discrete space, which allows for efficient binary operations. By increasing the number of bits used in the discretization of a metric space, BitBooster can provide more accurate estimations at a higher computational complexity, while still being competitive to alternative approximations and significantly faster than the exact computation. This is empirically evaluated in extensive experimental evaluations. We have further demonstrated the effectiveness of BitBooster in clustering algorithms, where it can drastically speed up the computation time compared to the exact Euclidean and Manhattan distance, with a small loss in cluster purity. BitBooster is also shown to be superior to alternative Euclidean distance approximations, both on synthetic and real-world datasets, and for different clustering algorithms.

It is further interesting to look beyond the application of clustering and investigate how effective BitBooster is for kNN classification tasks as well or how it may be combined with existing approaches such as Hashing. Additionally,

the principle of BitBooster, using discrete values to approximate distances can be extended to other distance metrics such as the Jaccard distance (quotient of the intersection and union set sizes) and higher L-norms (where Manhattan and Euclidean are the L_1 and L_2-norm, respectively).

Acknowledgements

6 Reproducibility

The implementation of BitBooster, and the scripts used to run the experiments are available at `https://github.com/AnonymousDataResearcher/ICDM2021.git` during the review period, and open-source afterwards. The repository provides additional details on the datasets used, where to download them, and how they are preprocessed. The repository further contains examples on how BitBooster can be used for clustering user-defined datasets outside of the experimental settings, and the derivation and implementation of Euclidean BitBooster for $n = 3$, and the Manhattan BitBooster variants.

References

[1] S. Jeong, S. W. Kim, K. Kim, and B. U. Choi, "An effective method for approximating the euclidean distance in high-dimensional space," in *Lecture Notes in Computer Science*, vol. 4080 LNCS, pp. 863–872, Springer Verlag, 2006.

[2] Y. Gong, S. Kumar, V. Verma, and S. Lazebnik, "Angular Quantization-based Binary Codes for Fast Similarity Search," in *Advances in Neural Information Processing Systems* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), vol. 25, pp. 1196–1204, Curran Associates, Inc., 2012.

[3] V. Mic, D. Novak, and P. Zezula, "Speeding up Similarity Search by Sketches," in *Similarity Search and Applications* (L. Amsaleg, M. E. Houle, and E. Schubert, eds.), vol. 9939 LNCS, (Cham), pp. 250–258, Springer International Publishing, 2016.

[4] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pp. 226–231, AAAI Press, 1996.

[5] S. P. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[6] T. Bottesch, T. Bühler, and M. Kächele, "Speeding up K-Means by Approximating Euclidean Distances via Block Vectors," in *Proceedings of the 33rd International Conference on ICML - Volume 48*, vol. 6 of *ICML'16*, pp. 2578–2586, JMLR.org, 2016.

[7] H. S. Park and C. H. Jun, "A simple and fast algorithm for K-medoids clustering," *Expert Systems with Applications*, vol. 36, pp. 3336–3341, mar 2009.

[8] E. Schubert and P. J. Rousseeuw, "Faster k-Medoids Clustering: Improving the PAM, CLARA, and CLARANS Algorithms," in *Lecture Notes in Computer Science*, vol. 11807 LNCS, pp. 171–187, Springer, oct 2019.

[9] G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, A. Nannarelli, M. Re, and S. Spano, "N-Dimensional Approximation of Euclidean Distance," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 3, p. 565, 2020.

[10] W. A. Kosmala, "Dot Product and Cross Product," in *A friendly introduction to Analysis*, ch. 9.3, p. 390, Pearson Education (US), 2004.

[11] H. Warren, "Counting 1-Bits," in *Hacker's Delight*, ch. 5.1, Addison-Wesley Professional, 1st ed., 2002.

[12] A. Z. Broder, "On the Resemblance and Containment of Documents," *Compression and complexity of sequences (sequences'97)*, pp. 21—-29, 1997.

[13] Y. H. Tsai and M. H. Yang, "Locality preserving hashing," in *2014 IEEE International Conference on Image Processing, ICIP 2014*, pp. 2988–2992, jan 2014.

[14] S. Ioffe, "Improved Consistent Sampling, Weighted Minhash and L1 Sketching," *2010 IEEE International Conference on Data Mining*, pp. 246–255, 2010.

[15] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," in *16th International World Wide Web Conference, WWW2007*, pp. 141–150, 2007.

[16] D. Dua and C. Graff, "UCI Machine Learning Repository," 2017.

[17] S. K. Lam, A. Pitrou, and S. Seibert, "Numba," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*, LLVM '15, (New York, New York, USA), pp. 1–6, ACM Press, 2015.

[18] E. Schubert, J. Sander, M. Ester, H.-P. Kriegel, and X. Xu, "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN," *ACM Transactions on Database Systems*, vol. 42, no. 3, 2017.

[19] C. D. Manning, P. Raghavan, and H. Schütze, "Evaluation of clustering," in *An Introduction to Information Retrieval*, ch. 16.3, pp. 356–360, Cambridge, England: Cambridge University Press, online ed., 2009.

[20] R. S. Anderssen, R. P. Brent, D. J. Daley, and P. A. P. Moran, "Concerning $\int_0^1 \cdots \int_0^1 (x_1^2 + \cdots + x_k^2)^{1/2} dx_1 \cdots dx_k$ and a Taylor Series Method," *SIAM Journal on Applied Mathematics*, vol. 30, no. 1, pp. 22–30, 1976.