EKM: An Exact, Polynomial-Time Divide-and-Conquer Algorithm for the K-Medoids Problem

Anonymous Authors[1]

Abstract

The K-medoids problem is a challenging combinatorial clustering task, widely used in data analysis applications. While numerous algorithms have been proposed to solve this problem, none of these are able to obtain an exact (globally optimal) solution for the problem in polynomial time. In this paper, we present EKM: a novel algorithm for solving this problem exactly with worst-case $O\left(N^{K+1}\right)$ time complexity. The algorithm is provably correct by construction, obtained using formal program derivation steps. We demonstrate the effectiveness of our algorithm by comparing it against various approximate methods and a state-of-the-art, exact branch-and-bound (BnB) algorithm on numerous real-world datasets. Our algorithm can not only provide provably exact solutions, but also consume much less time over all datasets compared with the BnB algorithm. We also show that the wall-clock time of our algorithm aligns with its worst-case time complexity analysis on synthetic datasets. In contrast, a state-of-the-art BnB algorithm not only exhibits exponential time complexity even for fixed K but also frequently produces erroneous solutions. This highlights the importance of employing formal, correct derivation steps when constructing exact algorithms.

1. Introduction

In machine learning (ML), K-medoids is the problem of partitioning a given dataset into K clusters where each cluster is represented by one of its data points, known as a *medoid*. A plethora of approximate/heuristic methodologies have been proposed to solve this problem, such as PAM (Partitioning Around Medoids) Kaufman (1990), CLARANS (Clustering Large Applications based on randomized Search) (Ng & Han, 2002), and variants of PAM (Schubert & Rousseeuw, 2021; Van der Laan et al., 2003).

While approximate algorithms are computationally efficient and have been scaled up to large dataset sizes, because they are approximate they provide no guarantee that the *exact* (*globally optimal*) *solution* can be obtained. In *high-stakes* or *safety-critical* applications, where errors are unacceptable or carry significant costs, we want the best possible partition given the specification of the clustering problem. Only an exact algorithm can provide this guarantee.

However, there are relatively few studies on exact algorithms for the K-medoids problem. The use of *branch-and-bound* (BnB) methods predominates research on this problem (Ren et al., 2022; Elloumi, 2010; Christofides & Beasley, 1982; Ceselli & Righini, 2005). An alternative approach is to use off-the-shelf *mixed-integer programming solvers* (MIP) such as Gurobi (Gurobi Optimization, 2021) or GLPK (GNU Linear Programming Kit) (Makhorin, 2008). These solvers have made significant achievements, for instance, Elloumi (2010); Ceselli & Righini (2005)'s BnB algorithm is capable of processing medium-scale datasets with a very large number of medoids. More recently, Ren et al. (2022) designed another BnB algorithm capable of delivering tight approximate solutions—with an optimal gap of less than 0.1%—on very large-scale datasets, comprising over one million data points with three medoids, although this required a massively parallel computation over 6,000 CPU cores.

These existing studies on the exact K-medoids problem have three defects. Firstly, most previous studies on exact algorithms impose a computation time limit, thus exact solutions are rarely actually calculated; the only rigorously exact cases are those computed by (Christofides & Beasley, 1982; Ceselli & Righini, 2005) for small datasets with a maximum size of 150 data items. Secondly, worst-case time and space complexity analyses are not reported in studies of BnB-based algorithms. Such BnB methods have exponential worst-case time/space complexity, yet this critical aspect is rarely discussed or analyzed rigorously. As a result, existing studies on the exact algorithms for the K-medoids problem either omit time complexity analysis (Elloumi, 2010;

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

Christofides & Beasley, 1982; Ceselli & Righini, 2005) or overlook essential details (Ren et al., 2022) such as algorithm complexity with respect to cluster size and the impact on performance of upper bound tightness. This omission undermines the reproducibility of findings and hinders advancement of knowledge in this domain. Thirdly, proofs of correctness of these algorithms is often omitted. Exact solutions require such mathematical proof of global optimality, yet many BnB algorithm studies rely on weak assertions or informal explanations that do not hold up under close scrutiny (Fokkinga, 1991).

In this report, we take a fundamentally different approach, by combining several modern, broadly applicable algorithm design principles from the theories of constructive algorithmics (Meertens, 1986; Jeuring & Pekela, 1993; Bird & de Moor, 1996). The derivation of our algorithm is obtained through a rigorous, structured approach known as the *Bird-Meertens formalism* (or the *algebra of programming*) (Meertens, 1986; Jeuring & Pekela, 1993; Bird & de Moor, 1996). This formalism enables the development of an efficient and correct algorithm by starting with an obviously correct but possibly inefficient algorithm and deriving an equivalent, efficient implementation by way of *equational reasoning* steps. Thus, the correctness of the algorithm is assured, skipping the need for error-prone, post-hoc induction proofs. Our proposed algorithm for the K-medoids problem has worst-case $O\left(N^{K+1}\right)$ time complexity and furthermore is amenable to optimally efficient vectorization and parallelization.

The paper is organized as follows. In Section 2, we explain in detail how our efficient EKM algorithm is derived through provably correct equational reasoning steps. Section 3 presents the results of empirical computational comparisons between our proposed algorithms, approximate algorithms, and the state-of-the-art exact algorithm developed by Ren et al. (2022). These comparisons were conducted on datasets from the UCI machine learning repository, and a few datasets from Ren et al. (2022). Furthermore, we provide a detailed time-complexity analysis comparing Ren et al. (2022)'s algorithm with our algorithm. The results demonstrate that, even for fix K, Ren et al. (2022)'s algorithm exhibits *exponential time complexity*. In contrast, our proposed algorithms consistently achieve predictable polynomial time complexity. In addition, the naive brute-force algorithm for solving this problem also has $O\left(N^{K+1}\right)$complexity. To highlight the difference between our algorithm and a brute-force algorithm, we also compared the performance of our algorithm in exhaustive generation and solving the K-medoids problem by comparing against the classical one-by-one enumeration strategy by using `itertools` library. Section 4 summarizes the contributions of this study, reviews related work. Finally, Section 5 suggests future research directions.

2. Theory

Our novel EKM algorithm for K-medoids is derived using rational algorithm design steps, described in the following subsections. There are two main advantages to this approach of deriving correct (globally optimal) algorithms from specifications. First, since the exhaustive search algorithm is correct for the MIP problem, any algorithm derived through correct equational reasoning steps from this specification is also (provably) correct. Second, the proof is short and elegant: a few generic theorems for shortcut fusion already exist and when the conditions of these shortcut fusion theorems hold, it is simple to apply them to a specific problem such as the K-medoids problem here. This provably correct design step is vitally important in the study of the exact algorithm, as we will see shortly in the experiments, where the state-of-art BnB algorithm produces solutions lower than exact solutions frequently. This highlights the importance of using a correct-by-construction algorithm derivation process rather than relying on ad-hoc BnB algorithms.

2.1. Specifying a mixed-integer program (MIP) for the K-medoids problem

We denote a data set $\mathcal{D}$ consists of N *data points* $\boldsymbol{x}_n = (x_{n1}, x_{n2}, \ldots, x_{nD}) \in \mathbb{R}^D, \forall n \in \{1, \ldots, N\} = \mathcal{N}$, where D is the dimension of the *feature space*. We assume the data items $\boldsymbol{x}_n \in \mathcal{D}$ are all stored in a ordered sequence (list) $\mathcal{D} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N]$. In clustering problem, we need to find a set of centroids $\mathcal{U} = \{\boldsymbol{\mu}_k : k \in \mathcal{K}\}$, where $\mathcal{K} = \{1, \ldots, K\}$ in $\mathbb{R}^D$, and each centroid $\boldsymbol{\mu}_k$ associated with a unique *cluster* C_k, which subsumes all data points $\boldsymbol{x}$ are the *closest* to this centroid than other centroids, this closeness is defined by a distance function $d(\boldsymbol{x}, \boldsymbol{\mu})$, there is no constrains on the distance function in the K-medoids problem, a common choice is the *squared Euclidean distance function* $d_2(\boldsymbol{x}, \boldsymbol{\mu}) = \|\boldsymbol{x} - \boldsymbol{y}\|_2^2$. For each set of centroids $\mathcal{U}$ we have a set of *disjoint clusters* $\mathcal{C} = \{C_1, C_2, \ldots C_K\}$, and set $\mathcal{K}$ is then called the *cluster labels.*

The K-medoids problem is usually specified as the following MIP

$$\begin{aligned} \mathcal{U}^* = &\underset{\mathcal{U}}{\operatorname{argmin}}\ E(\mathcal{U}) \\ &\text{subject to } \mathcal{U} \subseteq \mathcal{D}, |\mathcal{U}| = K, \end{aligned} \tag{1}$$

where $E(\mathcal{U}) = \sum_{k \in \mathcal{K}} \sum_{x_n \in C_k} d(\boldsymbol{x}_n, \boldsymbol{\mu}_k)$ is the *objective function* for the K-medoids problem, and $\mathcal{U}^*$ is a set of centroids that optimize the objective function $E(\mathcal{U})$. In our study, we make no assumptions about the objective function, allowing any arbitrary distance function to be applied without affecting the worst-case time complexity while still obtaining the exact solution.

2.2. Representing the solution as an (inefficient) exhaustive search algorithm

In the theory of transformational programming (Bird & de Moor, 1996; Jeuring & Pekela, 1993), combinatorial optimization problems such as (1) are solved using the following, generic, *generate-evaluate-select* algorithm,

$$s^* = sel_E\left(eval_E\left(gen\left(\mathcal{D}\right)\right)\right). \tag{2}$$

Here, the generator function $gen : \mathcal{D} \rightarrow \left[\left[\mathbb{R}^D\right]\right]$, enumerates *all* possible *combinatorial configurations* $s : \left[\mathbb{R}^D\right]$ (here, configurations consist of a list of data items representing the K medoids) within the *solution* (*search*) *space*, $\mathcal{S}$ (here, this is the set of all possible size K combinations). For most problems, gen is a recursive function so that the input of the generator can be replaced with the index set $\mathcal{N}$ and rewritten $gen\left(n\right), \forall n \in \mathcal{N}$. The *evaluator* $eval_E : \left[\left[\mathbb{R}^D\right]\right] \rightarrow \left[\left(\left[\mathbb{R}^D\right], \mathbb{R}\right)\right]$ computes the objective values $r = E\left(s\right)$ for all configurations s generated by $gen\left(n\right)$ and returns a list of *tupled configurations* (s, r). Lastly, the *selector* $sel_E : \left[\left(\left[\mathbb{R}^D\right], \mathbb{R}\right)\right] \rightarrow \left(\left[\mathbb{R}^D\right], \mathbb{R}\right)$ select the best configuration s^* with respect to objective E.

Algorithm (2) is an example of *exhaustive* or *brute-force search*: by generating all possible configurations in the search space $\mathcal{S}$ for (1), evaluating the corresponding objective E for each, and selecting an optimal configuration, it is clear that it must solve the problem (1) exactly. Taking a different perspective, (2) can be considered as a generic *program* for solving the MIP (1). However, program (2) is generally inefficient due to *combinatorial explosion*; the size of $gen\left(\mathcal{D}\right)$ is often exponential (or worse) in the size of $\mathcal{D}$.

To make this exhaustive solution practical, two aspects must be considered to improve the efficiency of the program. The first aspect is the design of an efficient generator. By recognizing that in many combinatorial problems, generators can be expressed as efficient recursions that take advantage of solving difficult problems by first addressing easier subproblems, Another principle is known as *shortcut fusion*. For many problems, it is often possible to fuse sel_E, $eval_E$, and gen into a single, fused, efficient program. This fusion can save substantial amounts of computation because it eliminates the need to generate and store every configuration. However, the fusion is only possible when the generator is defined as a recursive program, this immediately eliminates any one-by-one enumeration approach, as none of them are defined recursively. The remaining paper will be dedicated to explaining an efficient generator for solving the K-medoids problem and the fusion in this problem.

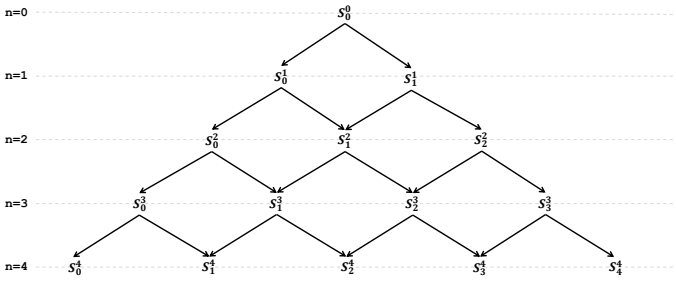

Figure 1. A generic recursive generator of all possible sublists for list $[\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4]$, in each recursive stage n, the sublists of size k (combinations) are automatically grouped into list $\mathcal{S}_k^n$, for all $k \in \{0, 1, \ldots n\}$.

2.3. Constructing an efficient, recursive combinatorial configuration generator

In K-medoids there are only $N \times (N-1) \times \cdots \times (N-K)$ ways of selecting centroids whose corresponding assignments are potentially distinct. Thus, the solution space $\mathcal{S}$ for the K-medoids problem consists of all size K *combinations* (K-*sublists*) of the data input $\mathcal{D}$, denoted as $s : \left[\mathbb{R}^D\right]$. If we can design an efficient, recursive combination generator $gen_{combs} : \mathcal{N} \times \mathcal{K} \rightarrow \left[\mathbb{R}^D\right]$ to enumerate all possible size K combinations, at least one set of such centroids corresponds to an optimal value of the clustering objective E. An efficient and structured combination generator which achieves this, will be described next.

Denoting $\mathcal{S}_k^n : \left[\mathbb{R}^D\right]$ as the list that stores all possible size k combinations from list $[\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots \boldsymbol{x}_n]$ of length n, and $\mathcal{S}^n = [\mathcal{S}_k^n \mid k \in \{0, \ldots n\}]$ as the list of all *sublists* of that list. The key to constructing an efficient combination generator can be reduced to solving the following problem: given all sublists $\mathcal{S}^n$ for list $\mathcal{D}_1 = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots \boldsymbol{x}_n]$, and $\mathcal{S}^m$ for list $\mathcal{D}_2 = [\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots \boldsymbol{y}_m]$, construct all sublists $\mathcal{S}^{n+m}$ for list $\mathcal{D}_1 \cup \mathcal{D}_2$. This problem captures the essence of Bellman's *principle of optimality* (Bellman, 1954) i.e. we can solve a problem by decomposing it into smaller *subproblems*, and the solutions for these subproblems are combined to solve the original, larger problem. In this context, constructing all combinations $\mathcal{S}^{n+m}$ for list $\mathcal{D}_1 \cup \mathcal{D}_2$ is the problem; all possible combinations $\mathcal{S}^n$, $\mathcal{S}^m$ for lists $\mathcal{D}_1$, $\mathcal{D}_2$ respectively, are the smaller subproblems.

In previous work of Little et al. (2024), it was demonstrated that the *constraint algebra* (typically a *group* or *monoid*) can be implicitly embedded within a generator semiring through the use of a *convolution algebra*. Subsequently, the filter that incorporates these constraints can be integrated into the generator via the *semiring fusion theorem* (for de-

tails of semiring lifting and semiring fusion, see Little et al. (2024)). In our case, we can substitute the convolution algebra used in Little et al. (2024) with the *generator semiring* $([[\mathbb{T}]], \cup, \circ, \emptyset, [[\,]])$ ($\mathbb{T}$ stands for type variable), then we have following equality

$$\mathcal{S}_k^{n+m} = \bigcup_{\substack{i+j=k \\ 0\leq i,j\leq k}} \left(\mathcal{S}_i^n \circ \mathcal{S}_j^m\right), \forall\, 0 \leq k \leq n+m \tag{3}$$

where $l_1 \circ l_2 = [s \cup s' \mid s \in l_1, s' \in l_2]$ is the *cross-join operator* on lists l_1 and l_2 obtained by concatenating each element s in configuration l_1 with each element s' in l_2. For instance, $[[1],[2]] \circ [[3],[4]] = [[1,3],[1,4],[2,3],[2,4]]$. Informally, (3) is true because the size k combinations for the list $\mathcal{D}_1 \cup \mathcal{D}_2$ should be constructed from all possible combinations $\mathcal{S}_i^n$ in $\mathcal{D}_1$ and $\mathcal{S}_j^m$ in $\mathcal{D}_2$ such that $i+j=k$ for all $0 \leq i,j \leq k$. In other words, all possible size k combinations should be constructed by joining all possible combinations with a size smaller than k.

Definition (3) is a special kind of *convolution product* for two lists $l_a = [a_0, a_2, \ldots, a_{n-1}]$ and $l_b = [b_0, b_2, \ldots, b_{m-1}]$,

$$conv\,(f, l_1, l_2, k) = [c_0, c_1, \ldots, c_k], \tag{4}$$

where c_k is defined as

$$c_k = \bigcup_{\substack{i+j=k \\ 0\leq i,j\leq k}} f\,(a_i, b_j), \quad 0 \leq k \leq n+m-1. \tag{5}$$

Given $\mathcal{S}^n$ and $\mathcal{S}^m$, it is not difficult to verify that $\mathcal{S}^{n+m} = conv\,(\circ, \mathcal{S}^n, \mathcal{S}^m, n+m-1)$, representing all sublists (combinations) for list $\mathcal{D}_1 \cup \mathcal{D}_2$. Furthermore, all combinations with size smaller than k can be obtained by calculating $conv\,(\circ, \mathcal{S}^n, \mathcal{S}^m, k)$, denoted as $\mathcal{S}_{\leq k}^{n+1}$. Using equation $\mathcal{S}_{\leq k}^{n+1} = conv\left(\circ, \mathcal{S}_i^n, \mathcal{S}_j'^m, k\right)$, a combination generator recursion $gen_{combs} : \mathcal{N} \times \mathcal{K} \to \left[\mathbb{R}^D\right]$ can be defined by following *pattern matching*

$$\begin{aligned}
&gen_{combs}\,([\,], k) = [[[\,]]] \\
&gen_{combs}\,([\boldsymbol{x}_n], k) = [[[\,]], [[\boldsymbol{x}_n]]] \\
&gen_{combs}\,(xs \cup ys, k) = \\
&\qquad conv\,(\circ, gen_{combs}\,(xs, k), gen_{combs}\,(ys, k), k).
\end{aligned} \tag{6}$$

where $\mathcal{D} = xs \cup ys$. The function $gen_{combs}\,(\mathcal{D}, k)$ generate all combinations with a size k or less. For instance, $gen_{combs}\,([\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3], 2) = \mathcal{S}_{\leq 2}^3\,([\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3]) = [[[]], [[\boldsymbol{x}_1]], [[\boldsymbol{x}_2]], [[\boldsymbol{x}_3]], [[\boldsymbol{x}_1, \boldsymbol{x}_2]], [[\boldsymbol{x}_1, \boldsymbol{x}_3]], [[\boldsymbol{x}_2, \boldsymbol{x}_3]]]$. See Figure 1 for an illustration of the process by which recursive combination generator (6) operates. The generator (6) is an instance of the *divide-and-conquer* (D&C) algorithm, for which the original problem ($xs \cup ys$) can be divided into arbitrary two disjoint subproblems xs and ys and the subproblems are solved independently.

One of the benefits of being a D&C algorithm is that all D&C algorithms are easy to parallelize. Moreover, our generator is specifically *designed for ease of vectorization and parallelization*, making it well-suited for modern hardware optimized for array-based operations. By deliberately organizing combinations of the same size into the same list, we enable storage in a pre-allocated array. Operations on these contiguous memories will decrease the possibility of cache misses, which makes our program more efficient. Alternative methods are typically list-based, such as the generator proposed by (He & Little, 2023) and the Gray code generator (Kreher & Stinson, 1999). These methods rely on dynamic memory allocation, making them significantly less efficient than our proposed generator. Alternatively, one-by-one generation approaches, such as lexicographical generation, are primarily designed for random generation and often involve substantial constant factors that are obscured within the big-O notation.

Moreover, the one-by-one enumeration strategy does not permit any form of fusion, rendering future speed-ups for this method unattainable. Consequently, both the best-case and worst-case complexities of the one-by-one enumeration strategy remain $O\left(N^{K+1}\right)$. In contrast, the recursive structure of our generator enables fusion. Previous research by He & Little (2023) demonstrated an order-of-magnitude speed-up using the simplest bounding techniques. While our focus is on illustrating the principles of program derivation, and we thus omit such optimizations, we show that even the plain version of our algorithm is significantly faster than the state-of-the-art BnB algorithm.

2.4. Applying shortcut fusion equational transformations to derive an efficient, correct implementation

The combination generator gen_{combs} constructed above, gives us an efficient recursive basis to use 2 to derive an efficient algorithm for solving the K-medoids problem. Thus, a provably correct algorithm for solving the K-medoids problem can be rendered as

$$s^* = sel_E\,(eval_E\,(gen_{combs}\,(xs, k))). \tag{7}$$

As we mentioned above, if both selector sel_E and evaluator $eval_E$ can be fused into the generator by identifying specific shortcut fusion theorems, then a significant amount of computational effort and memory can be saved.

For any configurations of size k, we can evaluate the objectives of the k-combinations immediately once its was generated. Since adding more medoids provably decreases

the objective value, we do not evaluate combinations of sizes smaller than k. However, this approach can be generalized to solve the k-medoids problem for all $k \leq K$. In other words, the following fusion condition holds:

$$sel_E\left(eval_E\left(gen_{combs}\left(xs,k\right)\right)\right) = sel_{E,k}\left(conv\left(\circ_{E,k}, gen_{combs}\left(xs\right), gen_{combs}\left(ys\right), k\right)\right), \tag{8}$$

where $sel_{E,k}$ select the configuration with lowest objective among all size k combinations, and $\circ_{E,k}$ is the cross-join operator described above augmented with evaluation updates,

$$l_1 \circ_{E,k} l_2 = \begin{cases} [(s_1 \cup s_2, E\left(s_1 \cup s_2\right)) \mid \\ \quad (s_1, r) \in l_1, (s_2, r) \in l_2] & \text{if } |s_1 \cup s_2| = k \\ [(s_1 \cup s_2, r) \mid \\ \quad (s_1, r) \in l_1, (s_2, r) \in l_2] & \text{otherwise,} \end{cases} \tag{9}$$

Due of the universal property (Bird & de Moor, 1996), the solution obtained by recursion

$$\begin{aligned} &EKM\left([\,], k\right) = [[[\,]]] \\ &EKM\left([\boldsymbol{x}_n], k\right) = [[[\,]\,], [[\boldsymbol{x}_n]]] \\ &EKM\left(xs \cup ys, k\right) = \\ &\qquad sel_{E,k}\left(conv\left(\circ_{E,k}, EKM\left(xs,k\right), EKM\left(ys,k\right), k\right)\right), \end{aligned} \tag{10}$$

is also a solution to the specification (7), without the proof of induction.

We now analyze the time and space complexity of the algorithm $EKM\left(\mathcal{D}, K\right)$ for a dataset of size N. Since the pairwise distances between data points can be stored in a pre-allocated matrix, accessing any element in this matrix requires only $O\left(1\right)$ time. Evaluating the objective of a K-combination requires indexing $K \times N$ entries. Thus evaluating each K-combination takes $O\left(N\right)$ time, and there are $\binom{N}{K} = O\left(N^K\right)$ combinations in total. Therefore, the overall running time of $EKM\left(\mathcal{D}, K\right)$ is $O\left(N^{K+1}\right)$. +Additionally, $EKM\left(\mathcal{D}, K\right)$ only stores partial configurations of sizes smaller than $K-1$ resulting in a space complexity of $O\left(N^{K-1}\right)$.

3. Experiments

In this section, we analyze the computational performance of our algorithm EKM on both synthetic and real-world data sets. Our evaluation aims to test the following predictions: (a) EKM always obtains the best objective value[1]; (b) wall-clock run-time matches the worst-case polynomial time complexity analysis; (c) the state-of-art BnB algorithm (Ren et al., 2022) for solving the K-medoids problem will have exponential time complexity even for fixed K in the worst-case. In our implementation, the matrix operations required at every recursive step are batch processed on a single GPU. We executed all the experiments on an Intel Core i9 CPU, with 24 cores, 2.4-6 GHz, 32 GB RAM and GeForce RTX 4060 Ti GPU.

[1]The squares Euclidean distance function was chosen for the experiments, any other proper metrics could also be used.

Table 1. Empirical comparison of the EKM algorithm, against widely-used approximate algorithms (PAM, Fast-PAM, and CLARANS) and the state-of-art exact BnB algorithm developed by Ren et al. (2022), for $K = 3$, in terms of sum-of-squared errors (E), smaller is better. For Ren et al. (2022)'s aglorithm, we include both the *upper bound* and the *optimal gap*. The best-performing algorithm is highlighted in **bold**, while incorrect solutions are marked in red. Although some of the upper bounds returned by BnB algorithm are exact, these values are not marked in bold, as the optimal gap has not yet converged to zero. Wall clock time in brackets (seconds), we set a time limit 1.08×10^4 for BnB algorithm.

UCI DATASET	N	D	EKM (OURS)	REN'S BNB	PAM	FASTER-PAM	CLARANS
LM	338	3	**3.96×10^1** (8.20×10^{-1})	1.21×10^1 (2.31×10^1) 0	3.99×10^1 (4.02×10^{-3})	4.07×10^1 (3.01×10^{-3})	5.33×10^1 (6.14)
UKM	403	5	**8.36×10^1** (1.37)	5.51×10^1 (1.62×10^3) $\leq 0.1\%$	8.44×10^1 (8.57×10^{-3})	8.40×10^1 (3.21×10^{-3})	1.16×10^2 (4.98×10^1)
LD	345	5	**3.31×10^5** (8.3×10^{-1})	1.21×10^1 (2.47×10^1) $\leq 0.1\%$	3.56×10^5 (4.11×10^{-3})	**3.31×10^5** (3.87×10^{-3})	4.68×10^5 (3.40)
ENERGY	768	8	**2.20×10^6** (1.37×10^1)	2.20×10^6 (1.68×10^1) $\leq 0.1\%$	2.28×10^6 (6.95×10^{-3})	2.28×10^6 (3.94×10^{-3})	2.97×10^6 (2.71)
VC	310	6	**3.13×10^5** (6.82×10^{-1})	1.50×10^5 (3.83×10^2) $\leq 0.1\%$	**3.13×10^5** (3.15×10^{-3})	3.58×10^5 (5.36×10^{-3})	5.27×10^5 (2.58)
WINE	178	13	**2.39×10^6** (2.22×10^{-1})	1.16×10^4 (5.17×10^4) $\leq 0.1\%$	**2.39×10^6** (1.06×10^{-3})	2.63×10^6 (2.34×10^{-3})	6.86×10^6 (5.56×10^{-1})
YEAST	1484	8	**8.37×10^1** (1.74×10^2)	6.57×10^1 (1.08×10^4) $\leq 39.19\%$	8.42×10^1 (9.54×10^{-2})	8.42×10^1 (6.08×10^{-2})	1.05×10^2 (1.73×10^2)
IC	3150	13	**6.9063×10^9** (4.53×10^3)	6.18×10^9 (6.37×10^3) 0	6.9105×10^9 (8.68×10^{-1})	**6.9063×10^9** (1.91×10^{-1})	1.44×10^{10} (2.70×10^1)
WDG	5000	21	**1.67×10^6** (5.23×10^4)	1.60×10^9 (1.08×10^4) $\leq 785.05\%$	**1.67×10^6** (1.34)	**1.67×10^6** (1.97×10^{-1})	2.77×10^6 (5.32×10^3)
IRIS	150	4	**8.40×10^1** (1.57×10^{-1})	8.46×10^1 (2.51×10^1) $\leq 27.1\%$	8.45×10^1 (2.51×10^{-3})	8.45×10^1 (1.03×10^{-3})	1.57×10^2 (2.32×10^{-1})
SEEDS	210	7	**5.98×10^2** (2.85×10^{-1})	5.98×10^2 (2.42×10^1) $\leq 0.1\%$	**5.98×10^2** (1.14×10^{-3})	**5.98×10^2** (3.59×10^{-3})	1.12×10^3 (7.82×10^{-1})
GLASS	214	9	**6.29×10^2** (2.90×10^{-1})	6.29×10^2 (3.13×10^1) $\leq 0.1\%$	**6.29×10^2** (1.01×10^{-3})	**6.29×10^2** (1.62×10^{-3})	1.04×10^3 (2.27)
BM	249	6	**8.63×10^5** (3.96×10^{-1})	8.63×10^5 (1.19×10^2) $\leq 0.1\%$	8.76×10^5 (4.12×10^{-3})	**8.63×10^5** (1.61×10^{-3})	1.33×10^6 (1.02×10^1)
HF	299	12	**7.83×10^{11}** (6.26×10^{-1})	**7.83×10^{11}** (5.20×10^1) 0%	**7.83×10^{11}** (1.00×10^{-3})	**7.83×10^{11}** (4.87×10^{-3})	1.88×10^{12} (6.55×10^{-1})
WHO	440	7	**8.33×10^{10}** (1.98)	8.33×10^{10} (3.71×10^2) $\leq 0.1\%$	**8.33×10^{10}** (5.62×10^{-3})	**8.33×10^{10}** (2.81×10^{-3})	1.21×10^{11} (8.12)
UK	258	5	**5.08×10^1** (3.78×10^{-1})	5.08×10^1 (1.43×10^3) $\leq 0.1\%$	**5.08×10^1** (2.47×10^{-3})	**5.08×10^1** (2.47×10^{-3})	6.89×10^1 (2.67×10^1)
HCV	572	12	**2.75×10^6** (5.48)	2.75×10^6 (8.59×10^1) $\leq 0.1\%$	**2.75×10^6** (5.79×10^{-3})	**2.75×10^6** (2.21×10^{-3})	4.75×10^6 (6.89×10^1)
ABS	740	19	**2.32×10^6** (1.17×10^1)	2.32×10^6 (6.23×10^2) $\leq 0.1\%$	**2.32×10^6** (2.11×10^{-2})	2.38×10^6 (5.00×10^{-3})	2.96×10^6 (7.80×10^1)
TR	980	10	**1.13×10^3** (2.53×10^1)	1.14×10^3 (1.08×10^4) $\leq 89\%$	**1.13×10^3** (5.14×10^{-2})	**1.13×10^3** (1.14×10^{-2})	1.38×10^3 (2.59×10^2)
SGC	1000	21	**1.28×10^9** (3.87×10^1)	1.28×10^9 (1.75×10^2) $\leq 0.1\%$	**1.28×10^9** (1.71×10^{-1})	**1.28×10^9** (4.22×10^{-2})	2.52×10^9 (2.24)
HEMI	1995	7	**9.91×10^6** (9.00×10^2)	9.91×10^6 (3.92×10^2) $\leq 0.1\%$	**9.91×10^6** (3.64×10^{-1})	**9.91×10^6** (6.99×10^{-2})	1.66×10^7 (9.53)
PR2392	2392	2	**2.13×10^{10}** (1.29×10^3)	2.13×10^{10} (1.54×10^3) $\leq 0.1\%$	**2.13×10^{10}** (3.66×10^{-1})	**2.13×10^{10}** (8.38×10^{-2})	3.47×10^{10} (1.15×10^2)

Remarks, all comparsion about the time complexity is executed in the sequential version of our algorithm over CPU only.

3.1. Performance on real-world datasets

We test the performance of our EKM algorithm against the approximate algorithms partition around medoids (PAM),

Faster-PAM and Clustering Large Applications based on RANdomized Search (CLARANS)[2] on 18 datasets from the UCI Machine Learning Repository, two datasets from Ren et al. (2022) (UK, HCV), and two open-source datasets (PR2392, HEMI) from (Wang et al., 2022; Padberg & Rinaldi, 1991; Ren et al., 2022). We show that, as expected, no other algorithms can achieve better objective function values (see Table 1), except in cases where Ren et al. (2022)'s BnB algorithm returned incorrect solutions, which were clearly invalid as they were several orders of magnitude lower than our exact solutions.

Our experiments included real-world datasets with a maximum size of $N = 5,000$. To the best of our knowledge, the largest dataset for which an exact solution has been previously obtained is $N = 150$, as documented by Ceselli & Righini (2005) with $K = 3$. Existing literature on the K-medoids problem has only reported exact solutions on very small datasets, primarily due to the use of BnB algorithms. Given their unpredictable run-time and worst-case exponential time complexity, most reported usage of BnB algorithms impose a hard computational time limit to avoid memory overflow or intractable run times.

In summary, Ren et al. (2022)'s algorithm returned only **approximate** solutions (with an optimality gap greater than zero), whereas our algorithm consistently produced **provably exact** solutions in **significantly less time**. Furthermore, for challenging datasets, such as WDG, Ren et al. (2022)'s algorithm produced a solution with an optimality gap of **800%** even after running for three hours! Moreover, in *nearly all* datasets tested in our experiments but not included in Ren et al. (2022) (e.g. IC, Yearst, WDG, wine, LD, VC, UKM and LM), their algorithm produces obvious errorness solutions where upper bounds that were **lower than** our **exact** solutions, which is fundamentally incorrect as an upper bound cannot be lower than the exact solution.

The only reason that Ren et al. (2022) claim their algorithm can handle datasets with over a million instances is that they test on datasets that are inherently easy to classify—so that even approximate algorithms can obtain exact solutions. As we have demonstrated, almost all the datasets they use can be solved exactly using PAM or Faster-PAM, which achieve exact solutions with significantly fewer resources. In contrast, Ren et al. (2022)'s algorithm requires an excessive amount of computational power (6,000 CPU cores) compared to approximate algorithms.

Moreover, we observed that Ren et al. (2022)'s algorithm exhibits **exponential** complexity even when K is fixed. This is evident from the **non-polynomial growth** in running time for experiments on datasets such as UK, BM, and Seeds. Although these datasets have nearly identical sizes, their running time differs significantly in Ren et al. (2022)'s algorithm. In the following section, we provide an empirical analysis of their algorithm to further investigate this behavior.

[2]We set the maximum number of neighbors examined as 4, and the number of iteration as 5.

3.2. Time complexity analysis without parallelization

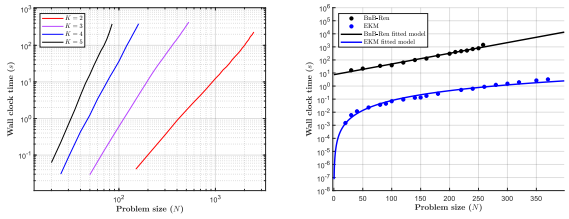

Figure 2. Log-log wall-clock run time (seconds) for our algorithm (EKM) tested on synthetic datasets (left panel). The run-time curves from left to right (corresponding to $K = 2, 3, 4, 5$ respectively), have slopes 3.005, 4.006, 5.018, and 5.995, an excellent match to the predicted worst-case run-time complexity of $O\left(N^3\right)$, $O\left(N^4\right)$, $O\left(N^5\right)$, and $O\left(N^6\right)$ respectively. Log-linear wall-clock run-time (seconds) comparing EKM algorithm against Ren et al. (2022)'s algorithm by sampling UK dataset with $K = 3$ (right panel). On this log-linear scale, exponential run-time appears as a linear function of problem size N, whereas polynomial run-time is a logarithmic function of N.

We test the wall-clock time of our novel EKM algorithm on a synthetic dataset with cluster sizes ranging from $K = 2$ to 5. When $K = 2$, the data size N ranges from 150 to 2,500, $K = 3$ ranges from 50 to 530, $K = 4$ ranges from 25 to 160, and $K = 5$ ranges from 30 to 200. The worst-case predictions are well-matched empirically (Figure 2, left panel).

As predicted, Ren et al. (2022)'s algorithm exhibits worst-case exponential time complexity even for a fixed $K = 3$ (Figure 2, right panel), whereas our algorithm runs in polynomial time in the worst case.

3.3. Performance compared with one-by-one enumeration

It is easy to confuse our algorithm with the one-by-one enumeration strategy, as both share a worst-case time complexity $O\left(N^{K+1}\right)$ time complexity in the worst-case. However, this classical approach of modeling algorithm performance using big-O notation is inaccurate in the context of combinatorial generation, as it ignores constant factors, which can significantly impact the actual performance of an algorithm in practice.

To provide a more detailed analysis, it is crucial to examine the wall-clock runtime of different generators in actual implementation. In this section, we compare our approach with

the classical combination generator often used in the machine learning community, the `itertools` package, which is based on the lexicographical generation algorithm (Kreher & Stinson, 1999). We demonstrate that although both generators have the same asymptotic complexity, the constant factors hidden in the big-O notation differ significantly, leading to a substantial disparity in performance.

Both experiments are based on CPU, but our algorithm is designed for easy implementation on a GPU. The performance difference would likely be even more pronounced with the additional speedup provided by GPU acceleration.

In Figure 3, the Python implementation of our algorithm achieves performance improvements of up to 10-fold for the task of exhaustive generation and up to 1000-fold for solving the K-medoids problem, compared to the C-based `itertools` library on CPU. Furthermore, the performance gap continues to expand as the combinatorial complexity of the problem increases. For exhaustive generation, the performance improvement grows from 3-fold to 10-fold, while for solving the K-medoids problem, it increases from 300-fold to 1000-fold.

4. Discussion

In this paper, we derived EKM, a novel exact algorithm for the K-medoids clustering problem with worst-case $O\left(N^{K+1}\right)$ time complexity. One immediate objection to our algorithm might be that it shares the same worst-case complexity as the naive one-by-one enumeration strategy. However, our algorithm demonstrates significantly better performance in optimization tasks and offers the potential for further speed-ups through fusion—an advantage that is unattainable with the one-by-one enumeration approach. Since the K-medoids problem is NP-hard, it is unlikely that the K in the exponent can be eliminated in the worst case. Therefore, our focus shifts to designing an efficient recursive algorithm suitable for modern hardware, such as GPUs. This approach paves the way for future acceleration through bounding techniques based on efficient recursion, rather than attempting to improve existing BnB algorithms, which exhibit exponential worst-case complexity. The bounding techniques used in BnB algorithms **cannot** improve the worst-case complexity, as no acceleration can be employed in the worst cases.

Besides presenting our novel EKM algorithm, we aim to prompt researchers to reconsider the metric for assessing the efficiency of exact algorithms to account for subtleties beyond simple problem scale. In discussing the "*goodness*" of exact algorithms for ML, it is critical to recognize that focusing solely on the scalability of these algorithms—for instance, their capacity to handle large datasets—does not provide a comprehensive assessment of their utility. This inclination to prioritize scalability when assessing exact algorithms arises from the perceived intractable combinatorics of many ML problems, and most of these problems are classified as NP-hard, so that no known algorithm can solve all instances of the problem in polynomial time.

Although many ML problems may be NP-hard in their most general form, they are often not specified in full generality. In such cases (which are quite common in practice), polynomial-time solutions may be available. Therefore, comparing algorithms solely on the basis of problem scale is neither fair nor accurate, as it ignores variations in actual run-times, such as the tightness of upper bounds, or parameters independent of problem scale.

However, for many ML problems, the problems specified for proving NP-hardness are not the same as their original definitions used in practical ML applications. Apart from the K-medoids problem, polynomial-time algorithms for solving the 0-1 loss classification problem (He & Little, 2023) and other K-clustering problems (Inaba et al., 1994; Tîrnăucă et al., 2018) have also been developed in the literature. If a polynomial-time algorithm does exist for these seemingly intractable problems, overemphasizing scalability can mislead scientific development, diverting attention from important measures such as memory usage, worst-case time complexity, and the practical applicability of the algorithm in real-world scenarios. For example, by setting the cluster size to one, the K-medoids problem can be solved exactly by choosing the closest data item to the mean of the data set, a strategy with $O\left(N\right)$ time complexity in the worst-case. While this $O\left(N\right)$ time algorithm can efficiently handle very large scale datasets, it does little to advance our understanding of the fundamental principles involved.

Therefore, judging an algorithm implementation solely by the scale of the dataset it can process is not an adequate measure of its effectiveness. Indeed, for large datasets, the use of exact algorithms may often be unnecessary as many high-quality approximate algorithms provide very good results, supported by solid theoretical assurances. If the clustering model closely aligns with the ground truth, the discrepancy between approximate and exact solutions should not be significant, provided the dataset is sufficiently large. Thus, it is not surprising that Ren et al. (2022)'s algorithm can achieve excellent approximate solutions with more than one million data points, a typical occurrence in studies involving exact algorithms. Past research has shown that while exact algorithms can quickly find solutions, most of the effort is expended on verifying their optimality (Dunn, 2018; Ustun, 2017). It is possible that the first configuration generated by the algorithm is optimal, but proving its optimality without exploring the entire solution space $\mathcal{S}$ is impossible (unless additional information is provided).

For the study of the K-medoids problem, while algorithms

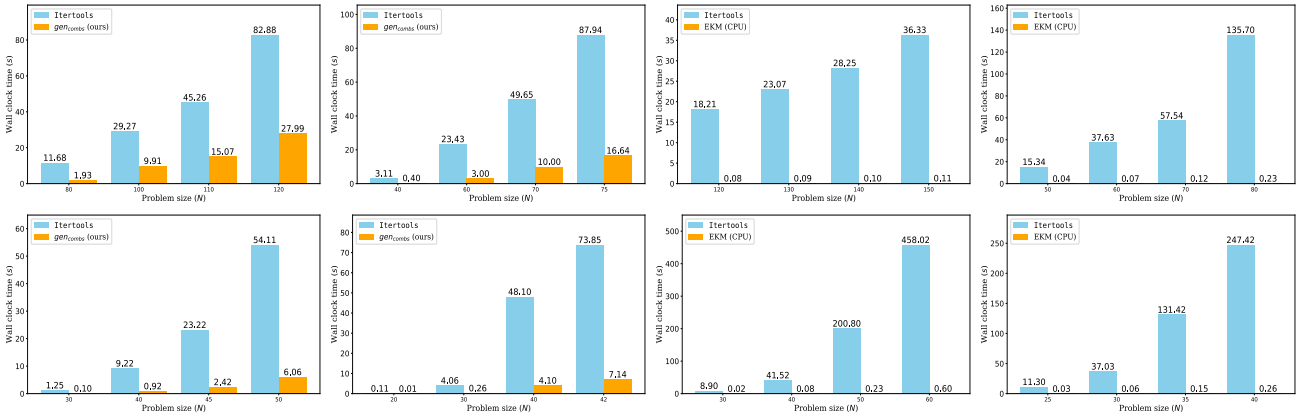

Figure 3. The running time comparison between the `itertools` library—a widely used Python library for generating combinations—and our combination generator for the task of exhaustive generation (The panels are arranged from left to right and top to bottom from $K = 5$ to K=8) and the task of solving the K-medoids problem ($K = 3$ to $K = 6$) demonstrates that our Python implementation achieves performance improvements of up to 10-fold and 1000-fold, respectively, over the C-based `itertools` library on the CPU.

presented in Ren et al. (2022); Ceselli & Righini (2005); Elloumi (2010); Christofides & Beasley (1982) are exact in principle, experiments reported by the authors do not demonstrate the actual computation of exact solutions, nor do they provide any theoretical guarantee on the computational time required to achieve satisfactory approximate solutions. If the application of the problem is concerned with only the approximate solution, it may be more beneficial to concentrate on developing more efficient or more robust heuristic algorithms. This could potentially offer more practical value in scenarios where approximate solutions are adequate.

5. Conclusions and future work

As our predictions and experiments show, our novel EKM algorithm clearly outperforms all other algorithms which can be guaranteed to obtain the exact globally optimal solution to the K-medoids problem. Whereas the approximate algorithms and the BnB algorithm designed by Ren et al. (2022) can only obtain approximate solutions (or wrong solutions) and use significantly more time.

Moreover, precise time and space complexity anlysis of the EKM algorithm precise prediction of time and space requirements before attempting to solve a clustering problem, in contrast to existing BnB algorithms which in practice require a hard computation time limit to prevent memory overflow or bypass the exponential worst-case run time. In our experiments we were able to process datasets of up to $N = 5,000$ data items, a considerable increase from the previous maximum of around $N = 150$. We have also shown that the state-of-the-art BnB algorithm, while claimed to be optimal, produces erroneous solutions on many datasets. Additionally, it only returns approximate solutions, requiring significantly more time across all datasets we tested. For difficult datasets, the algorithm by Ren et al. (2022)'s algorithm exhibits exponential complexity, even for fixed K.

The main disadvantage of our algorithm is that its space and time complexity is exponential in K. Thus for problems that involve a large number of medoids, our algorithm quickly becomes intractable. Currently, we only employs the plain form of our algorithm, in other words, we have not yet consider the speed-up by using bounding techniques. In the future, a more sophisticated bounding techniques could be developed; the inherently recursive structure of our combination generator makes this a relatively simple prospect.

With exact solutions for combinatorial ML problems, the memory-computation trade-off is always present. However, with BnB algorithms and MIP solvers, space complexity analysis is often omitted making it difficult to ascertain the actual memory requirements. Specifically, when using off-the-shelf MIP solvers, the memory required just to specify the problem can be substantial. For instance, to describe the K-medoids problem in MIP form (1), a constraint matrix[3] of size $\left(N^2 + N + 1\right) \times N$ is required (Ren et al., 2022; Vinod, 1969). Indeed, memory overflow issues have been reported in almost all practical usage of BnB algorithms (Ceselli & Righini, 2005; Elloumi, 2010; Christofides & Beasley, 1982). Therefore, setting a computation time limit is a necessary restriction for BnB algorithms, a restriction which only applies to EKM at large values of K or N.

[3]There are N^2 constraints to ensure each point is assigned to exactly one cluster, N constraints to identify which data items are medoids, and one additional constraint for the number of medoids

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Bellman, R. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

Bird, R. and de Moor, O. The algebra of programming. 1996. URL http://www.cs.ox.ac.uk/publications/Books/algebra/.

Ceselli, A. and Righini, G. A branch-and-price algorithm for the capacitated p-median problem. *Networks: An International Journal*, 45(3):125–142, 2005.

Christofides, N. and Beasley, J. E. A tree search algorithm for the p-median problem. *European Journal of Operational Research*, 10(2):196–204, 1982.

Dunn, J. W. *Optimal trees for prediction and prescription*. PhD thesis, Massachusetts Institute of Technology, 2018.

Elloumi, S. A tighter formulation of the p-median problem. *Journal of Combinatorial Optimization*, 19(1):69–83, 2010.

Fokkinga, M. M. An exercise in transformational programming: Backtracking and branch-and-bound. *Science of Computer Programming*, 16(1):19–48, 1991.

Gurobi Optimization, L. Gurobi optimizer reference manual. 2021.

He, X. and Little, M. A. An efficient, provably exact algorithm for the 0-1 loss linear classification problem. *ArXiv. /abs/2306.12344*, 2023.

Inaba, M., Katoh, N., and Imai, H. Applications of weighted voronoi diagrams and randomization to variance-based k-clustering. In *Proceedings of the Tenth Annual Symposium on Computational Geometry*, pp. 332–339, 1994.

Jeuring, J. T. and Pekela, T. *Theories for algorithm calculation*. Universiteit Utrecht, 1993.

Kaufman, L. Partitioning around medoids (program pam). *Finding groups in data*, 344:68–125, 1990.

Kreher, D. L. and Stinson, D. R. Combinatorial algorithms: generation, enumeration, and search. 30(1):33–35, 1999.

Little, M. A., He, X., and Kayas, U. Polymorphic dynamic programming by algebraic shortcut fusion. *Formal Aspects of Computing*, May 2024. ISSN 0934-5043. doi: 10.1145/3664828. URL https://doi.org/10.1145/3664828. (in press).

Makhorin, A. GLPK (gnu linear programming kit). *http://www.gnu.org/s/glpk/glpk.html*, 2008.

Meertens, L. Algorithmics: Towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1, pp. 289–334, 1986.

Ng, R. T. and Han, J. CLARANS: A method for clustering objects for spatial data mining. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1003–1016, 2002.

Padberg, M. and Rinaldi, G. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *Society for Industrial and Applied Mathematics Review*, 33(1):60–100, 1991.

Ren, J., Hua, K., and Cao, Y. Global optimal k-medoids clustering of one million samples. *Advances in Neural Information Processing Systems*, 35:982–994, 2022.

Schubert, E. and Rousseeuw, P. J. Fast and eager k-medoids clustering: O (k) runtime improvement of the pam, clara, and clarans algorithms. *Information Systems*, 101:101804, 2021.

Tîrnăucă, C., Gómez-Pérez, D., Balcázar, J. L., and Montaña, J. L. Global optimality in k-means clustering. *Information Sciences*, 439:79–94, 2018.

Ustun, B. T. B. *Simple linear classifiers via discrete optimization: learning certifiably optimal scoring systems for decision-making and risk assessment*. PhD thesis, Massachusetts Institute of Technology, 2017.

Van der Laan, M., Pollard, K., and Bryan, J. A new partitioning around medoids algorithm. *Journal of Statistical Computation and Simulation*, 73(8):575–584, 2003.

Vinod, H. D. Integer programming and the theory of grouping. *Journal of the American Statistical Association*, 64 (326):506–519, 1969.

Wang, E., Ballachay, R., Cai, G., Cao, Y., and Trajano, H. L. Predicting xylose yield from prehydrolysis of hardwoods: A machine learning approach. *Frontiers in Chemical Engineering*, 4:994428, 2022.