ABSTRACT

How can we efficiently and scalably cluster high-dimensional data? The $k$-means algorithm clusters data by iteratively reducing intra-cluster Euclidean distances until convergence. While it finds applications from recommendation engines to image segmentation, its application to high-dimensional data is hindered by the need to repeatedly compute Euclidean distances among points and centroids. In this paper, we propose MARIGOLD ($k$-means for high-dimensional data), a scalable algorithm for $k$-means clustering in high dimensions. MARIGOLD prunes distance calculations by means of (i) a tight distance-bounding scheme; (ii) a stepwise calculation over a multi-resolution transform; and (iii) exploiting the triangle inequality. To our knowledge, such an arsenal of pruning techniques has not been hitherto applied to $k$-means. Our work is motivated by time-critical Angle-Resolved Photoemission Spectroscopy (ARPES) experiments, where it is vital to detect clusters among high-dimensional spectra in real time. In a thorough experimental study with real-world data sets we demonstrate that MARIGOLD efficiently clusters high-dimensional data, achieving approximately one order of magnitude improvement over prior art.

1 INTRODUCTION

Data in very high-dimensional spaces arises in the health sciences, astronomy, physics, finance, surveillance, bioinformatics, and via the datafication of human society [24, 25, 29]. To analyze such data, we need to tailor data science methods for high-dimensional spaces. A prominent data science method used to automatically discover groupings in a raw data distribution and as an initial step before further analysis is $k$-means clustering by Lloyd’s algorithm [30]. As an unsupervised method, it does not necessitate any annotations on the raw data as input; it partitions the data into $k$ disjoint groupings, or clusters, such that each data record is closer to its allocated cluster’s center or representative, by a given distance function, than to others. The distance function is usually Euclidean distance [16, 47], forming a Voronoi/Dirichlet tessellation among cluster centers [41].

While $k$-means clustering has been extensively applied and enhanced in terms of choosing the value of $k$ [6, 26, 33, 39, 43], initializing cluster centers [44, 45, 48], and handling many small clusters [9], such methods evade scaling up the exact $k$-means solution to very high dimensions. The main bottleneck is calculating distances from each data record to each candidate cluster center in each iteration. For instance, a method focused on high-dimensional data records deals with data of only up to 100 features per document [9]. Some methods aim to eschew calculating high-dimensional distances [16, 18, 46], yet fully calculate distances that pass a filtering step. To our knowledge, no previous work aims to calculate high-dimensional distances only partially for the sake of time-efficiency without affecting the $k$-means clustering result.

In this paper we introduce MARIGOLD: a pruning-intensive reformulation of Lloyd’s algorithm that uses bounds based on the triangle inequality, as in [16, 18], and, unprecedentedly, a lightweight pre-processing and a tight double-bounding scheme along with a stepwise distance calculation over an energy-concentrating transformation to trim high-dimensional distance calculations during iterations. MARIGOLD delivers the exact result of Lloyd’s algorithm in at least one order of magnitude less time.

Motivating application. MARIGOLD addresses a real-world need in time-critical Angle-Resolved Photoemission Spectroscopy (ARPES) condensed-matter physics experiments, where the high-dimensional spectra of a solid’s electronic structure need to be clustered in real time, and even repetitively under modification of experimental parameters. In this application, clustering in reduced dimensionality can miss important physical properties of materials, while standard Lloyd’s takes a prohibitive amount of time.

MARIGOLD: Efficient $k$-means Clustering in High Dimensions

Kasper Overgaard Mortensen
Aarhus University
kmcs@cs.au.dk

Fateme Zardbani
Aarhus University
fateme.zardbani@cs.au.dk

Mohammad Ahsanul Haque
Aarhus University
mah@cs.au.dk

Steinn Ymir Agustsson
Aarhus University
agustsson@phys.au.dk

Davide Mottin
Aarhus University
davide@cs.au.dk

Panagiotis Karras
Aarhus University
piekarras@gmail.com

PVldb Reference Format:
doi:10.14778/3587136.3587147

PVldb Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/AU-DIS/scalable-kmeans.
2 BACKGROUND

A clustering partitions a set of $N$-dimensional points $X$ to disjoint sets or clusters. In $k$-means, a cluster $X_i \subseteq X$ is represented by a centroid $c_i$ being the mean of the points therein:

$$c_i = \frac{1}{|X_i|} \sum_{x \in X_i} x$$

The compactness of a cluster $i$ is the sum of the squared Euclidean distances among points in a cluster and its centroid:

$$\sum_{x \in X_i} d(x, c_i)^2 = \sum_{x \in X_i} \|x - c_i\|^2 = \sum_{i=1}^{N} \sum_{j=1}^{N} (x_j - c_{ij})^2$$  \hspace{1cm} (1)

The $k$-means problem is to find a clustering $C$ of $k$ clusters that minimizes total compactness.

Algorithm 1 LLOYD ($X, k$)

1: $C \leftarrow \text{SAMPLE}(k, X)$ \hspace{1cm} $\rightarrow$ sample $k$ points from $X$
2: while not converged do
3: for all $x$ in $X$ do $\rightarrow$ assign points to centroids
4: $a(x) \leftarrow \arg\min_c \{d(x, c)\}$
5: converged $\leftarrow \text{RECALCULATE}(X, C, a[X])$ $\rightarrow$ Algorithm 2
6: return $a[X]$

Algorithm 2 RECALCULATE ($X, C, a[X]$)

1: $C_{old} \leftarrow C$
2: converged $\leftarrow \text{true}$
3: for all $c$ in $C$ do $\rightarrow$ recalculate centroid positions
4: $c \leftarrow \text{MEAN}(\{x \in X | a(x) = c\})$ \hspace{1cm} $\rightarrow$ new centroid
5: $d_{old}[c] \leftarrow d(c, c_{old})$ \hspace{1cm} $\rightarrow$ centroid divergence
6: if $d_{new}[c] > \epsilon$ then converged $\leftarrow \text{false}$
7: return converged, $d_{new}[C]$

2.1 Lloyd’s algorithm

As the $k$-means problem is NP-hard [30], it is addressed by heuristics; Lloyd’s algorithm [17, 30], starting with a set of $k$ centroids, repetitively assigns each point to the nearest centroid’s cluster by Euclidean distance and recalculates each centroid as the mean of its members’ positions until convergence. In effect, it repetitively computes $O(kn)$ distances between points and centroids; this computation becomes prohibitive as dimensionality $N$ grows [32]. Algorithm 1 presents the pseudocode, while Algorithm 2 presents the subroutine that recalculates centroid positions until they converge.

Several enhancements on Lloyd’s algorithm have been proposed: choosing an appropriate value of $k$ [6, 33] by the Elbow method [26], gap statistics [43], or the Silhouette method [39]; selecting initial cluster centers [44, 48] by $k$-means++ [45]; addressing the problem with many small clusters by nearest-neighbor search from cluster centers, rather than from observation points [9]; avoiding distance computations by utilizing the triangle inequality [16, 18]. We review these triangle-inequality-based methods next.

2.2 Applying the triangle inequality

To prune the $O(kn)$ point-to-centroid Euclidean distance computations, Elkan [16] leverages the triangle inequality in a metric space, by which, for any point $x$ and centroids $c_i$ and $c_j$, it holds that:

$$d(c_i, c_j) \leq d(x, c_i) + d(x, c_j)$$  \hspace{1cm} (2)
$$d(x, c_i) \leq d(c_i, c_j) + d(x, c_j)$$  \hspace{1cm} (3)
$$d(x, c_j) \leq d(c_i, c_j) + d(x, c_i)$$  \hspace{1cm} (4)

The following lemma follows directly from Equations (2) and (3):

Lemma 2.1. $d(x, c_i) \geq |d(x, c_i) - d(c_i, c_j)|$

By Lemma 2.1, if we know $d(x, c_i)$ and $d(c_i, c_j)$, we can lower-bound $d(x, c_j)$. Further, if $d(c_i, x) \leq \frac{d(c_i, c_j)}{2}$, then, from Equation (2) it follows that $d(x, c_i) \leq \frac{1}{2} \left| d(x, c_i) + d(x, c_j) \right|$, which implies that $d(x, c_i) \leq d(x, c_j)$. Formally:

Lemma 2.2. If $d(x, c_i) \leq \frac{d(c_i, c_j)}{2}$, then $d(x, c_i) \leq d(x, c_j)$

By Lemma 2.2, if $c_i$’s distance from $x$ is at most half its distance from any other centroid, then $c_i$ is nearest to $x$. Thus, $x$ may be re-assigned from $c_i$ to $c_j$ only if $d(x, c_j) > d(c_i, c_j)$. In effect, these lemmata prune recomputing point-to-centroid distances using intercentroid distances and prior point-to-centroid distances.

Hammerly [18] proposed two further conditions to eschew distance computations. The first condition utilizes a lower bound $\ell_T$ on the distance to a point’s nearest non-assigned centroid. At any iteration, if an upper bound for the distance to a point x’s currently nearest centroid derived by Lemma 2.2 is more than $\ell_T$, then the nearest centroid to $x$ remains unchanged. The second condition uses centroid-to-centroid distances as in Lemma 2.2.

2.3 Other related work

We accelerate Lloyd’s algorithm in high dimensions by pruning its calculations. Other works, which do not address the same problem, render Lloyd more robust via seeding, accelerate it in low dimensions via indexing, and provide guarantees on top of a presumed approximation algorithm via dimensionality reduction. These goals are orthogonal to ours as we explain in the following.

Seeding. $k$-means++ [45] boosts the stability of Lloyd through a smart initialization, extended to a stream setting [2]; $k$-means [4] allows for parallelization of $k$-means++ via oversampling. These are orthogonal works, as they cater to initialization only.

Index-based. Some techniques accelerate $k$-means by indexing [35]. Others propose using cosine similarity and use an inverted index to assign points to centroids [9]. However, such works require an additional $O(n \log n)$ pre-computation for index building, while indexing in high dimensions is problematic in itself; thus, unfortunately, such methods do not address high-dimensional $k$-means.

Sampling. Sampling methods [21, 40] select a fixed amount of points to cluster. However, such techniques still require many samples when clusters are small. Kumar et al. [27] samples a fixed number of points to devise a $1/\epsilon$-approximate solution linear in $n$ and $N$ but, unfortunately, exponential in $k$. Our proposal are orthogonal to, and may be used along with, such sampling methods.

Dimensionality reduction. Several works study $k$-means on dimensionality reduced compared to the inherent dimensionality of
the data, and eventually independent thereof [5, 7, 12], leading to the currently best \((1 + \epsilon)\)-approximation with \(O(\log(k/\epsilon)/\epsilon^2)\) dimensions in [31]. However, such methods typically assume a pre-existing \(\gamma\)-approximation of the optimal \(k\)-means solution, yet resort to the Lloyd’s algorithm to obtain a good solution in practice [8]. Further, the assumed reduced dimensionality is still high in practice; it depends on \(1/\epsilon^2\), which corresponds at least to \(N > 10^4\) [8] to achieve a 10% deviation from the optimal solution, even assuming a \(\gamma\)-approximation algorithm were available. Our work is orthogonal and complementary to such approaches, which still require running Lloyd’s algorithm in a significant number of dimensions; thus, improving the scalability of Lloyd’s algorithm would provide a direct benefit on real-world data science applications and also enhance the usability of such theoretical advances.

3 APPLICATION DOMAIN

The need to scale \(k\)-means to high-dimensional data arises in Angle-Resolved Photo-Emission Spectroscopy (ARPES), an experimental method in condensed-matter physics that studies the electronic structure of materials [37]. In an ARPES experiment, electrons are excited from a sample by means of ultraviolet light eliciting the photoelectric effect. We focus on spatially-resolved ARPES, also called nanoARPES due to its high resolution [19, 38]: as Figure 1 shows, a finely focused light spot is raster-scanned across a sample to yield, at each position, a photoemission intensity image, or spectrum, that records photoemission intensity as a function of electron energy \(E_{\text{kin}}\) and emission angle \(\theta\); such spectra encode the material’s band structure and its electronic properties.

The development of photon sources and detectors has increased the measurable parameter space, data dimensionality, and acquisition rate. Machine learning methods have proven effective at spotting differences in ARPES data [24, 34, 36], yet they resort to dimensionality reduction that risks missing essential features. As the cost of acquisition remains high and the time of access to the required light sources such as synchrotrons or free-electron lasers is limited, physicists need to quickly extract clusters from a stack of photoemission images at high dimensionality to make real-time decisions on how to proceed during an experimental data-collection session; to this end, we aim at accelerating \(k\)-means clustering on high-dimensional data as those arising in nanoARPES.

4 MARIGOLD

While Elkan’s and Hamerly’s methods [16, 18] reduce distance calculations, their runtime is still prohibitive on high data dimensionality. After all, by those methods, once a distance passes the filtering step, it is fully calculated. Arguably, a method should prune distance calculations even after a first filtering step, i.e., perform partial distance calculations with early abandonment. Here we devise Marigold, a method that utilizes a hierarchical energy-concentrating feature transformation to gradually calculate distances over the feature hierarchy, on top of triangle-inequality-based pruning. We first show how to exploit the triangle inequality utilizing both of [16] and [18] and then illustrate the stepwise calculation of distances.

Algorithm 3 Triangle-based \(k\)-means (\(X, k\))

1. \(C \leftarrow \text{SAMPLE}(k, X)\) \hspace{1cm} \triangleright \text{initial cluster centers}
2. \(\alpha[X] = 0\) \hspace{1cm} \triangleright \text{null assignment}
3. \(\ell_E(X, C) = \ell_H(X) = \text{near}[C] = 0\) \hspace{1cm} \triangleright \text{initialize lower bounds}
4. \(u_E(X) = \infty\) \hspace{1cm} \triangleright \text{initialize upper bounds}
5. while not converged do \hspace{1cm} \triangleright \text{main loop}
6. for all \(x\) in \(X\) do \hspace{1cm} \triangleright \text{assign points to centroids}
7. if \(u_E(x) > \max(\text{near}[\alpha[x]], \ell_E(x))\) then \hspace{1cm} \triangleright \text{Hamerly’s LB}
8. \(\ell_E(x, \alpha[x]) \leftarrow d(x, \alpha[x])\) \hspace{1cm} \triangleright \text{Elkan’s LB}
9. \(u_E(x) \leftarrow d(x, \alpha[x])\) \hspace{1cm} \triangleright \text{Elkan’s UB}
10. for all \(c \in C\) \([c \neq \alpha[x]]\) do \hspace{1cm} \triangleright \text{Elkan check}
11. if \(u_E(x) > \max\{\ell_E(x, c), d(\alpha[x], c)/2\}\) then \hspace{1cm} \triangleright \text{calculate dist.}
12. \(\ell_E(x, c) \leftarrow d(x, c)\) \hspace{1cm} \triangleright \text{Elkan’s UB}
13. 14. 15. \(u_E(x) \leftarrow d(x, c)\) \hspace{1cm} \triangleright \text{Elkan’s UB}
16. converged, \text{div}[C] \leftarrow \text{RECALCULATE}(X, C, \alpha[X])\) \hspace{1cm} \triangleright \text{Alg. 2}
17. if not converged then \hspace{1cm} \triangleright \text{Alg. 4}
18. UpdateBounds \((X, C, \ell_E, u_E, \ell_H, \text{div, near})\) \hspace{1cm} \triangleright \text{Alg. 4}
19. return \(\alpha[X]\)

Algorithm 4 UpdateBounds \((X, C, \ell_E, u_E, \ell_H, \text{div, near})\)

1. for all \(x\) in \(X\) do \hspace{1cm} \triangleright \text{update bounds}
2. for all \(c \in C\) do \hspace{1cm} \triangleright \text{Elkan’s LB}
3. \(\ell_E(x, c) \leftarrow \max\{0, \ell_E(x, c) - \text{div}\[c]\}\)
4. \(u_E(x) \leftarrow u_E(x) + \text{div}[\alpha[x]]\) \hspace{1cm} \triangleright \text{Elkan’s UB}
5. \(\ell_H(x) \leftarrow \min\{\ell_E(x, \alpha[x])\}\) \hspace{1cm} \triangleright \text{Hamerly’s LB}
6. for all \(c \in C\) do \hspace{1cm} \triangleright \text{Elkan check}
7. \(\text{near}[c] \leftarrow \frac{1}{2} \min_{c \neq c'} d(c, c')\) \hspace{1cm} \triangleright \text{nearest other centroid}
8. return \(\ell_E, u_E, \ell_H, \text{div, near}\)

4.1 Applying the triangle inequality

Both Elkan [16] and Hamerly [18] exploit the triangle inequality to contain distance calculations, yet their methods have yet to be combined. In Algorithm 3, we put together all opportunities, derived from [16] and [18], to avoid distance calculations using bounds derived from the triangle inequality, while assigning data points to centroids. Algorithm 4 presents the bound update subroutine. We use the following bounds to eschew distance calculations:

Elkan’s lower bound \(\ell_E(x, c)\) bounds the distance from \(x\) to centroid \(c\) as the latter moves in one iteration so that \(d(x, c) \geq \ell_E(x, c)\).
we proceed to check whether \( u \).

### 4.2 Leveraging stepwise distance calculations

Pruning by the triangle inequality reduces the number of Euclidean distance calculations in each iteration. However, it leaves the cost of each such calculation intact: once a distance calculation passes the pruning stage, it has to be executed at full cost. Unfortunately, this cost grows prohibitively with dimensionality. More drastic measures are needed to discard or contain distance calculations.

We might use dimensionality reduction [42] hoping to obtain a data set that preserves the characteristics of the original. However, such reduction incurs information loss, which may severely affect the clustering result in critical applications where clustering aims to discover unknown data properties, such as nanoARPES which aims on an energy-preserving high dimensions is imperative even to use such methods.

Nevertheless, dimensionality reduction methods typically rest on an energy-preserving transform that unrolls the data in a hierarchy of progressively finer levels of resolution [22] and concentrates energy in lower-order coefficients. Full precision is available at the finest level of resolution, yet cluster assignments can be decided at coarser levels, without reading all features. We employ this property of a multi-resolution transform and a tight bounding scheme to calculate distances à la carte, as required to decide cluster membership; therewith we accelerate Lloyd while preserving its output.

We use the Discrete Cosine Transform (DCT), extensively used in image compression [11, 15]. Let \( F \) be an \( M \times N \) matrix representing a 2-dimensional data set and \( G \) be the matrix of its 2-dimensional DCT coefficients. Element \((u, v)\) of \( G \) is [28]:

\[
g(u, v) = \sum_{n=0}^{M-1} \sum_{m=0}^{N-1} f(m, n) \cos \left( \frac{(m+\frac{1}{2})u\pi}{M} \right) \cos \left( \frac{(n+\frac{1}{2})v\pi}{N} \right) \tag{5}
\]

while each element is normalized by \( \frac{2k_0}{MN} \) with \( k_0 = 1/\sqrt{2} \).

DCT preserves Euclidean distance on the transformed space [3] over DCT coefficient vectors \( x \) of raw data points; thus, the Euclidean distance \( d(x, c) \) of a point \( x \) from a centroid \( c \) is:

\[
d(x, c) = \sqrt{\sum x_i^2 + c_i^2 - 2x_ic_i} = \sqrt{\sum x_i^2 + \sum c_i^2 - 2\sum x_ic_i} \tag{6}
\]

where the summation is over all relevant DCT \((u, v)\) pairs, identified by \( i \). For any point \( x \) and centroid \( c \), we may pre-compute \( \sum x_i^2 \) and \( \sum c_i^2 \) terms. However, we should still compute the dot-product term \( x_ic_i \) on the fly. Still, we can bound this term using the Cauchy-Schwarz inequality [23] as follows:

\[
\left( \sum x_ic_i \right)^2 \leq \sum x_i^2 \sum c_i^2 \Rightarrow \sum x_ic_i \leq \sqrt{\sum x_i^2 \sum c_i^2} \tag{7}
\]

From Equations (6) and (7) it follows that:

\[
d(x, c)^2 \geq \sum x_i^2 + \sum c_i^2 - 2\sqrt{\sum x_i^2 \sum c_i^2} \tag{8}
\]

All terms in Equation (8) involve sums of squares over the dimensions of \( x \) and \( c \), which we only need to compute once. We utilize these bounds to calculate point-to-centroid distances in step-wise fashion, as in [23]: in each step, we derive exact distances up to an increasing number of DCT features, the known part, along with upper and lower bounds, by Equation (8), for the diminishing remainder, unknown part. While doing so, we discard from consideration, for each point, centroids whose lower bound gets higher than the lowest upper bound among other centroids. Starting with DCT term \((0, 0)\), in level \( t \) we extend the known DCT features from \( 4^t-1 \) to \( 2^t \), \( t > 0 \), spanning a square-shaped area from \((0, 0)\) to \((2^t, 2^t)\) in 2d, doubling the square side in each step. We pre-compute sums of \( x_{sq}^2 \) for \( 4^t \) squared feature values in the range from \( 4^{t-1} \) to \( 4^t \), for each \( t \), from the 2-dimensional DCT vector \( x \) of each data point in each such area, yielding a list \( x_{sq} \) of size \( O(\log N) \).

### Algorithm 5 Stepwise \((X, k)\)

1. \( L \leftarrow \log_4 N \) \hspace{1cm} \( \triangleright \) levels for stepwise distances
2. \( X \leftarrow \) Transform \((X)\) \hspace{1cm} \( \triangleright \) by DCT [1]
3. \( x_{sq} \leftarrow \) Squared \((X)\) \hspace{1cm} \( \triangleright \) sums of \( 4^t \) squares, \( t' = t, \ldots , L \)
4. \( C \leftarrow \) Sample \((k, X)\) \hspace{1cm} \( \triangleright \) initial cluster centers
5. while not converged do
   6. \( C_{sq} \leftarrow \) Squared \((C)\) \hspace{1cm} \( \triangleright \) recalculate \( C_{sq} \)
   7. for all \( x \) in \( X \) do
      8. \( a[x] \leftarrow \) SetLabel \((x, C, x_{sq}, C_{sq}, L)\) \hspace{1cm} \( \triangleright \) Algorithm 6
   9. converged \( \leftarrow \) Recalculate \((X, C, a[X])\) \hspace{1cm} \( \triangleright \) Algorithm 2
10. return \( a[X] \)

Algorithm 5 presents our \( k \)-means algorithm that leverages stepwise distance calculations over DCT-transformed data. Algorithm 6 shows the subroutine that assigns a point to a centroid, where
level \( \ell \) indicates the size of the square area of DCT features forming the known part in calculations. Starting from one known feature \( (\ell = 0) \), in each iteration it discards candidate centroids with a lower bound exceeding the lowest upper bound (Lines 7–8), expands the calculation for each candidate centroid (Line 11), and maintains the centroid of lowest upper bound (Lines 12–14). The iterations terminate when, at Line 5, we reach the last level, where bounds are exact distances, hence we select the centroid of lowest upper bound, or only one candidate centroid remains, hence we select it.

**Algorithm 6** SetLabel \((x, C, x_{sq}, C_{sq}, L)\)

```
1: \ell \leftarrow 0; \quad LB(C) \leftarrow 0; \quad UB_{min} \leftarrow \infty
2: \textbf{for all} \ (c \in C) \ \textbf{do} \quad \text{coarsest distance estimate}
3: \quad d[c] \leftarrow x_{sq}[0] + c_{sq}[0] \quad \text{candidate centroid} \ c \ \text{for} \ x
4: \quad \text{mask}[c] \leftarrow 1 \quad \text{prune centroid at previous level}
5: \quad \textbf{while} \ \ell \leq L \land \sum_{c} \text{mask}[c] > 1 \ \textbf{do} \quad \text{point undecided}
6: \quad \textbf{for all} \ (c \in C) \ (\text{mask}[c] = 1) \ \textbf{do} \quad \text{active candidates}
7: \quad \textbf{if} \ UB_{min} < LB(c) \ \textbf{then} \quad \text{move on to this level}
8: \quad \quad \text{prune centroid at previous level}
9: \quad \textbf{else} \quad \text{move on to this level}
10: \quad \textbf{if} \ c \in C \ \\
11: \quad \quad \text{initial cluster centers}
12: \quad \quad \alpha \leftarrow c \quad \text{Elkan check}
13: \quad \quad \text{Elkan bounds to check, in Line 7, whether to retain each candidate}
14: \quad \quad \text{centroid, and the Elkan lowest upper bound across centroids, used by}
15: \quad \quad \text{Line 7 with the next centroid, maintaining the centroid of lowest}
16: \quad \quad \text{upper bound as the running assignment. Upon reaching the last}
17: \quad \quad \text{return} \ \alpha \ [X]
18: \quad \ell \leftarrow \ell + 1
19: \textbf{return} \ \alpha
```

**Algorithm 7** DistToLevel \((x, c, x_{sq}, c_{sq}, \ell, d)\)

```
1: d \leftarrow d - 2 \cdot |x[\ell] - c[\ell]| \quad \text{known distance with all squares}
2: margin \leftarrow 2 \cdot \sqrt{x_{sq}[\ell + 1] \cdot c_{sq}[\ell + 1]} \quad \text{lower bound}
3: LB \leftarrow d - margin \quad \text{upper bound}
4: UB \leftarrow d + margin
5: \textbf{return} \ LB, UB, d
```

Algorithm 7 updates the distance for a given pair in each level of the stepwise distance computation; it updates the known distance term to the current level \( \ell \) and adds the Cauchy-Schwarz bounds for the unknown margin, using pre-computed sums of squares of the \( \frac{2 \cdot 3}{4} \) features from level \( \ell + 1 \) to level \( \ell \). Lower and upper bounds differ in one sign, as in Equation 8.

### 4.3 Putting it all together

Both triangle-inequality-based checks and stepwise calculations prune distance calculations, the former by an one-off filtering, the latter by progressively disqualifying centroid candidates in each iteration. Here, we merge these two methods to craft an integrated solution, Marigold (Algorithm 8). Marigold performs stepwise distance calculations on DCT-transformed data using pre-computed information by DistToLevel (Algorithm 7), as Stepwise (Algorithm 5) does, using the same bounds as Triangle-based k-means (Algorithm 3) and updating them by UpdateBounds (Algorithm 4), and recalculates centroids by Recalculate (Algorithm 2).

Still, Marigold keeps track of the tightest among the bounds obtained by stepwise calculations and those obtained by the triangle inequality, as Algorithm 9 illustrates, hence it has stronger pruning capacity than Stepwise (Algorithm 5).

**Algorithm 8** Marigold \((X, k)\)

```
1: L \leftarrow \log_2 N \quad \text{levels for stepwise distances}
2: X \leftarrow \text{Transform} (X) \quad \text{by DCT [1]}
3: x_{sq} \leftarrow \text{Squared} \quad \text{sums of} \ 4^\ell \text{ squares, } \ell = 1, \ldots, L
4: C \leftarrow \text{Sample} (k, X) \quad \text{initial cluster centers}
5: \alpha[X] = 0 \quad \text{null assignment}
6: \ell_f(X, C) = \ell_f(X) = \text{near} |C| = 0 \quad \text{initialize lower bounds}
7: \ u_g(X) = \infty \quad \text{initialize upper bounds}
8: \textbf{while} \ \text{not converged} \ \textbf{do} \quad \text{main loop}
9: \ C_{sq} \leftarrow \text{squared}(C) \quad \text{recur calculation}
10: \textbf{for all} \ x \in X \ \textbf{do} \quad \text{assign points to centroids}
11: \quad \text{if} \ u_g(x) > \max \{\alpha[x], \ell_f(x)\} \ \text{then} \quad \text{Hamery}
12: \quad \quad A \leftarrow (X, C, x_{sq}, C_{sq}, L, c|x], \ell_f(x, C), u_g(x)) \quad \text{Alg. 2}
13: \quad \text{SetLabelMG} (A) \quad \text{Alg. 9}
14: \quad \text{converged, } \text{do} \ [C] \quad \text{Alg. 4}
15: \quad \text{if not converged} \quad \text{Alg. 4}
16: \quad \text{UpdateBounds} (X, C, \ell_E, u_E, \ell_f, d_{in}, \text{near}) \quad \text{Alg. 4}
17: \quad \text{return} \ \alpha[X]
```

Algorithm 9 follows the pattern of SetLabel (Algorithm 6), yet introduces cardinal aspects of Triangle-based k-means (Algorithm 3) therein. In each level of stepwise calculations, it uses Elkan bounds to check, in Line 7, whether to retain each candidate centroid; thereby it injects the comparison of Lemma 2.2 using inter-centroid distances, into the bound comparison of Line 7, Algorithm 6. Line 12 of SetLabelMG obtains square roots of stepwise bounds to render them appropriate for triangle-inequality Euclidean-distance comparisons. Last, Lines 13–17 use stepwise bounds to tighten, if possible, both the Elkan lower bound for the examined centroid, used by Line 7 in the next iteration over centroids, and the Elkan lowest upper bound across centroids, used by Line 7 with the next centroid, maintaining the centroid of lowest upper bound as the running assignment. Upon reaching the last
level, where bounds are exact distances, or if only one candidate centroid remains (Line 5), we settle on the running assignment. Line 19 returns, along with the assigned centroid, the (possibly exact) Elkan lower bounds of distances to each centroid and Elkan upper bound of the distance to the assigned centroid; subsequently, Algorithm 2 recalculates centroid positions and Algorithm 4 updates Elkan bounds based on centroid divergences and performs a Hamerly bound to be used in Line 11 of Algorithm 8.

While Algorithm 3 derives Elkan and Hamerly bounds from exact distances, MARIGOLD (Algorithm 8) extracts them from stepwise bounds. The earlier MARIGOLD abandons a point-to-centroid distance calculation, the lesser the $E(x, c)$ bound it derives. Still, the next iteration tightens such a bound, if needed. Eventually, MARIGOLD performs fewer distance calculations than both TRIANGLE-based $k$-means and STEPWISE, at the cost of more inequality checks than STEPWISE (Line 11, Algorithm 8 and Line 13, Algorithm 9) and more square-root operations than both (Line 12, Algorithm 9).

In short, MARIGOLD prunes distance calculations effectively, incorporating Hamerly and Elkan bounds into bounds maintained by stepwise distance calculations. We emphasize that MARIGOLD provides the same results as Lloyd’s algorithm.

5 EXPERIMENTAL RESULTS

We conduct an experimental study of $k$-means methods on data from real-world nanoARPES experiments. To enhance the reproducibility of our results, we provide an implementation-independent measure, the number of feature distance calculations. We measure runtime in two environments: by Python 3.10 implementation on a Windows 10, Intel Core i7-1165G7 machine @2.80GHz with 48GB RAM, and by C++ implementation with -O3 flag on an Ubuntu 20.04.4, Intel Core i7-10610U machine @1.8GHz with 64GB RAM. We report runtimes including pre-computations, while we run with five different initializations and report averages.

Methods. We juxtapose the following methods for $k$-means:

- **LLOYD** [30], the classical iterative $k$-means algorithm.
- **ELKAN** [16], which uses Elkan bounds only as in Section 4.1.
- **HAMERLY** [18], using the Hamerly bound only as in Section 4.1.
- **TRIANGLE-based $k$-means**, as presented in Section 4.1.
- **KMEANS-G** [20], a state-of-the-art algorithm that exploits geometric primitives to perform $k$-means in high dimensions.
- **STEPWISE**, performing stepwise calculations as in Section 4.2.
- **MARIGOLD**, which integrates stepwise calculations with the bounds of ELKAN and HAMERLY, as in Section 4.3.

Data sets. We run $k$-means on four data sets obtained from two nanoARPES studies and four image datasets from other fields. To illustrate the need to run $k$-means in high dimensions we juxtapose the results by clustering in reduced dimensionality and those obtained in high dimensions to the physical ground truth [10, 13].

- **gr_flake** is a map of $49 \times 39 = 1911$ nanoARPES images, each with resolution 256×256, on a graphene sample. Clustering with $k=30$ separates areas in a way very similar to a systematic fit of the spectra. Figure 2(a) shows the original result of a fitted momentum (Figure 3(b) in [13]) whereas panels (b) and (c) show clustering at DCT dimensionalities 8×8 and 64×64. Using a higher DCT dimensionality gives a more accurate result.

- **misfit** comprises three maps of $12 \times 14 = 168$ nanoARPES images, each with resolution 1024×1024. misfit_VB, misfit_Se3d, and misfit_Bi5d, represent the same sample position observed at different energies (Figure S1 in [10] with a reversed color scale for misfit_Se3d). Figure 2(d)-(f) shows the photoemission intensity in a manually selected characteristic region of interest for the three data sets, while images (g)-(i) visualize $k$-means results with $k=16$ performed on entire spectra, confirming that high-dimensional $k$-means is appropriate for such data analysis.

- **MNIST** [14] comprises $28 \times 28 = 784$-size handwritten digits, in a training set of 60,000 images and a test set of 10,000 images. We upscale the test set to size $128 \times 128 = 16384$ and $256 \times 256 = 65536$.

- **Deep Globe Land Cover (DGLC)** contains three collections of RGB satellite images of land cover types, 1606 in DGLC Train, 171 in DGLC Validation and 172 in DGLC Test. We converted the images to grayscale and normalized resolutions to $256 \times 256$ in the first and $1024 \times 1024$ in the other collections.

5.1 Distance calculations

We first report results on the number of feature distance calculation operations each method performs vs. the number of DCT features representing the data and that of clusters $k$.

Varying DCT features. Figure 3(a) shows how the number of features entering distance calculations varies vs. the number of DCT features representing the data with $k=10$ clusters. All methods show linear growth, with a gap between LLOYD and HAMERLY, on the one hand, and other methods, on the other hand; this result reconfirms that HAMERLY falters [18] on dimensionality above 50.
where that overhead is more significant as part of the total cost due to fewer data points. Marigold performs the best overall.

5.2 Runtime

Having established the pattern of implementation-independent feature distance calculations, we examine wall-clock time, which is affected by operations we have hitherto ignored, including inequality checks, table look-ups, and comparisons.

Varying DCT features. Figure 4(a) shows how the total runtime varies vs. the number of DCT features used to represent the data with \( k = 10 \), with Python implementations. Performance follows the previously observed pattern, with two cardinal differences. First, gaps are now starker than before. Second, beyond the lowest dimensionality, the performance of Stepwise is consistently better than that of triangle-inequality-based methods. Even if Stepwise uses more features for point-to-centroid distance calculations, it has a smaller overhead of comparisons and bound updates; bound calculations involve calculating inter-centroid distances, used in both Elkan and Hamerly checks. Hamerly comes very close to Lloyd in terms of wall-clock time, as it is hard to compensate the overhead of keeping track of the nearest non-assigned centroid to each point, and nearest other centroid to each centroid. For the same reason, methods using Elkan bounds, which involve inter-centroid disances, fare worse then Stepwise. Marigold, benefiting from both forms of bounds, performs best.

Varying clusters. Figure 4(b) shows the runtime, per iteration and total, vs. the number of clusters \( k \) while using \( 128 \times 128 = 2^{14} \) DCT features. The results corroborate our previous observations. Moreover, the runtime of Hamerly approaches and even exceeds that of Lloyd as the number of clusters grows; that is reasonable, as more
These results indicate that geometry-based bounds are less helpful in large high-dimensional data sets. The advantage of Marigold is most pronounced on the gr_flake and MNIST data, which have more data points, reconfirming that Marigold manages high ratios of data points per cluster well.

6 CONCLUSIONS

The need to cluster high-dimensional data by $k$-means arises often in the natural sciences. Observing that the core $k$-means operation is a search for a point’s nearest centroid, we brought the know-how of nearest-neighbor search in high dimensions into $k$-means. We devised Marigold, an efficient algorithm for high-dimensional $k$-means that produces the same result as Lloyd’s algorithm. Marigold integrates triangle-inequality-based pruning with a powerful stepwise distance calculation method that progressively refines the data-representation granularity while maintaining tight upper and lower distance bounds to facilitate pruning centroid candidates. Our results demonstrate that Marigold reduces the volume of distance calculations and, consequently, runtime by approximately one order of magnitude, while its Stepwise component is also competitive when the data points per cluster ratio is low. Our results pave the way to real-time $k$-means in critical applications such as nanoARPES in condensed-matter physics.

ACKNOWLEDGMENTS

This research work was supported by VILLUM FONDEN (grant VIL40558) and partially by the Danish Council for Independent Research (grant DFF-1051-00062B). We also thank Bart Olsthorn and Alexander Balatsky for fruitful discussions on this topic.
REFERENCES