UNIVERSITÀ DEGLI STUDI DI PADOVA

Distributed k-Means

Implementation and performance analysis of distributed K-Means clustering algorithms

F. Bezzi, W. Conte, E. D'Amore, G. Gasparotto

September 18, 2025

Table of Contents

Table of Contents

UNIVERSITÀ DEGLI STUDI DI PADOVA

k-Means Clustering Objective

Università degli Studi di Padova

k-**Means problem** $\rightarrow$ find set of cluster centers C with $|C| = k$ that minimizes the **clustering cost** $\phi_X(C)$:

$$\min_C \sum_{x \in \mathcal{X}} \|f(C, x) - x\|^2,$$

where $f(C, x)$ returns the nearest cluster center in C to point x, using Euclidean distance.

Cost Function and Centroids

Let $\mathcal{X} = \{x_1, \dots, x_n\} \subset \mathbb{R}^d$ be the dataset and $C = \{c_1, \dots, c_k\} \subset \mathbb{R}^d$ the set of centers.

Centroid of a set $Y \subseteq \mathcal{X}$ (subset of the dataset) is given by the mean across all points:

$$\texttt{centroid}(Y) = \frac{1}{|Y|} \sum_{y \in Y} y$$

Cost of Y w.r.t. C:

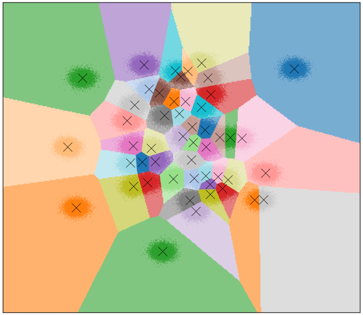
$$\phi_Y(C) = \sum_{y \in Y} \min_{c \in C} \|y - c\|^2$$

Lloyd's Iteration

Simplicity: start with a set of *randomly* chosen *initial centers*, repeatedly assign input points to nearest center, *recompute centers*.

Local search: Lloyd's iteration continues until convergence (early stopping).

Initialization: final solution is *locally optimal*. Better initialization algorithms get closer to *global optimum*.



Voronoi diagram of 2D-Gaussian Mixture synthetic dataset ($n = 10{,}000$ entries, $K = 50$ cluster centers, and variance $R = 100$). Labeling done using `KMeans` from `sklearn`.

k-Means++ Initialization

Algorithm: k-Means++ (k) Initialization

```
1. $C \leftarrow$ sample one point uniformly at random from $\mathcal{X}$
2. while $|C| < k$:
3.     Sample $x \in \mathcal{X}$ with probability $\frac{d^2(x,C)}{\phi_{\mathcal{X}}(C)}$
4.     $C \leftarrow C \cup \{x\}$
```

Algorithm is *sequential*

Computationally expensive depending on k: total running time $O(nkd)$.

k-Means$\|$ Initialization

Algorithm: k-Means$\|$ (k, ℓ) Initialization

```
1. $C \leftarrow$ sample one point uniformly at random from $\mathcal{X}$
2. $\psi \leftarrow \phi_{\mathcal{X}}(C)$
3. for $O(\log \psi)$ times:
4.     $C' \leftarrow$ sample $x \in \mathcal{X}$ independently with $p_x = \frac{\ell \cdot d^2(x, C)}{\phi_{\mathcal{X}}(C)}$
5.     $C \leftarrow C \cup C'$
6. For $x \in C$: assign weight $w_x$ as number of points in $\mathcal{X}$ closest to $x$
7. Recluster the weighted points in $C$ into $k$ clusters
```

Expected number of points in C is $\ell \log \psi$, typically more than k.

Mini-Batch k-Means Algorithm

Inputs: k, batch size b, iterations T, dataset $\mathcal{X}$
Steps:

```
1. Initialize $C$ (with $k$ random points from $\mathcal{X}$ or other init. algo.)
2. $\mathbf{v} \leftarrow \mathbf{0}$ (counts for updates)
3. for $t = 1$ to $T$:
    • Sample mini-batch $M \subset \mathcal{X}$ of size $b$
    • for $x \in M$:
        • Find closest center $c \in C$ and cache it
        • Update counts: $\mathbf{v}[c] \leftarrow \mathbf{v}[c] + 1$
    • for $c \in C$:
        • Learning rate: $\eta = \frac{1}{\mathbf{v}[c]}$
        • Update center: $c \leftarrow (1 - \eta)c + \eta x$
```

Avoids high *computational cost of full batch k-means* on large datasets.

Spark Overview

Spark Application:

- consists of a Driver program → runs user's main and executes parallel operations on a cluster.

Cluster:

- 1 master, 3 workers, 4 CPU cores and 6.8 GiB RAM per worker.
- Driver runs in the notebook process and orchestrates jobs.

Resilient Distributed Dataset (RDD):

- Slice dataset in `P` partitions to be distributed across the cluster.
- Operate on by:
  transformations → create a new dataset.
  actions → return a value to the driver program after running a computation on the dataset.

Spark Overview

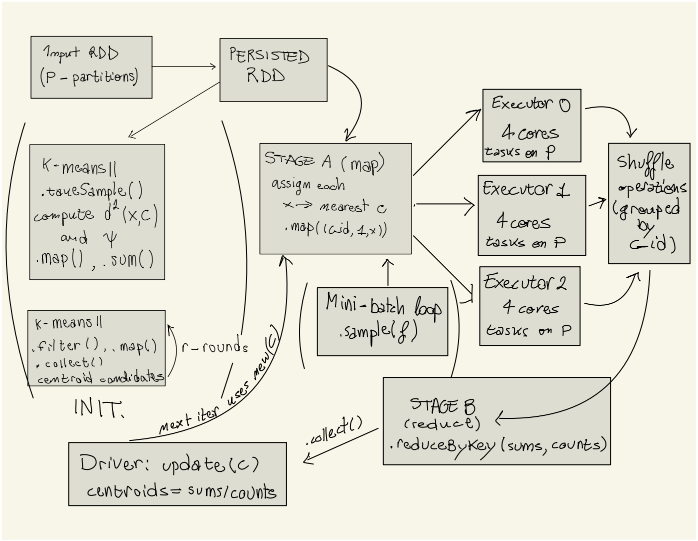
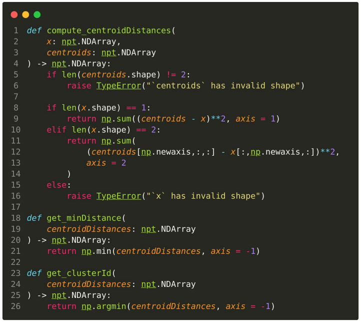


Table of Contents

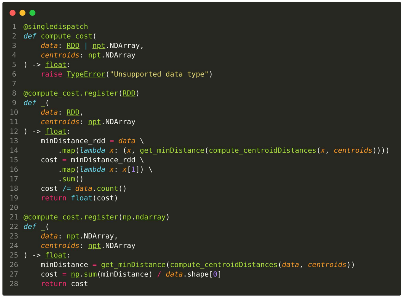
Basic functions



```python
def compute_centroidDistances(
    x: npt.NDArray,
    centroids: npt.NDArray
) -> npt.NDArray:
    if len(centroids.shape) != 2:
        raise TypeError("`centroids` has invalid shape")

    if len(x.shape) == 1:
        return np.sum((centroids - x)**2, axis = 1)
    elif len(x.shape) == 2:
        return np.sum(
            (centroids[np.newaxis,:,:] - x[:,np.newaxis,:])**2,
            axis = 2
        )
    else:
        raise TypeError("`x` has invalid shape")

def get_minDistance(
    centroidDistances: npt.NDArray
) -> npt.NDArray:
    return np.min(centroidDistances, axis = -1)

def get_clusterId(
    centroidDistances: npt.NDArray
) -> npt.NDArray:
    return np.argmin(centroidDistances, axis = -1)
```

- `compute_centroidDistances`: given data and centroids, for each point in data returns the distance to every centroid
- `get_minDistance`: given the distances to every centroid, returns the smallest one
- `get_clusterId`: given the distances to every centroid, returns the index of the smallest one, which is the corresponding `clusterId`

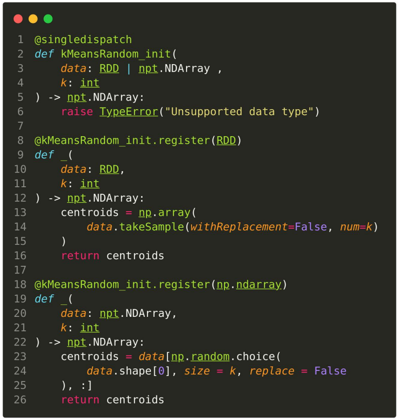
Basic functions



```python
@singledispatch
def compute_cost(
    data: RDD | npt.NDArray,
    centroids: npt.NDArray
) -> float:
    raise TypeError("Unsupported data type")

@compute_cost.register(RDD)
def _(
    data: RDD,
    centroids: npt.NDArray
) -> float:
    minDistance_rdd = data \
        .map(lambda x: (x, get_minDistance(compute_centroidDistances(x, centroids))))
    cost = minDistance_rdd \
        .map(lambda x: x[1]) \
        .sum()
    cost /= data.count()
    return float(cost)

@compute_cost.register(np.ndarray)
def _(
    data: npt.NDArray,
    centroids: npt.NDArray
) -> float:
    minDistance = get_minDistance(compute_centroidDistances(data, centroids))
    cost = np.sum(minDistance) / data.shape[0]
    return cost
```

- `cost` is computed as the sum of all the `minDistances`, normalized by the number of points

kMeansRandom

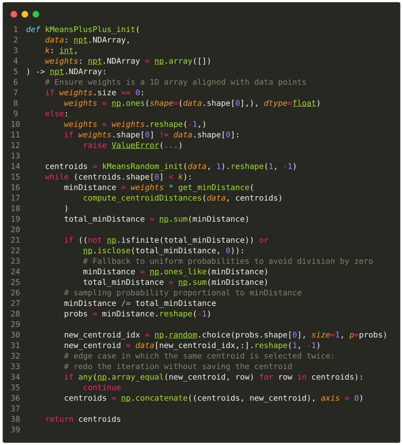


```python
@singledispatch
def kMeansRandom_init(
    data: RDD | npt.NDArray ,
    k: int
) -> npt.NDArray:
    raise TypeError("Unsupported data type")

@kMeansRandom_init.register(RDD)
def _(
    data: RDD,
    k: int
) -> npt.NDArray:
    centroids = np.array(
        data.takeSample(withReplacement=False, num=k)
    )
    return centroids

@kMeansRandom_init.register(np.ndarray)
def _(
    data: npt.NDArray,
    k: int
) -> npt.NDArray:
    centroids = data[np.random.choice(
        data.shape[0], size = k, replace = False
    ), :]
    return centroids
```

- simplest initialization algorithm: `centroids` are drawn uniformly at random from `data`
- parallel implementation: `takeSample` without replacement

kMeans++

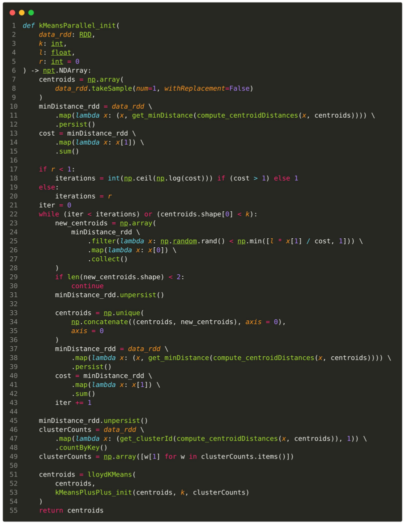
Università degli Studi di Padova



```python
def kMeansPlusPlus_init(
    data: npt.NDArray,
    k: int,
    weights: npt.NDArray = np.array([])
) -> npt.NDArray:
    # Ensure weights is a 1D array aligned with data points
    if weights.size == 0:
        weights = np.ones(shape=(data.shape[0],), dtype=float)
    else:
        weights = weights.reshape(-1,)
        if weights.shape[0] != data.shape[0]:
            raise ValueError(...)

    centroids = kMeansRandom_init(data, 1).reshape(1, -1)
    while (centroids.shape[0] < k):
        minDistance = weights * get_minDistance(
            compute_centroidDistances(data, centroids)
        )
        total_minDistance = np.sum(minDistance)

        if ((not np.isfinite(total_minDistance)) or
            np.isclose(total_minDistance, 0)):
            # Fallback to uniform probabilities to avoid division by zero
            minDistance = np.ones_like(minDistance)
            total_minDistance = np.sum(minDistance)
        # sampling probability proportional to minDistance
        minDistance /= total_minDistance
        probs = minDistance.reshape(-1)

        new_centroid_idx = np.random.choice(probs.shape[0], size=1, p=probs)
        new_centroid = data[new_centroid_idx,:].reshape(1, -1)
        # edge case in which the same centroid is selected twice:
        # redo the iteration without saving the centroid
        if any(np.array_equal(new_centroid, row) for row in centroids):
            continue
        centroids = np.concatenate((centroids, new_centroid), axis = 0)

    return centroids
```

- the sequential nature of the algorithm doesn't allow for a convenient parallel implementation
- variation with `weights`: each data point probability of being sampled is multiplied by its weight $\rightarrow$ used in `kMeans||`

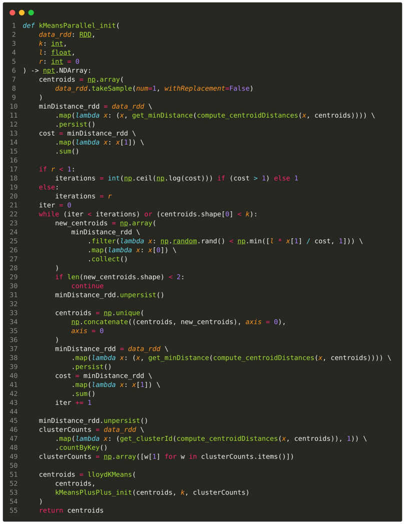
kMeans||



```python
def kMeansParallel_init(
    data_rdd: RDD,
    k: int,
    l: float,
    r: int = 0
) -> npt.NDArray:
    centroids = np.array(
        data_rdd.takeSample(num=1, withReplacement=False)
    )
    minDistance_rdd = data_rdd \
        .map(lambda x: (x, get_minDistance(compute_centroidDistances(x, centroids)))) \
        .persist()
    cost = minDistance_rdd \
        .map(lambda x: x[1]) \
        .sum()

    if r < 1:
        iterations = int(np.ceil(np.log(cost))) if (cost > 1) else 1
    else:
        iterations = r
    iter = 0
    while (iter < iterations) or (centroids.shape[0] < k):
        new_centroids = np.array(
            minDistance_rdd \
                .filter(lambda x: np.random.rand() < np.min([l * x[1] / cost, 1])) \
                .map(lambda x: x[0]) \
                .collect()
        )
        if len(new_centroids.shape) < 2:
            continue
        minDistance_rdd.unpersist()

        centroids = np.unique(
            np.concatenate((centroids, new_centroids), axis = 0),
            axis = 0
        )
        minDistance_rdd = data_rdd \
            .map(lambda x: (x, get_minDistance(compute_centroidDistances(x, centroids)))) \
            .persist()
        cost = minDistance_rdd \
            .map(lambda x: x[1]) \
            .sum()
        iter += 1

    minDistance_rdd.unpersist()
    clusterCounts = data_rdd \
        .map(lambda x: (get_clusterId(compute_centroidDistances(x, centroids)), 1)) \
        .countByKey()
    clusterCounts = np.array([w[1] for w in clusterCounts.items()])

    centroids = lloydKMeans(
        centroids,
        kMeansPlusPlus_init(centroids, k, clusterCounts)
    )
    return centroids
```

- designed with the MapReduce framework in mind: on each iteration multiple centroids are drawn independently
- `data_rdd` is an RDD where each row has a single element: a `np.ndarray`
    - benefits: more general approach to storage
    - downsides: executors require the `numpy` module to execute operations $\rightarrow$ compress and send virtual environment to executors

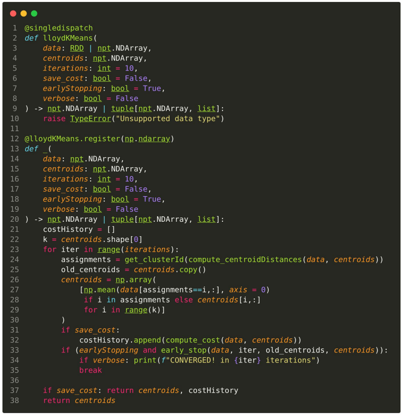
kMeans||



```python
def kMeansParallel_init(
    data_rdd: RDD,
    k: int,
    l: float,
    r: int = 0
) -> npt.NDArray:
    centroids = np.array(
        data_rdd.takeSample(num=1, withReplacement=False)
    )
    minDistance_rdd = data_rdd \
        .map(lambda x: (x, get_minDistance(compute_centroidDistances(x, centroids)))) \
        .persist()
    cost = minDistance_rdd \
        .map(lambda x: x[1]) \
        .sum()

    if r < 1:
        iterations = int(np.ceil(np.log(cost))) if (cost > 1) else 1
    else:
        iterations = r
    iter = 0
    while (iter < iterations) or (centroids.shape[0] < k):
        new_centroids = np.array(
            minDistance_rdd \
                .filter(lambda x: np.random.rand() < np.min([l * x[1] / cost, 1])) \
                .map(lambda x: x[0]) \
                .collect()
        )
        if len(new_centroids.shape) < 2:
            continue
        minDistance_rdd.unpersist()

        centroids = np.unique(
            np.concatenate((centroids, new_centroids), axis = 0),
            axis = 0
        )
        minDistance_rdd = data_rdd \
            .map(lambda x: (x, get_minDistance(compute_centroidDistances(x, centroids)))) \
            .persist()
        cost = minDistance_rdd \
            .map(lambda x: x[1]) \
            .sum()
        iter += 1

    minDistance_rdd.unpersist()
    clusterCounts = data_rdd \
        .map(lambda x: (get_clusterId(compute_centroidDistances(x, centroids)), 1)) \
        .countByKey()
    clusterCounts = np.array([w[1] for w in clusterCounts.items()])

    centroids = lloydKMeans(
        centroids,
        kMeansPlusPlus_init(centroids, k, clusterCounts)
    )
    return centroids
```

- `minDistance_rdd` gets cached as it is used for computing two different quantities during each iteration: `cost` and `new_centroids`
- `clusterCounts` represents the weight of each `centroid` (i.e. how many data points contains its cluster)

lloydKMeans

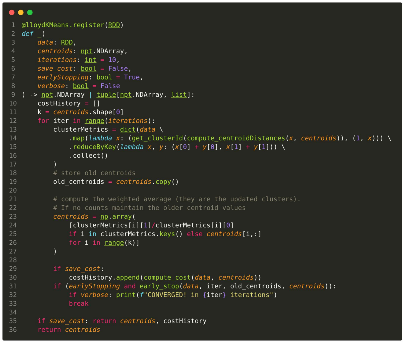


```python
@singledispatch
def lloydKMeans(
    data: RDD | npt.NDArray,
    centroids: npt.NDArray,
    iterations: int = 10,
    save_cost: bool = False,
    earlyStopping: bool = True,
    verbose: bool = False
) -> npt.NDArray | tuple[npt.NDArray, list]:
    raise TypeError("Unsupported data type")

@lloydKMeans.register(np.ndarray)
def _(
    data: npt.NDArray,
    centroids: npt.NDArray,
    iterations: int = 10,
    save_cost: bool = False,
    earlyStopping: bool = True,
    verbose: bool = False
) -> npt.NDArray | tuple[npt.NDArray, list]:
    costHistory = []
    k = centroids.shape[0]
    for iter in range(iterations):
        assignments = get_clusterId(compute_centroidDistances(data, centroids))
        old_centroids = centroids.copy()
        centroids = np.array(
            [np.mean(data[assignments==i,:], axis = 0)
             if i in assignments else centroids[i,:]
             for i in range(k)]
        )
        if save_cost:
            costHistory.append(compute_cost(data, centroids))
        if (earlyStopping and early_stop(data, iter, old_centroids, centroids)):
            if verbose: print(f"CONVERGED! in {iter} iterations")
            break

    if save_cost: return centroids, costHistory
    return centroids
```

- on each iteration all `data` points are considered in the update of the `centroids`
- `cost` monotonically decreases as iterations pass

lloydKMeans - Parallel implementation

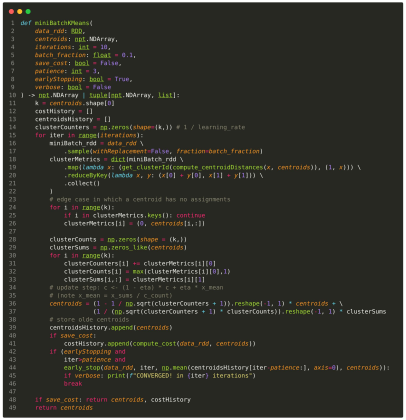
UNIVERSITÀ DEGLI STUDI DI PADOVA



```python
@lloydKMeans.register(RDD)
def _(
    data: RDD,
    centroids: npt.NDArray,
    iterations: int = 10,
    save_cost: bool = False,
    earlyStopping: bool = True,
    verbose: bool = False
) -> npt.NDArray | tuple[npt.NDArray, list]:
    costHistory = []
    k = centroids.shape[0]
    for iter in range(iterations):
        clusterMetrics = dict(data \
            .map(lambda x: (get_clusterId(compute_centroidDistances(x, centroids)), (1, x))) \
            .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) \
            .collect()
        )
        # store old centroids
        old_centroids = centroids.copy()

        # compute the weighted average (they are the updated clusters).
        # If no counts maintain the older centroid values
        centroids = np.array(
            [clusterMetrics[i][1]/clusterMetrics[i][0]
            if i in clusterMetrics.keys() else centroids[i,:]
            for i in range(k)]
        )

        if save_cost:
            costHistory.append(compute_cost(data, centroids))
        if (earlyStopping and early_stop(data, iter, old_centroids, centroids)):
            if verbose: print(f"CONVERGED! in {iter} iterations")
            break

    if save_cost: return centroids, costHistory
    return centroids
```

- natural porting of the classic Lloyd's algorithm to the parallel framework
- `reduceByKey` allows the parallel computation of the new centroid

miniBatchKMeans



```python
def miniBatchKMeans(
    data_rdd: RDD,
    centroids: npt.NDArray,
    iterations: int = 10,
    batch_fraction: float = 0.1,
    save_cost: bool = False,
    patience: int = 3,
    earlyStopping: bool = True,
    verbose: bool = False
) -> npt.NDArray | tuple[npt.NDArray, list]:
    k = centroids.shape[0]
    costHistory = []
    centroidsHistory = []
    clusterCounters = np.zeros(shape=(k,)) # 1 / learning_rate
    for iter in range(iterations):
        miniBatch_rdd = data_rdd \
            .sample(withReplacement=False, fraction=batch_fraction)
        clusterMetrics = dict(miniBatch_rdd \
            .map(lambda x: (get_clusterId(compute_centroidDistances(x, centroids)), (1, x))) \
            .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) \
            .collect()
        )
        # edge case in which a centroid has no assignments
        for i in range(k):
            if i in clusterMetrics.keys(): continue
            clusterMetrics[i] = (0, centroids[i,:])

        clusterCounts = np.zeros(shape = (k,))
        clusterSums = np.zeros_like(centroids)
        for i in range(k):
            clusterCounters[i] += clusterMetrics[i][0]
            clusterCounts[i] = max(clusterMetrics[i][0],1)
            clusterSums[i,:] = clusterMetrics[i][1]
        # update step: c <- (1 - eta) * c + eta * x_mean
        # (note x_mean = x_sums / c_count)
        centroids = (1 - 1 / np.sqrt(clusterCounters + 1)).reshape(-1, 1) * centroids + \
                    (1 / (np.sqrt(clusterCounters + 1) * clusterCounts)).reshape(-1, 1) * clusterSums
        # store olde centroids
        centroidsHistory.append(centroids)
        if save_cost:
            costHistory.append(compute_cost(data_rdd, centroids))
        if (earlyStopping and
            iter>patience and
            early_stop(data_rdd, iter, np.mean(centroidsHistory[iter-patience:], axis=0), centroids)):
            if verbose: print(f"CONVERGED! in {iter} iterations")
            break

    if save_cost: return centroids, costHistory
    return centroids
```

- a fraction of the dataset is considered on each iteration
- learning rate shrinks as iterations pass
- our modification: $\eta \sim v^{-1/2}$ instead of $\eta \sim v^{-1}$. This leads to slower shrinking and, generally, improved learning (i.e. lower cost)

Table of Contents

Introduction to the initialization analysis

UNIVERSITÀ DEGLI STUDI DI PADOVA

The goal is to evaluate the performance of different initialization strategies in terms of **initialization time** and **clustering cost**:

- **Random**
- **k-Means++**
- **k-Means**$\|$ with $\ell \cdot k = 0.5$
- **k-Means**$\|$ with $\ell \cdot k = 2$

Each initialization is followed by Lloyd's iterations.

This setup evaluates whether the **parallelization** of **k-Means**$||$ improves clustering, using **Random** and **k-Means++** as baselines. We compare two metrics:

- **Execution time**: Initialization + Lloyd updates
- **Final cost**: $C = \frac{1}{|Y|} \sum_{y \in Y} \min_{i=1,\dots,k} \|y - c_i\|^2$

GaussianMixture dataset

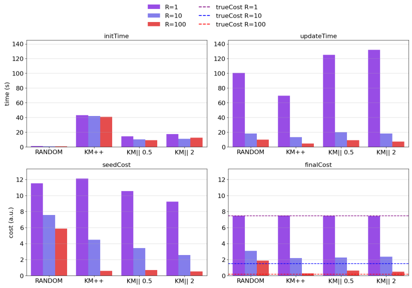
- Synthetic dataset
- k centers sampled from a 15-dimensional spherical Gaussian:

$$\mathcal{N}(0, R), \quad R \in \{1, 10, 100\}.$$

- Around each center, points are Normally distributed
- Result: a mixture of k spherical Gaussians with equal weights

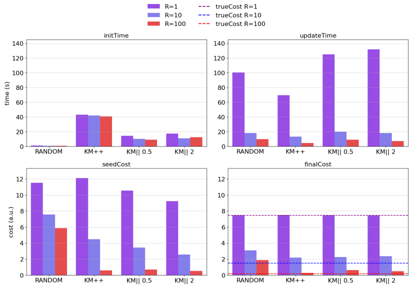
$\Longrightarrow$ Regular dataset with well separated clusters boundaries

Cost and Time analysis



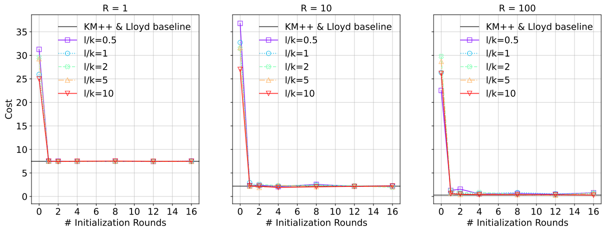
- **Random:** fastest initialization but with an high cost due to stochastic centroid selection
- **k-Means++:** achieves lower cost, but initialization is slower because of its sequential nature
- **k-Means**||: parallelization provides fast initialization and comparable or better cost

Cost and Time analysis: effect of R



- Higher R means that centers are farther apart and clusters are better separated $\Rightarrow$ clustering becomes easier: **lower seed and final costs** as R increases

Cost vs Rounds (GaussianMixture)



- Cost decreases rapidly within the first rounds ⇒ few iterations are sufficient to approach the baseline of k-Means++
- Larger oversampling ratio (ℓ/k) slightly improves centroid coverage

$\Longrightarrow$ **Parallelization is effective:** low and fast initialization cost, scalable for large datasets

KDDcup 1999 dataset (10%)

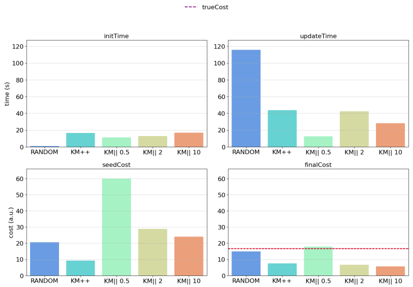
Now our goal is to demonstrate whether these improvements still hold on massive, real-world datasets.

KDDcup99:

- Real-world dataset for anomaly detection in network traffic
- Contains 5M records
- Each record: 41 features
- Clusters are not well-separated as before

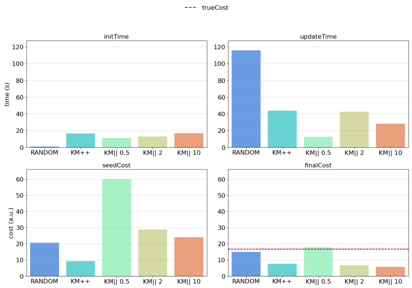
$\Longrightarrow$ Clustering is harder: noisy, heterogeneous, overlapping data

Seed and Final cost



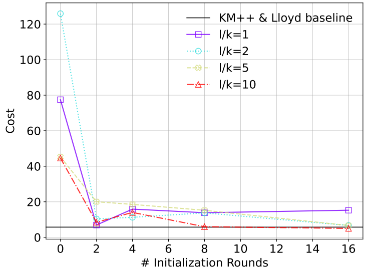
- **k-Means**$\|$: higher seed cost than **k-Means++**, but improves as ℓ/k increases (since we are sampling more points for each centroid k)
- Data heterogeneity makes it harder for parallel sampling to consistently capture the best centers
- All methods converge to similar final costs after Lloyd iterations

Initialization and update time



- **Init Time**:
  - **Random**: almost zero.
  - **k-Means++**: slowest
  - **k-Means||**: more time to sample as ℓ/k increases but better cost
- Lloyd iterations dominate the cost, so gains from faster initialization are less visible overall
- **k-Means||** offers a good balance between time and cost, scaling better than k-Means++

Cost vs Rounds



- The cost decreases sharply in the first rounds and quickly approaches the baseline
- Increasing ℓ/k improves stability but does not change the overall trend

Introduction to the update analysis

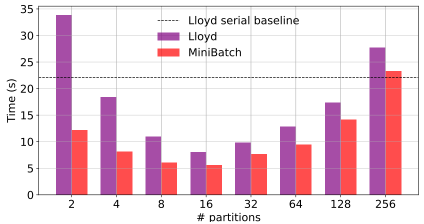
In the **“update”** analysis, we focus on two properties to evaluate algorithms:

- Execution time
- Iterations required to reach convergence

Why?

$\rightarrow$ **Time scaling with the number of partitions**
$\rightarrow$ **Convergence differences/similarities for each algorithm**

time vs $N_{partitions}$

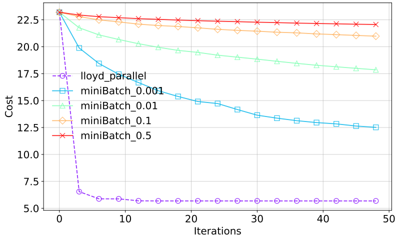


- Minibatch K-Means is **more efficient** than Lloyd-parallel.
- $N_{partitions} = 16$: $\rightarrow$ more than a $2\times$ speedup compared to Lloyd-serial.
- Choosing too many or too few $N_{partitions}$ $\rightarrow$ worse performance than Lloyd-serial.

Why this pattern?

- **Too few partitions:**
  - Large tasks, poor load balancing
  - Limited parallelism, resources underutilized
- **Too many partitions:**
  - Smaller tasks $\rightarrow$ proportionally higher **overhead**
  - Overhead sources: scheduling, communication, coordination
  - Overhead dominates computation

Cost vs iterations



- Lloyd-parallel converges “faster” in terms of iterations.
- Minibatch K-Means is slower and appears to approach different (and worse) “steady states”.
- For Minibatch K-Means, convergence improves when using smaller batch fractions.

$\rightarrow$ **Minibatch K-Means is more time efficient, but requires more iterations w.r.t. Lloyd-parallel**

Remark on Minibatch update

Looking at the centroids update formula for the Minibatch K-Means:

$$c_j^{(t+1)} = (1 - \eta_t)\, c_j^{(t)} + \eta_t\, \bar{x}_j^{(t)}$$

Where:

- η_t is the learning rate used for the update of centroid j at iteration t.
- η_t is proportional to $\left(N_j^{\text{samples}}\right)^{-\frac{1}{2}}$

Larger batch fractions $\rightarrow$ $\boldsymbol{\eta_t}$ drops close to zero more quickly.

$$\downarrow$$

Smaller centroid updates over time $\rightarrow$ slower convergence and poorer minimization of the cost function.

Final Remarks and further improvements

- Overall, the distributed versions of the algorithm are more efficient than the serial one when dealing with large datasets.
- There appears to be a tradeoff between time efficiency and cost minimization for both the initialization and update algorithms.
- Regarding initialization, it would be interesting to repeat the analysis with a dataset containing more centroids, in order to observe clearer performance improvements.
- With sufficient computing resources, an analysis on a larger dataset could also be valuable for the update step, where minibatch K-means generally outperformed the parallel version (assuming an appropriate number of partitions).