机器学习的数学理论课程项目作业：动态 Kmeans

王英枭
211840213
wangbottlecap@gmail.com

June 19, 2025

Abstract

本报告是 25 Spring 数学学院课程《机器学习的数学理论》项目作业。针对给定的数据集，我们首先实现了经典 Kmeans, Kmeans++, Hartigan & Wong 算法, Elkan 加速 Kmeans, miniBatch Kmeans, Bisecting Kmeans 等多种算法并给出它们的主要思想和算法概述。随后，我们引入动态 kmeans 的概念，考虑在动态情形下对数据进行动态聚类，并给出对应的分析。动态聚类的所有代码作为附录在报告的最后给出。

Contents

1 Classical Kmeans

1.1 问题背景与符号表示

令数据集记为

$$\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d,$$

其中 $\mathbf{x}_i \in \mathbb{R}^d$ 表示第 i 个样本，n 表示总样本数，d 表示特征维度。我们希望将 $\mathcal{X}$ 划分为 k 个不相交的子集（簇）

$$S_1, S_2, \ldots, S_k, \quad \bigcup_{i=1}^{k} S_i = \mathcal{X}, \quad S_i \cap S_j = \varnothing \ (i \neq j),$$

并为每个簇 S_i 找到一个中心 $\boldsymbol{\mu}_i \in \mathbb{R}^d$，以最小化以下目标函数（簇内平方误差和）：

$$J\big(S_1, \ldots, S_k; \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\big) \ = \ \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2.$$

1.2 目标函数的分块优化思想

KMeans 算法通过分块坐标下降（Block Coordinate Descent）策略，将优化拆分为两个交替步骤：

1. 给定当前中心 $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$，求解最优的簇分配 $S_1, \ldots, S_k$。
2. 给定当前簇分配 $S_1, \ldots, S_k$，求解最优的聚类中心 $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$。

通过在这两个步骤间迭代更新，我们不断降低目标函数，最终在有限步内收敛到某个局部最优解。

1.2.1 给定中心后的最优分配

设当前中心为 $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\}$，则最优的分配策略是将每个样本分到离它最近的中心所在的簇：

$$\mathbf{x} \in S_i \quad \Longleftrightarrow \quad i = \arg \min_{j \in \{1, \ldots, k\}} \|\mathbf{x} - \boldsymbol{\mu}_j\|.$$

1.2.2 给定分配后的最优中心

设当前的簇分配结果为 $S_1, \ldots, S_k$，则目标函数可表示为

$$J = \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 \ = \ \sum_{i=1}^{k} J_i,$$

其中 $J_i = \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2$. 要分别最小化各 J_i，只需对各簇独立求解，即

$$\boldsymbol{\mu}_i^* = \arg \min_{\boldsymbol{\mu}_i} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2.$$

可以证明此时的最优解是簇内所有样本的算术平均值：

$$\boldsymbol{\mu}_i^* = \frac{1}{|S_i|} \sum_{\mathbf{x} \in S_i} \mathbf{x}.$$

1.3 经典 KMeans 算法迭代流程

基于上述分块最优思想，KMeans 算法流程可概括如下：

1. **初始化**：随机选择 k 个样本作为初始中心，记为 $\boldsymbol{\mu}_1^{(0)}, \ldots, \boldsymbol{\mu}_k^{(0)}$。
2. **迭代**：对于第 t 次迭代：

   (a) 分配样本（Assignment Step）：

   $$S_i^{(t)} = \left\{ \mathbf{x} \mid i = \arg\min_j \|\mathbf{x} - \boldsymbol{\mu}_j^{(t-1)}\| \right\}, \quad i = 1, \ldots, k.$$

   (b) 更新中心（Update Step）：

   $$\boldsymbol{\mu}_i^{(t)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{\mathbf{x} \in S_i^{(t)}} \mathbf{x}, \quad i = 1, \ldots, k.$$

3. **收敛或终止条件**：若中心更新的变化量足够小（小于某个阈值），或达到最大迭代次数，停止迭代并输出结果。

2 Kmeans++

2.1 核心思想

与经典 KMeans 相同，令数据集

$$\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d,$$

其中 $\mathbf{x}_i \in \mathbb{R}^d$ 表示第 i 个样本。我们希望选取 k 个“初始中心”，以改善后续 KMeans 聚类的质量和收敛速度。KMeans++ 的核心思想是：

- 第一个中心等概率地从数据集中随机挑选；
- 后续中心依照与已选中心距离平方成正比的概率进行抽样，距离越远的点被选为新中心的概率越大。

这一策略能够让初始中心彼此分散，从而加速收敛并减少陷入不良局部最优的风险。

2.2 初始化算法的数学步骤

为选取第 k 个中心（假设已经选出 $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_{k-1}$）时，执行以下步骤：

1. **计算每个点到最近已选中心的距离平方**：对每个 $\mathbf{x}_i \in \mathcal{X}$，定义

$$D(\mathbf{x}_i) \;=\; \min_{1 \le j \le k-1} \|\mathbf{x}_i - \boldsymbol{\mu}_j\|^2.$$

2. **基于距离平方构建抽样分布**：

$$P(\mathbf{x}_i) \;=\; \frac{D(\mathbf{x}_i)}{\sum_{r=1}^{n} D(\mathbf{x}_r)}, \quad i = 1, 2, \ldots, n.$$

3. **按此分布随机抽样**：随机生成一个 $\mathbf{x}_q$ 使 $\Pr[q = i] = P(\mathbf{x}_i)$，并令 $\boldsymbol{\mu}_k = \mathbf{x}_q$.

当所有 k 个中心均选出后，便得到一组初始中心。

2.3 算法描述

Algorithm 1 KMeans++ Initialization

Require: Data set $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$, number of centers k.
Ensure: k initial centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$.

Randomly pick one sample $\mathbf{x}_p$ from $\mathcal{X}$, set $\boldsymbol{\mu}_1 \leftarrow \mathbf{x}_p$.
for $t = 2 \to k$ **do**
    For each i, $D(\mathbf{x}_i) \leftarrow \min_{1 \le j \le t-1} \|\mathbf{x}_i - \boldsymbol{\mu}_j\|^2$.
    $P(\mathbf{x}_i) \leftarrow \dfrac{D(\mathbf{x}_i)}{\sum_{r=1}^{n} D(\mathbf{x}_r)}$.
    Sample $\mathbf{x}_q$ from $\mathcal{X}$ with probability $P(\mathbf{x}_i)$.
    $\boldsymbol{\mu}_t \leftarrow \mathbf{x}_q$
end for
return $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$

3 Hartigan & Wong (1979)

3.1 逐点搬移的思想

设当前数据集为

$$\mathcal{X} \;=\; \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d,$$

被划分成 k 个簇 $S_1, S_2, \ldots, S_k$，对应中心为 $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_k$。我们考虑将单个样本 $\mathbf{x}$ 从其当前所在簇 A 移动到另一个簇 B 时，对目标函数

$$J \;=\; \sum_{i=1}^{k} \sum_{\mathbf{z} \in S_i} \|\mathbf{z} - \boldsymbol{\mu}_i\|^2$$

所带来的变化量。具体地，令 S_A, S_B 分别表示簇 A 和簇 B 的成员集合：

$$S_A' = S_A \setminus \{\mathbf{x}\}, \quad S_B' = S_B \cup \{\mathbf{x}\}.$$

相应的新中心记为

$$\boldsymbol{\mu}_A' = \frac{1}{|S_A| - 1} \sum_{\mathbf{y} \in S_A \setminus \{\mathbf{x}\}} \mathbf{y}, \quad \boldsymbol{\mu}_B' = \frac{1}{|S_B| + 1} \left(\sum_{\mathbf{z} \in S_B} \mathbf{z} + \mathbf{x} \right).$$

若将样本 $\mathbf{x}$ 从簇 A 搬移到簇 B 后导致目标函数值严格下降，即

$$\Delta = \left(\sum_{\mathbf{z} \in S_A'} \|\mathbf{z} - \boldsymbol{\mu}_A'\|^2 + \sum_{\mathbf{z} \in S_B'} \|\mathbf{z} - \boldsymbol{\mu}_B'\|^2 \right) - \left(\sum_{\mathbf{z} \in S_A} \|\mathbf{z} - \boldsymbol{\mu}_A\|^2 + \sum_{\mathbf{z} \in S_B} \|\mathbf{z} - \boldsymbol{\mu}_B\|^2 \right) < 0,$$

则执行此“逐点搬移”可使 J 单调下降，从而向局部最优解靠近。

3.2 算法步骤

Algorithm 2 Simple Hartigan & Wong KMeans (Point Reassignment)

Require: Data set $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, number of clusters k, max iterations M.
Ensure: Final cluster centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ and assignments $S_1, \ldots, S_k$.

Initialize centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ using some method (*e.g.*, KMeans++).
Assign each $\mathbf{x}_i$ to the nearest center.
for $t = 1 \to M$ **do**
    changed $\leftarrow$ False
    for each sample $\mathbf{x}$ in $\mathcal{X}$ **do**
        Let A be the cluster containing $\mathbf{x}$, and $\boldsymbol{\mu}_A$ be its center.
        $\delta_{\text{best}} \leftarrow 0, \quad B_{\text{best}} \leftarrow A$
        for each cluster $B \neq A$ **do**
            Compute the new centers if $\mathbf{x}$ were moved from A to B:

$$\boldsymbol{\mu}'_A \quad \text{and} \quad \boldsymbol{\mu}'_B.$$

            Evaluate $\Delta = J_{\text{after}} - J_{\text{before}}$ for the local change (see text).
            if $\Delta < \delta_{\text{best}}$ **then**
                $\delta_{\text{best}} \leftarrow \Delta$
                $B_{\text{best}} \leftarrow B$
            end if
        end for
        if $B_{\text{best}} \neq A$ **then**
            Move $\mathbf{x}$ from A to B_{best}, update $\boldsymbol{\mu}_A$ and $\boldsymbol{\mu}_{B_{\text{best}}}$.
            changed $\leftarrow$ True
        end if
    end for
    if not *changed* **then**
        break
    end if
end for
return final centers $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\}$ and cluster sets $\{S_1, \ldots, S_k\}$.

4 Elkan 加速版

4.1 主要思想

与经典 KMeans 相同，假设数据集为

$$\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d,$$

聚类中心为 $\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_k$, 目标函数为

$$J = \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{c}_i\|^2.$$

Elkan 加速版的关键在于利用对偶边界（上下界）和三角不等式减少不必要的距离计算。大体上，为每个样本 $\mathbf{x}$ 维护：

- **上界** $u(\mathbf{x})$：估计 $\mathbf{x}$ 与其当前分配中心 $\boldsymbol{c}_{\text{assigned}(\mathbf{x})}$ 的距离,

$$u(\mathbf{x}) \;\geq\; \|\mathbf{x} - \boldsymbol{c}_{\text{assigned}(\mathbf{x})}\|.$$

- **下界** $l_j(\mathbf{x})$：对任意中心 $\boldsymbol{c}_j$ 的距离,

$$l_j(\mathbf{x}) \;\leq\; \|\mathbf{x} - \boldsymbol{c}_j\|.$$

记中心间的距离为

$$d_{ij} = \|\boldsymbol{c}_i - \boldsymbol{c}_j\|.$$

若对样本 $\mathbf{x}$ 有

$$u(\mathbf{x}) < \frac{d_{ij}}{2},$$

则可推断 $\|\mathbf{x} - \boldsymbol{c}_j\| > \|\mathbf{x} - \boldsymbol{c}_i\|$, 因此中心 j 不会是 $\mathbf{x}$ 更好的分配对象，无需计算真实距离。三角不等式保障了此结论的正确性，从而实现大规模剪枝。此外，当中心 $\boldsymbol{c}_i$ 在两次迭代间移动量 m_i 较小，则可依据

$$u(\mathbf{x}) \leftarrow u(\mathbf{x}) + m_i$$

等方式修正上界，而依旧保留安全的剪枝判断。此即 Elkan 加速算法的根本逻辑：**以有限的上/下界更新与三角不等式将大量“无关距离”排除在外**，从而减少时间复杂度。

4.2 算法框架

Algorithm 3 Elkan Accelerated KMeans

Require: Data set $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, number of clusters k, max iterations M.

Ensure: Cluster centers $\{\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k\}$ and partition $S_1, \ldots, S_k$.

Initialize centers $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k$ (e.g., KMeans++), and assign each sample to the nearest center.
For each $\mathbf{x}_i$, set $u(\mathbf{x}_i) \leftarrow \|\mathbf{x}_i - \boldsymbol{c}_{\text{assigned}(\mathbf{x}_i)}\|$, and initialize $l_j(\mathbf{x}_i)$ for $j \neq \text{assigned}(\mathbf{x}_i)$.
for $t = 1 \to M$ **do**
    Update bounds: compute center movements $m_i = \|\boldsymbol{c}_i^{(\text{old})} - \boldsymbol{c}_i^{(\text{new})}\|$ and correct $u(\mathbf{x})$ or $l_j(\mathbf{x})$ accordingly.
    Assignment step:
    for each sample $\mathbf{x}_i$ **do**
        Let $i_0 = \text{assigned}(\mathbf{x}_i)$.
        for each center $j \neq i_0$ **do**
            Pruning check: if $u(\mathbf{x}_i) < d_{i_0 j}/2$, skip distance computation for center j.
            Otherwise, compute $\|\mathbf{x}_i - \boldsymbol{c}_j\|$ and update $u(\mathbf{x}_i), l_j(\mathbf{x}_i)$ if needed.
        end for
        Re-assign $\mathbf{x}_i$ to the closest center found if it differs from i_0.
    end for
    Update step: recalculate each center $\boldsymbol{c}_j \leftarrow \frac{1}{|S_j|} \sum_{\mathbf{x} \in S_j} \mathbf{x}$.
    Check convergence: if center changes are small enough or no re-assignments, break.
end for
return $\{\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k\}$ and final partition $\{S_1, \ldots, S_k\}$.

5 Mini-Batch KMeans

5.1 主要思想

与经典 KMeans 相同，假设数据集

$$\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d,$$

我们希望寻找 k 个中心 $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ 并将 $\mathcal{X}$ 划分为 k 个不相交子集 $S_1, \ldots, S_k$，使得以下目标函数（簇内平方误差和）最小化：

$$\min_{S_1,\ldots,S_k;\, \boldsymbol{\mu}_1,\ldots,\boldsymbol{\mu}_k} \sum_{j=1}^{k} \sum_{\mathbf{x} \in S_j} \|\mathbf{x} - \boldsymbol{\mu}_j\|^2.$$

然而，在大规模数据场景或流式数据场景中，每次迭代都处理全部样本 $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ 成本非常高。为此，Mini-Batch KMeans 在每次迭代只抽取一个 **小批量**（mini-batch），记为

$$B^{(t)} = \{\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_m}\} \subset \mathcal{X}, \quad |B^{(t)}| = m \ll n,$$

并执行类似 KMeans 的分配与更新步骤，但仅基于 $B^{(t)}$ 更新当前的中心 $\{\boldsymbol{\mu}_j\}$。

令在 $B^{(t)}$ 中分配到簇 j 的那些样本记为

$$B_j^{(t)} \;=\; \left\{\mathbf{x} \in B^{(t)} \mid \text{assigned to } \boldsymbol{\mu}_j\right\},$$

则 $\boldsymbol{\mu}_j$ 的更新常采取如下形式：

$$\boldsymbol{\mu}_j^{(t+1)} \;=\; \boldsymbol{\mu}_j^{(t)} \;+\; \eta\left(\bar{\mathbf{x}}_{\text{batch},\,j} \;-\; \boldsymbol{\mu}_j^{(t)}\right),$$

其中

$$\bar{\mathbf{x}}_{\text{batch},\,j} \;=\; \frac{1}{\left|B_j^{(t)}\right|}\sum_{\mathbf{x}\in B_j^{(t)}} \mathbf{x}, \quad \eta \in (0,1]$$

为更新步长（又可视为学习率）。这使得中心向本批次样本的均值方向移动，而无需处理全部 n 个样本。

通过在每次迭代使用较小规模的批数据 $B^{(t)}$，算法单次迭代的复杂度降至 $O(mk)$，明显低于经典 KMeans 的 $O(nk)$；同时在多轮迭代后，聚类解可近似逼近全量 KMeans 的结果。Mini-Batch KMeans 在大规模数据或在线学习环境中非常实用，能在保证一定精度下显著降低计算开销。

5.2 算法框架

Algorithm 4 Mini-Batch KMeans

Require: Data set $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$, number of clusters k, mini-batch size m, max iterations M.
Ensure: Final centers $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\}$ and assignments $S_1, \ldots, S_k$.

Initialize centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ (e.g., random or KMeans++).
for $t = 1 \to M$ **do**
  Sample a mini-batch $B^{(t)} = \{\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_m}\} \subset \mathcal{X}$.
  Assignment step (on mini-batch):
  for each $\mathbf{x} \in B^{(t)}$ **do**
    Find $j^* = \arg\min_j \|\mathbf{x} - \boldsymbol{\mu}_j\|^2$.
    Assign $\mathbf{x}$ to cluster j^*.
  end for
  Update step (on mini-batch):
  for $j = 1 \to k$ **do**
    Let $B_j^{(t)}$ be the subset of $B^{(t)}$ assigned to center j.
    if $B_j^{(t)} \neq \varnothing$ **then**
      $\overline{\mathbf{x}}_j^{(t)} \leftarrow \frac{1}{|B_j^{(t)}|} \sum_{\mathbf{x} \in B_j^{(t)}} \mathbf{x}$.
      $\boldsymbol{\mu}_j \leftarrow \boldsymbol{\mu}_j + \eta \left(\overline{\mathbf{x}}_j^{(t)} - \boldsymbol{\mu}_j \right)$,
        where η *is a learning rate in* $(0, 1]$.
    end if
  end for
  (Optional) Check convergence if center movement is small or no changes.
end for
Final assignment: reassign every $\mathbf{x}_i \in \mathcal{X}$ to its nearest center $\boldsymbol{\mu}_j$.
return $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\}$ and $S_1, \ldots, S_k$.

6 Bisecting KMeans

6.1 主要思想

Bisecting KMeans（又称二分 KMeans）将 KMeans 与层次聚类思想结合，通过 **自顶向下** (top-down) 的分裂方式得到 k 个簇。初始时将所有数据视为一个大簇，然后反复挑选其中“最需要细分”的簇，做 2-means 将其分成两个簇，直至达到 k 个簇。

具体而言，令 $\mathcal{X}$ 初始在一个簇 $S_{\text{root}} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ 中，中心

$$\boldsymbol{\mu}_{\text{root}} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i.$$

若当前聚类数小于 k，则：

1. 选出一个簇 $S_{\max}$（可根据簇内误差最大的原则），其中包含若干样本 $\{\mathbf{x}_{\alpha_1}, \ldots, \mathbf{x}_{\alpha_r}\}$。

2. 在该簇上运行 **2-means**（$k = 2$ 的 KMeans），解为

$$\min_{S_a, S_b, \boldsymbol{\mu}_a, \boldsymbol{\mu}_b} \left(\sum_{\mathbf{x} \in S_a} \|\mathbf{x} - \boldsymbol{\mu}_a\|^2 + \sum_{\mathbf{x} \in S_b} \|\mathbf{x} - \boldsymbol{\mu}_b\|^2 \right),$$

   使 $S_a \cup S_b = S_{\max}$, $S_a \cap S_b = \varnothing$。

3. 用 S_a 与 S_b 替换 $S_{\max}$，整体簇数加 1。

通过这种不断“二分”最大的簇，Bisecting KMeans 最终可得到 k 个簇。其核心思想是：

逐次选出最需要被拆分的簇，用2-means 精细地对它进行划分。

在每次二分时，仍遵循 KMeans 的优化目标，即最小化簇内误差和，只不过将问题限制在某个子簇 $S_{\max}$ 上。这种自顶向下的层次聚类与 KMeans 结合的方式在实践中具有较好的可解释性和灵活度，同时每次只在局部簇做细分，能在一定程度上减少计算量。虽然与一次性 k-means 一样只能得到局部最优，但常能更好地针对层次结构做可视化与分析。

6.2 算法框架

Algorithm 5 Bisecting KMeans (二分聚类)

Require: Data set $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$, number of clusters k, max iterations for each 2-means M_2.
Ensure: k final clusters $S_1, \ldots, S_k$ with centers $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\}$.

Initialize a single cluster $S_{\text{root}} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$.
Let the set of active clusters $C \leftarrow \{S_{\text{root}}\}$.
while $|C| < k$ **do**
    Select the cluster $S_{\max} \in C$ with the largest SSE (or largest cardinality).
    Remove $S_{\max}$ from C.
    Run 2-means on $S_{\max}$ (e.g., use Lloyd or Mini-Batch with $k = 2$, M_2 iterations):
        Split $S_{\max}$ into two subsets S_a and S_b with centers $\boldsymbol{\mu}_a$ and $\boldsymbol{\mu}_b$.
    Add S_a and S_b to C.
end while
Let $\{S_1, \ldots, S_k\} = C$, and compute each center as $\boldsymbol{\mu}_j \leftarrow \frac{1}{|S_j|} \sum_{\mathbf{x} \in S_j} \mathbf{x}$.
return $S_1, \ldots, S_k$ and $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\}$.

7 动态聚类

7.1 问题描述

随着海量数据以流式方式到达，本节所述算法旨在对数据进行动态聚类。假设数据流被划分为等长批次，每批含 $B = 100$ 条样本。记在时刻 t 到达的新批次为 $\Delta\mathcal{D}_t$，总体数据集为

$$\mathcal{D}_t = \bigcup_{s=1}^{t} \Delta\mathcal{D}_s.$$

我们要求：

- 每批到达后立即更新聚类结构；
- 保证每个簇包含的数据条数在 $[N_{\min}, N_{\max}] = [60, 150]$ 范围内；
- 若某簇大小超过 $N_{\max}$，则将其拆分为两个子簇；
- （可扩展）若删除过期数据后某簇小于 $N_{\min}$，则与最近邻簇合并。

7.2 数学建模

令第 t 时刻的聚类中心集合为

$$C_t = \{\, c_{t,1}, \ldots, c_{t,K_t} \},\quad c_{t,i} \in \mathbb{R}^d,$$

样本点集合被分配至簇 $\mathcal{D}_{t,i} = \{\, x \mid \text{assign}(x) = i \}$。我们仍以 KMeans 惯性（Inertia）作为损失函数：

$$\mathcal{L}_t \;=\; \sum_{i=1}^{K_t} \sum_{x \in \mathcal{D}_{t,i}} \|x - c_{t,i}\|^2.$$

动态更新包含以下运算：

1. **增量指派**：对新到达点 $x \in \Delta\mathcal{D}_t$，

$$\text{assign}(x) = \arg\min_{1 \le i \le K_{t-1}} \|x - c_{t-1,i}\|.$$

2. **中心更新**：重新计算

$$c_{t,i} = \frac{1}{|\mathcal{D}_{t,i}|} \sum_{x \in \mathcal{D}_{t,i}} x.$$

3. **簇拆分**：对所有 i 满足 $|\mathcal{D}_{t,i}| > N_{\max}$，在该簇内部再执行一次 $k = 2$ 的 KMeans，得子中心 $\{c'_{i,1}, c'_{i,2}\}$，用 $c'_{i,1}$ 替换原中心并新增 $c'_{i,2}$。

Algorithm 6 Dynamic KMeans Clustering Algorithm

Require: Data stream $\{\Delta\mathcal{D}_t\}_{t=1}^{T}$, batch size B, thresholds $N_{\min}, N_{\max}$
Ensure: Final cluster centers C_T

$t \leftarrow 1$
Perform standard KMeans on $\Delta\mathcal{D}_1$ to obtain C_1, A_1
for $t = 2, 3, \ldots, T$ **do**
    Read new batch $\Delta\mathcal{D}_t$
    for all $x \in \Delta\mathcal{D}_t$ **do**
        $\text{assign}(x) \leftarrow \arg\min_i \|x - c_{t-1,i}\|$
    end for
    Update centers: $c_{t,i} \leftarrow \frac{1}{|\mathcal{D}_{t,i}|} \sum_{x \in \mathcal{D}_{t,i}} x$
    for all clusters i with $|\mathcal{D}_{t,i}| > N_{\max}$ **do**
        Run 2-cluster KMeans on $\mathcal{D}_{t,i}$ to get $\{c'_{i,1}, c'_{i,2}\}$
        Replace $c_{t,i} \leftarrow c'_{i,1}$ and add new center $c'_{i,2}$
    end for
end for

7.3 算法流程

7.4 复杂度分析

该算法包含三个主要步骤：增量指派、中心更新、簇拆分，现逐一给出详细复杂度推导。

7.4.1 增量指派 Complexity of Assignment

在第 t 批处理中，新数据量为 B，当前簇数为 K_{t-1}，数据维度为 d。对每个点计算到 K_{t-1} 个中心的欧氏距离，时间复杂度为

$$O\left(B \times K_{t-1} \times d\right).$$

由于 K_{t-1} 保持在总数据条数 n_t 的 1% 左右，即 $K_{t-1} \approx 0.01\, n_t/B$，故该步骤总体接近线性。

7.4.2 中心更新 Complexity of Update

中心更新需遍历所有簇及其成员，总数据量为 $n_t = \sum_i |\mathcal{D}_{t,i}|$，对每簇计算均值耗时 $O(|\mathcal{D}_{t,i}| \times d)$，合计

$$\sum_{i=1}^{K_t} O\left(|\mathcal{D}_{t,i}| \times d\right) = O\left(n_t \times d\right).$$

7.4.3 簇拆分 Complexity of Split

最坏情况下，每个簇都可能超过阈值并被拆分，拆分过程对簇内点再执行一次 $k = 2$ KMeans。对簇 i，若其规模为 m_i，一次 2-簇 KMeans 时间为 $O(m_i \times d)$（假设迭代次数有限常数次）。合

并所有簇的拆分成本为

$$\sum_{i=1}^{K_t} O\left(m_i \times d\right) = O\left(n_t \times d\right).$$

7.5 整体复杂度 Overall Complexity

综合上述三步，每批次迭代的总复杂度为

$$O\left(BK_{t-1}d + n_t d + n_t d\right) = O\left(\left(BK_{t-1} + 2n_t\right) d\right).$$

鉴于 $B \ll n_t$ 且 $K_{t-1} = O(n_t/B)$，可简化为

$$O\left(n_t\, d\right),$$

即对流式数据的聚类更新具有近线性时间复杂度，适用于大规模在线场景。

8 程序结果

在给定迭代步数和给定收敛时间的收敛曲线中，Minibatch 和 Bisecting 聚类算法都比经典 kmeans 有了明显的提速。同时，由于算法自身设计，Bisecting 的聚类速度相对更稳定，MiniBatch 算法更加依赖挑选 Batch 的策略。

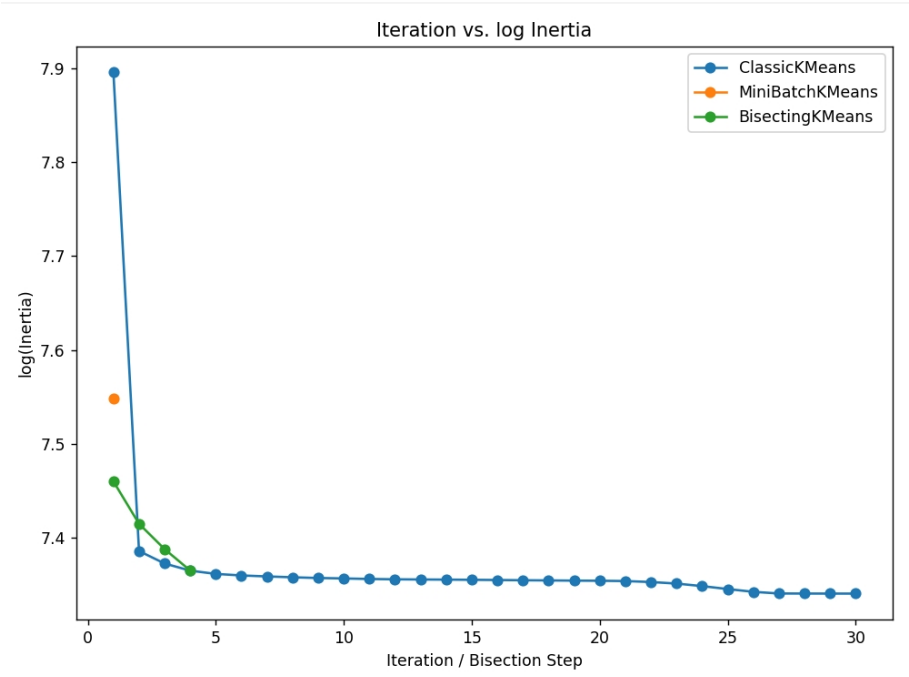

Figure 1: IterVSlogLoss

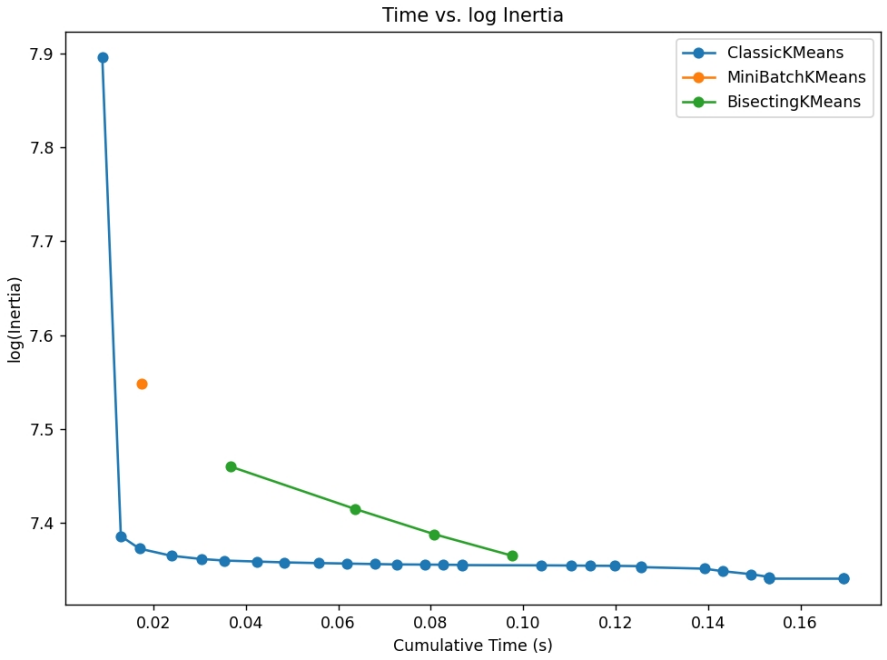

Figure 2: TimeVSlogLoss

聚类效果如下：

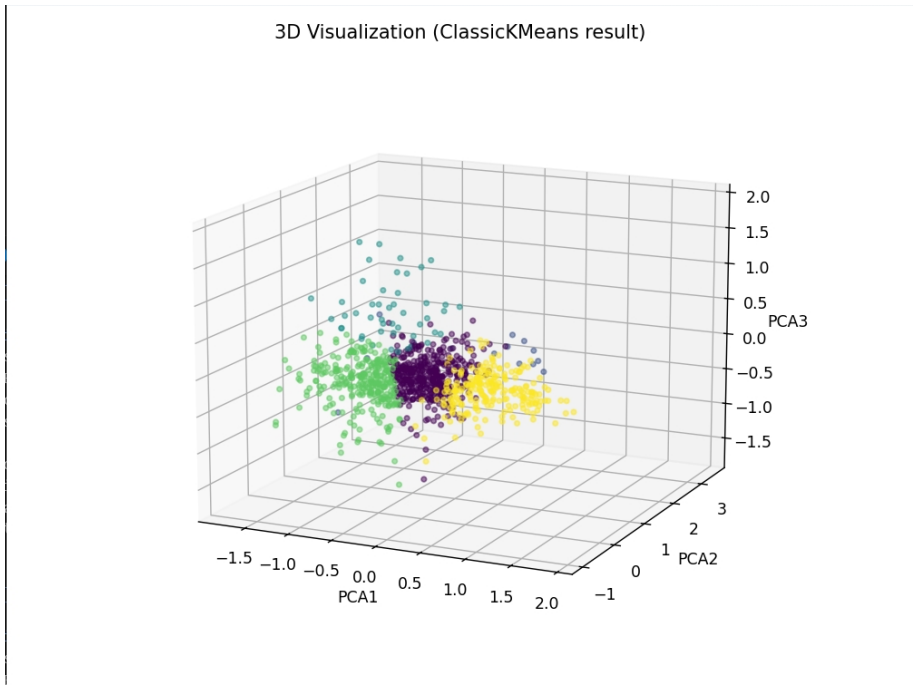

Figure 3: TimeVSlogLoss

随后，针对动态聚类问题，我们计算得出在相同的中心数量下，动态方法和静态方法平均最近距离为 0.3699。随着引入数据的增大，动态方法的 Loss 函数也在递增，这种递增方式无

法避免，因而需要更加具有代表性的 loss 函数设计以表达在动态过程中聚类的效果；时间方面，动态方法引入时间较长，这一方面是因为动态方法需要多次调用 kmeans 和裂分裂函数，另外一方面是因为动态方法也会对原始数据进行重复读取。当使用数据量较大（如百万条以上）时，动态方法因其快速迭代初始解和分批读取数据对内存需求较低等特性会相较本次实验出现不同的表现。

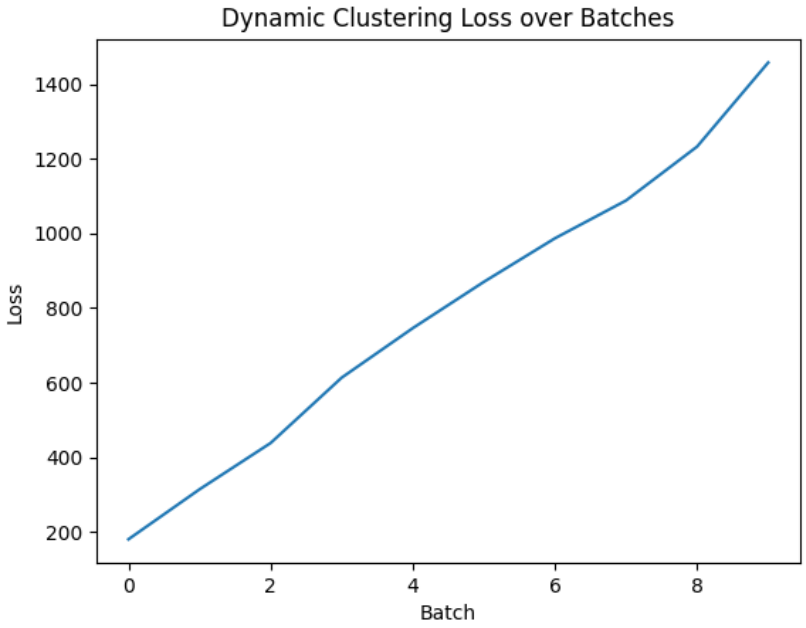

Figure 4: Dynamic Method: Loss

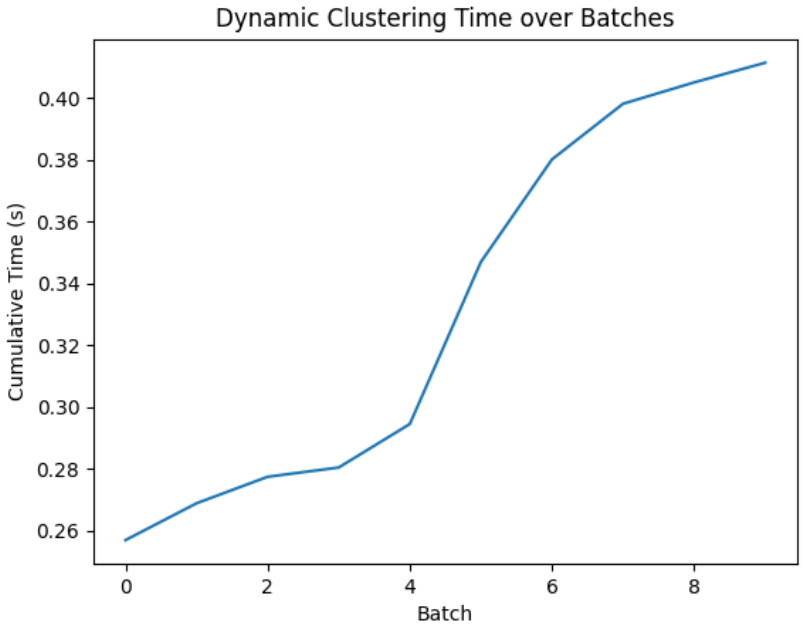

Figure 5: Dynamic Method: Time

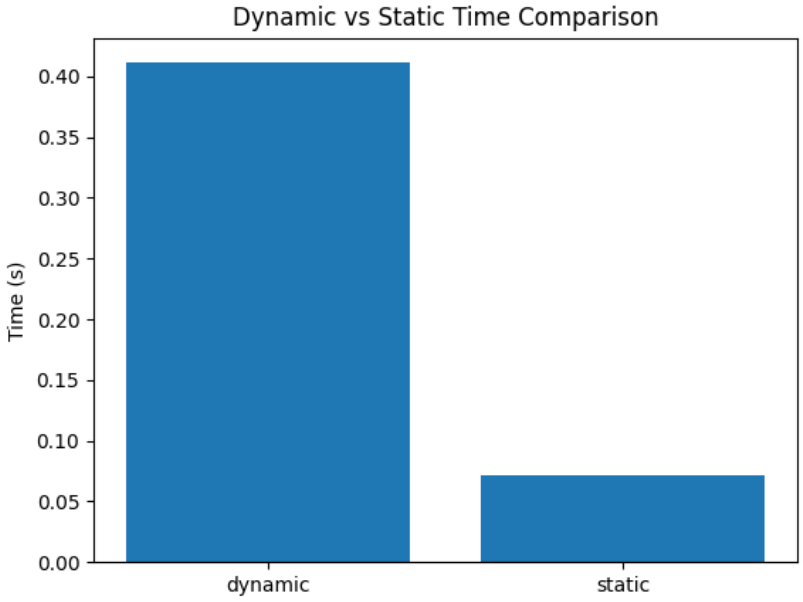

Figure 6: Time Compare

DynCluster

Release 0.0.1

Yingxiao Wang

Jun 19, 2025

CONTENTS:

Add your content using `reStructuredText` syntax. See the reStructuredText documentation for details.

CHAPTER

ONE

SOURCE CODE

1.1 main.py

Listing 1: main.py

```python
"""
This project is a course project for Prof. Qinwu Xu's "Mathematical Theory of Machine Learning" (Spring 2025), School of Mathematics, Nanjing University. It mainly focuses on the problem of dynamic clustering.

The core process is as follows:
→ New data stream arrives
→ Perform clustering on the new data
→ Update cluster centers and adjust previous clustering results
→ If any cluster becomes too large, it is split
→ Adjust the clustering result accordingly
→ Wait for the next batch of data

Author: Yingxiao Wang
Email: wangbottlecap@gmail.com
"""

import logger
from constants.constants import *
from backgrounds import DataReader
from optimizer import DynamicKmeansController, StaticKmeansController
from processer import ComparisonProcessor
from plotter import ComparisonPlotter

def main():
    """
```

(continues on next page)

(continued from previous page)

```
    """
    LOGGER = logger.setup_logging()
    LOGGER.info(f"This project is a course project for Prof. Qinwu Xu's Mathematical␣
↪Theory of Machine Learning (Spring 2025), School of Mathematics, Nanjing University.
↪ It mainly focuses on the problem of dynamic clustering.")

    rd = DataReader()
    rd.read_fvecs()
    print(rd.data)

    dyn = DynamicKmeansController(data=rd.data)
    dyn.run()

    sta = StaticKmeansController(data=rd.data, k=dyn.getCentroidsNum())
    sta.run()

    processer = ComparisonProcessor(dyn, sta)
    plotter = ComparisonPlotter(processer)
    plotter.print_center_distance()
    plotter.plotAll()

    return


if __name__ == "__main__":
    main()
```

1.2 optimizer.py

Listing 2: optimizer.py

```
import numpy as np
import time
from sklearn.cluster import KMeans

import logging
LOGGER = logging.getLogger(__name__)

```

(continues on next page)

(continued from previous page)

```
class DynamicKmeansController():
    """
    Dynamic KMeans clustering controller for streaming data.

    Core process:
    → New data batch arrives
    → Assign new points to existing clusters (or initialize)
    → Update cluster centers
    → Split clusters exceeding max_size into two
    → Compute and print current loss
    → Repeat for next batch

    Attributes:
        data (np.ndarray): full dataset for indexing.
        batch_size (int): number of points per incoming batch.
        min_size (int): minimum cluster size before merge (future extension).
        max_size (int): maximum cluster size before split.
        centroids (list[np.ndarray]): current cluster centers.
        cluster_data_indices (dict[int, list[int]]): mapping cluster IDs to data indices.
        assignments (dict[int, int]): mapping data index to its cluster ID.
        loss_history (list[float]): recorded loss after each batch.
        time_history (list[float]): cumulative processing time after each batch.
    """

    def __init__(self, data:np.ndarray, batch_size=100, min_size=60, max_size=150, ratio = 0.01,threshold = 0.01):
        self.data = data                                    # 保存整个数据集
        self.batch_size = batch_size                        # 每次处理的批大小
        # self.threshold = threshold
        # self.ratio = ratio
        self.min_size = min_size                            # 最小簇大小（留作合并条件）
        self.max_size = max_size                            # 最大簇大小（超过需分裂）
        self.centroids = []                                 # 存储当前簇中心列表
        self.cluster_data_indices = {}                      # 存储簇 ID 到样本索引的映射
        self.assignments = {}                               # 存储样本索引到簇 ID 的映射
        self.loss_history = []                              # 记录每批的损失值（inertia）
        self.time_history = []                              # 记录累计时间
        self._current_time = 0.0                            # 内部跟踪累计运行时间

```

(continues on next page)

(continued from previous page)

```
    def getCentroidsNum(self): return len(self.centroids)

    def run(self):
        """Run dynamic clustering over the entire dataset in streaming batches."""
        n_points = self.data.shape[0]                       # 数据总样本数
        # 按批次遍历数据
        for batch_start in range(0, n_points, self.batch_size):
            # 计算本批样本全局索引范围
            batch_indices = list(range(batch_start, min(batch_start + self.batch_size,
↪ n_points)))
            batch = self.data[batch_indices]                # 提取本批数据
            t0 = time.time()                                # 计时开始

            # 增量聚类、更新中心、拆分过大簇
            self._add_batch(batch, batch_indices)
            self._update_clusters()
            self._split_large_clusters()

            batch_loss = self._compute_loss()               # 计算本批后总损失
            self._current_time += time.time() - t0          # 更新累计时间

            self.loss_history.append(batch_loss)            # 记录损失
            self.time_history.append(self._current_time)    # 记录时间
            print(f"Batch {batch_start//self.batch_size + 1}: Loss = {batch_loss:.4f},
↪ Time = {self._current_time:.4f}s")

    def _add_batch(self, batch, indices):
        """Assign a new batch of points to clusters or initialize clustering."""
        if not self.centroids:                              # 如果还没有簇中心（首次调用）
            # 计算初始簇数：确保至少 1 个簇
            k_init = max(1, len(batch) // ((self.min_size + self.max_size) // 2))
            # k_init = max(1, int(self.batch_size * self.ratio))
            # 使用 KMeans 对首批数据做静态聚类初始化
            km = KMeans(n_clusters=k_init, init='k-means++', random_state=42).
↪fit(batch)
            self.centroids = list(km.cluster_centers_)      # 保存初始簇中心
            # 记录每个样本的簇标签
            for idx, lbl in zip(indices, km.labels_):
                self.assignments[idx] = lbl
            # 构建簇到样本索引的映射
```

(continues on next page)

(continued from previous page)

```
            for cid in range(k_init):
                self.cluster_data_indices[cid] = [idx for idx in indices if self.
↪assignments[idx] == cid]
        else:
            # 对后续批次，遍历每个新点
            for idx, point in zip(indices, batch):
                # 计算该点到所有簇中心的欧氏距离
                distances = [np.linalg.norm(point - c) for c in self.centroids]
                lbl = int(np.argmin(distances))             # 选择最近的簇
                self.assignments[idx] = lbl                  # 更新样本-簇映射
                # 将样本索引追加到对应簇的列表中
                self.cluster_data_indices.setdefault(lbl, []).append(idx)

    def _update_clusters(self):
        """Recompute centroids based on current cluster memberships."""
        # 遍历所有簇
        for cid, members in self.cluster_data_indices.items():
            # 提取该簇对应的所有样本点
            pts = np.array([self.data[i] for i in members])
            if pts.size > 0:
                # 重新计算簇中心为样本均值
                self.centroids[cid] = pts.mean(axis=0)

    def _split_large_clusters(self):
        """Split any cluster larger than max_size into two subclusters."""
        # 下一个可用簇 ID
        next_cid = max(self.cluster_data_indices.keys(), default=-1) + 1
        # 找出需要分裂的簇 ID 列表
        oversized = [cid for cid, members in self.cluster_data_indices.items() if␣
↪len(members) > self.max_size]
        for cid in oversized:
            members = self.cluster_data_indices.pop(cid)  # 暂时移除该簇
            pts = np.array([self.data[i] for i in members])
            # 对过大簇内样本再次做 2 簇 KMeans 拆分
            km2 = KMeans(n_clusters=2, init='k-means++', random_state=42).fit(pts)
            centers2 = km2.cluster_centers_
            labels2 = km2.labels_

            # 更新原簇中心及成员
            self.centroids[cid] = centers2[0]
```

(continues on next page)

(continued from previous page)

```
            self.cluster_data_indices[cid] = [members[i] for i, l in enumerate(labels2) if l == 0]

            # 添加新簇中心及成员
            self.centroids.append(centers2[1])
            self.cluster_data_indices[next_cid] = [members[i] for i, l in enumerate(labels2) if l == 1]

            # 更新分裂后所有样本的簇归属
            for idx, lbl in zip(members, labels2):
                self.assignments[idx] = cid if lbl == 0 else next_cid

            next_cid += 1                               # 更新下一个可用簇 ID

    def _compute_loss(self):
        """Compute total inertia (sum of squared distances) as loss."""
        total_loss = 0.0
        # 遍历每个簇计算簇内平方和
        for cid, members in self.cluster_data_indices.items():
            center = self.centroids[cid]                # 当前簇中心
            pts = np.array([self.data[i] for i in members])
            total_loss += np.sum((pts - center) ** 2)    # 累加平方误差
        return total_loss



class StaticKmeansController:
    """
    Static KMeans clustering controller using faiss for full-dataset clustering.

    Attributes:
        data (np.ndarray): Full dataset of shape (N, D), dtype float32.
        k (int): Number of clusters.
        niter (int): Number of iterations for faiss KMeans.
        seed (int): Random seed for reproducibility.
        centroids (np.ndarray): Final cluster centers of shape (k, D).
        assignments (dict[int, int]): Mapping from data index to assigned cluster ID.
        cluster_data_indices (dict[int, list[int]]): Mapping from cluster ID to list of data indices.
        loss (float): Final inertia (sum of squared distances to centroids).
```

(continues on next page)

(continued from previous page)

```
        time_taken (float): Time spent on clustering (seconds).
    """

    def __init__(self, data: np.ndarray, k: int, niter: int = 100, seed: int = 42):
        self.data = data.astype('float32')
        self.k = k
        self.niter = niter
        self.seed = seed
        self.centroids = None
        self.assignments = {}
        self.cluster_data_indices = {}
        self.loss = None
        self.time_taken = None

    def run(self):
        """
        Run static KMeans clustering using faiss.Kmeans.
        Records centroids, assignments, cluster_data_indices, loss, and time_taken.
        """
        try:
            import faiss
        except ImportError:
            raise ImportError("faiss library not installed. Please install faiss to use StaticKmeansController.")

        # Initialize faiss KMeans
        d = self.data.shape[1]
        kmeans = faiss.Kmeans(d, self.k, niter=self.niter, verbose=False, seed=self.seed)

        # Train and time
        start_time = time.time()
        kmeans.train(self.data)
        self.time_taken = time.time() - start_time

        # Retrieve centroids
        self.centroids = kmeans.centroids.copy()

        # Assign each point to the nearest centroid
        distances, labels = kmeans.index.search(self.data, 1)
```

(continues on next page)

(continued from previous page)

```
        labels = labels.flatten()

        # Build assignments and cluster-to-indices mapping
        for idx, lbl in enumerate(labels):
            self.assignments[idx] = int(lbl)
            self.cluster_data_indices.setdefault(int(lbl), []).append(idx)

        # Compute inertia (loss)
        total_loss = 0.0
        for idx, lbl in self.assignments.items():
            center = self.centroids[lbl]
            point = self.data[idx]
            total_loss += np.sum((point - center) ** 2)
        self.loss = float(total_loss)

    def report(self):
        """
        Return a summary of clustering results.
        """
        return {
            'Method': 'Static faiss.Kmeans',
            'Time (s)': self.time_taken,
            'Loss': self.loss,
            'Num Clusters': self.k,
            'Cluster Sizes': {cid: len(idxs) for cid, idxs in self.cluster_data_
↪indices.items()}
        }
```

1.3 background.py

1.4 logger.py

Listing 3: logger.py

```
import os
import logging

from constants.filepath_constants import RESULTS_DIR

```

(continues on next page)

(continued from previous page)

```
def setup_logging(fileName = 'app.log'):
    # set up logging file, format and location, information level

    # logFormat = "%(asctime)s - %(levelname)s - %(message)s"
    logFormat = "%(message)s"
    dateFormat = "%m/%d/%Y %H:%M:%S %p"

    # fileHandler
    path = os.path.join(RESULTS_DIR, fileName)
    file_handler = logging.FileHandler(path, mode='w')

    logging.basicConfig(
        format=logFormat,
        datefmt=dateFormat,
        level=logging.DEBUG,
        handlers= [
            file_handler,
            # stream_handler
        ]
    )

    LOGGER = logging.getLogger()
    return LOGGER
```

1.5 processer.py

Listing 4: processer.py

```
import logging
from optimizer import DynamicKmeansController, StaticKmeansController
from util import *
LOGGER = logging.getLogger(__name__)

# 2. 数据处理类
class ComparisonProcessor:
    """
    对比动态 vs 静态 聚类结果，存储以下信息：
      - center_distance: 两组簇中心平均最近距离
      - dynamic_loss, static_loss
```

(continues on next page)

(continued from previous page)

```
      - dynamic_time, static_time
      - dynamic_history: loss & time 历史
    """
    def __init__(self, dynController:DynamicKmeansController,␣
↪staController:StaticKmeansController):
        self.dyn = ComparisonProcessor.summarize_dynamic(dynController)
        self.sta = ComparisonProcessor.summarize_static(staController)
        self.results = {}
        self._compute()

    def _compute_center_distance(self):
        # 对于每个动态中心，找到距离最近的静态中心，计算平均距离
        dyn_c = self.dyn['centroids']
        sta_c = self.sta['centroids']
        dists = []
        for c in dyn_c:
            diff = sta_c - c
            dist_to_sta = np.linalg.norm(diff, axis=1)
            dists.append(np.min(dist_to_sta))
        return float(np.mean(dists))

    def _compute(self):
        self.results['center_distance'] = self._compute_center_distance()
        self.results['dynamic_loss'] = float(self.dyn['loss'])
        self.results['static_loss'] = float(self.sta['loss'])
        self.results['dynamic_time'] = float(self.dyn['time'])
        self.results['static_time'] = float(self.sta['time'])
        self.results['dynamic_loss_history'] = self.dyn['loss_history']
        self.results['dynamic_time_history'] = self.dyn['time_history']


    @staticmethod
    def summarize_dynamic(controller:DynamicKmeansController):
        """
        输出动态聚类摘要信息
        Returns a dict with:
        - num_clusters: 最终簇数
        - loss: 最终 loss
        - time: 累计运行时间
        - loss_history: 每批 loss 的列表
```

(continues on next page)

(continued from previous page)

```
        - time_history: 每批累计时间的列表
        - centroids: 簇中心数组, shape=(k, D)
        """
        centroids = np.stack(controller.centroids, axis=0)
        return {
            'method': 'dynamic',
            'num_clusters': centroids.shape[0],
            'loss': controller.loss_history[-1],
            'time': controller.time_history[-1],
            'loss_history': np.array(controller.loss_history),
            'time_history': np.array(controller.time_history),
            'centroids': centroids
        }

    @staticmethod
    def summarize_static(controller:StaticKmeansController):
        """
        输出静态聚类摘要信息
        Returns a dict with:
        - num_clusters: 簇数 k
        - loss: 最终 loss
        - time: 聚类耗时
        - centroids: 簇中心数组, shape=(k, D)
        """
        return {
            'method': 'static',
            'num_clusters': controller.k,
            'loss': controller.loss,
            'time': controller.time_taken,
            'centroids': controller.centroids
        }
```

1.6 plotter.py

Listing 5: plotter.py

```python
import matplotlib.pyplot as plt

import logging
from processer import ComparisonProcessor
from constants.filepath_constants import RESULTS_DIR

LOGGER = logging.getLogger(__name__)









# 3. 画图类
class ComparisonPlotter:
    """
    接收 ComparisonProcessor, 绘制:
      - loss 对比 (bar)
      - time 对比 (bar)
      - 中心距离 (单个文本展示)
      - 动态聚类 loss & time 历史曲线
    """
    def __init__(self, comp: ComparisonProcessor):
        self.comp = comp

    def plotAll(self):
        self.plot_dynamic_history()
        self.plot_loss_comparison()
        self.plot_time_comparison()

    def plot_loss_comparison(self):
        fig = plt.figure()
        labels = ['dynamic', 'static']
        values = [self.comp.results['dynamic_loss'], self.comp.results['static_loss']]
        plt.bar(labels, values)
        plt.ylabel('Loss')
        plt.title('Dynamic vs Static Loss Comparison')
        plt.show()
```

(continues on next page)

(continued from previous page)

```

    def plot_time_comparison(self):
        fig = plt.figure()
        labels = ['dynamic', 'static']
        values = [self.comp.results['dynamic_time'], self.comp.results['static_time']]
        plt.bar(labels, values)
        plt.ylabel('Time (s)')
        plt.title('Dynamic vs Static Time Comparison')
        plt.show()

    def print_center_distance(self):
        dist = self.comp.results['center_distance']
        print(f"Average nearest-center distance between dynamic and static: {dist:.4f}
↪")

    def plot_dynamic_history(self):
        # Loss history
        fig1 = plt.figure()
        plt.plot(self.comp.results['dynamic_loss_history'])
        plt.xlabel('Batch')
        plt.ylabel('Loss')
        plt.title('Dynamic Clustering Loss over Batches')
        plt.show()
        # Time history
        fig2 = plt.figure()
        plt.plot(self.comp.results['dynamic_time_history'])
        plt.xlabel('Batch')
        plt.ylabel('Cumulative Time (s)')
        plt.title('Dynamic Clustering Time over Batches')
        plt.show()
```

1.7 util.py

Listing 6: util.py

```
import numpy as np
```