SE3352: Algorithm Design	Nov 27, 2021

Notes 1 – Clustering

Instructor: *Guoqiang Li*	Scribes: *Haotian Wang, Yiyan Li*

1 Introduction

Clustering can be considered the most important unsupervised learning problem; As every other problem of this kind, it deals with finding a structure in a collection of unlabeled data.

Definition 1. *A cluster is a collection of objects which are "similar" between them and are "dissimilar" to the objects belonging to other clusters. Clustering is the algorithm that recognizes clusters from a given data set.*

Note that this is a very rough definition. Part of common application domains in which the clustering problem arises are as follows:

- **Multimedia Data Analysis:** Learning image or video representations without manual annotations. e.g. When using Streaming media platform, face clustering can recognize all the actors in any frame. [**?**, **?**]
- **Responding to public health crises:** With the increasing number of samples, the manual clustering of COVID-19 data samples becomes time-consuming. Clustering helps classify medical datasets deterministically.[**?**]
- **Social Network Analysis:** Clustering provides an important understanding of the community structure in the network. Results can be used for customer segmentation and sending ads. Put it in a formal way, **clustering groups the nodes of the graph into clusters**, taking into account the edge structure of the graph in such a way that there are several edges within each cluster and very few between clusters. [**?**]
- **Intermediate Step for other fundamental data mining problems:** Clustering can be considered as a form of data summarization. Many clustering methods are closely related to dimensionality reduction methods. Such methods can be considered a form of data summarization.
- **Intelligent Transportation:** Under **the online scenario**, data is in the form of streams, i.e., the whole dataset could not be accessed at the same time. In future intelligent transportation, low-latency online vehicle tracking is essential and can be solved by online clustering.[**?**]

Today we'll start from the naive K-means clustering and improve the algorithm step by step. The lecture has X main topics that we'll go through, i.e. TODO AL LAST!!

2 Problem Description

The k-means clustering problem is one of the oldest and most important questions in all of computational geometry. Given an integer k and a set of n data points in $\mathbb{R}^d$, the goal of this problem is to choose k centers so as to minimize the total squared distance between each point and its closest center.[**?**]

There are several kinds of k-means algorithms among which the most common algorithm , also called naive k-means algorithm, was first proposed by Stuart Lloyd[**?**] of Bell Labs in 1957.

For a k-means problem, we are given an integer k and a set of data vector $(x_1, x_2, x_3 \dots x_n)$ in d-dimension. And we need to choose k centroids to partition the n vectors into k types T $(T_1, T_2 \dots T_k)$ with the minimum within-cluster sum of squares $(WCSS)$

$$\mathcal{WCSS} = \arg\min \sum_{i=1}^{k} \sum_{x \in S_i} \|x - c_i\|^2$$

where μ_i is the mean of vector in set S_i.

Lemma 2. *Let S be a set of points with its mean to be μ, and let c to be and arbitrary point.Then* $\sum_{x \in S} \|x - c\|^2 = \sum_{x \in S} \|x - \mu\|^2 + |S| \cdot \|z - \mu\|^2$

So we minimize the function only when $c_i = \mu_i$

$$\begin{aligned}
\mathcal{WCSS} &= \arg\min \sum_{i=1}^{k} \sum_{x \in S_i} \|x - c_i\|^2 \\
&= \arg\min \sum_{i=1}^{k} (\sum_{x \in S_i} \|x - \mu_i\|^2 + |S_i| \cdot \|c_i - \mu_i\|^2) \\
&= \arg\min \sum_{i=1}^{k} |S_i| \cdot VarS_i \\
&= \arg\min \sum_{i=1}^{k} \frac{1}{2 \cdot |S_i|} \sum_{x,y \in S_i} \|x_i - y_i\|^2
\end{aligned}$$

3 Algorithms

3.1 The K-means algorithm

The k-means algorithm is a simple and fast algorithm for this problem, although it offers no approximation guarantees at all. It iteratively calculates the sum of distance within a cluster and updates the partition.The details are as follows.[**?**]

1. Arbitrarily choose and initial k centers $\mathcal{C} = \{c_1, c_2 \dots c_k\}$

2. For each $i \in \{1, 2 \dots k\}$, set the cluster C_i to be the set of points that are closer to c_i than they are to c_j for all $j \neq i$

3. For each $i \in \{1, 2 \dots k\}$, set c_i to be the center of all points in C_i

Algorithm 1 K-means

Input: k: number of output cluster; Data: input data

Output: $\mathcal{S}$: set of all clusters S_i

Arbitrarily initialize k centroids $C = \{c_1, c_2 \dots c_k\}$

repeat

  for each point x in Data S **do**

    for $i = 0 \to k$ **do**

      for $j = 0 \to k$ **do**

        set x to be a member of cluster S_i where $\|x - c_i\|^2 < \|x - c_j\|^2$

      end for

    end for

  end for

  for $i = 0 \to k$ **do**

    $c_i \leftarrow \frac{1}{|S_i|} \sum_{x \in S_i} x$

    set c_i to be the centroid of all points in cluster S_i

  end for

until S stays unchanaged

Output: $\mathcal{S}$: set of all clusters S_i

4. Repeat Step 2 and Step 3 until $\mathcal{C}$ no longer changes.

证明. Let $x_1, x_2 \dots x_n$ be n vectors in $\mathbb{R}^d$, then $f(x) = \sum_{i=1}^{n} \|x_i - x\|^2$ gets its minimum iff. $x = \frac{1}{n} \sum_{i=1}^{n} x_i$

$$\begin{aligned}
\frac{df(x)}{dx} &= \frac{d \sum_{i=1}^{n} \|x_i - x\|^2}{dx} \\
&= -2 \sum_{i=1}^{n} (x_i - x) \\
&= 0 \\
x &= \frac{1}{n} \sum_{i=1}^{n} x_i
\end{aligned}$$

$x = \frac{1}{n} \sum_{i=1}^{n} x_i$ is a stationary point of this function. Owing that it is a strictly convex function, the stationary point is alse the only minimun point of that function. So the function gets its minimum at $x = \frac{1}{n} \sum_{i=1}^{n} x_i$. □

证明. Updated value $f(x'')$ is strictly less than the original $f(x')$ where $x'' = \frac{1}{n} \sum_{i=1}^{n} x_i$.

As described above, each centroid is updated to the center of all points in cluster C_i. That is to say, once the centroid of one cluster changed from x' to $x'' = \frac{1}{n} \sum_{i=1}^{n} x_i$, the function gets its minimum in this iteration at x'' and $f(x'') < f(x')$. □

4 Key properties

K means problem is an NP Hard problem, and two teams have proved them using 3-SAT and Exact Cover by 3-Sets respectively.[**?**, **?**] Next, I'll try to describe the reduction from 3-SAT to k-means since

NP-Complete problem is an inescapable topic in algorithm course and try my best to get rid of copying original material.

4.1 Reduction from 3-SAT to K-means

Let F be the given planar 3-SAT instance with n variables and m clauses. We construct an instance I of planar k-means corresponding to F. Properties of layout I are listed below:

1. Each variable x_i corresponds to a simple circuit s_i in the plane and each circuit has an even number Q of vertices. Each vertex on such a circuit have M copies of a point. Note that M and Q will be stricted below.
2. bdd
3. cc

5 Median Trick

So far, we have an algorithm A which estimates in correct range of ϵ with probability ≥ 0.9. Our new algorithm A^* will output in range of ϵ with probability $1-\delta$. Algorithm:

- Repeat A for $m = O(log(1/\delta))$ times
- Take median of all the m answers.

To prove the correctness, we'll use Chernoff/Hoeffding bounds.

Definition 3 (Chernoff/Hoeffding Bound)**.** *Let $X_1, X_2, \ldots, X_m$ be independent random variables $\in \{0,1\}$, $\mu = E[\Sigma_i X_i], \epsilon \in [0,1]$. Then $Pr[|\Sigma_i X_i - \mu| > \epsilon\mu] \leq 2e^{-\epsilon^2\mu/3}$*

Define $X_i = 1$ iff the i^{th} answer of A is correct (i.e. estimated value of A lies in correct range).

Claim 4. *$E[X_i] = 0.9$, and $E[\mu] = 0.9m$*

证明. Since A is correct with probability 0.9, $E[X_i] = 0.9$. And $E[\mu] = 0.9m$ due to linearity of expectation. □

Claim 5. *New algorithm A^* is correct when $\Sigma_i X_i > 0.5m$*

证明. Since we are considering median value to be our answer, if more than half the trials of A are correct, algorithm A^* is also correct. □

Claim 6. *To prove, $Pr[\Sigma_i X_i \geq 0.5m] \geq 1-\delta$ or $Pr[\Sigma_i X_i < 0.5m] < \delta$*

证明.

$$\begin{aligned} Pr[\Sigma_i X_i < 0.5m] &= Pr[\Sigma_i X_i - 0.9m < -0.4m] \\ &\leq Pr[|\Sigma_i X_i - \mu| > 0.4m] \\ &= Pr[|\Sigma X_i - \mu| > 0.4/0.9\mu] \end{aligned} \tag{1}$$

Using Chernoff bound,

$$\begin{aligned} &\leq e^{-c*0.9m} \\ &< \delta \end{aligned} \tag{2}$$

Above equation holds for $m = O(log(1/\delta))$ □

6 Distinct Elements

Given, a stream of size m containing numbers from $[n]$, we have to approximate the number of elements with non-zero frequency. To calculate the exact value the space required:

- $O(n)$ bits. (maintain a vector of length n).
- $O(m \log(n))$ bits. (save m numbers, each taking $log(n)$ bits).

Since, this complexity is not feasible as m,n can be very large, we'll look at algorithm for approximating the distinct count value.

6.0.1 Hash Function

- $h : [n] \to [0, 1]$
- $h(i)$ is uniformly distributed in $[0, 1]$.

6.1 Algorithm [Flajolet-Martin 1985]

We maintain a variable z.

1. Initialize $z = 1$.
2. Whenever i is encountered: $z = \min(z, h(i))$
3. When done, output $1/z - 1$.

Now, we'll prove the algorithm works in a similar fashion followed in previous lecture. Let d be number of distinct elements.

Claim 7. $E[z] = d + 1$

证明. z is the minimum of d random numbers in $[0, 1]$. Pick another random number $a \in [0, 1]$. The probability $a < z$:

1. exactly z
2. probability it's smallest among $d + 1$ reals : $1/(d + 1)$

Equating these two, one can prove the claim. □

Claim 8. $var[z] \leq 2/d^2$

证明. It can be done in a similar fashion described in previous lecture. □

6.1.1 $(1+\epsilon)$ approximation Algorithm

We can take $Z = (z_1 + z_2 + ...z_k)/k$ for independent $z_1, ...z_k$

6.2 Alternate Algorithm: Bottom-k

Instead of just use the minimum value of hash function for i inputs, we'll maintain the k smallest hashes seen.

1. Initialize $(z_1, z_2, ...z_k) = 1$.
2. Keep k smallest hashes seen, s.t. $z_1 \leq z_2 \leq ...z_k$
3. When done, output $\hat{d} = k/z_k$

Claim 9. *The following claims are stated:*

- $Pr[\hat{d} > (1+\epsilon)d] \leq 0.05$
- $Pr[\hat{d} < (1-\epsilon)d] \leq 0.05$
- *Overall probability that $\hat{d}$ outside range is at most 0.1*

证明. To compute $Pr[\hat{d} > (1+\epsilon)d]$:

- Define $X_i = 1$ iff $h(i) < \dfrac{k}{(1+\epsilon)d}$
- Then $\hat{d} > (1+\epsilon)d$ iff $\Sigma_i X_i > k$
- if $\Sigma_i X_i > k$
  $\iff \exists$ at least k numbers for which $h(i) < \dfrac{k}{(1+\epsilon)d}$

$$\iff z_k < \frac{k}{(1+\epsilon)d} \iff \frac{k}{z_k} > (1+\epsilon)d \iff \hat{d} > (1+\epsilon)d \tag{3}$$

- $E[X_i] = \dfrac{k}{(1+\epsilon)d}$
  $E[\Sigma_i X_i] = dE[X_i] = \dfrac{k}{1+\epsilon}$
  $\text{var}[\Sigma_i X_i] = d\text{var}[X_i] \leq dE[X_1^2] \leq \dfrac{k}{1+\epsilon} \leq k$
  (Since $X_1 \in \{0,1\}$, $E[X_1^2] = E[X_i]$)
- By Chebyshev: $Pr[|\Sigma X_i - \dfrac{k}{1+\epsilon}| > \sqrt{20k}] \leq 0.05 \implies Pr[\Sigma X_i > \dfrac{k}{1+\epsilon} + \sqrt{20k}] \leq 0.05$
  - (For $\epsilon < 1/2$ and $k = c/\epsilon^2$)
    $\dfrac{k}{1+\epsilon} + \sqrt{20k} \leq k(1-\epsilon+\epsilon^2) + \sqrt{20k}$ (Taylor Series Expansion)
    $\leq k - k\epsilon/2 + 5\sqrt{c}/\epsilon = k - c/2\epsilon + 5\sqrt{c}/\epsilon$
    $< k$ where $c > 100$

– Since $k > \dfrac{k}{1+\epsilon} + \sqrt{20k}$ in our case and ΣX_i is monotonically increasing, $Pr[\Sigma X_i > k] \leq Pr[\Sigma X_i > \dfrac{k}{1+\epsilon} + \sqrt{20k}] \leq 0.05$

□

6.3 Hash functions in stream

The hash function we used has two practical issues: (1) the return value should be a real number. (2) how do we store it?

Discretization can solve the first issue. Instead of all the real numbers in $[0, 1]$, we use hash function with range $\{0, \frac{1}{M}, \frac{2}{M}, \frac{3}{M}, \ldots, 1\}$. For large $M \gg n^3$, the probability that $d \leq n$ random numbers collide is at most $\frac{1}{n}$.

For the second issue, we use pairwise independent function instead of independent function.

Definition 10. *$h : [n] \to \{1, 2, \ldots M\}$ is pairwise independent if for all $i \neq j$ and $a, b \in [M]$, $Pr[h(i) = a \wedge h(j) = b] = \frac{1}{M^2}$*

It works because in previous calculation, we only care about pairs. We defined $X_i = 1$ iff $h(i)$ is small than a threshold, then we computed $\text{var}[\Sigma X_i] = E[(\Sigma X_i)^2] - E[(\Sigma X_i)^2] = E[X_1X_1 + X_1X_2 + \ldots] - E[(\Sigma X_i)^2]$. Notice that $E[X_iX_j]$ is the same for fully random h and pairwise independent h.

Example 11 (Construct a pairwise independent hash)**.** *Assume M is a prime number (if not, we can always pick a larger M that is a prime number). We pick $p, q \in \{0, 1, 2, \ldots M - 1\}$ and the hash function $h(i) = pi + q \mod M$. In this construction we only need $O(\log M) = O(\log n)$ space (to store p, q, M).*

证明. $h(i) = a, h(j) = b$ is equivalent to $pi+q \equiv a, pj+q \equiv b$. So $p(i-j) \equiv a-b$ and $p \equiv (a-b)(i-j)^{-1}, q \equiv a - pi$. Since M is a prime number, the unique inverse implies that there is only one pair (p, q) satisfies it. And the probability that pair is chosen is exactly $\frac{1}{M^2}$. □

7 Impossibility Results

We have used both approximation and randomization to solve the distinct counting problem with space much less than $\min(m, n)$. Now we are wondering: can we omit either approximation or randomization to achieve the same space efficiency? The answer is no.

7.1 Deterministic Exact Won't Work

First, we will show that there is no deterministic (no randomization) and exact (no approximation) way to solve it.

Suppose there do exists a deterministic and exact algorithm A and an estimator function R that use space $s \ll n, m$. That is, for a given integer stream, we first run the algorithm A on the stream. As the stream goes A will return middle memory steps, and we obtain the final memory state σ after the stream ends. Then we apply R on σ to obtain our estimator $\hat{d}$. Since both A and R are deterministic and exact, $\hat{d}$ must equals to the distinct count for the stream.

We now build a binary representation x of the stream with the following rules: (1) $x \in \{0, 1\}^n$, (2) i in stream iff $x_i = 1$. For example, if 1, 3, 5, 6, 7 are in the stream and 2, 4 are not, x will start with

1, 0, 1, 0, 1, 1, 1. Notice that each stream has a corresponding representation and streams containing different numbers have different representations.

Claim 12. *We can recover the x of the stream given the memory state σ*

证明. Denote $d = R(\sigma)$ be the original estimator. Now we treat σ as a middle snapshot of the memory and add integer i as the next element of the stream. Now A will return another memory state σ', and $d' = R(\sigma)'$ will be our new estimator. If $d' = d$, i must have appeared in the stream before since A and R are deterministic and exact. Similarly, if $d' > d$, i must have not appeared in the stream before. Using this method with $i = 1, 2, 3 \ldots$ and we can recover the x. □

Since we can recover x from σ, we can treat σ as an encoding of a string x of length n. But σ has only $s \ll n$ bits! Furthermore, we can treat A, the function that produces σ, as a function with domain $\{0,1\}^n$ and $\{0,1\}^s$. We can see that A must be injective because if $A(x) = A(x') = \sigma$, the recoverability implies $x = x'$.

Hence $s \geq n$. Which implies that there is no deterministic and exact algorithm A and an estimator function R that use space $s \ll n, m$.

7.2 Deterministic Approx. Won't Either

We can use the similar strategy to prove that deterministic approx. won't work. We pick $T \subset \{0,1\}^n$ that satisfies the following conditions: (1) for all distinct $x, y \in T$, the number of digits i that $y_i = 1$ and $x_i = 0$ should $\geq \frac{n}{6}$. (2) $|T| \geq 2^{\Omega(n)}$. Now we use algorithm A to encode an input x into $\sigma = A(x)$ and our estimator would be $\hat{d} = R(\sigma)$.

Now we want to recover x based on σ, as what we have done in the last section. For a given σ and any $y \in T$, we append y to the stream and apply A on it, and A will return a memory state σ'. Using σ' we have new estimator $\hat{d'} = R(\sigma')$.

Claim 13. *If $\hat{d'} > 1.01\hat{d}$, then $x \neq y$, else $x = y$.*

证明. The idea is that when $x = y$, $\hat{d}$ would be really close to $\hat{d'}$ (up to $(1+\epsilon)^2$ because both of them are ϵ-approximated) and when $x \neq y$, the construction of T guarantee that $\hat{d} \geq \hat{d} + \frac{n}{6}$. So we can pick an ϵ that works for our claim. □

We can use this method to check every element $y \in T$ to see if $y = x$, and eventually we can recover x from it. Similar to last section, we can show that A is an injective function and it implies that $2^s \geq |T|$ or $s = \Omega(n)$.

8 Concluding Remarks

- We can use median trick and Chernoff bound to improve the probability of an existing algorithm.
- For distinct elements problem, we can also store the hashes $h(i)$ approximately. One example is to store the number of leading zeros, and it only cost $O(\log \log n)$ bits per hash value, and that is the idea behind another algorithm called HyperLogLog.
- For the impossibility results, we can also prove that randomized exact algorithm won't work.

Appendix

A K-means Algorithm Code in Python

```
import math
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np


def loadData():
    df = pd.read_csv("./data/data.csv")
    return df.values


def euclideanDistance(vector1, vector2):
    return math.sqrt(sum(np.power(vector1 - vector2, 2)))


def initRandomCentroids(data, k):
    count, dim = data.shape
    centroids = np.zeros((k, dim))
    colMax = np.max(data, axis=0)
    colMin = np.min(data, axis=0)
    colRange = colMax - colMin
    for i in range(k):
        centroid = colMin + np.random.rand(dim) * colRange
        centroids[i, :] = centroid
    print(centroids)
    return centroids


def kmeans(k):
    data = loadData()
    count = data.shape[0]
    centroids = initRandomCentroids(data, k)
    clusterBound = np.zeros((count, 2))
    index = np.zeros((count, 1))
    processing = True
    while processing:
        processing = False
        for i in range(count):
            minIndex = 0
            minDist = float("inf")
```

```
            for j in range(k):
                distance = euclideanDistance(centroids[j, :], data[i, :])
                if distance < minDist:
                    minDist = distance
                    minIndex = j

            if clusterBound[i, 0] != minIndex:
                processing = True
                clusterBound[i, :] = minIndex, minDist ** 2
        index[:, 0] = clusterBound[:, 0]
        for j in range(k):
            newCentroid = data[np.all(index == j, axis=1), :]
            centroids[j, :] = np.mean(newCentroid, axis=0)
    print("k means finished!")
    visualization(centroids, clusterBound, data)


def visualization(centroids, clusterBound, data):
    plotMarkList = ['oy', 'og', 'or', 'oc', '^m', '+y', 'sk', 'dw', '<b', 'pg']
    centroidMarkList = ['Dr', 'Dc', 'Dm', 'Dy', '^k', '+w', 'sb', 'dg', '<r', 'pc']
    k = centroids.shape[0]
    count = data.shape[0]
    if data.shape[1] != 2:
        print("too many dimensions to draw :(")
        return
    if k > len(plotMarkList):
        print("too many centroids to draw :(")
        return
    for i in range(count):
        mark = plotMarkList[int(clusterBound[i, 0])]
        plt.plot(data[i, 0], data[i, 1], mark)
    for i in range(k):
        mark = centroidMarkList[i]
        plt.plot(centroids[i, 0], centroids[i, 1], mark)
    plt.show()


if __name__ == "__main__":
    kmeans(3)
```