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

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

Problem Description

Definition

Cluster centroid is the middle of a cluster. A centroid is a vector that contains one number for each variable, where each number is the mean of a variable for the observations in that cluster. The centroid can be thought of as the multi-dimensional average of the cluster.

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 $(x_1, x_2, x_3 \ldots x_n)$ in d-dimension. And we need to choose k centroids to partition the n vectors into k types T $(T_1, T_2 \ldots T_k)$ 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

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 $c_i = \mu_i$

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

Algorithms

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 $\mathcal{C} = \{c_1, c_2 \ldots c_k\}$
2. For each $i \in \{1, 2 \ldots k\}$, 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 $i \in \{1, 2 \ldots k\}$, set c_i to be the center of all points in C_i
4. Repeat Step 2 and Step 3 until $\mathcal{C}$ no longer changes.

Proof.

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 $f(x'') < f(x')$. □

The naive K-means algorithm do great work for the simplicity and efficiency, but it also has drawbacks. In the caes of initializing k centroids using naive K-means algorithm (usually Lloyd's algorithm), we use randomization. The initial k centroid are picked in the range of data set randomly. However, this initialization strategy could result in initialization sensitivity. The final formed clusters could be affected greatly by the initial picked centroids.

Here are a few figures showing the potential:

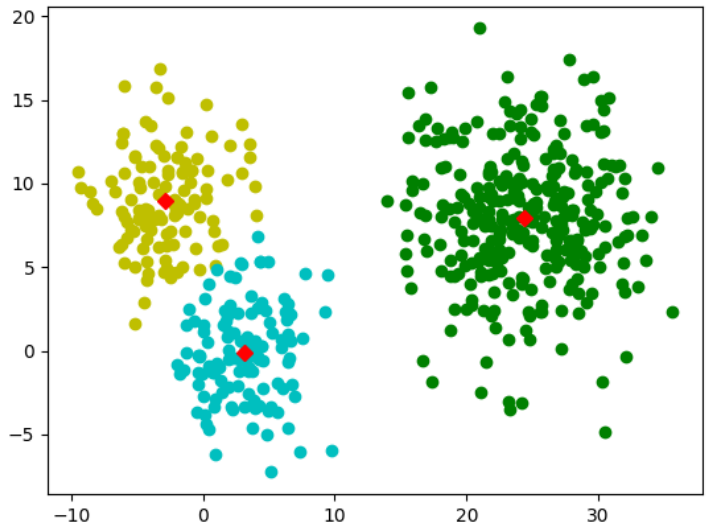

Fig: Good Cluster

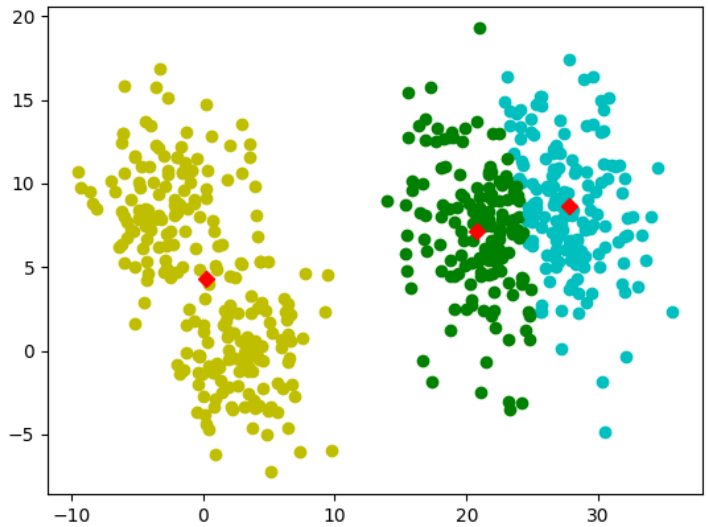

Fig: Bad Cluster

In the above images, the final formed clusters are pretty different. The good cluster's[?] initial k-centroids are initialized in different clusters, leading to a good result. While the other one's[?] initial centroids were unfortunately initialized in the same cluster and got a result which was not as expected.

In last several decades, there has been significant work on improving Lloyd's algorithm both in terms of reducing MSE and running time. The follow up work on Lloyd's algorithm can be broadly divided into three categories:

1. Better seed selection
2. Selecting ideal value for number of clusters
3. Bounds on data point to cluster centroid distance

Next, we will introduce some typical work in these categories. Section 3.3 introduces K means++ for seed selection, section 3.4 and 3.5 helps bound on data point to cluster centroid distance and section 3.6 introduces several typical methods to select ideal value for number of clusters briefly. Let's begin.

The K-means++ algorithm was proposed by David Arthur and Sergei Vassilvitskii in 2006, which outperforms K-means in terms of both accuracy and speed [?].

The K-means++ algorithm uses totally a different way to initialize the k centroids. Rather than uniformly randomly pick point in the range of all points, it uses special to make the k centroids as far away from each other as possible. And the updated algorithm does exactly the same in the iteration steps. The algorithm goes as follows:

1. Pick the first centroid c_1 randomly from the dataset S

2. Compute the distance of all points in the dataset from centroid c_1. The distance of a point x could be calculated by

$$\mathrm{Dist}(x, c_i) = \|x - c_i\|$$

3. Take a new centroid c_i, choosing $x_j \in$ from all points with probability p_j

$$p_j = \frac{\mathrm{Dist}^2(x_j)}{\sum_{x_j \in S} \mathrm{Dist}^2(x_j)}$$

4. Repeat the above steps until k cetnroids are found
5. Iteration steps are exactly the same as navie K-means algorithm

k-means++ consistently outperformed k-means, both by achieving a lower potential value, in some cases by several orders of magnitude, and also by completing faster. With the synthetic examples, the k-means method does not perform well, because the random seeding will inevitably merge clusters together, and the algorithm will never be able to split them apart. The careful seeding method of k-means++ avoids this problem altogether, and it almost always attains the optimal results on the synthetic datasets.[?]

In both K-means and K-means++ algorithm, if there are n data points in $\mathbb{R}^d$ space and k clusters for partition, each iteration involves $n * k$ distance

computations, which would significantly slow down algorithm. One contribution from Siddhesh Khandelwal[?] helps reduce this cost to $n * k'$ ($k' << k$) by generating candidate cluster list (CCL) of size k' for each data point. We'll show how this heuristic works in detail. The optimization is based on the observation that across all iterations of K-means or K-means++, a data point changes its membership only among a small subset of clusters. The heuristic considers only a subset of nearby cluster as candidates for deciding membership for a data point. This heuristic has advantage of speeding up K-means and K-means++ clustering with marginal increase in loss functuion(MSE in our case). Note that the optimization can be applied in any algorithm to solve K-means problem with steps to caculate distance between points and centers, acting as augmentation. Let A be any variant algorithm of K-means problem and B be the same variant augmented with this geuristic. Let T be the time required for A to converge to MSE value of E. Let T' be the time required for B to converge to MSE value of E'. Our target is:

- For convergence time: $T' < T$
- For loss: $E' \leq E$ or $E' \overset{\text{marginally}}{>} E$

We assume that k' is significantly smaller than k. We will show how to

choose k' later.We build candidate cluster list (CCL) based on top k' nearest clusters to the data point after first iteration of K-means.

Consider a data point p1 and cluster centroids represented as c_1, c_2..., c_k. We assume that $k' = 4$, and $k' << k$. After first iteration of K-means c_5, c_6, c_8, and c_{11} are the top four closest centroids to p_1 in the increasing order of distance. This is the candidate cluster list for p_1. If we run K-means for second iteration, p_1 will compute distance to all k centroids. After second iteration, there are two possible cases:

1. The list do not change but only members'ranking changes.
2. Several members of the centroids in the previous list are replaced with other centroids which were not in the list.

It seems that the augmentation makes no sense. But what makes this method succeed is real world data rarely makes case 2 happen. That is, the set of top few closest centroids for a data point remains almost unchanged even though order among them might change. Overhead analysis:

- Computation overhead: O(nklog(k)) for creating CCL at first. We have to compute the distance to each cluster's centroid for each point and sort them to create CCL.

- Memory overhead:O(nk′) to maintain CCL.

In fact, the previous augmentation [?] makes trade-off between loss function and running time. Still based on the observation of unnecessary calculation in the process of iteration, Charles Elkan put forward a more strict method to avoid unnecessary computation.[?] The key idea is to bound on data point to cluster centroid distance and use triangle inequality to avoid redundant computations of distance between data points and cluster centroids. The accelerated algorithm applyies the triangle inequality in two different ways, and keeps track of lower and upper bounds for distances between points and cluster centroids. Its inspiration is based on the fact that most distance calculations in standard K-means are redundant. If a point is far away from a centroid, it is not necessary to calculate the exact distance between the point and the centroid in order to know that the point should not be assigned to this centroid. Conversely, if a point is much closer to one center than to any other, calculating exact distances is not necessary to know that the point should be assigned to the first center. There are 3 properties the accelerated K-means algorithm should satisfy:

- Start with any initial centers, so that all existing initialization methods incuding original K-means and K-means++ can continue to be used.

- Correct results, it always produces exactly the same final centers as the standard algorithm.

- Support any black-box distance metric, so it should not rely for example on optimizations specific to Euclidean distance.

Obviously, It's more strict than previous method. This stronger properties mean that more basic K-means algorithms are able to install this augmentation. e.g., heuristics for merging or splitting centers can be used together with the new algorithm.(e.g. ISODATA, which not memtioned in our note)

The algorithm firstly find the only black box property of any distance metrics.That is: for any 3 points x, y, z, satisfy triangle inequality:

$$\mathrm{d}(\mathrm{x}, \mathrm{z}) \leq \mathrm{d}(\mathrm{x}, \mathrm{y}) + \mathrm{d}(\mathrm{y}, \mathrm{z})$$

in which d() means distance function. Let x be a point and let b and c be centers, we need to know that $\mathrm{d}(\mathrm{x}, \mathrm{c}) \geq \mathrm{d}(\mathrm{x}, \mathrm{b})$ in order to avoid calculating $\mathrm{d}(\mathrm{x}, \mathrm{c})$. Now I'd like to introduce 2 Lemma.

Lemma

Let x be a point and let b and c be centers. If $d(b, c) \geq 2d(x, b)$ then $d(x, c) \geq d(x, b)$.

Proof.

Use triangle inequality, $d(b, c) - d(x, b) \leq d(x, c)$. Use "if" in lemma, we can get the conclusion. □

Lemma

Let x be a point and let b and c be centers.$d(x, c) \geq \max(0, d(x, b) - d(b, c))$.

Proof.

Use triangle inequality, with $d(x, c) \geq 0$, easily get lemma 5. □

Let x be any data point, let c be the center to which is currently assigned, let s become any other center. So with the lemma, we can assert that:

Claim

if $\frac{1}{2}d(c,s) \geq d(x,c)$ then $d(x,s) \geq d(x,c)$, and we don't need to compute $d(x,s)$. (Proved by lemma)

Claim

Suppose that we don't know $d(x,c)$ exactly, and we do know an upper bound u such that $u \geq d(x,c)$: For any other possible choice, we only need to compute $d(x,c), d(x,s)$ iff $u > \frac{1}{2}d(c,s)$.(Proved by claim)

Claim

Suppose that $u \leq \frac{1}{2}d(c,s)$ for any possible s, all distance calculations for x can be avoided. (Proved by claim)

Next, Let x be any data point, let c be any center, let s become previous version of smae center. Suppose that in the previous iteration we knew a lower bound g such that $d(x,s) \geq g$. Then we can infer a lower bound h for current iteration:

$$d(x,c) \geq \max\{0, d(x,s) - d(s,c)\} \geq \max\{0, g - d(s,c)\} = h$$

This can be easily proved by lemma 2. So we can assert that:

Claim

If center moved a small distance(d(s, c)issmall), the lower bound only make a small move.

In practical application, as the centers are converging to their final positions, the vast majority of the data points have the same closest center from one stage to the next. A good algorithm would exploit this coherence to improve running time. We use u(x) to represent upper bound of distance between a given point x and its currently assigned center c. $l(x, c')$ is the lower bound on the distance between x and some other center c'. If $u(x) \leq l(x, c')$, we don't need to calculate $d(x, c), d(x, c')$.

Initially, we set $l(x, c) = 0$ for each point x and center c. Then assign each x to its closest initial center, using Lemma 1 to firstly reduce redundant distance computations. Each time d(x, c) is computed, set $l(x, c) = d(x, c)$. At last, set upper bounds $u(x) = \min_c(d(x, c))$. Then repeate this until convergence.(Each time d(x, c) is calculated, we update $l(x, c)$. Similarly u(x) when computing d(x, c(x)).)

1. For all centers c and c', compute $d(c, c')$. Set $s(c) = \frac{1}{2}\min_{c \neq c'} d(c, c')$.
2. Identify all points x such that $u(x) \leq s(c(x))$. (Refer to claim 8)
3. For each pair of remaining x and c, which satisfy: i) $c \neq c(x)$ and ii) $u(x) > l(x, c)$(obviously) and iii) $u(x) > \frac{1}{2}d(c(x), c)$(Claim7):
   - 3.1 If $r(x) =$ true, compute $d(x, c(x))$ and assign $r(x) =$ false. Otherwise, $d(x, c(x)) = u(x)$. (We don't have to update according to Claim 9)
   - 3.2 If $d(x, c(x)) > l(x, c)$ or $d(x, c(x)) > \frac{1}{2}d(c(x), c)$, then compute $d(x, c)$ and decide if swap c for x. (Claim 7)
4. For each center c, compute centroid, store in $m(c)$.
5. For each pair of x and c, set $l(x, c) = \max(l(x, c) - d(c, m(c)), 0)$
6. For each point x, set $u(x) = u(x) + d(m(c(x)), c(x))$
7. For each point x, set $r(x) =$ True
8. Really replace c by $m(c)$

Logically, step (2) is redundant because its effect is achieved by condition 3(iii). Computationally, step (2) is beneficial in real experiment because reduce x's size for later steps. And note that $u(x)$ and $c(x)$ may change during the execution of step (3), so we can't discard condition 3(iii).

The most significant part is step (3), we use many strictions to avoid redundant computations. In 3(a) step, we only compute $d(x, c(x))$ for at

most one time. Just when in pervious step (7) r(x) is set to True, we update d(x, c(x)), or else we use u(x) to replace. Why step 3's strictions are effcient is that at the start of each iteration, the upper bounds and lower bounds for x are tight enough. If at j^{th} iteration is tight, it'll be tight at $(j+1)^{th}$ iteration, because the location of most centers changes only slightly, and hence the bounds change only slightly.

Compared to original algorithm, in 6 typical benchmark, using this optimazation speeds up algorithms from 11.3× to 351×.

How to evaluate goodness of clustering for various potential values of number of clusters? We will introduce several common methods.[?, ?, ?, ?] Let's review our optimization. We speed up the algorithm with either lenient or strict conditions and seed the initial K center properly. But still exists the only hyper-parameter we don't know how to tune. So let's begin.

We choose a set of K and compare their performance. In this set, we need to choose k significantly smaller than the number of objects in the data sets and let it be resonably large.

There are several statistical measures available for selecting K. These measures are often applied in combination with probabilistic clustering

approaches. They are calculated with certain assumptions about the underlying distribution of the data. The Bayesian information criterion is calculated on data sets which are constructed by a set of Gaussian distributions. Monte Carlo techniques, which are associated with the null hypothesis, are used for assessing the clustering results and also for determining the number of clusters.

Visual verification is applied widely because of its simplicity and explanation possibilities. In my own practice, I usually use PCA or other methods to draw points on a planar graph to check how many clusters exist in Machine Learning course's project. But it has many restrictions. e.g. The application of visualization techniques implies a data distribution continuity in the expected clusters. In fact, Visual examples are often used to illustrate the drawbacks of an algorithm.

When K-means clustering is used as a pre-processing tool, the number of clusters is determined by the specific requirements of the main processing algorithm. No attention is paid to the effect of the clustering results on the performance of this algorithm. In such applications, the K-means algorithm

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.[?, ?] For lack of space, we won't describe their proof in detail. For reduction from planar 3-SAT to k-means, the author corresponds a simple circuit to each variable x_i in 3-SAT. And each circuit has an even number of vertices marked on it. Circuits are partitioned into pairs of adjacent vertices according to the value of x_i. Then, each edge becomes a representation of value. The authors set the distance between vertices and uniquely determine a layout that gives a correct reduction from planar 3-SAT to planar k-means. Then we can use K-means to solve 3-SAT. That is, K-means problem is

Conclusion

Application

In real world, K-means are widely used in clustering for its simplicity and effcient. Its core concept is extremely easy to understand and the updated version algorithm itself only cost a few iterations to output pretty good results. And here are a few applications of K-means algorithm in real life.

The segmentation of the customers' base allows operators to better serve customers and target advertising precisely to them[?]. The concept of segmentation relies on the high probability of persons grouped into segments based on common demands and behaviours to have a similar response to marketing strategies.

And the concept of segmentation coincides with that of clustering. A company can run K-means analysis base on customers' consumption habits, browsing histories, financial conditions and many other features to precisely partition their customers into different clusters so that they can implement different strategies.[?] The K-means algorithm is used to optimize truck-drone in tandem delivery network. One of the objectives is to investigate the time, energy, and costs associated to a truck-drone delivery network compared to standalone truck or drone.

In 2016, Sergio Ferrández, Timothy Harbison, Troy Weber, Robert H. Sturges and Robert Rich proposed a method[?], optimizing the delivery

netword of drone powerd by K-means algorithm to find the appropriate launch locations with the minimum cost. The optimal solution is determined by finding the minimum cost associated to the parabolic convex cost function. Clustering documents in multiple categories based on tags, topics, and the content of the document is a very standard classification problem and k-means is a highly suitable algorithm for this purpose.[?]

All we need to do is to pre-processing the documents, representing each of them as a high-dimension vector and using term frequency to identify commonly used terms that help classify the document. The document vectors are then treated as input of K-means algorithm to help identify similarities in document groups.[?]

Now we'll present a k-means clustering-based COVID-19 analysis to determine the clusters according to the health care quality of the countries. In fact we don't think this research meaningful, but just refer to their way to use K-Means in application.[?]

The team uses we use Principal Component Analysis (PCA) whiledetermining the centroids.

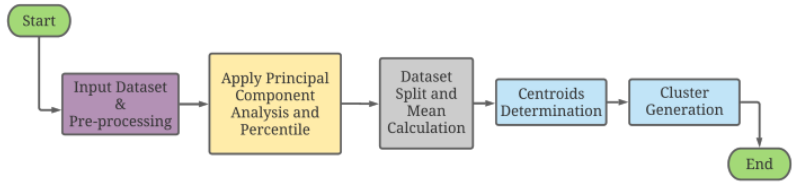

Fig: Flowchart of centroids deciding

With this method to decide on centroids, the author argues that their method uses much less iterations and execution time for convergence. Compared with existing K-means++, their algorithm's performance is constant and faster.

But to be honest, we have to argue that their research just used K-means to generate clusters at last. In fact, after PCA, we can clearly observer that there are 3 kinds of COVID-19 situations over the world.(which fits our impression) What K-means does just verifies countries in detail. That is, without enough knowledge of dataset, we have to decide on cluster number with other methods. After that, K-means will generate clusters for us

Median Trick

So far, we have an algorithm A which estimates in correct range of with probability ≥ 0.9. Our new algorithm A^* will output in range of 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 (Chernoff/Hoeffding Bound)

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

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

Claim

$E[X_i] = 0.9$, and $E[\mu] = 0.9m$

Proof.

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

Claim

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

Proof.

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

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

Proof.

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

Using Chernoff bound,

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

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

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.

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

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

$E[z] = d + 1$

Proof.

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

$var[z] \leq 2/d^2$

Proof.

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

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

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

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

Claim

The following claims are stated:

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

Proof.

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

- Define $X_i = 1$ iff $h(i) < \frac{k}{(1+\epsilon)d}$
- 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 $h(i) < \frac{k}{(1+\epsilon)d}$

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

- $E[X_i] = \frac{k}{(1+\epsilon)d}$

  $E[\Sigma_i X_i] = dE[X_i] = \frac{k}{1+\epsilon}$

  $var[\Sigma_i X_i] = dvar[X_i] \leq dE[X_1^2] \leq \frac{k}{1+\epsilon} \leq k$

  (Since $X_1 \in \{0, 1\}$, $E[X_1^2] = E[X_i]$)
- By Chebyshev:

  $\Pr[|\Sigma X_i - \frac{k}{1+\epsilon}| > \sqrt{20k}] \leq 0.05 \implies \Pr[\Sigma X_i > \frac{k}{1+\epsilon} + \sqrt{20k}] \leq 0.05$
  - (For $\epsilon < 1/2$ and $k = c/\epsilon^2$)

    $\frac{k}{1+\epsilon} + \sqrt{20k} \leq k(1 - \epsilon + \epsilon^2) + \sqrt{20k}$ (Taylor Series Expansion)

    $\leq k - k/2 + 5\cdot\sqrt{c}/\epsilon = k - c/2\epsilon^2 + 5\cdot\sqrt{c}/\epsilon$

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

$h : [n] \to \{1, 2, \ldots M\}$ 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 (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).

Proof.

$h(i) = a, h(j) = b$ 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 $\frac{1}{M^2}$. □

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.

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

We can recover the x of the stream given the memory state σ

Proof.

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 $s \ll n, m$.

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

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

Proof.

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

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

K-means Algorithm Code in Python

[language=Python] import math import matplotlib.pyplot as plt import pandas as pd import numpy as np

def loadData(): df = pd.read$_c$sv(" ./data/data.csv")returndf.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]ifdata.shape[1]! = 2 :
print(”toomanydimensionstodraw : (”)returnifk > len(plotMarkList) :
print(”toomanycentroidstodraw : (”)returnforiiinrange(count) : mark =
plotMarkList[int(clusterBound[i, 0])]plt.plot(data[i, 0], data[i, 1], mark)foriiinrange(k
mark =
centroidMarkList[i]plt.plot(centroids[i, 0], centroids[i, 1], mark)plt.show()

if $_{\mathrm{name}_{==”}\,_{\mathrm{main}_{”}}:\mathrm{kmeans}(3)}$