Adaptive Monte Carlo

Applications on k-nearest neighbors and Medoid Computation

Group 7
Ziyi Song
Kexin Zhang

Era of Big Data

e.g. : 1.3 Million Brain Cells from E18 Mice. 10x Genomics, 2017.

1.3 million cells	(# points)
28 thousands genes	(dimensions)

e.g.: Netflix prize dataset

480 thousands users	(# points)
17,770 movies	(dimensions)

- Traditional Monte Carlo method cannot solve these large-scale problems
- But do we really need to use all dimensions or points to compute an estimate?
- A new technique: Adaptive Monte Carlo Computation
- Adaptive Monte Carlo computation bases on Monte Carlo method and Multi-armed Bandit (MAB) problem

Multi-armed Bandit

A gambler is facing at a row of slot machines. At each time step, he chooses one of the slot machines to play and receives a reward. The goal is to maximize his return.

Multi-armed Bandit

Mean Reward μ_1 μ_2 μ_3 $\cdots$ μ_n Unknown

Round 1

Round 2

Round 3

$\vdots$

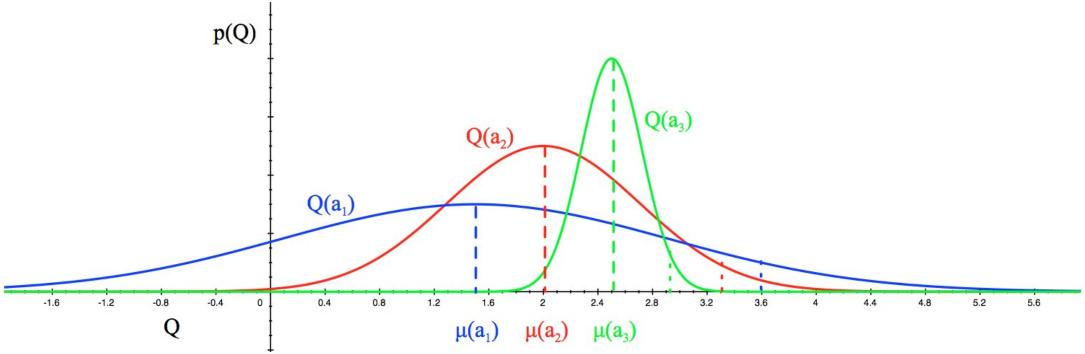
Everytime we pull an arm, the machine provides a random reward from a probability distribution specific to that machine.

Best arm is the arm with the largest reward. So we want to select the Best Arm with high probability and minimum number of arm pulls.

Bandit Strategies

➔ Exploring vs. Exploiting:
  - No exploration: the most naive approach and a bad one
  - Exploration at random
  - Exploration smartly with preference to uncertainty - **U**pper **C**onfidence **B**ound Algorithm

➔ Be optimistic about options with high uncertainty:



K-nearest neighbor problem

Assume we have an $(n \times d)$ data matrix corresponding to n points: $x_1, ..., x_n \in \mathbb{R}^d$

Goal: find the nearest neighbors in l_p distance.

$$f(i) = \frac{1}{d} \sum_{t=1}^{d} (x_{1,t} - x_{i,t})^2$$

Design a sequence of estimators for f(i) with increasing accuracy:

- We can use evaluations of $\hat{f}_l(i)$ to construct confidence interval on f(i).
- Updating $\hat{f}_l(i)$ to $\hat{f}_{l+1}(i)$ is computationally cheap.

Reformulate KNN as a multi-armed bandit problem

Create a UCB object and initialize it by adding other data points as arms;
For each arm A_l, compute a (1-δ) confidence intervals;
$\boldsymbol{B} = \{\}$ ◇ Set of k best arms;
$\boldsymbol{S} = \{A_l : l \in [n]\}$ ◇ Set of arms under consideration;
for *t in 1:MAX_Iteration* **do**
  Pick top arm A_t from $\boldsymbol{S}$;
  if A_t *is not evaluated enough times* **then**
    Pull it again: improve the CI and mean estimate of the arm A_t by updating the estimator by one more step;
  else
    Use brute force evaluation;
  end
  if *the UB of A_t is smaller than the LB of any other arms in $\boldsymbol{S}$* **then**
    Add it to $\boldsymbol{B}$; Remove A_t from $\boldsymbol{S}$;
    Check if already have k nearest neighbors;
  else
end

- ➔ Each arm corresponds to each contending point
- ➔ Each arm's reward corresponds to f(i)
- ➔ Each pull of arm i corresponds to generating a sample to update estimate of f(i)
- ➔ An estimator after l pulls of an arm i:

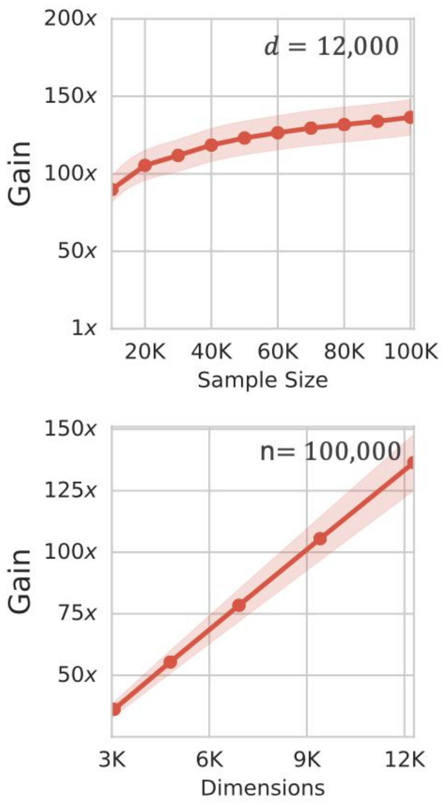
$$\hat{f}_l(i) = \frac{1}{l}\sum_{k=1}^{l}(x_{1,t_k} - x_{i,t_k})^2$$

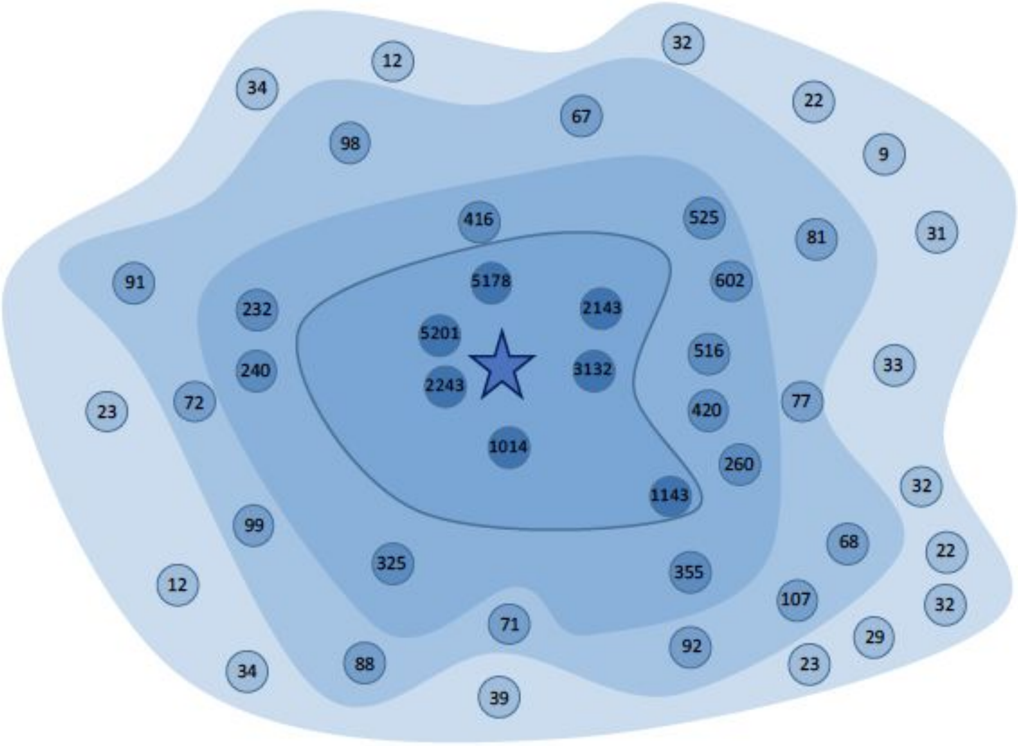
$t_1, \ldots, t_l$ are uniformly sampled.

Implementation of knn with UCB

- Data Set: Tiny-imagenet-200 (val), 10000*12288
- Create reference classes
  - ‘Arm’ class:
    - Each data point (except the reference point) is an arm
    - Stores information like upper and lower bounds, how many times this arm has been pulled, and the estimator of distance of this arm, etc.
  - ‘UCB’ class:
    - For each data point, use UCB algorithm to find k-nearest neighbors
    - Stores information like arms (ranked based on lower bound), sample size, delta and sigma, etc.
- Compare with brute force method

Time complexity analysis





arXiv:1805.08321

Medoid Computation

Medoid is a representative point of a cluster whose average distance to all other points in the cluster is minimal

$$\text{Medoid} = \min_{y \in \{x_1, \ldots, x_n\}} \frac{1}{n} \sum_{i=1}^{n} dist(x_i, y)$$

- n = number of points
- d = dimension

Medoid Computation

Comparison

Multi-armed Bandits	Medoid Computation
arms	each point
mean reward	average distance of a point to all the other points
arms pulls	evaluating the distance of that point to a randomly chosen point

Medoid Computation (UCB) Algorithm

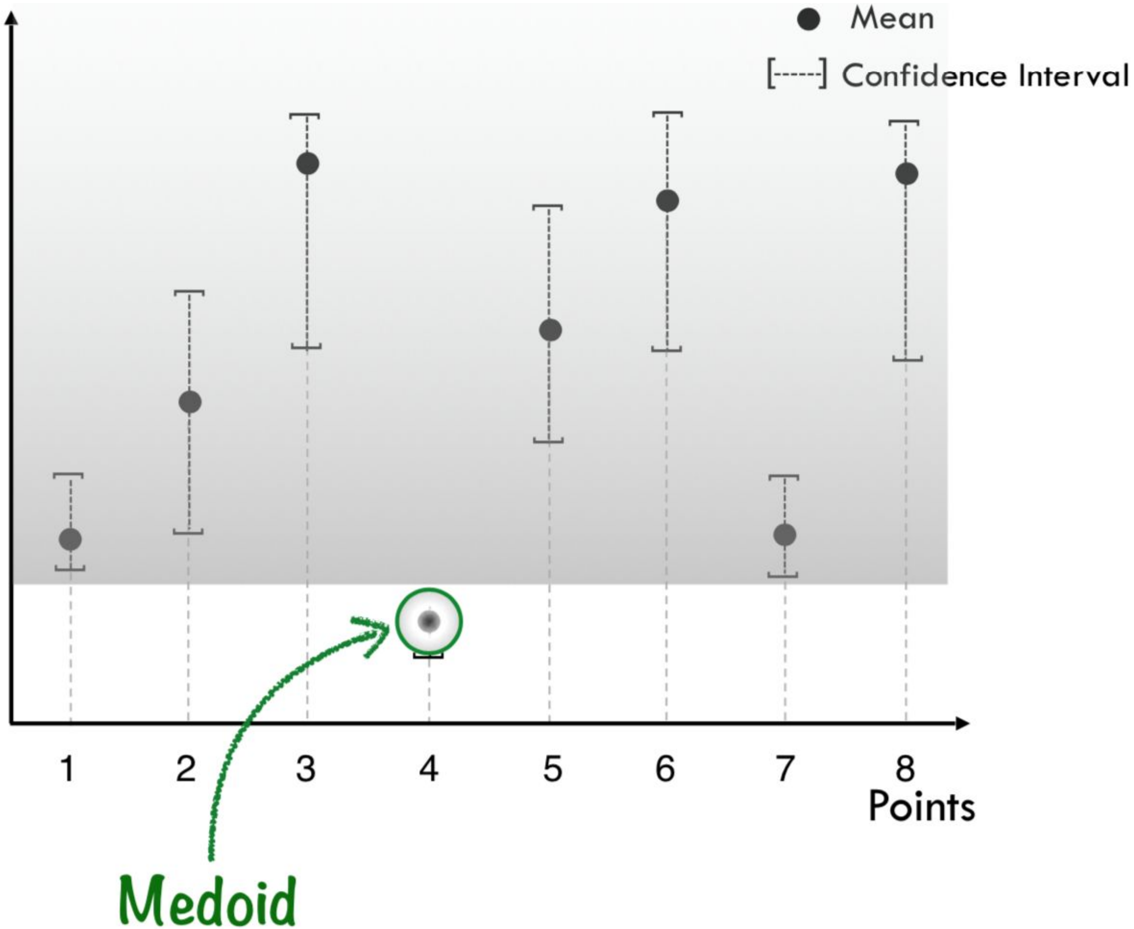
Simplification:

1. Initialization: evaluate distances of each point to a randomly chosen point and build a (1-δ)-confidence interval (CI) for the mean distance of each point i

2. **While True**

   - At every iteration, pick a point that has minimal lower CI bound among all points
   - Evaluate the distance of this point to a randomly chosen point and update its CI
   - **If** there exists a point such that its upper CI bound is smaller than the lower CI bounds of all other points, **break**

Just to give you some sense



Complexity analysis and comparison

To find the medoid of a group of high-dimensional data:

- **PAM** algorithm takes $O(n^2)$ distance evaluations
- **RAND** algorithm takes $O\left(\frac{n \log n}{\epsilon^2}\right)$ distance evaluations to approximate the medoid
- **TOPRANK** algorithm takes $O(n^{\frac{5}{3}} \log^{\frac{4}{3}} n)$ distance evaluations to find the medoid
- **Trimed** algorithm takes $O(n^{\frac{3}{2}} 2^{\Theta(d)})$ distance evaluations
- **Adaptive Monte Carlo** takes $O(n \log n)$ distance evaluations, **almost linear !**

My experimental results, comparing with pam() in R cluster package

Dataset: 1.3 Million Brain Cells from E18 Mice. 10x Genomics, 2017.

Sub-dataset: 5,000 points, 27,998 dimensions

- pam() takes more than **2 hours** to find the 636-th point as the medoid
- Adaptive Monte Carlo implementation find the 636-th point:

  in around 100 seconds

  73 distance evaluations per point on average

  stable, return the right answer every trial

My experimental results

Sub-dataset: 20,000 points, 27,998 dimensions

- Adaptive Monte Carlo implementation find the 7375-th point

in around 1,000 seconds

95 distance evaluations per point on average