Adaptive Monte-Carlo Optimization

Group 7: Kexin Zhang, Ziyi Song

12/17/2019

1 INTRODUCTION

K-nearest neighbors (KNN) and medoid computation are computationally expensive problems in clustering, especially for huge data points in high dimensions. A general problem in KNN and medoid computation is: $\min_{i \in S} f(i)$, where S is a data set, i is each point, and f is a computationally expensive function, e.g., average distance of point i to all other points. *Monte Carlo method* will accurately estimate each $f(i)$ by random samplings and then compute the minimum, which is computationally prohibitive and unnecessary. To solve such discrete optimization problem, *adaptive Monte Carlo method* was proposed[1], combining *Monte Carlo method* and the celebrated *Multi-armed Bandit* (MAB) problem.

In our project, we utilize *adaptive Monte Carlo method* to improve KNN and medoid computation implementation on time complexity savings with accuracy guaranteed. In Section 2.3.1 and 2.4.1 we elaborate how we reformulate KNN and medoid computation into MAB problem framework and our solutions for the reformulated problems. Our solutions are essentially different variants of the well-known upper confidence bound (UCB) algorithm[2], which solves MAB problem with great computation savings and high accuracy. In Section 3.1 and 3.2 we present our improvements based on experimental results in R implementation.

2 METHODS

2.1 MAB description

We have a bandit machine with many arms in a row. Each arm has its unknown mean reward. Every time we pull an arm, the arm gives us a random reward with respect to its probability distribution. The Best Arm is the arm with the largest mean reward. We want to find the Best Arm with high probability and minimal arm pulls.

2.2 Upper Confidence Bound Algorithm

The most naive strategy to solve a *MAB* problem is to try all arms at each time step and select the one that gives the maximum reward. This strategy only exploits and does not explore other options. Another commonly used strategy is ϵ-greedy algorithms because it chooses an arm that gives maximum reward with probability (1-ϵ) so that we can explore some other options.

However, one major concern with these two strategies is inefficiency since the history of each arm is not considered during the decision making step. And in this project, the strategy we implemented is **U**pper **C**onfidence **B**ound (UCB) algorithm which chooses an arm based on an uncertainty estimation. In other words, we are optimistic about the arm that is most uncertain. At each step, the arm with highest variance is pulled and its confidence interval will be updated

2.3 *k*-nearest neighbors problem

k-nearest neighbors, or k-NN problem is to find the k-nearest data points for a reference data point based on a certain distance matrix. Assuming we have a $n \times d$ matrix X with n points $(x_1, x_2, ..., x_n \in \mathbb{R}^d)$, to simplify the implementation, we will fix x_1 as the reference point and identify its first k nearest points from $x_2, ..., x_n$ based on squared euclidean distance $f(i)$:

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

where $i \in \{2, ..., n\}$.

2.3.1 Reformulate *k*-NN into MAB problem

To reformulate k-NN problem into a MAB problem, we considered each contending point x_i $(i \in \{2, ..., n\})$ as an arm and the distance between the contending point and the reference point as the reward of an arm. We generated an estimator $\hat{f}(i)$ for distance $f(i)$ during each pull and based on that estimator, we constructed (or updated) the confidence interval.

An estimator after l pulls of arm i is:

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

where t_k is the uniformly sampled dimension from original d dimensions for estimating distance. The computation cost for updating arm from l pulls to $l+1$ pulls is $\mathcal{O}(1)$:

$$\hat{f}_{l+1}(i) = \frac{l}{l+1}\hat{f}(i) + \frac{1}{l+1}(x_{1,t_{l+1}} - x_{i,t_{l+1}})^2. \tag{3}$$

According to the paper[1], $\hat{f}(i)$ has σ-sub-Gaussian tails, which essentially allowed us to build confidence intervals. The modified UCB algorithm (for k-NN problem) is:

Algorithm 1: k-NN UCB

Output: Find k nearest neighbors of x_1 based on squared euclidean distance
. **Input:** $\{\mathcal{A}_l : l \in [n]\}$, k, δ, p (number of initial pulls), s (sample size)
initialization For each arm, pull p times to generate initial estimator and $(1-\delta)$ confidence interval;
$\mathcal{B} = \{\}$ ⋆ Empty set of best arms;
$\mathcal{S} = \{\mathcal{A}_l\}$ ⋆ Set of remaining points;
while $\mathcal{B}$ *has fewer than* k *elements* **do**
  At iteration t, pick the "best" arm $\mathcal{A}_t$ from S which has smallest lower confidence bound;
  if $\mathcal{A}_t$ *is not evaluated enough times* **then**
    Pull $\mathcal{A}_t$ again to update the estimator and confidence interval;
  else
    Calculate exact distance by brute force function;
  end
  if *the upper bound of* $\mathcal{A}_l$ *is smaller than the lower bounds of the remaining arms* **then**
    Add it to $\mathcal{B}$;
    Remove it from $\mathcal{S}$;
  else
end

2.3.2 Time complexity analysis

The naive brute force method will find k-nearest neighbors for all n points in $\mathcal{O}(n^2d)$ since it will calculate pair-wise distances for all points. In comparison, according to the paper[1], the modified UCB algorithm should be able to find the answer in $\mathcal{O}(nklog(dn)log(k))$ which will significantly reduce computation time when n or d is very large.

This speed-up is due to the fact that UCB searches with a preference to the uncertainty of an arm. In other words, instead of pulling each arm equal times, it will focus on arms that are more promising, that is, closer to the reference point, since we only look for top k arms.

2.4 Medoid

Without loss of generality, we consider finding the single medoid of a cluster. In a cluster, medoid is the point with the smallest average distance to other points, i.e., medoid $= \min_{y \in \{x_1,\ldots,x_n\}} \frac{1}{n}\sum_{i=1}^{n} dist(x_i, y)$

2.4.1 Reformulate Medoid into MAB problem

According to the paper[3], We take each point as an arm whose unknown mean reward is its accurate average distance to all other points. At each iteration, pulling an arm is evaluating the distance of that point to a randomly chosen point. It helps us to update the estimation of the arm's accurate average distance. The Best Arm is the medoid with the smallest average distance.

Algorithm 2: Medoid UCB

Input: $\{A_l : l \in [n]\}$, δ, σ, m (step size, i.e., make m distance evaluations of x_i at a time)

Output: Find the medoid of a cluster of data

initialization For each point, evaluate its distance to m randomly chosen points, generate its initial estimator and its $(1-\delta)$ confidence interval;

while *TRUE* **do**

  At iteration t, pick the "best" point $\mathcal{A}_t$ which has the smallest lower confidence bound;

  if $\mathcal{A}_t$ *is evaluated less than* $n-1$ *times* **then**

    evaluate $\mathcal{A}_t$ again to a randomly picked point and update estimation and confidence interval of $\mathcal{A}_t$;

  else

    compute exact average distance of $\mathcal{A}_t$ and set its confidence interval as 0;

  end

  if *there exist* $\mathcal{A}_t$ *such that its upper bound is smaller than the lower bounds of all the other points* **then**

    return this $\mathcal{A}_t$

  end

end

Simply speaking, our philosophy here is that, during iterative estimation on each point's true average distance, we select promising points, focus on them, and continuously narrow down the set of promising points. We regard points whose average distance closes to that of the medoid as promising points, and accurately estimate average distance of these promising ones, while coarsely estimate other points.

For point x_i, $d_{i,j}$ denotes distance from point x_i to point x_j, so the accurate average distance is $\mu_i = \frac{1}{n}\sum_{j=1}^{n} d_{i,j}$. Every time We randomly choose a point $x_{l,l\neq i}$, from all other points with replacement, compute the distance from point x_i to this x_l, which completes one distance

evaluation on point x_i. After k distance evaluations on point x_i, we estimate $\hat{\mu}_i = \frac{1}{k}\sum_{j=1}^{k} d_{i,j}$. Thus we can easily update $\hat{\mu}_i$ in $\mathcal{O}(1)$ after each distance evaluation on x_i.

We assume that the random samples $d_{i,1}, \ldots, d_{i,k}$ and independent and follow σ-sub-Gaussian, so we can construct confidence interval for $\hat{\mu}_i$. Let $T_i(t)$ denote the number of distance evaluation on point x_i up to time t. With confidence interval of $1-\delta$, $\mu_i \in [\hat{\mu}_i(t) - C_i(t), \hat{\mu}_i(t) + C_i(t)]$, where $C_i(t) = \sqrt{\frac{2\sigma^2 log\frac{2}{\delta}}{T_i(t)}}$. It has been proved that the number of distance evaluations has an upper bound before obtaining the medoid.

2.4.2 Time complexity analysis

The cost of naive exact computation and the famous PAM algorithm is $\mathcal{O}(n^2)$ because there are n^2 pairwise distances in a set of n points. With regard to other algorithms for medoid computation, for example, TOPRANK takes $\mathcal{O}(n^{\frac{5}{3}} log^{\frac{4}{3}} n)$ distance evaluations to find the medoid; Trimed algorithm takes $\mathcal{O}(n^{\frac{3}{2}} 2^{\Theta(d)})$ distance evaluations, where d is the number of data dimensions. In comparison, the modified UCB algorithm for Medoid takes $\mathcal{O}(nlogn)$ distance evaluations, almost linear.

In the modified UCB for medoid computation, the number of distance evaluations is independent of data dimension d. This algorithm can perform even much better if points are in higher dimensions. It is because that, with higher dimensions, the distances are more likely to be Gaussian-like distributed.

3 Experimental Data and Results

3.1 Experiments for k-NN

The data set we tested our algorithm on is the tiny-imagenet-200 dataset with 10000 images and $64 \times 64 \times 3$ dimensions.

To implement the UCB algorithm for k-NN and other similar use cases, we created two reference classes in R: *Arm* and *UCB*. Class *Arm* stores information for a data point including upper and lower confidence bounds, number of pulls, estimation of distance etc. Each contending point (or reference point) will be created as an arm and using the member functions we can pull it to generate an estimator using s uniformly sampled dimensions and improve confidence interval. Class *UCB* stores information for a UCB run including a sorted arm queue, δ, sample size etc.In a UCB run, we first initialized the UCB object by adding and pulling arms to get a crude estimation and build initial confidence interval for each point. Then, based on uncertainty, we chose the "best" arm at each iteration to further evaluate.

We tested our modified UCB algorithm on the tiny-imagenet data set. As shown in Table 1, the running time is greatly reduced when using UCB algorithm. Note that for simplicity, we fixed x_1 as the reference point but the results should be similar when calculating k-NN for all data points. To test the accuracy of the algorithm, we compared the nearest neighbors found by UCB and the nearest neighbors found by brute force method with the output of external package "nabor" (using k-d tree algorithm). All three methods identified the first 5 nearest neighbors for x_1: x_{919}, x_{3722}, x_{5979}, x_{4542}, x_{7334}.

Table 1: Running Time for Tiny-Imagenet-200 Data Set

Method	user	system	elapsed
UCB	2.881	0.014	2.894
Brute Force	9.044	1.717	10.761

3.2 Experiments for Medoid

As for experiments on medoid computation, We utilized the celebrated *1.3 Million Brain Cells from E18 Mice* data set, containing 109,140 points in 27,998 dimensions. We manually and randomly selected two subsets from the original big data set. One subset, named after *gene5k*, is of 5,000 points in 27,998 dimensions; another subset, named after *gene20k*, is of 20,000 points in 27,998 dimensions. Let us focus on *gene5k* and *gene20k* hereafter.

To make a comparison, we conducted the same data with R's pam() function in Cluster package, a famous algorithm which was well developed in 1990s and has been continuously updated in R. We regard experimental results from pam() are correct results. The primitive computation time of pam() is $\mathcal{O}(n^2)$ while it is much more delicately constructed in R.

Here are some details in our experiments. According to data features, we use ℓ_1 distance to compute medoid. For the sub-Gaussian parameter σ, we estimate it by cross-validation with randomly chosen points. Deciding the value of the confidence interval parameter δ is a speed-accuracy trade-off. In practice, we set δ as $\frac{1}{n}$, where n is the number of data points. All the medoid experiments are run on our MacBook Pro (Early 2015 Version).

For *gene5k* data, R's pam() takes more than 2 hours to find the 636-th point as the medoid. Our modified UCB Medoid algorithm finds the 636-th point in around 100 seconds, with around 73 distance evaluations per point on average. Remember that 5,000 distance evaluations will be needed if we use the most naive way to compute medoid here. Our implementation is also very stable, returning the right answer every trial.

For *gene20k* data, our modified algorithm finds the 7375-th point as medoid, in around 1,000 seconds, with around 95 distance evaluations per point on average. We did not verify this answer with pam() because it definitely will take too much time to wait for the results.

Since pam() takes too much time on the above two data sets, to just quickly verify correctness of our modified algorithm, we create other subsets in less dimensions. For example, we create a new subset, named after *gene5k700d*, with 5,000 points in 700 dimensions from the big original data set so that pam() can finish quickly. For this *gene5k700d*, pam() returns the 3629-th point as medoid in around only 30 seconds. Our implementation finds the same 3629-th point as medoid in around 880 seconds. The correctness of our method has been proved so far. Notice that, in low-dimensional data (e.g. 700d), we can see pam() is faster. It can be explained that, on the one hand, our modified algorithm performs better on high-dimensional data, because distances in high-dimensions more likely to have Gaussian-like behaviour corresponding to our sub-Gaussian assumption, while the algorithm takes more time to convergence in low dimensions; on the other hand, R's pam() has been built to be very powerful to deal with data in such scale.

4 CONCLUSIONS

In this project we have implemented the modified Upper Confidence Bound (UCB) algorithms proposed by Bagaria etc. in R and applied the algorithms to two reformulated multi-armed bandit (MAB) problems: k-nearest neighbor problem and medoid problem. Using adative Monte-Carlo optimization, the modified UCB algorithms allowed us to focus on candidates which exhibited high variance/uncertainty. By generating samples to update estimator and improve confidence interval, the algorithms evaluated data points that were more promising and thus greatly reducing the computational cost when we have high dimensional data. In our experiments, we applied these two modified algorithms to: the tiny-imagenet data set and subsets of 1.3 Million Brain Cells data. The results in Table 1 and Section 3.2 confirmed the speed up when using the modified UCB algorithms compared to the brute force algorithms.

5 AUTHOR CONTRIBUTIONS

- Ziyi Song selected the paper that we implemented.
- Kexin Zhang implemented k-NN modified UCB algorithm and tested it on tiny-imagenet data set.
- Ziyi Song implemented medoid modified UCB algorithm and tested it on subsets of 1.3 Million Brain Cells data.
- We both worked on the presentation slides and final project report.

6 REFERENCES

1. Bagaria, Vivek, Govinda M. Kamath, and David N. Tse. "Adaptive Monte-Carlo Optimization." *arXiv preprint, arXiv:1805.08321* (2018).
2. Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.
3. Vivek Bagaria, Govinda Kamath, Vasilis Ntranos, Martin Zhang, and David Tse. Medoids in almost-linear time via multi-armed bandits. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84, pages 500–509, 2018.