11/6/2019 Lecture 17

Sampling and Sketching

Where are we:

Talking about what to do when you need more performance from your data

Have talked about a general approach to performance optimization, storage, and parallelism in previous lectures.

Today we're going talk about algorithmic techniques we can apply -- in particular we will focus on two: random sampling and sketching.

We've actually seen a number of algorithmic techniques already, e.g.:

- When we have an large, we can break it into blocks, solve the problem on blocks, and then combine (or not, in some cases). We applied this when doing blocking for:

- entity resolution, because we only compare entities in the same block together -- avoids the N^2 search across all pairs of items

- LSH for similarity search, since LSH effectively hashes similar items together, which allows us to avoid an N^2 search -- we just have to compare items in the same buckets

- repartitioning for distributed joins, because items that join together are in the same partition, and repartitioning can be fully parallelized

- replacing a nested loops join with a hash join

(Show slides for these)

But we will see two more today. Spend most of our time on sampling.

Sampling:

Incredibly powerful way to help you save time! Traditionally used in settings where it's impossible to observe all the data (e.g., a survey of voters in a presidential election -- we can't talk to them all, so how do we estimate what they will vote.)

But powerful in scientific and observational settings too - for example, suppose we want to know the average age of students in this room? We could ask everyone, but we could also get a pretty good estimate by taking a random sample and taking their average.

How close will taking an average of a sample be?

Central limit theorem tells us!

mean = sample mean
std dev = sample std dev / sqrt(n)

(Show example code)

Note that std dev can be used to get *confidence intervals* on the mean, i.e., with 95% probability, the true mean will be within 2 x std dev of the sample mean. This is powerful because it allows me to estimate the chances I am wrong.

Note that this is also super helpful even in computing! Suppose for example I want to estimate the result of some expensive computation, such as:

- the number of dogs in a image database, using an image recognition algorithm
- the frequency with which people say the word "banana" is a bunch of audio recordings,
- an simulation of the number of cars passing through some intersection in a traffic simulator...
...

If these are expensive computations, I can use a sample to estimate statistics over them -- i.e., sampling allows me to avoid many expensive function invocations, often saving a ton of time, especially in the development phase of a model.

Problem: what if I don't want to just estimate the mean? What if I've got some other statistic i want to compute?

Some have closed forms, e..g, count, sum, like mean. But others don't! E.g., median

Cool idea: bootstrap

Intuition for method: suppose we could sample from the same dataset over and over

We want a 95% confidence interval on some statistic, say median of the dataset.

We could get a 95% confidence interval by taking N samples, compute the statistic (median) on each sample, and looking at the top 5% and bottom 5% of the medians. If N is large enough, we know our median lies in this interval.

Of course, if N is large we're going to have to take a lot of samples, which defeats the point of sampling!

Surprising result:

We can just take 1 sample, and then resample it with replacement, and this will be an accurate estimate of the statistic for most statistic.

We'll talk about why this works in a minute, but let's first look at algo and try out some code.

(Show slides)

(Show code)

Why does this work?

A random sample is an approximation of the distribution of the data.

If it’s big enough, it’s a good approximation, so resampling the sample is close to resampling from the original data.

Variation in those samples captures variation in the original data

Of course, it will miss outliers, extrema, etc.

But it will work well for a variety of descriptive statistics, including quantiles, regression errors, precision/recall estimates, etc.

When doesn't it work:

(Show slides)

BlinkDB Slides

Topic #2 -- Sketching

When we have a very large dataset, there are some statistics that can be easily computed in a bounded amount of space on the entire data set, e.g., counting the number of values, or computing the average.

What about statistics where their storage requirements grow with the size of the data, for example, counting the number of unique values in a dataset, or finding the top K largest values ("heavy hitters"). These can require a lot of space.

Why does this matter? Imagine you're trying to count the number of users who have visited each web page in your website, or who have seen each add. If you have

millions of visitors, this will amount to a lot of storage for each page.

There are a number of so called sub-linear or "sketching" algortithms for solving these problems.

(Show slide)

hyperloglog idea -- count number of distinct values using very little state

Gist of the idea: represent items in your list by their hash value, so they become large random integers, with duplicates having the same value

Measure the number of leading (trailing) 0's in each integer; call it k

An estimate of the count is 2^k --because it will take 2^k random samples before we see a random number has that many zeros on average.

This is noisy, however -- sometimes we'll get such a number very quickly. So we need to make more robust, which we do in a clever way.