CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Databricks and Stanford.

Lecture 16, 5/20/2015. Scribed by Carlos Riquelme and Sajid Zaidi.

16 Bootstrap

Let us first introduce the ideas underlying the Bootstrap algorithm.

Suppose we have a dataset $B = \{X_i, y_i\}_{i=1}^n$, where $X_i \in \mathbb{R}^d$ and $Y_i \in \mathbb{R}$. Most of times we are interested in estimating some property of the data. For example, if we assume that the data was generated according to some distribution from a parametric family $\mathcal{F}$, we may want to estimate the true parameter θ^*. People commonly choose a function to approximate this parameter given the data:

$$\widehat{\theta}_n = \widehat{\theta}_n(X_1, y_1, \ldots, X_n, y_n) = \widehat{\theta}_n(B).$$

This function is called an estimator. Once we see the data, we compute its value. Many important and commonly used estimators are unbiased, that is, they satisfy: $\mathbf{E}[\widehat{\theta}_n] = \theta^*$, where the expectation is taken over the data. In this case, an important question is how does the distribution of $\widehat{\theta}_n$ behave? —apart from its mean—. Bootstrapping methods help to answer this question.

Given B, we will create a set of datasets $B_1, B_2, \ldots, B_m$. In order to create each artificial dataset B_i we sample independently and with replacement n elements from B. In other words, each element of B_i is independently and identically distributed according to the empirical distribution that was observed in B.

Once we have the m datasets, we can compute the estimator $\widehat{\theta}_n$ for each of them, leading to:

$$\widehat{\theta}_n^{(1)}, \ldots, \widehat{\theta}_n^{(m)}.$$

There are at least two ways to read this information. We can compute their mean:

$$\overline{\theta}_n = \frac{1}{m} \sum_{i=1}^{m} \widehat{\theta}_n^{(i)}.$$

Also, we can study other properties of their distribution, like the sample variance of the estimator and confidence intervals.

How can we apply these ideas in a distributed fashion?

The most obvious approach would be to assign each sampled dataset B_i to one machine, which could subsequently compute the estimator $\widehat{\theta}_n^{(i)}$. Unfortunately, this could be tremendously expensive or even unfeasible for large datasets B.

An alternative is to use the *b out of n* bootstrap.

In this case, we first fix a small $b < n$. Then, we sample from B the datasets $B_1, B_2, \ldots, B_m$, where each dataset has size b. We distribute those datasets to the m machines in the cluster. At each machine i, locally, we perform the standard Bootstrap using dataset B_i but sampling new datasets of the original size n:

$$B_{i,1}, B_{i,2}, \ldots, B_{i,m}.$$

Note that these datasets, for fixed i, do not contain all the distinct elements of B, in general. We locally compute the estimator over the datasets: $\widehat{\theta}_{i,j} = \widehat{\theta}_n(B_{i,j})$. Finally, the estimator at machine i is the average of the previous ones:

$$\widehat{\theta}_i = \frac{1}{m} \sum_{j=1}^{m} \widehat{\theta}_{i,j}.$$

Every machine computes and returns its local estimator, and we average them in a similar manner to find that:

$$\widehat{\theta}_i = \frac{1}{m} \sum_{i=1}^{m} \widehat{\theta}_i.$$

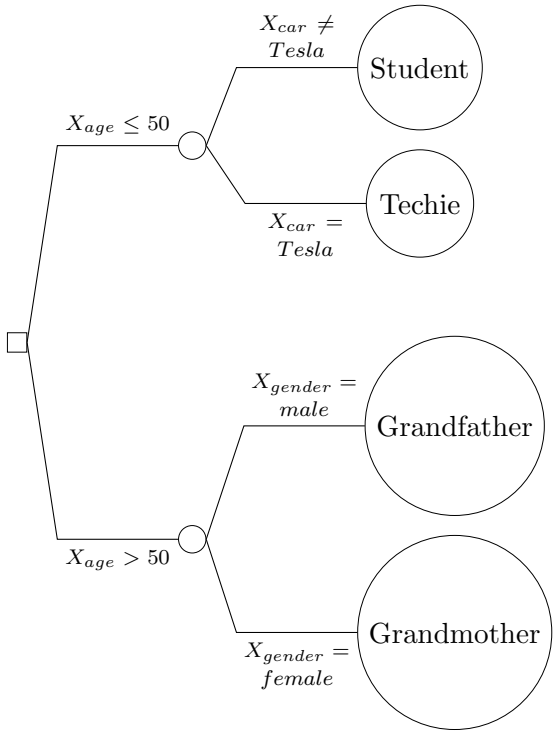
That way we keep unbiasedness, but we can reduce the variance of the estimator.

17 Decision Trees

17.1 Single Machine Algorithm

Decision trees are a popular machine learning algorithm that can be used for either classification or regression. A decision tree consists of a series of binary splits which partition the feature space. At each interior node, the data in that node is split into two based on the value of one variable. At each terminal node (leaf), a prediction is made for all the data in that leaf.

The features can be categorical or numeric. For example, we could have features $x_i \in \{male, female\}$ and $x_j \in \mathbb{R}$. An example of a decision tree:



The objective in building a decision tree is to increase the purity of the resulting partition of the training data. For classification problems, there are multiple measures of purity, such as the Gini impurity ($\sum_{i=1}^{C} f_i(1-f_i)$) or the entropy ($\sum_{i=1}^{C} f_i \log(f_i)$), where f_i is the frequency of class i. For regression problems, the objective is to minimize some loss function, often squared error loss.

There are many different variations of algorithms to build decision trees. We will focus on the ID3 algorithm. We begin with all the training data in the root node. At each step of the algorithm, for every node that is impure we pick the best split to split the data in that node into two daughter nodes. To find the best split, we have to search for the variable and the split value of that variable that most increases the overall purity. The algorithm continues recursively partitioning the data until the resulting nodes are pure (i.e., all of one class). This is a greedy algorithm that may get stuck in local minima. It is also prone to badly overfitting, so the usual practice is to use a held out cross-validation data set to prune the tree.

If we have d categorical features, and at most k nominal values for each feature, then at each node we must consider at most dk possible splits.

For numeric features, we generally want to discretize the data into buckets to reduce the amount of computation. Usually the data isn't sorted, so we would be unable to do a binary search to consider all possible split values. To avoid this problem, we use an equidepth histogram to bucket the data. For example, if the data were $1, 2, 2, 3, 4, 7, 8, 9, 10, 10, 10, 10$, and we wanted four points in each bucket, the new categories would be $1, 2, 2, 3|4, 7, 8, 9|10, 10, 10, 10$.

17.2 Distributed Algorithm

We consider the case where the data is parallel, but the model isn't (so dk isn't too large). At each step of the algorithm:

1. Each machine loads the current model
2. For impure nodes, consider all possible splits
3. Each machine can compute sufficient statistics for each possible split on data local to that machine

For each possible split, each machine looks at its own local training examples which fall into that node (if any), and computes the number of examples in each class, for both sides of the split. This data is then communicated, and the information from all the machines is used to choose the best split.

The key is that purity is only a function of the class counts in each node, so each machine only needs to report the class counts resulting from each possible split.

The algorithm continues in this way until the data in a node can fit in a machine's local memory, at which point we proceed with the single machine algorithm for that node. Note that this involves sending data around the network in order to get all the examples in a node onto one machine. It is worth the cost of this shuffle of the data to obtain the speed increase from running the algorithm locally.

References

[1] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.