Decision Trees on MapReduce

CS246: Mining Massive Datasets
Jure Leskovec, Stanford University
http://cs246.stanford.edu
Input features:
- \(d\) features: \(X_1, X_2, \ldots, X_d\)
- Each \(X_j\) has domain \(O_j\)
  - Categorical: \(O_j = \{\text{red, blue}\}\)
  - Numerical: \(H_j = (0, 10)\)
- \(Y\) is output variable with domain \(O_Y\):
  - Categorical: Classification
  - Numerical: Regression

Data D:
- \(n\) examples \((x_i, y_i)\) where \(x_i\) is a \(d\)-dim feature vector, \(y_i \in O_Y\) is output variable

Task:
- Given an input data vector \(x\) predict \(y\)
**Decision Trees (1)**

- **Decision trees:**
  - Split the data at each internal node
  - Each leaf node makes a prediction

- **Lecture today:**
  - Binary splits: $X_j < v$
  - Numerical attrs.
  - Regression
How to make predictions?

- **Input:** Example $x_i$
- **Output:** Predicted $y_i'$

- “Drop” $x_i$ down the tree until it hits a leaf node

- Predict the value stored in the leaf that $x_i$ hits

$Y = 0.42$
How to construct a tree?

- **Training dataset D**, \(|D*|=100 \) examples

![Decision Tree Diagram]

- **A**: \(|D|=10\)
- **C**: \(|D|=90\)
- **D**: \(|D|=45\)
- **E**: \(|D|=30\)
- **F**: \(|D|=20\)
- **G**: \(|D|=25\)
- **H**: \(|D|=15\)
- **I**: \(|D|=20\)

Y = 0.42

# of examples traversing the edge
Imagine we are currently at some node $G$
  - Let $D_G$ be the data that reaches $G$
There is a decision we have to make:
  **Do we continue building the tree?**
  - If *yes*, which variable and which value do we use for a split?
  - If *not*, how do we make a prediction?
    - We need to build a “predictor node”
How to construct a tree?

- **Alternative view:**

![Diagram of a tree constructed through alternative view]

- $X_1$
- $X_2$
How to construct a tree?

Algorithm 1: BuildSubtree

Require: Node $n$, Data $D \subseteq D^*$

1. $(n \rightarrow \text{split}, D_L, D_R) = \text{FindBestSplit}(D)$ (1)
2. if StoppingCriteria($D_L$) then (2)
3. $n \rightarrow \text{left\_prediction} = \text{FindPrediction}(D_L)$ (3)
4. else
5. $n \rightarrow \text{left\_prediction} = \text{BuildSubtree}(n \rightarrow \text{left}, D_L)$ (n)
6. if StoppingCriteria($D_R$) then
7. $n \rightarrow \text{right\_prediction} = \text{FindPrediction}(D_R)$
8. else
9. $n \rightarrow \text{right\_prediction} = \text{BuildSubtree}(n \rightarrow \text{right}, D_R)$

- Requires at least a single pass over the data!
How to construct a tree?

(1) How to split? Pick attribute & value that optimizes some criterion

- Classification:
  Information Gain

  - $IG(Y|X) = H(Y) - H(Y|X)$
  - Entropy: $H(Z) = - \sum_{j=1}^{m} p_j \log p_j$
  - Conditional entropy:
    $H(W|Z) = - \sum_{j=1}^{m} P(Z = v_j) H(W|Z = v_j)$
    - Suppose $Z$ takes $m$ values ($v_1 \ldots v_m$)
    - $H(W|Z=v)$ ... Entropy of $W$ among the records in which $Z$ has value $v$
(1) How to split?

- **Regression:**
  - Find split \((X_i, v)\) that creates \(D, D_L, D_R\): parent, left, right child datasets and maximizes:
    \[
    |D| \cdot Var(D) - (|D_L| \cdot Var(D_L) + |D_R| \cdot Var(D_R))
    \]
  - \(Var(D) = \frac{1}{n} \sum_{i=1}^{\mid D \mid} (y_i - \bar{y})^2\) ... variance of \(y_i\) in \(D\)
  - For ordered domains sort \(X_i\) and consider a split between each pair of adjacent values
  - For categorical \(X_i\) find best split based on subsets
### (2) When to stop?

- Many different heuristic options
- Few ideas:
  1. **When the leaf is “pure”**
     - The target variable does not vary too much: $\text{Var}(y_i) < \varepsilon$
  2. **When # of examples in the leaf is too small**
     - For example, $|D| \leq 10$
(3) How to predict?

- Many options

  - Regression:
    - Predict average $y_i$ of the examples in the leaf
    - Build a linear regression model on the examples in the leaf

  - Classification:
    - Predict most common $y_i$ of the examples in the leaf
Announcement:
-- Machine Learning Gradiance quiz has been posted.
Due 2013-03-02 23:59 Pacific time
-- HW4 will be released today

Building Decision Trees Using MapReduce
Problem: Building a tree

- Given a large dataset with hundreds of attributes
- Build a decision tree!

General considerations:
- **Tree is small** (can keep it memory):
  - Shallow (~10 levels)
- **Dataset too large to keep in memory**
- **Dataset too big to scan over on a single machine**
- **MapReduce to the rescue!**
Parallel Learner for Assembling Numerous Ensemble Trees [Panda et al., VLDB ’09]

- A sequence of MapReduce jobs that builds a decision tree
- **Setting:**
  - Hundreds of **numerical** (discrete & continuous, but not categorical) attributes
  - Target variable is **numerical**: Regression
  - Splits are **binary**: $X_j < v$
  - Decision tree is small enough for each Mapper to keep it in memory
  - Data too large to keep in memory
PLANTET Architecture

Input data → Model → Master → Attribute metadata → Intermediat e results

MapReduce

FindBestSplit

BuildSubTree
We build the tree level by level

- One MapReduce step builds one level of the tree
- **Mapper** considers a number of possible splits \((X_i, v)\) on its subset of the data
- For each split it stores partial statistics
- Partial split-statistics is sent to Reducers
- **Reducer** collects all partial statistics and determines the best split
- Tree grows for one level
**Mapper** loads the model and info about which attribute splits to consider

- Each mapper sees a subset of the data $D^*$
- Mapper “drops” each datapoint to find the appropriate leaf node $L$
- For each leaf node $L$ it keeps statistics about
  1. the data reaching $L$
  2. the data in left/right subtree under split $S$

**Reducer** aggregates the statistics (1), (2) and determines the best split for each tree node
PLANT: Components

- **Master**
  - Monitors everything (runs multiple MapReduce jobs)

- **Three types of MapReduce jobs**
  - **(1) MapReduce Initialization** (run once first)
    - For each attribute identify values to be considered for splits
  - **(2) MapReduce FindBestSplit** (run multiple times)
    - MapReduce job to find best split when there is too much data to fit in memory
  - **(3) MapReduce InMemoryBuild** (run once last)
    - Similar to BuildSubTree (but for small data)
    - Grows an entire sub-tree once the data fits in memory

- **Model file**
  - A file describing the state of the model
PLANET: Components

(1) Master Node
(2) MapReduce Initialization (run once first)
(3) MapReduce FindBestSplit (run multiple times)
(4) MapReduce InMemoryBuild (run once last)
PLANTET: Master

- **Controls the entire process**
- **Determines the state of the tree and grows it:**
  - (1) Decides if nodes should be split
  - (2) If there is little data entering a tree node, Master runs an **InMemoryBuild** MapReduce job to grow the entire subtree
  - (3) For larger nodes, Master launches MapReduce **FindBestSplit** to evaluate candidates for best split
    - Master also collects results from FindBestSplit and chooses the best split for a node
  - (4) Updates the model
Master keeps two node queues:

- **MapReduceQueue (MRQ)**
  - Nodes for which $D$ is too large to fit in memory
- **InMemoryQueue (InMemQ)**
  - Nodes for which the data $D$ in the node fits in memory

The tree will be built in levels

- Epoch by epoch
Two MapReduce jobs:

- **FindBestSplit**: Processes nodes from MRQ
  - For a given set of tree nodes $S$, computes quality of a candidate split (for each node in $S$)

- **InMemoryBuild**: Processes nodes from InMemQ
  - For a given set of nodes $S$, completes tree induction at nodes in $S$ using the InMemoryBuild algorithm (discussed later)

Start the process by executing **FindBestSplit** on full data $D^*$
(1) Master Node
(2) MapReduce Initialization (run once first)
(3) MapReduce FindBestSplit (run multiple times)
(4) MapReduce InMemoryBuild (run once last)
Initialization job: Identifies all the attribute values which need to be considered for splits

- Initialization process generates “attribute metadata” to be loaded in memory by other tasks

Which splits to even consider?

- For small data we can sort the values along a particular feature and consider every possible split
- But data values may not be uniformly populated so many splits may not really make a difference

\[ x_j: 1.2, 1.3, 1.4, 1.6, 2.1, 7.2, 8.1, 9.8, 10.1, 10.2, 10.3, 10.4, 11.5, 11.7, 12.8, 12.9 \]

- Idea: Consider a limited number of splits such that splits “move” about the same amount of data
Splits for numerical attributes:

- For attribute $X_j$ we would like to consider every possible value $v \in O_j$
- Compute an approx. equi-depth histogram on $D^*$
  - **Idea:** Select buckets such that counts per bucket are equal

Use boundary points of histogram as splits
Goal:
- Equal number of elements per bucket (B buckets total)
- Construct by first **sorting** and then taking B-1 equally-spaced splits

Faster construction:
Sample & take equally-spaced splits in the sample
- Nearly equal buckets
PLANET: Components

(1) Master Node
(2) MapReduce \textit{Initialization} (run once first)
(3) MapReduce \textit{FindBestSplit} (run multiple times)
(4) MapReduce \textit{InMemoryBuild} (run once last)
**FindBestSplit**

- **MapReduce** job to find best split when there is too much data to fit in memory
- **Goal:** For a particular split node \( j \) find attribute \( X_j \) and value \( v \) that maximize:

\[
|D| \times \text{Var}(D) - (|D_L| \times \text{Var}(D_L) + |D_R| \times \text{Var}(D_R))
\]

- \( D \) ... training data \((x_i, y_i)\) reaching the node \( j \)
- \( D_L \) ... training data \( x_i \), where \( x_i^{(j)} < v \)
- \( D_R \) ... training data \( x_i \), where \( x_i^{(j)} \geq v \)

\[
\text{Var}(D) = \frac{1}{(n-1)} \sum_i y_i^2 - \left( \sum_i y_i \right)^2 / n
\]

**Note:** Variance can be computed from sufficient statistics: \( n, \Sigma y_i, \Sigma y_i^2 \)
FindBestSplit: Map

- **Mapper:**
  - Initialized by loading results of **Initialization task**
    - **Current model** (to find which node each datapoint $x_i$ ends up)
    - **Attribute metadata** (all split points for each attribute)
  - For each data record run the Map algorithm
  - For each node store statistics of the data entering the node and at the end emit (to all reducers):
    - `<NodeId, { Σy, Σy², Σ1 } >`
  - For each split store statistics and at the end emit:
    - `<SplitId, { Σy, Σy², Σ1 } >`
      - `SplitId = (node n, attribute X_j, split value v)`
FindBestSplit: Map

- **Requires**: Split node set $S$, Model file $M$
- **For every training record** $(x_i, y_i)$ **do**:
  
  Node $n = \text{TraverseTree}(M, x_i)$
  
  if $n \in S$:
    
    update $T_n \leftarrow y_i$ //store $\{\Sigma y, \Sigma y^2, \Sigma 1\}$ for each node

    for $j = 1 \ldots d$:
      // $d$... number of features
      
      $v = x_i^{(j)}$ // value of feature $X_j$ of example $x_i$
      
      for each split point $s$ of feature $X_j$, such that $s < v$:
        
        update $T_{n,j}[s] \leftarrow y_i$ //store $\{\Sigma y, \Sigma y^2, \Sigma 1\}$ for (node $n$, $X_j$, split $s$)

- **MapFinalize: Emit**
  
  - $<$NodeId, $\{\Sigma y, \Sigma y^2, \Sigma 1\}>$ // sufficient statistics (so we can later
  
  - $<$SplitId, $\{\Sigma y, \Sigma y^2, \Sigma 1\}>$ // compute variance reduction
FindBestSplit: Reducer

Reducer:

1. Load all the \(<\text{NodeId}, \text{List}\{\Sigma y, \Sigma y^2, \Sigma 1}\>\) pairs and aggregate the per node statistics.
2. For all the \(<\text{SplitId}, \text{List}\{\Sigma y, \Sigma y^2, \Sigma 1}\>\) aggregate and run the reduce algorithm.
3. For each NodeId, output the best split found:

Reduce(SplitId, values):

\[
\begin{align*}
\text{split} &= \text{NewSplit}(\text{SplitId}) \\
\text{best} &= \text{BestSplitSoFar}(\text{split.nodeid}) \\
\text{for } \text{stats} \text{ in } \text{values} & \quad \text{split.stats.AddStats(stats)} \\
\text{left} &= \text{GetImpurity(split.stats)} \\
\text{right} &= \text{GetImpurity(split.node.stats–split.stats)} \\
\text{split.impurity} &= \text{left} + \text{right} \\
\text{if } \text{split.impurity} < \text{best.impurity} & \quad \text{UpdateBestNodeSplit(split.nodeid, split)}
\end{align*}
\]
Back to the Master

- Collects outputs from FindBestSplit reducers `<Split.nodeid, feature, value, impurity>`
- For each node decides the best split
  - If data in $D_L/D_R$ is small enough put the nodes in the InMemoryQueue
    - To later run a MapReduce job `InMemoryBuild` on the node
  - Else put the new nodes into MapReduceQueue
PLANT: Components

(1) Master Node
(2) MapReduce Initialization (run once first)
(3) MapReduce FindBestSplit (run multiple times)
(4) MapReduce InMemoryBuild (run once last)
**InMemoryBuild: Map and Reduce**

- **Task:** Grow an entire subtree once the data fits in memory
- **Mapper:**
  - Initialize by loading current model file
  - For each record identify the node it falls under and if that node is to be grown, output `<NodeId, (x_i, y_i)>`
- **Reducer:**
  - Initialize by loading attribute file from Initialization task
  - For each `<NodeId, List{(x_i, y_i)}` run the basic tree growing algorithm on the data records
  - Output the best splits for each node in the subtree

---

Algorithm 1 InMemoryBuildNode

Require: Node n, Data D ⊆ D*

1: \((n \rightarrow \text{split}, D_L, D_R) = \text{FindBestSplit}(D)\)
2: if StoppingCriteria\((D_L)\) then
3: \(n \rightarrow \text{left\_prediction} = \text{FindPrediction}(D_L)\)
4: else
5: InMemoryBuildNode\((n \rightarrow \text{left}, D_L)\)
6: if StoppingCriteria\((D_R)\) then
7: \(n \rightarrow \text{right\_prediction} = \text{FindPrediction}(D_R)\)
8: else
9: InMemoryBuildNode\((n \rightarrow \text{right}, D_R)\)
**Overall system architecture**

- **Example:** Need to split nodes F, G, H, I
  - F, I small, run InMemoryGrow
  - G, H too big, run FindBestSplit({G, H}):
- **Map and Reduce:**
  - **FindBestSplit::Map** (each mapper)
    - Load the current model $M$
    - Drop every example $x_i$ down the tree
    - If it hits G or H, update in-memory hash tables:
      - For each node: $T_n: \text{(node)} \rightarrow \{\Sigma y, \Sigma y^2, \Sigma 1\}$
      - For each (split,node): $T_{n,js}: (\text{node, attribute, split_value}) \rightarrow \{\Sigma y, \Sigma y^2, \Sigma 1\}$
  - **Map::Finalize:** output the key-value pairs from above hashables
  - **FindBestSplit::Reduce** (each reducer)
    - Collect:
      - $T1: \langle \text{node, List}\{\Sigma y, \Sigma y^2, \Sigma 1\} \rangle \rightarrow \langle \text{node, } \{\Sigma \Sigma y, \Sigma \Sigma y^2, \Sigma \Sigma 1\} \rangle$
      - $T2: \langle (\text{node, attr. split}), \text{List}\{\Sigma y, \Sigma y^2, \Sigma 1}\rangle \rightarrow \langle (\text{node, attr. split}), \{\Sigma \Sigma y, \Sigma \Sigma y^2, \Sigma \Sigma 1\}\rangle$
    - Compute impurity for each node using $T1$, $T2$
    - Return best split to Master (which then decides on globally best split)
Practical considerations

- We need one pass over the data to construct one level of the tree!
- MapReduce set up and tear down cost
  - Per-MapReduce overhead is significant
    - Starting/ending MapReduce job costs time
  - Reduce tear-down cost by polling for output instead of waiting for a task to return
  - Reduce start-up cost through forward scheduling
    - Maintain a set of live MapReduce jobs and assign them tasks instead of starting new jobs from scratch
Practical considerations

- **Very high dimensional data**
  - If the number of splits is too large the Mapper might run out of memory
  - Instead of defining split tasks as a set of nodes to grow, define them as a set of nodes to grow and a set of attributes to explore
    - This way each mapper explores a smaller number of splits (needs less memory)
Learning Ensembles

- Learn multiple trees and combine their predictions
  - Gives better performance in practice
- Bagging:
  - Learns multiple trees over independent samples of the training data
  - Predictions from each tree are averaged to compute the final model prediction
Bagged Decision Trees

- **Model construction for bagging in PLANET**
  - When tree induction begins at the root, nodes of all trees in the bagged model are pushed onto the MRQ queue
  - Controller does tree induction over dataset samples
    - Queues will contain nodes belonging to many different trees instead of a single tree
- **How to create random samples of D***?
  - Compute a hash of a training record’s id and tree id
  - Use records that hash into a particular range to learn a tree
  - This way the same sample is used for all nodes in a tree
  - **Note**: This is sampling D* without replacement (but samples of D* should be created with replacement)
SVM vs. DT

**SVM**
- **Classification**
  - Usually only 2 classes
- **Real valued features**
  - (no categorical ones)
- **Tens/hundreds of thousands of features**
- **Very sparse features**
- **Simple decision boundary**
  - No issues with overfitting
- **Example applications**
  - Text classification
  - Spam detection
  - Computer vision

**Decision trees**
- **Classification & Regression**
  - Multiple (~10) classes
- **Real valued and categorical features**
- **Few (hundreds) of features**
- **Usually dense features**
- **Complicated decision boundaries**
  - Overfitting! Early stopping
- **Example applications**
  - User profile classification
  - Landing page bounce prediction