Decision Trees

- **Input features:**
  - N features: $X_1, X_2, \ldots X_N$
  - Each $X_j$ has domain $D_j$
    - Categorical: $D_j = \{\text{red, blue}\}$
    - Numerical: $D_j = (0, 10)$
  - $Y$ is output variable with domain $D_Y$:
    - Categorical: Classification
    - Numerical: Regression

- **Task:**
  - Given input data vector $x_i$ predict $y_i$
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
How to construct a tree?

- Training dataset $D^*$, $|D^*| = 100$ examples

```
A
  /  \
|D|=10 |D|=90
   / \
  X_1<v_1

Y=0.42

D
  /  \
|D|=45 |D|=45
   / \
  X_2<v_2

C
  /  \
|D|=45 |D|=45
   / \
  X_1<v_1

E
  /  \
|D|=15 |D|=30
   / \
  X_2<v_5

|D|=25 |D|=20
  /  \
F  G

|D|=10 |D|=15
  /  \
H  I
```

# of examples traversing the edge
Imagine we are currently at some node $G$

- Let $D_G$ be the data reaches $G$
- There is a decision we have to make:
  
  **Do we continue building the tree?**

  - If so, 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:
How to construct a tree?

Algorithm 1 InMemoryBuildNode

Require: Node $n$, Data $D \subseteq D^*$
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($n \rightarrow \text{right}, D_R$)

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

- **How to split?** Pick attribute & value that optimizes some criterion

- **Classification:** Information Gain

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

- **How to split?** Pick attribute & value that optimizes some criterion

  - **Regression:**
    - Find split \((X_i, v)\) that creates \(D, D_L, D_R\): parent, left, right child datasets and maximizes:
      \[
      |D| \cdot \text{Var}(D) - \left( |D_L| \cdot \text{Var}(D_L) + |D_R| \cdot \text{Var}(D_R) \right)
      \]
    - 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 (Breiman’s algorithm)
How to construct a tree?

- **When to stop?**
  - 1) When the leaf is “pure”
    - E.g., $\text{Var}(y_i) < \epsilon$
  - 2) When # of examples in the leaf is too small
    - E.g., $|D| \leq 10$

- **How to predict?**
  - **Predictor:**
    - **Regression:** Avg. $y_i$ of the examples in the leaf
    - **Classification:** Most common $y_i$ in the leaf
Building a tree 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!**

---

```plaintext
Algorithm 1 FindBestSplit
Require: Node n, Data D ⊆ D*
1: (n →split,DL,DR) = FindBestSplit(D)
2: if StoppingCriteria(DL) then
3:     n →left prediction = FindPrediction(DL)
4: else
5:     FindBestSplit(n →left,DL)
6: if StoppingCriteria(DR) then
7:     n →right prediction = FindPrediction(DR)
8: else
9:     FindBestSplit(n →right,DR)
```
MapReduce

Can use a secondary key to control ordering in which reducers see key-value pairs.
Parallel Learner for Assembling Numerous Ensemble Trees [Panda et al., VLDB ’09]

- A sequence of MapReduce jobs that build a decision tree

**Setting:**
- Hundreds of numerical (discrete & continuous) attributes
- Target (class) 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
**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) and (2) and determines the best split for each node
PLANT: Components

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

- **MapReduce Initialization**
  - For each attribute identify values to be considered for splits

- **MapReduce FindBestSplit**
  - MapReduce job to find best split when there is too much data to fit in memory

- **MapReduce InMemoryBuild**
  - Similar to FindBestSplit (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

---

Algorithm 1: **FindBestSplit**

```plaintext
Require: Node n, Data D \subseteq D^*
1: (n \rightarrow \text{split}, D_L, D_R) = \text{FindBestSplit}(D)
2: if StoppingCriteria(D_L) then
3: \quad n \rightarrow \text{left prediction} = \text{FindPrediction}(D_L)
4: else
5: \quad \text{FindBestSplit}(n \rightarrow \text{left}, D_L)
6: if StoppingCriteria(D_R) then
7: \quad n \rightarrow \text{right prediction} = \text{FindPrediction}(D_R)
8: else
9: \quad \text{FindBestSplit}(n \rightarrow \text{right}, D_R)
```

---

Hardest part
Initialization: Attribute metadata

- Identifies all the attribute values which need to be considered for splits
- **Splits for numerical attributes:**
  - Would like to consider very possible value \( v \in D \)
  - Compute an approximate equi-depth histogram on \( D^* \)
    - **Idea:** Select buckets such that counts per bucket are equal
  - Use boundary points of histogram as potential splits
- Generates an “attribute metadata” to be loaded in memory by other tasks
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
PLANT: Master

- Controls the entire process
- **Determines the state of the tree and grows it:**
  - Decides if nodes should be split
  - If there is little data entering a node, runs an InMemory-Build MapReduce job to grow the entire subtree
  - For larger nodes, launches MapReduce FindBestSplit to find candidates for best split
  - Collects results from MapReduce jobs and chooses the best split for a node
  - Updates 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 the MRQ
  - For a given set of nodes $S$, computes a candidate of good split predicate for each node in $S$

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

Start by executing FindBestSplit on full data $D^*$
FindBestSplit

- MapReduce job to find best split when there is too much data to fit in memory
- **Goal:** For a particular split node 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
- $D_L$ ... training data $x_i$, where $x_{i,j} < v$
- $D_R$ ... training data $x_i$, where $x_{i,j} \geq v$
- $\text{Var}(D) = 1/(n-1) \sum_i y_i^2 - (\sum_i y_i)^2/n$

**Note:** Can be computed from sufficient statistics: $\Sigma y_i, \Sigma y_i^2$
FindBestSplit: Map

- **Mapper:**
  - Initialize by loading from Initialization task
    - Current Model (to find which node each $x_i$ ends up)
    - Attribute metadata (all split points for each attribute)
  - For each record run the Map algorithm
  - For each node store statistics and at the end emit (to all reducers):
    - `<Node.Id, \{ \Sigma y, \Sigma y^2, \Sigma 1 \}>`
  - For each split store statistics and at the end emit:
    - `<Split.Id, \{ \Sigma y, \Sigma y^2, \Sigma 1 \}>`
    - `Split.Id = (node, feature, split value)`
FindBestSplit: Map

- Requires: Split node set S, Model file M, Training record \((x_i, y_i)\)

Node \(n = \text{TraverseTree}(M, x_i)\)

if \(n \in S:\)

- Update \(T_n \leftarrow y_i\) //stores \(\{\Sigma y, \Sigma y^2, \Sigma 1\}\) for each node

for \(j = 1 \ldots N:\) // \(N\)... number of features

- \(v = \text{value of feature } X_j \text{ of example } x_i\)

for each split point \(s \text{ of feature } X_j, s.t. s < v:\)

- Update \(T_{n,j}[s] \leftarrow y_i\) //stores \(\{\Sigma y, \Sigma y^2, \Sigma 1\}\) for each (node, feature, split)

- MapFinalize: Emit

  - <Node.Id, \{ \Sigma y, \Sigma y^2, \Sigma 1 \}> // sufficient statistics (so we can later
  - <Split.Id, \{ \Sigma y, \Sigma y^2, \Sigma 1\}> // compute variance reduction)
FindBestSplit: Reducer

Reducer:

1) Load all the `<Node_Id, List {Σy, Σy^2, Σ1}>` pairs and aggregate the per node statistics
2) For all the `<Split_Id, List {Σy, Σy^2, Σ1}>` aggregate and run the reduce algorithm

For each Node_Id, output the best split found:

Reduce(Split_Id, values):

split = NewSplit(Split_Id)
best = BestSplitSoFar(split.node.id)
for stats in values
    split.stats.AddStats(stats)
left = GetImpurity(split.stats)
right = GetImpurity(split.node.stats–split.stats)
split.impurity = left + right
if split.impurity < best.impurity:
    UpdateBestSplit(Split.Node.Id, split)
Collects outputs from FindBestSplit reducers

\(<\text{Split.Node.Id, feature, value, impurity}>\)

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 InMemoryBuild on the node
- Else put the nodes into MapReduceQueue
**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 `<Node_Id, Record>`

**Reducer:**
- Initialize by loading attribute file from Initialization task
- For each `<Node_Id, List{Record}>` run the basic tree growing algorithm on the records
- Output the best splits for each node in the subtree
Overall system architecture

- Need to split nodes F, G, H, I
- \( D_1, D_4 \) small, run InMemoryGrow
- \( D_2, D_3 \) too big, run FindBestSplit\(\{G, H\}\):
  - **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: \( T_{n,j,s}: (\text{node}, \text{attribute}, \text{split_value}) \rightarrow \{\Sigma y, \Sigma y^2, \Sigma 1\} \)
    - Map::Finalize: output the key-value pairs from above hashtables
  - **FindBestSplit::Reduce** (each reducer)
    - Collect:
      - \( T1:<\text{node}, \text{List}\{\Sigma y, \Sigma y^2, \Sigma 1\}> \rightarrow <\text{node}, \{\Sigma \Sigma y, \Sigma \Sigma y^2, \Sigma \Sigma 1\}> \)
      - \( T2:<(\text{node, attr. split}), \text{List}\{\Sigma y, \Sigma y^2, \Sigma 1\}> \rightarrow <(\text{node, attr. split}), \{\Sigma \Sigma y, \Sigma \Sigma y^2, \Sigma \Sigma 1\}> \)
    - Compute impurity for each node using \( T1, T2 \)
    - Return best split to Master (that decides on the globally best split)
Practical considerations

- We need one pass over the data to construct one level of the tree!

- **Set up and tear down**
  - 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
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
  - 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
  - Real valued and categorical features
  - Few (hundreds) of features
  - Usually dense features
  - Complicated decision boundaries
    - Overfitting!
  - Example applications
    - User profile classification
    - Landing page bounce prediction
Experiments: Bounce Rate Prediction

- **Google:** Bounce rate of ad = fraction of users who **bounced from ad landing page**
  - Clicked on ad and quickly moved on to other tasks
  - Bounce rate high --> users not satisfied

- **Prediction goal:**
  - Given an new add and a query
  - Predict bounce rate using query/ad features

- **Feature sources:**
  - Query
  - Ad keyword
  - Ad creative
  - Ad landing page
Experimental Setup

- **MapReduce Cluster**
  - 200 machines
  - 768MB RAM, 1GB Disk per machine
  - 3 MapReduce jobs forward-scheduled

- **Full Dataset:** 314 million records
  - 6 categorical features, cardinality varying from 2-500
  - 4 numeric features

- Compare performance of PLANET on whole data with *R* on sampled data
  - *R* model trains on 10 million records (~ 2GB)
  - Single machine: 8GB, 10 trees, each of depth 1-10
  - Peak RAM utilization: 6GB
Results: Scalability
Results: Prediction accuracy

- Prediction accuracy (RMSE) of PLANET on full data better than $R$ on sampled data