David Katz

Eli Friedman

Harrison Zhao

**Distributed C4.5 Decision Tree**

**Overview**

Our application creates and displays a decision tree using hadoop’s mapreduce. The motivation behind this application is that C4.5 is amongst the most used data mining algorithms and large datasets on the order of millions of lines would cause memory overflows on a single computer. The application takes as command line arguments input of a training dataset directory, an output directory, both stored on HDFS and the location of the output class attribute in a training line. Our application builds a decision tree using the C4.5 algorithm, which decides on what attributes to split on and where to split each attribute based on highest normalized information gain. The motivation behind this pro

**Where are input and output files stored**

All files are stored on the HDFS. The input training sets must be stored on /tmp/input. Intermediate output and input files from mapreduce jobs The output tree file will be stored on one’s local machine and it will also be stored on HDFS in the output directory’s tree sub-directory. The decision tree is formated as an XML file.

**How the algorithm works**

There are three mapreduce jobs: defining attributes, finding the best splits for each tree leaf node-attribute pair, and finding the best split for each leaf node. Of the three, finding the best splits for each tree leaf node-attribute pair and finding the best split for each leaf node are run repeatedly until tree is fully grown.

The first job is defining attributes. The map class performs the following transformation: <Does not matter, Training line> -> <AttributeId, List of attribute values>. The reduce class takes as input the output of the mapping and creates attribute definitions. It defines all attributes of a training file and produces counts for categorical attributes and defines a minimum and maximum range for numerical attributes. Below are examples of how attributes are defined and listed in the tree file:

For numerical attributes:

<attribute index="8" isCategorical="false" minValue="1.0" maxValue="10.0" count="541" />

For categorical attributes:

<attribute index="0" isCategorical="true" count="2417">

<category value="f" count="2141" />

<category value="t" count="276" />

</attribute>

After the attributes are defined, the application creates the tree and begins growing the tree. The tree grows incrementally by depth.

The second job finds the best splits for each tree leaf node-attribute pair. The mapper performs the following transformation: <Does not matter, Training line> -> <(leafnodeId, attributeId), [(leafnodeId,attributeId,minval,maxval) if numerical] or [(attributeValue,outputClass) if categorical>. The reduce takes the output of the map as input and finds the maximum information gain split for each attribute corresponding to a given leaf node. While the original C4.5 algorithm calculates the information gain at each split for numerical attributes, our algorithm buckets numerical attributes and limits the number of possible splits to evaluate information gain to the number of buckets used. It outputs each split as a string of the form: (leaf,attributeId, attributeSplitRule, information gain, [classcounts], [classcounts]).

The last job finds the best split for each leaf node. It uses the output of the previous job as input. The mapper performs the following transformation:  <Does not matter, (leaf id ,attributeId, attributeSplitRule, information gain, [classcounts for lessThan(numeric) / for isEqualTo(categorical)] [classcounts for greaterThan(numeric)/ for isNotEqualTo(categorical)]> -> <leafNodeId, Mapper input without leaf id>. The reducer takes the output of the mapper and emits the best split for each given leaf based on highest information gain.

After running the second and third job, the splits are added to the tree and if any splits were added, the the second and third job is run again. Eventually, the tree will be fully grown and the application will terminate. The overall tree file is laid out as follows:

<tree outputClassAtIndex="[index output class is at (min index is 0, max is #attributes-1)]">

 <attributes>

   [attribute definitions]

 </attributes>

 <root id="0" isLeaf="false">

   <classCounts>

     [class counts]

   </classCounts>

   <split [split criteria]/>

   <trueChild id="[nodeId]" isLeaf="[true/false]">

     [node info]

   </trueChild>

   <falseChild id="[nodeId]" isLeaf="[true/false]">

     [node info]

   </falseChild>

 </root>

</tree>

**Results**

Running the C4.5 algorithm on the chess dataset (<https://archive.ics.uci.edu/ml/datasets/Chess+(King-Rook+vs.+King-Pawn)>), we achieved 94.1% accuracy. The algorithm was also ran on the SUSY dataset (<https://archive.ics.uci.edu/ml/datasets/SUSY>), which has five million lines of data. We generated graphs from varying the input data from the SUSY dataset.

**Displaying results**

David Galles’s Data Structure Visualizer library [<http://www.cs.usfca.edu/~galles/visualization/Algorithms.html>] was used to display the results. The xml file representing the decision tree was parsed using the jkl-parsexml library. The tree was then recursively traversed and output to the screen.  The Data Structure Visualizer library was used to animate the tree making a decision. The attributes that the tree decides are chosen randomly. When one presses the “decide” button, one may watch the decision tree decide which class the set of attributes is in.