Introduction

This document explains in more detailed terms the process of inducing decision tree. I will explain the probability model and related formulas inductively and then use that to explain the design of the data structures and the algorithm. The algorithm is designed to run feasibly in parallel GPU environment and I will address some of the performance consideration there. However, there are still a number of practical GPU architecture specific issues that need to be ironed out to get the best performance.

Basic Concepts

Assume that we have a base data set with T instances showing positive result and F instances showing negative result. Knowing T and F, what is the probability that we will guess correctly whether the result is positive or negative given a random instance?

<table>
<thead>
<tr>
<th>T</th>
<th>F</th>
</tr>
</thead>
</table>

The answer is \( \frac{\text{MAX}(T, F)}{(T+F)} \). If \( T>F \), the rational guess will be \( T \) and you will be correct \( \frac{T}{T+F} \)% of the time. If \( F>T \), the rational guess will be \( F \) and you will be correct \( \frac{F}{T+F} \)% of the time. If \( T=F \), you will be correct 50% of the time.

Now, assume that we have a split condition called A-B that can be used to group the instances into A and B. We now have four subsets, Positive samples in A, Negative samples in A, Positive samples in B, Negative samples in B. Knowing their sizes of these samples, and assuming that we can measure whether an instance belongs to A or B, what is the probability that we will guess correctly whether the result is positive or negative given a random instance?

<table>
<thead>
<tr>
<th>TA</th>
<th>TB</th>
<th>FA</th>
<th>FB</th>
</tr>
</thead>
</table>

It depends on whether the instance is A or B. If it is an A, then the answer is \( \frac{\text{MAX}(TA, FA)}{(TA+FA)} \). If it is a B, then the answer is \( \frac{\text{MAX}(TB, FB)}{(TB+FB)} \).

But note that we actually know also the probability of whether we will get an A or B. The probability of getting an A sample, \( P(A) \), is \( \frac{(TA+FA)}{(TA+TB+FA+FB)} \). The probability of getting a B sample, \( P(B) \), is \( \frac{(TB+FB)}{(TA+TB+FA+FB)} \). And, \( P(A) + P(B) = 1 \).

Therefore, the probability of guessing right is:

\[
P(A) \times \frac{\text{MAX}(TA, FA)}{(TA+FA)} + P(B) \times \frac{\text{MAX}(TB, FB)}{(TB+FB)}
\]

\[
= \frac{(TA+FA)\times\text{MAX}(TA, FA)}{((TA+FA)\times(T+F))} + \frac{(TB+FB)\times\text{MAX}(TB, FB)}{((TB+FB)\times(T+F))}
\]

\[
= \frac{\text{MAX}(TA, FA)}{(T+F)} + \frac{\text{MAX}(TB, FB)}{(T+F)}
\]

The first term is the odd given A. The second term is the odd given B.
We can speak of the gain in probability of guessing right given an instance $A$ as the first term minus the original odd scaled to the proportional size of $A$.

Gain(for $A$ instances) = \( \frac{\text{MAX}(T_A, F_A) - P(A) \times \text{MAX}(T, F)}{T+F} \)

Similarly,

Gain(for $B$ instances) = \( \frac{\text{MAX}(T_B, F_B) - P(B) \times \text{MAX}(T, F)}{T+F} \)

By induction, we can replace the formula using a general parent to notate the original set which we know the ratio of $T$ and $F$.

Gain(for $A$ children) = \( \frac{\text{MAX}(T_A, F_A) - P(A \mid parent) \times \text{MAX}(T_{parent}, F_{parent})}{\text{All}} \)

[Note that $P(A \mid parent)$ means the probability of $A$ in parent]

For the purpose of comparison, we can use a scale of $1/\text{All}$. And the formula becomes:

Gain(A over Parent) = \( \text{MAX}(T_A, F_A) - P(A \mid Parent) \times \text{MAX}(T_{parent}, F_{parent}) \) … (1)

In enough words, adding a binary split gains us that much for one side (the “A” side) of the split.

The more powerful split is the one that gives us a bigger gain.

In a 3-layer split, we can re-apply the formula. Suppose $A$ can be further divided into $AA$ and $AB$.

Gain(AA over $A$) = \( \text{MAX}(T_{AA}, F_{AA}) - P(AA \mid A) \times \text{MAX}(T_A, FA) \)

Total probability broken down in this form:

\[ \text{MAX}(T_{AA}, F_{AA}) - P(AA \mid A) \times \text{MAX}(T_A, FA) + \text{MAX}(T_A, FA) - P(A \mid Parent) \times \text{MAX}(T_{parent}, F_{parent}) + \text{MAX}(T_{parent}, F_{parent}) \]

or

\[ \text{MAX}(T_{AA}, F_{AA}) + P(\text{NOT AA} \mid A) \times \text{MAX}(T_A, FA) + P(\text{NOT A} \mid Parent) \times \text{MAX}(T_{parent}, F_{parent}) \] … (2)

By induction, it is not hard to see that this formula can be simply expanded to arbitrary levels deep. Say we have 5 layers.

\[ \text{MAX}(\text{TAAAAA, FAAAAA}) + P(\text{NOT AAAAA} \mid AAAA) \times \text{MAX}(\text{TAAAA, FAAAA}) + P(\text{NOT AAAA} \mid AAA) \times \text{MAX}(\text{TAAA, FAAA}) + P(\text{NOT AAA} \mid AA) \times \text{MAX}(\text{TAA, FAA}) + P(\text{NOT AA} \mid A) \times \text{MAX}(\text{TA, FA}) + P(\text{NOT A} \mid \text{Root}) \times \text{MAX}(\text{T Root, F Root}) \] … (3)
This is one of many possible split paths. We observe nevertheless that if we figure out a way to maximize the one path, we can use the same technique to maximize all the paths.

**Maximizing the Path**

ID3 and C4.5 are the two most common decision tree induction algorithms. Both of them start from the top of the tree, find a split condition that gives it the most gain, and then recursively apply the procedure on both sides of the split.

In terms of the equations above: suppose there is already a split A-B in parent and A has given a gain of (1) on top of MAX(T Parent, F Parent). The goal is to find an AA-AB split in A that maximize (2) as well as the corresponding equation for AB. It is done by simply trying all the possible splits until the one that gives the maximum gain.

Looking only for the maximum immediate gain does not always give the best overall results. Moreover, a serial, top-down, split driven strategy is not easy to turn into a parallel algorithm.

Our goal is to leverage parallel computing hardware to implement strategies that lead to generally better results while being relatively efficient.

We start by establishing certain conventions.

First, suppose we have available K conditions that can be used for splitting the data, we can represent each of the leaf node, or the unit “data cube”, using a bit string of K bits, with each bit denoting the conditional value, 1 being true and 0 being false. There are 2^K number of such strings. For example, if there are 16 conditions, we have 65536 such leaves or data cubes.

Second, we can describe the possible union super-sets or aggregate data cubes by using a tertiary number notation. In addition to having 0 meaning condition false and 1 meaning condition true, * can mean wild-card (do not care). If there are K conditions, there are 3^K number of possible strings. For example, if there are 16 conditions, we have 43,046,721 such leaves or data cubes.

Third, for all leaf nodes/unit data cube as well as super-set/aggregate data cubes, we can compute a T size and a F size. We assume that a statistics table of 3^K rows and 2 columns to hold such information. For example, suppose we have 2 conditions, this is how the table can look like.

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

Fourth, in order to find the strategy to find the optimal split path to a given “target” leaf node or data cube, we define a new binary notation, where * means wild-card and _ (underscore) means the same as the target leaf node. If there are K conditions, there are 2^K of possible strings constructed from this notation of * and _. 
All possible paths, including the optimal, that travel from the top of the tree to the target leaf node have two basic characteristics in terms of the * and _ notations:

1. All such paths start from one end to the other. For example, it goes from ** to _ _ if there are 2 conditions.
2. For a given path, there are K steps. Each step removes a *.

Each row corresponds to a subset in the $3^K$ possible sets described in the second point.

Fifth, in order to record the paths to determine the optimum, we keep a two columns table with * and _ to define the $2^K$ rows. The last wild-card and the accumulated gains are the columns. For example, this is how a 3-condition path-table looks like:

<table>
<thead>
<tr>
<th>Last Wild-card</th>
<th>Accumulated Gains</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ _ _</td>
<td></td>
</tr>
<tr>
<td>_ _ *</td>
<td></td>
</tr>
<tr>
<td>_ * _</td>
<td></td>
</tr>
<tr>
<td>* _ _</td>
<td></td>
</tr>
<tr>
<td>_ * _</td>
<td></td>
</tr>
<tr>
<td>_ * *</td>
<td></td>
</tr>
<tr>
<td>* _ *</td>
<td></td>
</tr>
<tr>
<td>_ * _</td>
<td></td>
</tr>
<tr>
<td>* * _</td>
<td></td>
</tr>
<tr>
<td>* * *</td>
<td></td>
</tr>
</tbody>
</table>

The last wild-card records which of the condition turned from * to _ to make it to the present row. For example, if _ _ * came from * _ *, we might have a value of 1, indicating that it is the first condition. But if _ _ * came from _ * *, it would be a value of 2.

The accumulated gain records the accumulated gain by computing the formula logic of the form of (3) above. In other words, it uses the table computed in the second step to determine its own:

\[ \text{MAX}(T \text{ this-subset, F this-subset}) \ldots (4) \]

It uses that table also to compute the

\[ \text{P(this-subset | parent-set) \times MAX(T parent-set, F parent-set)} \ldots (5) \]

Then it computes its own accumulated gain as:

\[ [4] - [5] + \text{parent_accumulated_gains} \ldots (6) \]

Sixth, it is up to a row to determine where it should come from, or in other words, who its parent should be – where the last wild-card is. Unlike a look-ahead algorithm, here we have a logically multiple-path parallel search. It is a row (or a node candidate in the tree’s) responsibility to look-backward to see find a parent set which will give it the maximum value of (6).

Incidentally, it does not necessarily mean finding the parent with the greatest accumulated gains, the last term in (6). Since (4) is fixed for a row, it is really about finding (5) and (6) together.

Each row has between 1 to K parents that it need to try before it finds the “best” parent. At the end, all will converge back to the first all _ row. Once that is reached, we can hop backward by chasing the last
wild-card to create the full path that got us from all * to all _.

**Parallel GPU Consideration**

The algorithm is designed to run in parallel. Specifically, there can be multiple targets that can be sought after in parallel. The path table can be one 32-bit word per row, using most bits to represent accumulated gains and a few for the last wild-card. In a parallel environment, multiple tables will be placed side by side so that threads access the columns together in contiguous memory access.

The chief bottleneck of the algorithm comes from the table look up of T this-subset, F this-subset, T parent-set, F parent-set. Anyway that can optimize this look up with intelligent caching can help the algorithm tremendously.

One obvious strategy is to store it as part of the path table, effectively expanding it to four columns. It doesn't mean the child does not need to go to the parent row to look up. It still does. And the parent row is in global memory just like the union data set table. The difference is that the parent row is a fixed address. And multiple parallel threads can access them contiguously. The union data set table look up, in contrast, will likely diverge anyway since we are dealing with different targets. But if it is looked up once per row, the performance drag is smaller.

There are clear trade offs in terms of memory and parallelism. However, those are also the issues that current parallel hardware evolution is focusing on. We will likely get help as parallel computing becomes more memory access friendly.

Because we are searching target nodes in parallel, and we explore many parallel paths by visiting all rows, there are a lot of redundant calculations. There may be some opportunities to gain performance by removing some redundancies.

**Conclusion**

This document describes a model for decision tree induction. It seeks to produce better results than one-step look-ahead algorithm by exploring many more paths. It does incur extra computation and memory cost. However, not only should this method give more predictive decision tree (which is after all what matters most), it is also a strategy that works well in a parallel computing environment.