LightGBM cheat sheet

GBT is a popular machine learning algorithm and has quite a few effective implementations such as XGBoost. Although many engineering optimizations have been adopted in these implementations, the efficiency is still unsatisfactory when the feature dimension is high and the data size is large. A major reason is that for each feature thez need to scan all the data instances to estimate the information gain of all possible split points (or build the histogram for histogram-based algorithm). To tackle this problem, LightGBM propose two novel techniques : Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bunding (EFB). All this work is based on author's paper : http://www.audentia-gestion.fr/MICROSOFT/lightgbm.pdf

Prerequisites : GBT, XGBoost

1) Histogram-based model

First, as XGBoost, LightGBM support histogram-based algorithm (also called approximate algorithm in opposite of greedy exact algorithm where all the instances are scanned for split finding) for trees construction :

```
Algorithm 1: Histogram-based Algorithm
Input: I: training data, d: max depth
Input: m: feature dimension
nodeSet ← {0} ▷ tree nodes in current level
rowSet ← {{0, 1, 2, ...}} ▷ data indices in tree nodes
for i = 1 to d do
    for node in nodeSet do
        usedRows ← rowSet[node]
        for k = 1 to m do
            H ← new Histogram()
            ▷ Build histogram
            for j in usedRows do
                bin ← I.f[k][j].bin
                H[bin].y ← H[bin].y + I.y[j]
                H[bin].n ← H[bin].n + 1
            Find the best split on histogram H.
            ...
    Update rowSet and nodeSet according to the best
    split points.
    ...
```

2) Gradient-based One-Side Sampling (GOSS)

GOSS is a new sampling method that can achieve a good balance between reducing the number of data instances and keeping a good accuracy.

It's based on the fact that the larger the gradient of an instance is, the more it contributes to the information gain computation in the split finding. (recall: GBT is an iterative addition of a regression tree model that fits the negative gradients).

Indeed, as we have the information gain for a split equals to :

$$gain_{split} = V(I) - [(\#I_L / \#I) * V(I_L) + (\#I_R / \#I) * V(I_R)] \quad (1)$$

with $V(I)$ the variance in the node I, and $I = I_R \cup I_L$ the right and left children nodes. As :

$$V(I) = \frac{1}{n} \sum_{x_i \in I} (g_i - \overline{g}_i)^2$$

by developing (1) we find :

$$V(I) = \frac{1}{\#I} \left[\frac{1}{\#I_R} \left(\sum_{x_i \in I_R} g_i \right)^2 + \frac{1}{\#I_L} \left(\sum_{x_i \in L} g_i \right)^2 \right] - cste$$

which shows that larger the gradient is the more it contributes to information gain computation.

GOSS is set up with two parameters : it select the top $\alpha * 100\ \%$percent of the top gradient absolute value instances, and randomly sample $\beta * 100\ \%$ from the rest. After that, it amplifies the "small gradient" sampled data by a constant $\frac{1-\alpha}{\beta}$ to balance the two data sets and not overfit the model on the top gradient data by giving them too much weight.

Author's shows that this strategy outperform in many case a random sampling, and that with n high the accuracy is close to a model with all the instances.

Algorithm 2: Gradient-based One-Side Sampling

```
Input: I: training data, d: iterations
Input: a: sampling ratio of large gradient data
Input: b: sampling ratio of small gradient data
Input: loss: loss function, L: weak learner
models ← {}, fact ← (1−a)/b
topN ← a × len(I) , randN ← b × len(I)
for i = 1 to d do
    preds ← models.predict(I)
    g ← loss(I, preds), w ← {1,1,...}
    sorted ← GetSortedIndices(abs(g))
    topSet ← sorted[1:topN]
    randSet ← RandomPick(sorted[topN:len(I)],
    randN)
    usedSet ← topSet + randSet
    w[randSet] × = fact ▷ Assign weight fact to the
    small gradient data.
    newModel ← L(I[usedSet], − g[usedSet],
    w[usedSet])
    models.append(newModel)
```

3) Exclusive Feature Bundling (EFB)

In a sparse feature space, many features are mutually exclusive, i.e., they never take nonzero values simultaneously. We can safely bundle exclusive features into a single feature (which we call an
exclusive feature bundle).

By a carefully designed feature scanning algorithm, we can build the same feature histograms from the feature bundles as those from individual features. In this way, the complexity of histogram building changes from $O(\#data \times \#feature)$ to $O(\#data \times \#bundle)$, while $\#bundle \ << \ \#feature$. Then we can significantly speed up the training of GBDT without hurting the accuracy.

→ Bundling can be set up with a small recovery rate γ.

Algorithm 3: Greedy Bundling

```
Input: F: features, K: max conflict count
Construct graph G
searchOrder ← G.sortByDegree()
bundles ← {}, bundlesConflict ← {}
for i in searchOrder do
    needNew ← True
    for j = 1 to len(bundles) do
        cnt ← ConflictCnt(bundles[j],F[i])
        if cnt + bundlesConflict[i] ≤ K then
            bundles[j].add(F[i]), needNew ← False
            break
    if needNew then
        Add F[i] as a new bundle to bundles
Output: bundles
```

Algorithm 4: Merge Exclusive Features

```
Input: numData: number of data
Input: F: One bundle of exclusive features
binRanges ← {0}, totalBin ← 0
for f in F do
    totalBin += f.numBin
    binRanges.append(totalBin)
newBin ← new Bin(numData)
for i = 1 to numData do
    newBin[i] ← 0
    for j = 1 to len(F) do
        if F[j].bin[i] ≠ 0 then
            newBin[i] ← F[j].bin[i] + binRanges[j]
Output: newBin, binRanges
```